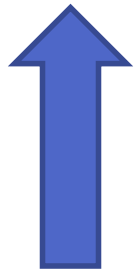
<functional>: What's New, And Proper Usage

Stephan T. Lavavej ("Steh-fin Lah-wah-wade")
Senior Developer - Visual C++ Libraries
stl@microsoft.com

Getting Started



- Please hold your questions until the end
  - Write down the slide numbers
- Everything here is Standard
  - Unless otherwise specified
- Almost everything here is available in VS 2015
  - Rewrote `<functional>` for Standard conformance
  - Minor issues will be listed at the end

Lambdas (C++11)

C++14: init-captures, generic lambdas

Example

```
vector<string> v{ "hydrogen", "helium",
  "lithium", "beryllium", "boron", "carbon",
  "nitrogen", "oxygen", "fluorine", "neon" };

stable_sort(v.begin(), v.end(),
  [](const auto& l, const auto& r) {
    return l.size() < r.size(); });

for (const auto& e : v) {
  cout << e << endl;
}
```

Lambdas Aren't Magical

- A lambda expression:
  - Defines a class
    - With `operator()(Args)` and maybe data members
  - Constructs an object
- Lambda syntax is convenient
  - Handwritten function objects are verbose
- Remember:
  - Lambdas are a Core Language feature
  - `std::function` is a Standard Library feature
  - Lambdas aren't `std::functions`

Lambdas Aren't Magical, Really

- Stateless lambdas: convertible to function pointers
  - Handwritten function objects can do that too
  - It's just `operator FunctionPointer()`
- Remember:
  - A lambda defines a class and constructs an object
  - Lambdas aren't functions
  - Lambdas aren't function pointers
- Never refer to lambdas as "anonymous functions"
  - Or I will make this face `>:-[/]`

invoke() (C++17)

Terminology: Function Objects

- Usable like functions: `func(args)`
  - Function pointers
    - Even in C, `fp(args)` means `(*fp)(args)`, thanks to DMR
  - Classes with `operator()(Args)`
    - Including lambdas
  - Classes with conversions to function pointers
    - Obscure!
- References to functions are similarly usable
  - Technically, reference types aren't object types

Terminology: Callable Objects

- Callable in a generalized sense
  - Function objects: `func(args)`
  - Pointers to member functions (PMFs): `(obj.*pmf)(args)`
  - Pointers to member data (PMDs): `obj.*pmd`
- The Core Language wants different syntax (hiss!)
  - `pmf(obj, args)` could be permitted, but isn't
- The Standard Library wants uniform syntax (purr!)
  - *`INVOKE`*`()` was imaginary in TR1 and C++11/14
  - `invoke()` is available in C++17

`invoke(callable, args...)`

- Function objects
  - `invoke(func, args...)` is `func(args...)`
- PMFs
  - `invoke(pmf, obj, rest...)` is `(obj.*pmf)(rest...)`
  - `invoke(pmf, ptr, rest...)` is `((*ptr).*pmf)(rest...)`
- PMDs
  - `invoke(pmd, obj)` is `obj.*pmd`
  - `invoke(pmd, ptr)` is `(*ptr).*pmd`
- Base PMFs/PMDs can be invoked on Derived things
- `invoke()` handles both raw pointers and smart pointers

Example

```
template <typename Range, typename Callable>
  void transform_print(const Range& r, Callable c) {
  for (const auto& e : r) {
    cout << invoke(c, e) << endl;
  }
}
vector<pair<int, int>> v{{4, 40}, {5, 50}, {6, 60}};
transform_print(v,
  [](const auto& p) { return p.first * p.first; });
transform_print(v,
  &pair<int, int>::second);
```

Things That Use invoke()

- <functional>
  - std::function
  - reference_wrapper
  - bind()
  - mem_fn()
- <type_traits>
  - result_of
- <future>
  - packaged_task
  - async()
- <mutex>
  - call_once()
- <thread>
  - std::thread

