PERFECT FORWARDING

Plan for Today

- Refresher on rvalue references
- Refresher on std::move
- Template argument deduction
- What is perfect forwarding and how does it work
- Factories
- How does auto work
- How does decltype work

Lvalues and Rvalues

- *Every expression is either an lvalue or an rvalue*
- Lvalue expressions name objects that persist beyond single expression
  - For example, a variable expression
- Rvalue expressions are temporaries that evaporate at end of full expression in which they live – at semicolon
  - Either literals or temporary objects created in the course of evaluating expressions

Lvalues and Rvalues


```
int x = 10, *px = &x;    // x and px are lvalues
int foo(std::string s); // foo and s are lvalues
std::string s{"hello"};
x = foo(s);    // ok, foo()'s return value is rvalue
px = &foo();  // error: foo()'s return value is rvalue
foo("hello"); // temp string created for call is rvalue

std::vector<int> vi(10); // vi is lvalue
vi[5] = 11; // vector<T>::op[] returns T&

int& foobar();
foobar() = 11; // ok, foobar()'s return value is lvalue
*px = &foobar(); // ditto
```

Lvalue References

- In type name, notation `X&` means “reference to `X`”
  - Binds to modifiable lvalues
  - Can’t bind to `const` lvalues – violates `const` correctness
  - Can’t bind to modifiable rvalues – modifying temporaries that evaporate along with modifications can lead to bugs
  - Can’t bind to `const` rvalues - for above reasons
- Main purpose: to refer to objects that we wish to change

`const` Lvalue References

- In type name, notation `const X&` means "non-modifiable (`const`) reference to `X`"
  - Binds to modifiable lvalues
  - Binds to `const` lvalues
  - Binds to modifiable rvalues
  - Binds to `const` rvalues
- Main purpose of `const` references: to refer to objects whose values we don't want to change (or shouldn't change)

const Lvalue References


```
int& ri{1}; // error: lvalue needed
int const& rci{1};  // binds to rvalue
int x{10};  // lvalue
int const& rcx1{x}; // binds to lvalue
int const x{11};
int const& rcx2{y}; // binds to const lvalue
int foo();
int const& rcx3{foo()}; // binds to rvalue
int const boo();
int const& rcx4{boo()}; // binds to const rvalue
```

Do You Know Your Lvalues, Rvalues, And References?


```
void mutate(string& s) { ... }
void observe(string const& s) { ... }

string one("one");
string const two("two");

string three() { return "three"; }
string const four() { return "four"; }
```

```
// classify calls as valid or non-valid
observe(one); observe(two);
observe(three()); observe(four());

mutate(one); mutate(two);
mutate(three()); mutate(four());
```

Rvalue References

- If X is any type, then X&& is called rvalue reference to X
- Behavior exactly opposite of lvalue reference:
  - Can only bind to an rvalue (a temporary object), but not to an lvalue

Rvalue References


```
string var{"Iowa"};
string f();

string& r1{var};     // bind r1 to lvalue var
string& r2{f()};     // error: f() is rvalue
string& r3{"Ohio"}; // error: cannot bind to temporary

string&& rr1{f()};  // bind rr1 to rvalue (a temporary)
string&& rr2{var};  // error: var is lvalue
string&& rr3{"Iowa"}; // rr3 refers to temporary
                      // encapsulating resource "Iowa"
string const& cr{"Iowa"}; // ok: make temporary
                          // and bind to cr
```

Why Three Kinds of References

- Basic idea of having more than one kind of reference is to support different uses of objects:
  - Non-`const` lvalue references: to refer to objects whose value we want to change
  - `const` lvalue references: to refers to objects whose value we don't want to change
  - Rvalue references: to refer to objects whose value we don't need to preserve after usage

Rvalue References: Main Purpose

- Invented to allow a function to branch at compile time via overload resolution on condition “Am I being called on an lvalue or an rvalue?”
  - Function with rvalue reference parameter can (and typically will) modify object assuming it will not be used again
  - This is what is known as *move semantics*

Rvalue References: Main Purpose


```
void f(vector<int>&);             // 1
void f(vector<int> const&); // 2
void f(vector<int>&&);           // 3

void g(vector<int>& vi,
      vector<int> const& cvi) {
  f(vi);                        // call ???
  f(cvi);                       // call ???
  f(vector<int>{1,2,3}); // call ???
}
```

