# Don't Help The Compiler

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

# Are You Smarter Than A Compiler?

- Programmers have high-level knowledge
- Compilers have low-level information
  - Lifetimes: When to construct and destroy objects
  - Value Categories: Whether expressions refer to objects...
    - ... that the rest of the program can access (lvalues)
    - ... that the rest of the program can't access (rvalues)
  - Types: What operations can be performed on objects
    - `vector` has `operator[]()`, `list` has `splice()`
    - `a += b` performs addition for `int`
    - `a += b` performs concatenation for `string`
    - `a += b` performs atomic addition for `atomic<int>`

# Technology Marches On

- Increasingly powerful ways to work with the compiler's info
- Lifetimes:
  - C++98: `auto_ptr`, `vector`
  - TR1 '05: `shared_ptr`
  - C++11: `make_shared`, `unique_ptr`
  - C++14: `make_unique`
- Value Categories:
  - C++11: move semantics, perfect forwarding, ref-qualifiers
- Types:
  - C++98: overloads, templates
  - TR1 '05: `function`, type traits
  - C++11: `auto`, `decltype`, lambdas, `nullptr`, variadic templates
  - C++14: generic lambdas, return type deduction

# Lifetimes

# string Concatenation

- What's wrong with this code?

```
string str("meow"); const char * ptr = "purr";
foo(string(str) + string(", ") + string(ptr));
bar(string(ptr) + string(", ") + string(str));
```

# string Concatenation

- What's wrong with this code?

```
string str("meow"); const char * ptr = "purr";
foo(string(str) + string(", ") + string(ptr));
bar(string(ptr) + string(", ") + string(str));
```

- Fix:

```
foo(str + ", " + ptr);
bar(string(ptr) + ", " + str); // C++98/03/11
bar(ptr + ", "s + str);        // C++14
```

# vector<string> Insertion

- What's wrong with this code?

```
vector<string> v; string str("meow");
v.push_back(string(str));
v.push_back(string(str + "purr"));
v.push_back(string("kitty"));
v.push_back(string());
```

# vector<string> Insertion

- What's wrong with this code?

```
vector<string> v; string str("meow");
v.push_back(string(str));
v.push_back(string(str + "purr"));
v.push_back(string("kitty"));
v.push_back(string());
```

- Fix:

```
v.push_back(str);             // push_back(const T&)
v.push_back(str + "purr"); // push_back(T&&)
v.emplace_back("kitty");    // emplace_back(Args&&...)
v.emplace_back();            // emplace_back(Args&&...)
```

# Recommendations

- Use explicit temporaries only when necessary
  - Ask: Can the compiler use my original types?
  - Ask: Can the compiler implicitly construct temporaries?
  - OK: `return hash<string>()(m_str);`
- For `vector<T>`, etc.:
  - Call `push_back()` for `T` lvalues/rvalues and braced-init-lists
  - Call `emplace_back()` for other types, 0 args, and 2+ args
    - Emplacement can invoke `explicit` constructors

# PNG Image Decompression

- What's wrong with this code?

```
png_image img;
memset(&img, 0, sizeof(img));
img.version = PNG_IMAGE_VERSION;
png_image_begin_read_from_memory(&img, input_data, input_size);
if (PNG_IMAGE_FAILED(img)) { throw runtime_error(img.message); }
img.format = PNG_FORMAT_RGBA;
vector<uint8_t> rgba(PNG_IMAGE_SIZE(img));
png_image_finish_read(&img, nullptr, rgba.data(), 0, nullptr);
if (PNG_IMAGE_FAILED(img)) { throw runtime_error(img.message); }
upload_texture(img.width, img.height, rgba);
png_image_free(&img);
```

# PNG Image Decompression

- What's wrong with this code?

```
png_image img;
memset(&img, 0, sizeof(img));
img.version = PNG_IMAGE_VERSION;
png_image_begin_read_from_memory(&img, input_data, input_size);
if (PNG_IMAGE_FAILED(img)) { throw runtime_error(img.message); }
img.format = PNG_FORMAT_RGBA;
vector<uint8_t> rgba(PNG_IMAGE_SIZE(img));
png_image_finish_read(&img, nullptr, rgba.data(), 0, nullptr);
if (PNG_IMAGE_FAILED(img)) { throw runtime_error(img.message); }
upload_texture(img.width, img.height, rgba);
png_image_free(&img);
```