Recommendations

- In non-generic code, `invoke()` isn't very useful
  - Unless you really hate PMF syntax
- In generic code, `invoke()` can simplify things
  - Take/store arbitrary callable objects and arguments
  - Give them to `invoke()`, let it decide what to do
- Don't special-case PMFs/PMDs
  - Inspecting PMF types is a headache
  - Implementing `invoke()` behavior is extremely difficult

result_of (C++11)

C++14: SFINAE, result_of_t

result_of: Type Trait For invoke()

- C++11 <type_traits> (was TR1 <functional>)
- result_of_t<Callable> is incorrect
- result_of_t<Callable(Args...)> is:
  - decltype(invoke(declval<Callable>(), declval<Args>()...))
- declval() is declared but never defined:
  - template <typename T> add_rvalue_reference_t<T> declval() noexcept;
- In C++14, the ::type SFINAEs away

Example

```
template <typename T, typename Callable>
  auto transform_sort(const vector<T>& v, Callable c) {
  vector<decay_t<result_of_t<Callable&(const T&)>>> ret;
  for (const T& t : v) { ret.push_back(invoke(c, t)); }
  sort(ret.begin(), ret.end());
  return ret;
}
const vector<string> v{ "hydrogen", "helium", "etc." };
auto lambda = [](const string& s) { return s.size(); };
for (const auto& e : transform_sort(v, lambda)) {
  cout << e << endl;
}
```

`result_of` Is Tricky

- It answers "what's the type of this invocation?"
  - But it uses **different syntax** than the real invocation
- cv-qualifiers and value categories can matter
  - `declval<Callable/Args>()` must match real `callable/args`
  - Especially when C++11 ref-qualifiers are involved
- The real invocation might do extra work
  - `bind()` extensively manipulates its arguments
  - `async()` decays its arguments
- `result_of` is TR1-era tech, predating `decltype`

Recommendations

- Avoid using `result_of`
  - If you must use it, be careful
  - Audit existing usage for bugs, I bet you'll find some
- Use `decltype(STUFF)`, `decltype(auto)`, or `auto`
  - Which generic programmers already need to understand
- In general, avoid computing the same thing through different mechanisms
  - If repetition is necessary, prefer exactly repeating text
  - `decltype(STUFF)` matching `return STUFF;` is easy to see

mem_fn() (C++11)

Example

```
struct Element {
  bool is_metallic() const;
};

size_t count_metals(const vector<Element>& v) {
  return count_if(v.begin(), v.end(),
    mem_fn(&Element::is_metallic));
}
```

`mem_fn()` Isn't Fun, Really

- Good: Usually terse
- Bad: Resistant to optimization
- Ugly: Won't compile in certain situations
  - Overloaded member functions (need `static_cast`)
  - Templated member functions (use `static_cast`)
    - Avoid explicit template arguments, Don't Help The Compiler
  - Default arguments (no workaround)
- Unnecessary with anything powered by `invoke()`

Recommendations

- Avoid using `mem_fn()`
  - Algorithm inner loops often affect performance
  - As code evolves, `mem_fn()` is fragile
  - Auditing existing usage is low priority, though
- Use lambdas, especially generic lambdas
  - They're slightly more verbose
  - But they optimize away
  - And they always compile, like other member function calls

Transparent Operator Functors (C++14)

Example

```
vector<int> ints{ 6, 3, 10, 5, 16, 8, 4, 2, 1 };
vector<string> strs{
  "O'Neill", "Carter", "Jackson", "Teal'c" };

sort(ints.begin(), ints.end(), greater<>());
sort(strs.begin(), strs.end(), greater<>());

for (const auto& e : ints) { cout << e << endl; }
for (const auto& e : strs) { cout << e << endl; }
```

Would You Like To Know More?

