# PERFECT FORWARDING 2

# Plan for Today



- Template argument deduction
- What is perfect forwarding and how does it work
- Factories
- How does auto work
- ~~How does decltype work~~

# Perfect Forwarding: The Problem



- Source for this lecture: Modern Effective C++
- See course webpage for specific list of items

# 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

# Template Type Deduction



- Entire discussion is based on the excellent material presented here
- Consider function template and call to that function template …

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

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

# Template Type Deduction



- Template type deduction is process during compilation when compilers use `expr` to deduce types for `T` and `ParamType`
- Three cases to consider:
  - `ParamType` is pointer or reference type
  - `ParamType` is neither a pointer nor reference
  - `ParamType` is forwarding reference

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

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

# ParamType: Pointer/Reference (1/8)



- If expr's type is reference, ignore reference part and then pattern-match expr's type against ParamType to determine T

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

int x{27};          // x is an int
int const cx{x};    // cx is const int
int const& rx{x};   // rx is reference to
                    // x as a const int

// what are deduced types for param and T?
f(x);  // T: ???, param: ???
f(cx); // T: ???, param: ???
f(rx); // T: ???, param: ???
```

# ParamType: Pointer/Reference (2/8)



- If expr's type is reference, ignore reference part and then pattern-match expr's type against ParamType to determine T

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

int x{27};           // x is an int
int const cx{x};     // cx is const int
int const& rx{x};    // rx is reference to
                     // x as a const int

// what are deduced types for param and T?
f(x);  // T: int, param: int&
f(cx); // T: const int, param: const int&
f(rx); // T: const int, param: const int&
```

# ParamType: Pointer/Reference (3/8)



- ParamType's type now changes from T& to T const&
- If expr's type is reference, ignore reference and pattern-match expr's type against ParamType to determine T

```
template <typename T> void f(T const& param);

int x{27};          // x is an int
int const cx{x};    // cx is const int
int const& rx{x};   // rx is reference to
                    // x as a const int

// what are deduced types for param and T?
f(x);  // T: ???, param: ???
f(cx); // T: ???, param: ???
f(rx); // T: ???, param: ???
```

# ParamType: Pointer/Reference (4/8)



- ParamType's type now changes from T& to T const&
- If expr's type is reference, ignore reference and pattern-match expr's type against ParamType to determine T

```
template <typename T> void f(const T& param);

int x{27};          // x is an int
int const cx{x};    // cx is const int
int const& rx{x};   // rx is reference to
                    // x as a const int

// what are deduced types for param and T?
f(x);  // T: int, param: int const&
f(cx); // T: int, param: int const&
f(rx); // T: int, param: int const&
```

# ParamType: Pointer/Reference (5/8)



- ParamType's type is T*
- Ignore reference in expr and then pattern-match expr's type against ParamType to determine T

```
template <typename T> void f(T *param);

int x{27};            // x is an int
int const *px{&x}; // px is pointer to x
                      // as a const int

// what are deduced types for param and T?
f(&x);  // T: ???, param: ???
f(&px); // T: ???, param: ???
```

# ParamType: Pointer/Reference (6/8)



- ParamType's type is T*
- Ignore reference in expr and then pattern-match expr's type against ParamType to determine T

```
template <typename T> void f(T *param);

int x{27};             // x is an int
int const *px{&x}; // px is pointer to x
                       // as a const int

// what are deduced types for param and T?
f(&x);  // T: int, param: int*
f(&px); // T: int const, param: int const*
```

# ParamType: Pointer/Reference (7/8)



- ParamType's type is T const*
- Ignore reference in expr and then pattern-match expr's type against ParamType to determine T

```
template <typename T> void f(T const *param);

int x{27};          // x is an int
int const *px{&x};  // px is pointer to x
                    // as a const int

// what are deduced types for param and T?
f(&x);  // T: ???, param: ???
f(&px); // T: ???, param: ???
```