# PNG Image Decompression

- Fix:

```
png_image img;
memset(&img, 0, sizeof(img));
img.version = PNG_IMAGE_VERSION;
BOOST_SCOPE_EXIT_ALL(&) { png_image_free(&img); };
png_image_begin_read_from_memory(&img, input_data, input_size);
if (PNG_IMAGE_FAILED(img)) { throw runtime_error(img.message); }
img.format = PNG_FORMAT_RGBA;
vector<uint8_t> rgba(PNG_IMAGE_SIZE(img));
png_image_finish_read(&img, nullptr, rgba.data(), 0, nullptr);
if (PNG_IMAGE_FAILED(img)) { throw runtime_error(img.message); }
upload_texture(img.width, img.height, rgba);
```

# PNG Image Decompression

- Fix:

```
png_image img;
memset(&img, 0, sizeof(img));
img.version = PNG_IMAGE_VERSION;
BOOST_SCOPE_EXIT_ALL(&) { png_image_free(&img); };
png_image_begin_read_from_memory(&img, input_data, input_size);
if (PNG_IMAGE_FAILED(img)) { throw runtime_error(img.message); }
img.format = PNG_FORMAT_RGBA;
vector<uint8_t> rgba(PNG_IMAGE_SIZE(img));
png_image_finish_read(&img, nullptr, rgba.data(), 0, nullptr);
if (PNG_IMAGE_FAILED(img)) { throw runtime_error(img.message); }
upload_texture(img.width, img.height, rgba);
```

Failure → Destruction

Success → Destruction

# PNG Image Decompression

- Encapsulation (&img ➔ this):

```
png_image img;
memset(&img, 0, sizeof(img));
img.version = PNG_IMAGE_VERSION;
BOOST_SCOPE_EXIT_ALL(&) { png_image_free(&img); };
png_image_begin_read_from_memory(&img, input_data, input_size);
if (PNG_IMAGE_FAILED(img)) { throw runtime_error(img.message); }
img.format = PNG_FORMAT_RGBA;
vector<uint8_t> rgba(PNG_IMAGE_SIZE(img));
png_image_finish_read(&img, nullptr, rgba.data(), 0, nullptr);
if (PNG_IMAGE_FAILED(img)) { throw runtime_error(img.message); }
upload_texture(img.width, img.height, rgba);
```

Constructor

Member Function

Destructor

Member Functions

Member Function

# shared_ptr Construction

- What's wrong with this code?

```
void f(const shared_ptr<X>& sp,
       const vector<int>& v);
f(shared_ptr<X>(new X(args)), { 11, 22, 33 });
f(new X(args), { 11, 22, 33 }); // ILL-FORMED
```

# shared_ptr Construction

- What's wrong with this code?

```
void f(const shared_ptr<X>& sp,
       const vector<int>& v);
f(shared_ptr<X>(new X(args)), { 11, 22, 33 });
f(new X(args), { 11, 22, 33 }); // ILL-FORMED
```

- Fix:

```
f(make_shared<X>(args), { 11, 22, 33 });
```

# Multiple Resources

- What's wrong with this code?

```
class TwoResources {                // Lots of stuff omitted
    TwoResources(int x, int y) // to save space
        : m_a(nullptr), m_b(nullptr) {
        m_a = new A(x); m_b = new B(y);
    }
    ~TwoResources() {
        delete m_b; delete m_a;
    }
    A * m_a; B * m_b;
};
```

# Multiple Resources

- What's wrong with this code?

```
class TwoResources {            // Lots of stuff omitted
    TwoResources(int x, int y) // to save space
        : m_a(nullptr), m_b(nullptr) {
        m_a = new A(x); m_b = new B(y);
    }
    ~TwoResources() {
        delete m_b; delete m_a;
    }
    A * m_a; B * m_b;
};
```