- Transparent Operator Functors
  - "Don't Help The Compiler"
  - GoingNative 2013, slides 37-48
  - Gained `constexpr` before C++14 shipped
- Heterogeneous Associative Lookup
  - "STL Features And Implementation Techniques"
  - CppCon 2014, slides 33-40

Recommendations

- Use `greater<>` etc. by default
  - Except when you need implicit conversions (very rare)
- Advantages:
  - Avoids truncation/signedness bugs
  - Avoids unnecessary temporaries
  - Avoids unnecessary copies
  - Less verbose
- Unlike `mem_fn()`, library machinery is OK here
  - Operators are known in advance, perfectly special-cased

bind() (C++11)

Example

```
const vector<int> v{ 1, 4, 9, 16, 25, 36, 49,
  64, 81, 100, 121, 144 };

cout << count_if(v.begin(), v.end(),
  bind(less<>(), _1, 50)) << endl; // 7

cout << count_if(v.begin(), v.end(),
  bind(less<>(), 50, _1)) << endl; // 5
```

How `bind()` Works

- `auto b = bind(callable, bound_args...);`
  - Later: `b(unbound_args...);`
- `callable` and `bound_args` are copied or moved
  - Then they're passed as **lvalues** to `invoke()`
    - So `b` can be called repeatedly
  - With `b`'s `const`ness, via `const`-overloaded `operator()`
- Some bound arguments are special:
  - Placeholders: `_1` perfectly forwards first unbound arg
  - `reference_wrapper<T>`: unwrapped via `get()` to `T&`
  - Nested `bind()`: called with perfectly fwded unbound args
- Unused unbound arguments are ignored

`bind()` Problems

- Same performance/compiler issues as `mem_fn()`
  - With function pointers in addition to PMFs/PMDs
- Misuse emits ultra-disgusting compiler errors
- Syntax isn't normal C++, especially nested `bind()`
- No short-circuiting for `logical_and/logical_or`
- Surprising behavior: bound args passed as lvalues
  - Affects `unique_ptr`, etc.
- Surprising behavior: immediate vs. delayed calls
- Placeholders and nested `bind()` can move twice

Recommendations

- Avoid using `bind()`
- Use lambdas, especially generic lambdas
- `bind()`: good idea in 2005, bad idea in 2015
  - In C++, we usually prefer Library solutions to Core
  - But the Library is terrible at building up function objects
  - Lambdas were added to the Core Language for a reason
  - STL maintainers rarely recommend avoiding the STL
  - `bind()`'s terseness just isn't worth the price

reference_wrapper (C++11)

C++17: trivially copyable

Class Definition

```
template <typename T> class reference_wrapper {
public:
  typedef T type;
  reference_wrapper(T&) noexcept;
  reference_wrapper(T&&) = delete;
  operator T&() const noexcept;
  T& get() const noexcept;
  template <typename... Args>
    result_of_t<T&(Args&&...)>
    operator()(Args&&...) const;
};
```

Example

```
vector<int> v(8);
const auto b = v.begin(); const auto e = v.end();
typedef uniform_int_distribution<int> Dist;
auto d20 = [urng = mt19937(1729), dist = Dist(
  1, 20)]() mutable { return dist(urng); };
generate(b, e, d20); // 2 11 8 18 10 11 16 2
generate(b, e, d20); // 2 11 8 18 10 11 16 2
generate(b, e, ref(d20)); // 2 11 8 18 10 11 16 2
generate(b, e, ref(d20)); // 18 4 12 10 8 13 14 10
```

Recommendations

- Almost all algorithms take function objects by value
  - And are allowed to copy them
- Use `ref()` to pass function objects by reference
  - Avoid explicit template arguments, Don't Help The Compiler
- `reference_wrapper` is useful elsewhere
  - Like `std::thread`'s constructor
  - But first, learn why it uses *`DECAY_COPY`*`()`
- Be aware of the 3 functions that unwrap
  - `make_pair()`, `make_tuple()`, and `bind()`
  - `make_tuple(x, ref(y), cref(z))` is `tuple<X, Y&, const Z&>`

Removed Old <functional> Stuff (C++17)

Erased... From Existence

- Deprecated in C++11, removed in C++17
- `unary_function/binary_function`
  - Provided `result_type`, etc.
- `ptr_fun()`
  - Wrapped function pointers with `result_type`, etc.
- `mem_fun()/mem_fun_ref()`
  - Strictly superseded by `mem_fn()`
- `bind1st()/bind2nd()`
  - Strictly superseded by `bind()`

