PERFECT FORWARDING

Plan for Today

- Motivation for perfect forwarding
- Reference collapsing rules
- Template argument deduction for rvalue references
- Implementing perfect forwarding

Perfect Forwarding: The Problem (1/8)

- Generic factory function returning `std::unique_ptr` for newly constructed type

```
template <typename T> // no argument version
std::unique_ptr<T> factory() { return std::make_unique<T>(); }

template <typename T, typename Param> // one argument version
std::unique_ptr<T> factory(Param param) {
  return std::make_unique<T>(param); // pass-by-value ...
}

// two argument version
template <typename T, typename Param1, typename Param2>
std::unique_ptr<T> factory(Param1 param1, Param2 param2) {
  return std::make_unique<T>(param1, param2);
}

// all other versions
```

Perfect Forwarding: The Problem (2/8)

- We want to *forward* parameter `param` from `factory` to `T`'s ctor

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

Ideally, from `param`'s perspective, everything should behave just as if `factory` wasn't there and `T`'s ctor was called directly: perfect forwarding

Perfect Forwarding: The Problem (3/8)

- `factory` doesn't solve the problem!!!
  - Introduces extra call by value
  - Incorrect if ctor takes its parameter by reference

`factory` creates `param` via copy ctor and in turn passes `param` by value to `T`'s ctor

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

What is to be done if `T`'s ctor takes its parameter by reference? Things will go wrong because `T`'s ctor parameter would be a reference to `param` rather than a reference to caller's argument

Perfect Forwarding: The Problem (4/8)

- Possible solution is to let outer function `factory` take parameter by reference
- Problem is `factory` cannot be called on *rvalues*

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

Perfect Forwarding: The Problem (5/8)

- Problem is `factory` cannot be called on *rvalues*
- Fix by providing overload

```
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 (6/8)

- This can be fixed by providing an overload that takes its parameter by `const` reference
- Two problems

Perfect Forwarding: The Problem (7/8)

- Combinatorial complexity
  - Scales poorly for functions with several parameters - overloads for all combinations of non-`const` and `const` references for various parameters are required
  - If a class `X` has n data members [each of different class type with each type having 3 ctors], `X` will have total of 3^n ctors [to accommodate every single variation]
  - Therefore, to construct objects of type `X`, `factory` must have 3^n overloads

Perfect Forwarding: The Problem (8/8)

- Move semantics are suppressed
  - Function parameters that are *rvalue* references are themselves *lvalues*
  - Less than perfect because move semantics are blocked

```
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: How To Solve?

- Use *rvalue* references to solve both problems
- To understand how, need to look at two more rules for *rvalue* references

What Are Forwarding References? (1/9)

- To declare *rvalue* reference to some type T, you type T&&
- Wrong to assume T&& in source code means *rvalue* reference

What Are Forwarding References? (2/9)


```
void f(Widget&& param);             // rvalue reference

Widget&& var1 = Widget{};           // rvalue reference

auto&& var2 = var1;                 // not rvalue reference

template <typename T>
void f(std::vector<T>&& param);     // rvalue reference

template <typename T>
void f(T&& param);                  // not rvalue reference
```

What Are Forwarding References? (3/9)

- In fact, T&& has two meanings
  - As *rvalue* reference which binds only to *rvalues* to identify objects that may be moved from
  - Either *rvalue* reference or *lvalue* reference
    - Looks like *rvalue* reference in source code (T&&) but can behave like *lvalue* reference (T&)
    - Dual nature allows them to bind to *anything*
    - Known as *forwarding references* [or as *universal references*]

What Are Forwarding References? (4/9)

- Forwarding references arise in contexts involving *type deduction*
  - Function template parameters

```
// no type deduction; param is rvalue reference
void f(Widget&& param);

// param is forwarding reference
template <typename T> void f(std::vector<T>&& param);
```

  - auto declarations

```
// no type deduction; param is rvalue reference
Widget&& var1 = Widget{};

// param is forwarding reference
auto&& var2 = var1;
```

What Are Forwarding References? (5/9)

- For reference to be forwarding, type deduction is necessary, but it's not sufficient
- *Form* of reference declaration must also be correct, and that form is constrained to be T&&

What Are Forwarding References? (6/9)

When `f` *is invoked, type* `T` *will be deduced.*
But form of `param` *isn't* `T&&`*, it's* `std::vector<T>&&`*.*
That rules out possibility that `param` *is forwarding reference.*
`param` *is therefore an rvalue reference.*

```
template <typename T>
void f(std::vector<T>&& param);

std::vector<int> v;
f(v); // error!
```