Understanding `std::move`

- Compiler triggers move semantics when function's parameter is rvalue reference

Understanding `std::move`

- Compiler triggers move semantics when function's parameter is rvalue reference
- Sometimes, you know object won't be used again, even though compiler does not

```
template<typename T>
void swap(T& a, T& b) { // old-style swap
  T tmp{a}; // 2 copies of a
  a = b;    // 2 copies of b
  b = tmp;  // now, we've 2 copies of tmp
}
```

Understanding `std::move`

- You can use move semantics at your discretion on lvalues

```
template <typename T>
void swap(T& lhs, T& rhs) {
  T tmp{static_cast<T&&>(lhs)};
  lhs = static_cast<T&&>(rhs);
  rhs = static_cast<T&&>(lhs);
}
```

Result is rvalue of type T&& for `lhs`

If type T has move assignment, it will be used

Understanding std::move

- We're rescued from verbosity by standard library function std::move

```
template<class T>
void swap(T& a, T& b) {
  T tmp{std::move(a)}; // steal from a
  a = std::move(b);    // steal from b
  b = std::move(tmp);  // steal from tmp
}
```

Understanding `std::move`

- `std::move(x)` passes its argument right thro' by reference, doing nothing with it all
- Expression `std::move(x)` is declared as rvalue reference and doesn't have a name and hence it's an rvalue
- Thus, `std::move()` "*turns its argument into an rvalue even if it isn't*" by "*hiding the name*"

Understanding `std::move`

- `move(x)` doesn't move anything – it simply means "*give me an rvalue reference to* `x`"
- Compiler can then trigger move semantics on lvalues
- Only safe use of `x` after `move(x)` is destruction or as target for assignment

Is An Rvalue Reference An Rvalue?

- Assuming class X implements move semantics, which ctors are called in following example?

```
void foo(X&& x) {
  X moreX = x; // which ctor
  // …
}

X&& goo();
X x = goo(); // which ctor
```

Is An Rvalue Reference An Rvalue?

- Don't be surprised - use C++ standard as a guide:
  - *Every expression is either an lvalue or an rvalue*
  - *Lvalue expressions name objects that persist beyond expression*
  - *Rvalue expressions are temporaries that evaporate at end of full expression in which they live – at semicolon*
- Hence the rule:
  - "*If it has a name, then it's an lvalue*"
  - Otherwise, "*it is an rvalue*"

"If it has a name, then it's an lvalue"

- It is *very* important to remember this rule …

Don't Make Things Worse …

- Assuming class X implements move semantics, which return is more optimal?

```
X foo() {
  X x;
  // ...
  return x;
}
```

```
X foo() {
  X x;
  // ...
  return std::move(x);
}
```

Perfect Forwarding: The Problem

- We want to forward parameter `param` from `factory()` to `T`'s constructor
- Ideally, from `param`'s perspective, everything should behave just as if `factory()` wasn't there and ctor was called directly: *perfect forwarding*

```
// factory function
template <typename T, typename Param>
std::unique_ptr<T> factory(Param param) {
  return std::make_unique<T>(param);
}
```

Perfect Forwarding: The Problem

- This factory function doesn't solve the problem – it introduces extra call by value – especially bad if ctor takes its parameter by reference

```
// factory function
template <typename T, typename Param>
std::unique_ptr<T> factory(Param param) {
  return std::make_unique<T>(param);
}
```

Perfect Forwarding: The Problem

- Most common solution is to let outer function take parameter by reference
- Problem is factory function cannot be called on rvalues

```
// factory function
template <typename T, typename Param>
std::unique_ptr<T> factory(Param& param) {
  return std::make_unique<T>(param);
}
```

Perfect Forwarding: The Problem

- Problem is factory function cannot be called on rvalues
- This can be fixed by providing an overload that takes its parameter by `const` reference

```
// factory function
template <typename T, typename Param>
std::unique_ptr<T> factory(Param& param) {
  return std::make_unique<T>(param);
}

template <typename T, typename Param>
std::unique_ptr<T> factory(Param const& param) {
  return std::make_unique<T>(param);
}
```

Perfect Forwarding: The Problem

- This can be fixed by providing an overload that takes its parameter by `const` reference
- Two problems:
  - Scales poorly for functions with several parameters - overloads for all combinations of non-`const` and `const` references for various parameters are required
  - Not perfect forwarding because move semantics are blocked - parameter of copy ctor in function body is lvalue