# ParamType: Pointer/Reference (8/8)



- ParamType's type is T const*
- Ignore reference in expr and then pattern-match expr's type against ParamType to determine T

```
template <typename T> void f(T const *param);

int x{27};          // x is an int
int const *px{&x}; // px is pointer to x
                    // as a const int

// what are deduced types for param and T?
f(&x);  // T: int, param: int const*
f(&px); // T: int, param: int const*
```

# ParamType: Neither Pointer Nor Reference (1/2)



- ParamType's type is T
- Fact that param is newly constructed object motivates rules governing how T is deduced from expr:
  - If expr's type is reference, ignore reference part
  - If expr is now const (or volatile), ignore that too

```
template <typename T> void f(T param);

int x{27};          // x is an int
int const cx{x};    // cx is const int
int const& rx{x};   // rx is reference to const int
int const * const rprx{&x};
// what are deduced types for param and T?
f(x);     // T: ???, param: ???
f(cx);    // T: ???, param: ???
f(rx);    // T: ???, param: ???
f(rprx);  // T: ???, param: ??
```

# ParamType: Neither Pointer Nor Reference (2/2)



- ParamType's type is T
- Fact that param is new object motivates rules governing how T is deduced from expr:
  - If expr's type is reference, ignore reference part
  - If expr is now const (or volatile), ignore that too

```
template <typename T> void f(T param);

int x{27};           // x is an int
int const cx{x};     // cx is const int
int const& rx{x};    // rx is reference to const int
int const * const rprx{&x};
// what are deduced types for param and T?
f(x);     // T: int, param: int
f(cx);    // T: int, param: int
f(rx);    // T: int, param: int
f(rprx);  // T: int const*, param: int const*
```

# ParamType: Forwarding Reference



- ParamType's type is T&&
- Situation is bit complicated because expr can be lvalue or rvalue expression!!!

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

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

# Forwarding References



- T&& means rvalue reference to some type T
- However, T&& has two different meanings
  - One meaning is rvalue reference
  - 2nd meaning is either lvalue reference or rvalue reference

```
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
```

# Forwarding References



- If you see T&& without type deduction, you're looking at rvalue references

```
void f(Widget&& param);   // rvalue reference
                          // no type deduction
Widget&& var1 = Widget(); // rvalue reference
                          // no type deduction
auto&& var2 = var1;       // not rvalue reference

template <typename T>
void f(std::vector<T>&& param); // rvalue reference
                                // no type deduction
template <typename T>
void f(T&& param);  // not rvalue reference
```

# Forwarding References



- Universal references arise in context of function template parameters
- In both cases, type deduction is taking place

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

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

# Forwarding References



- Because universal references are references, they must be initialized
- Initializer determines whether lvalue or rvalue reference

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

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

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

# Forwarding References



- In addition to type deduction, form of reference declaration must be precisely T&& for a reference to be universal

```
template <typename T>
void f(std::vector<T>&& param); // rvalue reference
                                // form of param is not T&&
template <typename T>
void f(T const&& param); // rvalue reference

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

# Forwarding References



- Being in a template doesn't guarantee type deduction

```
template <typename T,
          class Allocator = std::allocator<T>>
class vector {
public:
  void push_back(T&& x);
  ...
}
```

# Forwarding References



- Here, type parameter Params is independent of vector's type so each tie

```
template<typename T,
         class Allocator = std::allocator<T>>
class vector {
public:
  template <class... Params>
  void emplace_back(Params&&... params);
  ...
};
```

# Reference Collapsing Rules



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

# ParamType: Forwarding Reference



- ParamType's type is T&&
- 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&&
- ParamType is called *forwarding reference*

# ParamType: Forwarding Reference



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

int x{27};          // x is int
int const cx{x};    // cx is const int
int const& rx{x};  // rx is reference to int const

f(x);  // x:  ??? => T: ???, param: ???
f(cx); // cx: ??? => T: ???, param: ???
f(rx); // rx: ??? => T: ???, param: ???
f(27); // 27: ??? => T: ???, param: ???
```