When `f` *is passed an lvalue* `v`*, the compiler will complain since parameter* `param` *is an rvalue reference which cannot bind to an lvalue.*

What Are Forwarding References? (7/9)

Even presence of `const` *qualifier is enough to disqualify a reference from being forwarding reference. Remember the form of the reference declaration must be precisely* `T&&`.

```
// is param an rvalue or forwarding reference?
template <typename T>
void f(T const&& param);
```

What Are Forwarding References? (8/9)

Just because you're in a template and you see a member function parameter of type T&&, you can't assume that it's a forwarding reference.

That's because being in a member function template doesn't guarantee presence of type deduction.

push_back*'s parameter has right form for forwarding reference, but there's no type deduction in this case.*

```
// is param an rvalue or forwarding reference?
template <typename T, class Allocator = allocator<T>>
class vector {
public:
  void push_back(T&& param);
  // other stuff ...
};
```

What Are Forwarding References? (9/9)

That's because `push_back` *can't exist without particular* `vector` *instantiation for it to be part of, and type of that instantiation fully determines declaration for* `push_back`*. That is, saying*

```
std::vector<Widget> v;
```

causes `std::vector` *template to be instantiated as follows:*

```
// you can see that push_back employs no type deduction
// thus, this push_back for vector<T> always declares a
// parameter of type rvalue-reference-to-T
class vector<Widget, allocator<Widget>> {
public:
  void push_back(Widget&& param); // rvalue reference
  // other stuff ...
};
```

Forwarding References: Initializers (1/2)

- Initializer for forwarding reference determines whether it represents an *rvalue* reference or an *lvalue* reference

Forwarding References: Initializers (2/2)

If initializer is an lvalue reference, forwarding reference corresponds to an lvalue reference.
If initializer is an rvalue reference, forwarding reference corresponds to an rvalue reference.

```
// param is forwarding reference
template <typename T> void f(T&& param);

Widget w;

// lvalue passed to f; param's type is Widget&
f(w);

// rvalue passed to f; param's type is Widget&&
f(std::move(w));
```

Rule 1 for *Rvalue* References: Reference Collapsing

- C++98 did not allow taking a reference to a reference
- C++11 introduces following collapsing rules for references to type T:
  - T& & becomes T&
  - T& && becomes T&
  - T&& & becomes T&
  - T&& && becomes T&&

Rule 2 for Rvalue References: Template Argument Deduction

- When `f` is called with `expr` being an:
  - *lvalue* of type `A`, then `T` resolves to `A&`, and by reference collapsing rules, `param`'s type is `A&`
  - *rvalue* of type `A`, then `T` resolves to `A`, and hence `param`'s type is `A&&`

```
// function template declaration
template <typename T>
void f(T&& param);

// call f with some expression
f(expr);
```

Perfect Forwarding: The Solution

- Given the two rules for *rvalue* references perfect forwarding problem is solved like this:

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

std::forward<Param>

- Has following overloads:

```
// forwards lvalues as either lvalues or rvalues depending on U
template <typename U>
U&& forward(typename std::remove_reference<U>::type& u) noexcept {
  return static_cast<U&&>(u);
}
```

```
// forwards rvalues as rvalues and
// prohibits forwarding of rvalues as lvalues
// see cppreference for more details
template <typename U>
U&& forward(typename std::remove_reference<U>::type&& u) noexcept {
  return static_cast<U&&>(u);
}
```

Perfect Forwarding: The Solution


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

```
template <typename U>
U&& forward(typename std::remove_reference<U>::type& u) noexcept {
  return static_cast<U&&>(u);
}
```

```
template <typename U>
U&& forward(typename std::remove_reference<U>::type&& u) noexcept {
  return static_cast<U&&>(u);
}
```

Perfect Forwarding: The Perfect Solution

- Perfect forwarding problem for unknown number of parameters is solved like this:

```
template <typename T, typename... Params>
std::unique_ptr<T> factory(Params&&... params) {
  return std::make_unique<T>(std::forward<Params>(params)...);
}
```

```
template <typename T, typename... Params>
unique_ptr<T> make_unique(Params&&... params) {
  return unique_ptr<T>(new T(std::forward<Params>(params)...));
}
```

Perfect Forwarding: The Problem (6/6)

- See source files *forward?.cpp* to see progression of solutions to perfect forwarding problem