lvalue/rvalue

Perfect forwarding - motivation

- a not completely correct implementation of emplace

```
template< typename ... TList>
iterator emplace( const_iterator p, TList && ... plist)
{
  void * q = /* the space for the new element */;

  value_type * r = new( q) value_type( plist ...);

  /* ... */
}
```

- Note: Decoupling allocation and construction
  - new(q) - *placement new*
    - run a constructor at the place pointed to by q
      - returns q converted to value_type *
    - a special case of user-supplied allocator with an additional argument q

```
void * operator new( std::size_t, void * q) { return q; }
```

Perfect forwarding - motivation

```
template< typename ... TList>
iterator emplace( const_iterator p, TList && ... plist)
{
  void * q = /* the space for the new element */;

  value_type * r = new( q) value_type( plist ...);

  /* ... */
}
```

- How the emplace arguments are passed to the constructor?
  - Pass by reference for speed, but lvalue or rvalue?
    - Pass an rvalue as rvalue-reference to allow move
    - Never pass an lvalue as a rvalue-reference
    - Properly propagate const-ness of lvalues
  - Three ways of passing required: T &, const T &, T &&
    - The number of emplace variants would be exponential

Perfect forwarding - rules

- Reference collapsing rules
  - Applied only when template inference is involved

X & &	X &
X && &	X &
X & &&	X &
X && &&	X &&

- “Forwarding reference”, also called "Universal reference"
  - T && where T is a template argument

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

X lv;
f( lv);
```

- When the actual argument is an lvalue of type X
  - Compiler uses T = X &, type of p is then X & due to collapsing rules

```
f( std::move( lv));
```

- When the actual argument is an rvalue of type X
  - Compiler uses T = X, type of p is X &&

Perfect forwarding - motivation

- Forwarding a universal reference to another function

```
template< typename T> void f( T && p)
{
  g( p);
}

X lv;
f( lv);
```

- If an lvalue is passed: T = X & and p is of type X &
  - p appears as lvalue of type X in the call to g

```
f( std::move( lv));
```

- If an rvalue is passed: T = X and p is of type X &&
  - p appears as lvalue of type X in the call to g
  - Inefficient – move semantics lost

Perfect forwarding – std::forward

- Perfect forwarding

```
template< typename T> void f( T && p)
{
  g( std::forward< T>( p));
}
```

  - std::forward< T> is simply a cast to T &&

```
X lv;
f( lv);
```

- T = X &
  - std::forward< T> returns X & due to reference collapsing
  - The argument to g is an lvalue