# Multiple Resources

- Best fix:

```
class TwoResources { // Lots of stuff omitted
    TwoResources(int x, int y)
        : m_a(make_unique<A>(x)),
          m_b(make_unique<B>(y)) { }
    unique_ptr<A> m_a; unique_ptr<B> m_b;
};
```

- Also OK: `m_a = make_unique<A>(x);`
- Also OK: `shared_ptr/make_shared`

# Multiple Resources

- Possible fix:

```
class TwoResources { // Lots of stuff omitted
    TwoResources() : m_a(nullptr), m_b(nullptr) { }
    TwoResources(int x, int y) : TwoResources() {
        m_a = new A(x); m_b = new B(y);
    }
    ~TwoResources() { delete m_b; delete m_a; }
    A * m_a; B * m_b;
};
```

# Multiple Resources

- C++11 15.2 [except.ctor]/2: "if the non-delegating constructor for an object has completed execution and a delegating constructor for that object exits with an exception, the object's destructor will be invoked."
- Just because you **can** take advantage of this rule doesn't mean that you **should**!

# Recommendations

- `new` and `delete` are radioactive
  - Use `make_shared` and `make_unique`
- `new[]` and `delete[]` are radioactive
  - Use `vector` and `string`
- Manual resource management is radioactive
  - At the very least, use Boost.ScopeExit/etc.
  - Ideally, write classes for automatic resource management
- Each automatic resource manager...
  - ... should acquire/release exactly one resource, or
  - ... should acquire/release multiple resources **very** carefully

# Value Categories

# Returning By `const` Value

- Unintentionally modifying state is bad!
- `const` is good!
- Should we say `const` whenever possible?
- What's wrong with this code?

```
// Intentionally forbid meow().append("purr")
const string meow() { return "meow"; }
```

- Fix:

```
string meow() { return "meow"; }
```

# Returning Locals By Value (1/2)

- What's wrong with this code?

```
string meow() {
    string ret;
    stuff;
    return move(ret);
}
```

RVO: Return Value Optimization

NRVO: Named Return Value Optimization

- Fix:

```
return ret;
```

# Returning Locals By Value (2/2)

- What's wrong with this code?

```
tuple<string, string> meow() {
    pair<string, string> p;
    stuff;
    return move(p);
}
```

- Nothing!
  - As long as there's a reason for the types to be different

# Returning By Rvalue Reference (1/2)

- What's wrong with this code?

```
string&& meow() { string s; stuff; return s; }
```

- What's wrong with this code?

```
string& meow() { string s; stuff; return s; }
```

- C++11 8.3.2 [dcl.ref]/2: "Lvalue references and rvalue references are distinct types. Except where explicitly noted, they are semantically equivalent and commonly referred to as references."
- Fix:

```
string meow() { string s; stuff; return s; }
```

# Returning By Rvalue Reference (2/2)

- What's wrong with this code?

```
string&& join(string&& rv, const char * ptr) {
    return move(rv.append(", ").append(ptr)); }
string meow() { return "meow"; }
const string& r = join(meow(), "purr");
// r refers to a destroyed temporary!
```

- Fix:

```
  string join(string&& rv, const char * ptr) {
    return move(rv.append(", ").append(ptr)); }
```

# Recommendations

- Don't return by `const` value
  - Inhibits move semantics, doesn't achieve anything useful
- Don't `move()` when returning local `X` by value `X`
  - The NRVO and move semantics are designed to work together
  - NRVO applicable ➔ direct construction is optimal
  - NRVO inapplicable ➔ move semantics is efficient
- Don't return by rvalue reference
  - For experts only, extremely rare
  - Even the Standardization Committee got burned
  - Valid examples: `forward`, `move`, `declval`, `get(tuple&&)`

# Types

# Null Pointers… And Other Dangers

- Why doesn't this compile?

```
void meow(const pair<A, B *>& p);
struct C { explicit C(D * d); };
meow(make_pair(a, 0));
make_shared<C>(0);
```