Recommendations

- Never use the old `<functional>` stuff
- Remove any existing usage
- C++98/03 algorithms/containers never needed `unary_function/binary_function/ptr_fun()`
- VS 2015: `/D_HAS_AUTO_PTR_ETC=0`
  - Also controls `auto_ptr` and `random_shuffle()`

std::function (C++11)

C++14: SFINAE

C++17: Converts non-void to void

Example

```
int sum_squares(int x, int y) {
  return x * x + y * y; }
vector<function<int (int, int)>> v;
v.emplace_back(plus<>());
v.emplace_back(multiplies<>());
v.emplace_back(&sum_squares);
for (int i = 10; i <= 1000; i *= 10) {
  v.emplace_back([i](int x, int y) {
    return i * x + y; }); }
for (const auto& f : v) { cout << f(4, 5) << endl; }
```

How `std::function` Works

- `function<Ret (Args)>` is a wrapper
  - Stores a callable object of arbitrary type
  - Templated on call signature, not callable object type
  - Type erasure, powered by virtual functions (or equivalent)
- Useful when code can't be templated
  - Separately compiled code
  - Virtual functions
  - Container elements

CopyConstructible Required

- The STL usually delays requirements
  - `list<T>` doesn't require `T` to have `operator<()`
  - `list<T>::sort()` requires `T` to have `operator<()`
- `std::function` is special, due to type erasure
  - `function(F)` requires `F` to be CopyConstructible
  - Even though it stores `move(f)`
  - Even if you never copy `std::function`
- Can't store movable-only function objects
  - Design limitation; alternatives are being investigated

Small Functor Optimization

- Arbitrary callable objects can be arbitrarily large
  - Eventually, dynamic memory allocation is necessary
- Small callable objects can be stored locally
  - Guaranteed for function pointers and `reference_wrapper`
  - Otherwise, "small" is the implementer's decision
- `is_nothrow_move_constructible` needed for SFO
  - Because `function::swap()` must be `noexcept`

Magic Numbers

Toolset	VS 2015 x86	VS 2015 x64	libc++ 3.7.0 x64	libstdc++ 5.2.0 x64
std::function	40	64	48	32
SFO Max	32	48	24	16
std::string	24-28	32-40	24	32

Ambiguous C++11, Valid C++14

```
void meow(const function<int (int)>& f) {
  cout << f(3) << endl; }
void meow(const function<int (int, int)>& g) {
  cout << g(4, 5) << endl; }
meow([](int n) { return n * 11; });
meow([](int x, int y) { return x * 10 + y; });
```

- `function<Ret (Args)>(F)` is now constrained
  - `F` must be Callable as `Ret (Args)`

Invalid C++14, Valid C++17

```
int x = 1729;
function<void (int&)> f(
  [](int& r) { return ++r; });
f(x);
cout << x << endl;
```

- Non-`void` can't be implicitly converted to `void`
  - According to Core - but the Library makes its own rules!
- Also applies to `packaged_task`, obscure `bind<R>()`

Unresolved Library Issues

- `std::function`'s `operator()` is `const`
  - Can store function objects with non-`const` `operator()`
  - Violates the STL's `const`/multithreading conventions
- This compiles, but really shouldn't:

```
string meow(int x) { return to_string(x); }
function<const string& (int)> hiss(&meow);
```

  - VS warns: returning address of local variable or temporary
  - I think I can fix this in the Standard, but it's not trivial

Recommendations

- `std::function` is awesome!
- But use it **only when necessary**
  - When possible, use templates and `auto`
- `std::function` inherently has nonzero costs
  - Time: Type erasure prevents inlining
  - Space: SFO buffer and type erasure consume bytes
  - Codegen: Emits code whether you use it or not
- Avoid unnecessary copies, moves, temporaries

More Info

More Info

- Almost everything here is available in VS 2015
  - Not yet implemented: C++14 `result_of`/`function` SFINAE
  - Bugs: `async()` uses `bind()`, `packaged_task` uses `function`
  - My `mem_fn()` was sneaky in RTM, fixed in Update 1
- C++17 Working Paper
  - `http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/n4527.pdf`

Questions?

stl@microsoft.com