# ParamType: Forwarding Reference



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

int x{27};          // x is int
int const cx{x};    // cx is const int
int const& rx{x}; // rx is reference to int const

f(x);  // x:  lvalue => T: int&, param: int&
f(cx); // cx: lvalue => T: int const&, param: int const&
f(rx); // rx: lvalue => T: int const&, param: int const&
f(27); // 27: rvalue => T: int, param: int&&
```

# Type Deduction: Array Arguments (1/3)

- Array types are different from pointer types – even though they seem interchangeable
- Array *decays* into pointer to its first element:

```
char const name[] = "Clint";

// array decays to pointer
char const *ptr{name};
```

# Type Deduction: Array Arguments (2/3)

- What happens if array is passed to template taking by-value parameter?

```
template <typename T>
void f(T param); // param is passed by value

char const name[] = "Clint";

// what type deduced for T and param?
f(name);
```

# Type Deduction: Array Arguments (3/3)

- Although functions can't declare parameters that are arrays, they can declare parameters that are references to arrays!

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

char const name[] = "Clint";

// what type deduced for T and param?
f(name);
```

# Deducing Array Size

- Ability to declare references to arrays enables creation of a template that deduces number of elements that an array contains:

```
// return array size as compile-time constant
template <typename T, std::size_t N>
constexpr std::size_t array_size(T (&)[N]) noexcept {
  return N;
}

int keys[] {1,3,5,7,9};

// vals has size 7
std::array<int, array_size(keys)> vals;
```

# Type Deduction: Function Arguments

- Just like arrays, functions also decay into function pointers
- Type deduction is similar to arrays

```
void func(int, double);

template <typename T> void f1(T param);

template <typename T> void f2(T& param);

// what is type of T and param?
f1(func);
// what is type of T and param?
f2(func);
```

# auto Type Deduction (1)

- auto type deduction *is* template type deduction
- There is direct algorithmic transformation from template type deduction to auto type deduction

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

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

```
type specifier with auto identifier = initializer;
```

# `auto` Type Deduction (2)



- Three cases to consider:
  - *`type specifier`* is pointer or reference type
  - *`type specifier`* is neither a pointer nor reference
  - *`type specifier`* is forwarding reference

# auto Type Deduction (3)



- Three cases to consider:
  - *type specifier* is pointer or reference type
  - *type specifier* is neither a pointer nor reference
  - *type specifier* is forwarding reference

```
// case 2
auto x = 27;              // x: ???
auto const cx = x;        // cx: ???

// case 1
auto const& rx = x;       // rx: ???

// case 3
auto&& uref1 = x;         // uref1: ???
auto&& uref2 = cx;        // uref2: ???
auto&& uref3 = 27;        // uref3: ???
```

# auto Type Deduction (4)



```
// arrays
char const name[] {"hello"};
auto  arr1 = name;          // arr1: ???
auto& arr2 = name;          // arr2: ???

// functions
void  func(int, double); // arr1: ???
auto  fun1 = func;          // fun1: ???
auto& fun2 = func;          // fun2: ???
```

# auto Type Deduction (5)



- We can initialize `int`s in 4 ways but replacing `int` with `auto` is not equivalent!!!

```
int x1 = 27;
int x2(27);
int x3 = {27};
int x4{27};
```

```
auto x1 = 27;
auto x2(27);
auto x3 = {27};
auto x4{27};
```

# auto Type Deduction (5)



- When initializer for `auto`-declared variable is enclosed in braces, deduced type is `std::initializer_list`

```
auto x1 = 27;    // type is int
auto x2(27);     // type is int
auto x3 = {27};  // type is std::initializer_list<int>
                 // value is {27}
auto x4{27};     // type is std::initializer_list<int>
                 // value is {27}
```