# Null Pointers... And Other Dangers

- Why doesn't this compile?

```
void meow(const pair<A, B *>& p);
struct C { explicit C(D * d); };
meow(make_pair(a, 0)); // returns pair<A, int>
make_shared<C>(0); // calls make_shared(int&&)
```

- Literal `0` is a null pointer constant with a bogus type
- `NULL` is a macro for `0`, it's equally bad
- Fix:

```
meow(make_pair(a, nullptr)); // pair<A, nullptr_t>
make_shared<C>(nullptr); // make_shared(nullptr_t&&)
```

# Explicit Template Arguments (1/3)

- What's wrong with this code?

```
make_pair<A, B>(a, b)
```

- Declaration:

```
template <class T1, class T2> // C++03
    pair<T1, T2> make_pair(T1 x, T2 y);
template <class T1, class T2> // changed in C++11
    pair<V1, V2> make_pair(T1&& x, T2&& y);
```

- Fix:

`make_pair(a, b)` or sometimes `pair<X, Y>(a, b)`

# Explicit Template Arguments (2/3)

- What's wrong with this code?

```
transform(first1, last1, first2, result, max<int>);
```

- Declarations (without `Compare comp`):

```
template <class T>
    const T& max(const T& a, const T& b); // C++98
template <class T>
    T max(initializer_list<T> t); // added in C++11
```

- Fix:

`static_cast<const int& (*)(const int&, const int&)>(max)` or a lambda

# Explicit Template Arguments (3/3)

- What's wrong with this code?

```
generate<uint32_t *, mt19937&>(first, last, gen);
```

- Implementation:

```
template <class FwdIt, class Gen> void
    _Generate(FwdIt f, FwdIt l, Gen g) { stuff; }
template <class FwdIt, class Gen> void
    generate(FwdIt f, FwdIt l, Gen g) {
        _Generate(f, l, g); }
```

- Fix:

```
generate(first, last, ref(gen));
```

# Recommendations

- Always use `nullptr`, never use `0`/`NULL`
  - The type is what matters, not the spelling
  - Bonus: `int i = nullptr;` won't compile
- Rely on template argument deduction
  - You control its inputs - the function arguments
  - Change their types/value categories to affect the output
- Avoid explicit template arguments, unless required
  - Required: `forward<T>(t)`, `make_shared<T>(a, b, c)`
  - Wrong: `make_shared<T, A, B, C>(a, b, c)`

# C++98/03/11 Operator Functors

- STL algorithms (and containers) take functors
  - A functor is anything that's callable like a function
    - `func(args)`
  - OK: function pointers, function objects, lambdas
  - Not OK: pointers to member functions
    - `(obj.*pmf)(args)`, `(ptr->*pmf)(args)`
  - Not OK: operators
    - `x < y`

# C++98/03/11 Operator Functors

- STL algorithms (and containers) take functors
  - A functor is anything that's callable like a function
    - `func(args)`
  - OK: function pointers, function objects, lambdas
  - Not OK: pointers to member functions
    - `(obj.*pmf)(args)`, `(ptr->*pmf)(args)`
  - Not OK: operators
    - `x < y`
- `less<T>` adapts operator syntax to functor syntax

# Lambdas Aren't Always Better

- What's more verbose?

```
sort(v.begin(), v.end(), [](const Elem& l,
    const Elem& r) { return l > r; });
sort(v.begin(), v.end(), greater<Elem>());
```

- Also, consider:

```
map<Key, Value, greater<Key>>
```

- Can you even use a lambda here?
    - Yes, but it's **much** more verbose
    - And it doesn't compile in VS 2012/2013 due to an STL bug

# What Could Possibly Go Wrong?

- Today, someone writes:

```
vector<uint32_t> v = stuff;
sort(v.begin(), v.end(), greater<uint32_t>());
```

- Tomorrow, someone else changes this to:

```
vector<uint64_t> v = stuff;
sort(v.begin(), v.end(), greater<uint32_t>());
```

- Truncation!
  - Some compilers will warn about `uint64_t` ➔ `uint32_t` here
  - Equally dangerous, warnings rare: `int32_t` ➔ `uint32_t`

# More Problems

- Implementation:

```
template <class T> struct plus {
    T operator()(const T& x, const T& y) const {
        return x + y; } };
```

# More Problems

- Implementation:

```
template <class T> struct multiplies {
    T operator()(const T& x, const T& y) const {
        return x * y; } };
```

- Units library: `Watts * Seconds` returns `Joules`

# More Problems

- Implementation:

```
template <class T> struct greater {
 bool operator()(const T& x, const T& y) const {
        return x > y; } };
```

- Units library: `Watts * Seconds` returns `Joules`
- `greater<string>` called with `const char *`, `string`
    - Unnecessary temporary

# More Problems

- Implementation:

```
template <class T> struct plus {
    T operator()(const T& x, const T& y) const {
        return x + y; } };
```

- Units library: `Watts * Seconds` returns `Joules`
- `greater<string>` called with `const char *`, `string`
  - Unnecessary temporary
- `plus<string>` called with `string` lvalue, `string` rvalue
  - Unnecessary copy

# Transparent Operator Functors

```
template <class T = void> struct plus { /* same */ };

template <> struct plus<void> {
    template <class T, class U>
        auto operator()(T&& t, U&& u) const
        -> decltype(forward<T>(t) + forward<U>(u)) {
        return forward<T>(t) + forward<U>(u);
    }
};
```

# Diamonds Are Forever

```
// Voted into C++14, implemented in VS 2013
vector<const char *> v{ "cute", "fluffy", "kittens" };
set<string, greater<>> s{ "HUNGRY", "EVIL", "ZOMBIES" };
vector<string> dest;
transform(v.begin(), v.end(), s.begin(),
    back_inserter(dest), plus<>());
// cuteZOMBIES
// fluffyHUNGRY
// kittensEVIL
```

# Do You See What I See?

```
template <class T = void> struct plus {
    T operator()(const T& x, const T& y) const;
};


template <> struct plus<void> {
    template <class T, class U>
        auto operator()(T&& t, U&& u) const
        -> decltype(forward<T>(t) + forward<U>(u));
};
```

# Do You See What I See?

```
template <class T = void> struct plus {
    T operator()(const T& x, const T& y) const;
};
```

User provides type too early, before the call happens

Compiler deduces types at the right time, when the call happens

```
template <> struct plus<void> {
    template <class T, class U>
        auto operator()(T&& t, U&& u) const
        -> decltype(forward<T>(t) + forward<U>(u));
};
```

# Conclusions

# When To Help The Compiler

- Casts (avoid when possible, never use C casts)
  - Are you certain that briefly ignoring the type system is safe?
- Moves (use carefully)
  - Are you certain that stealing from lvalues is safe?
  - In general, "moved-from" == "unspecified state"
    - Special: moved-from `shared_ptrs`/`unique_ptrs` are empty
  - `move()` is friendly syntax for a value category **cast**
- `enable_if`/SFINAE (prefer tag dispatch, `static_assert`)
  - Are you certain that you're smarter than overload resolution?
- Common theme: Do you have high-level knowledge?

# Let The Compiler Help You

- C++'s rules are building blocks of systems for...
  - ... automatically acquiring/releasing resources
    - Nobody else has solved this problem
    - Nobody else **understands** this problem
    - Garbage collection is neither necessary nor sufficient
  - ... identifying resources that can be stolen (versus copied)
  - ... selecting run-time actions at compile-time (with inlining!)
    - Types flow from function calls to overloads/templates
- Use these systems, don't interfere with them
  - Compilers cannot feel pain, but you can!
  - Compilers cannot enjoy programming, but you can!

# Bonus Slides

# C++11 20.3.3 [pairs.spec]/8

```
template <class T1, class T2>
    pair<V1, V2> make_pair(T1&& x, T2&& y);
```

*Returns:* `pair<V1, V2>(std::forward<T1>(x), std::forward<T2>(y));` where `V1` and `V2` are determined as follows: Let `Ui` be `decay<Ti>::type` for each `Ti`. Then each `Vi` is `X&` if `Ui` equals `reference_wrapper<X>`, otherwise `Vi` is `Ui`.

- My example:

```
int n = 1729; make_pair(n, "Hardy-Ramanujan")
//     returns pair<int, const char *>
// instead of pair<int&, const char (&)[16]>
```

# static_cast vs. Lambdas

```
// Verbose:
static_cast<const int& (*)(const int&, const int&)>(max)

// Shorter, but different (takes/returns by value):
[](int l, int r) { return max(l, r); }

// Equivalent, but more verbose:
[](const int& l, const int& r) -> const int& { return max(l, r); }

// Upper bound on verbosity for long type names:
[](const auto& l, const auto& r) -> const auto& { return max(l, r); }
// (This permits L and R to be different, but max() will require L == R)
```

# Transparent Operator Functors

```
template <class T = void> struct plus { /* same */ };

template <> struct plus<void> {
    template <class T, class U>
        auto operator()(T&& t, U&& u) const
        -> decltype(forward<T>(t) + forward<U>(u)) {
        return forward<T>(t) + forward<U>(u);
    }
};
```

# Transparent Operator Functors

```
template <class T = void> struct plus { /* same */ };

template <> struct plus<void> {
    template <class T, class U>
        decltype(auto) operator()(T&& t, U&& u) const {
        return forward<T>(t) + forward<U>(u);
    }
};
```

- C++14 `decltype(auto)` avoids code duplication

RVO: Return Value Optimization

NRVO: Named Return Value Optimization

# Standardese

- C++11 12.8 [class.copy]/31: "in a `return` statement in a function with a class return type, when the expression is the name of a non-volatile automatic object [...] with the same cv-unqualified type as the function return type, the copy/move operation can be omitted by constructing the automatic object directly into the function's return value"
- /32: "When the criteria for elision of a copy operation are met [...] and the object to be copied is designated by an lvalue, overload resolution to select the constructor for the copy is first performed as if the object were designated by an rvalue."

# Categorizing Types

- Copyable and movable
  - Like `vector`, expensive copies and cheap moves
  - Like `string`, with the Small String Optimization
  - Like `array<int, 256>`, copies and moves just copy bits
- Copyable, but non-movable
  - "Non-movable" - Like C++98, expensive copies and moves
  - Really non-movable - Nobody does this
- Non-copyable, but movable
  - Like `unique_ptr`, cheap moves
- Non-copyable and non-movable
  - Like `atomic` and `lock_guard`, useful but not of interest here

# Parameter Types

- Observers should take `const X&`
  - Binds to everything
- Mutators should take `X&`
  - Binds to modifiable lvalues only
- Consumers should choose between:
  - Taking `const X&` - always copies like C++98/03
  - Moving `X` - copy+move or 2 moves - can be better **or worse**
  - Overloading `const X&` and `X&&` - copy or move, usually optimal
  - Perfectly forwarding `T&&` or `Args&&...` - always optimal
- Non-generic non-mutators should take scalars by value
  - For increased safety, `const` value (definitions only)

# const X&& Is Not Entirely Useless

- const X&& isn't useful for move semantics
  - Can't steal resources from const objects
- const T&& isn't useful for perfect forwarding
  - Only T&& triggers the template argument deduction tweak
- But const X&& can reject temporaries!

```
// Each declaration starts with: template <class T>
        reference_wrapper<T> ref(T&) noexcept;
                       void ref(const T&&) = delete;
reference_wrapper<const T> cref(const T&) noexcept;
                      void cref(const T&&) = delete;
```

- regex_match(s, m, r) should reject temporary strings

# Recommendations

- Learn rvalue references, then follow the patterns
  - Overload `const X&` and `X&&` for move semantics
    - Named rvalue references are lvalues for a reason
  - Perfectly forward `T&&` (1 arg) or `Args&&...` (0+ args)
    - `forward<T>(t)` might be an rvalue, so don't say it twice!
    - Don't overload perfect forwarders - they're intentionally greedy
  - Use `const X&&` to reject temporaries
- Write code only when you know how it'll behave
  - Always true, but especially true for rvalue references
  - If you don't know, ask an expert
  - Example: inspecting `T` in a perfect forwarder