```
f( std::move( lv));
```

- T = X
  - std::forward< T> returns X &&
  - The argument to g is an rvalue
  - std::forward< T> acts as std::move in this case

Perfect forwarding

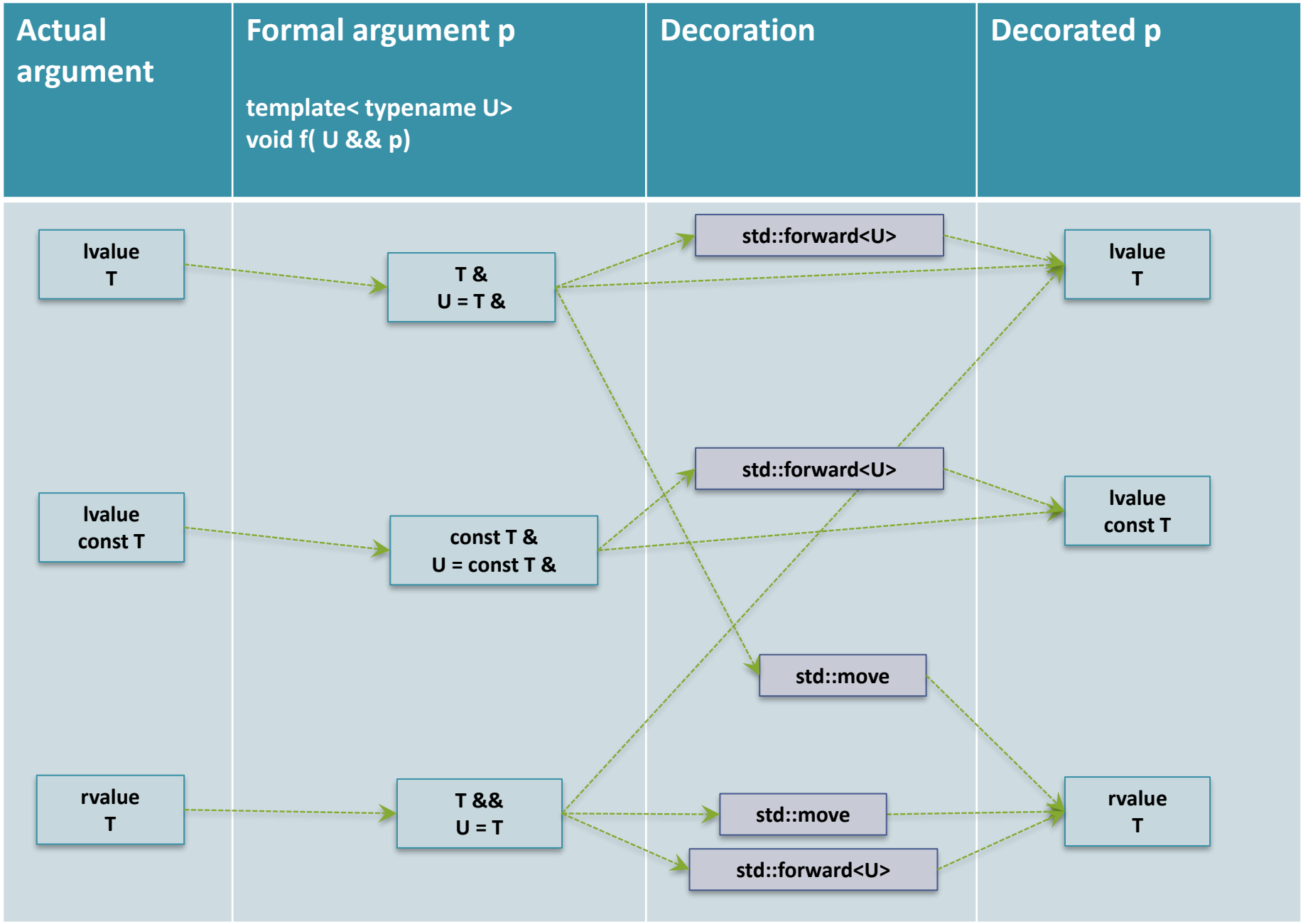
- A correct implementation of emplace

```
template< typename ... TList>
iterator emplace( const_iterator p, TList && ... plist)
{
  void * q = /* the space for the new element */;

  value_type * r = new( q) value_type( std::forward< TList>( plist) ...);

  /* ... */
}
```

Forwarding references



Forwarding (universal) references

- Forwarding references may appear
  - as function arguments

```
template< typename T>
void f( T && x)
{
  g( std::forward< T>( x));
}
```

  - as auto variables

```
auto && x = cont.at( some_position);
```

- Beware, not every T && is a forwarding reference
  - It requires the ability of the compiler to select T according to the actual argument
- The use of reference collapsing tricks is (by definition) limited to T &&
  - The compiler does not try all possible T's that could allow the argument to match
  - Instead, the language defines exact rules for determining T

Forwarding (universal) references

- In this example, T && is **not** a forwarding reference

```
template< typename T>
class C {
  void f( T && x) {
    g( std::forward< T>( x));
  }
};
C<X> o; X lv;
o.f( lv);// error: cannot bind an rvalue reference to an lvalue
```

- The correct implementation

```
template< typename T>
class C {
  template< typename T2>
  void f( T2 && x) {
    g( std::forward< T2>( x));
  }
};
```

Removing references when storing values

Example – storing values of any type

- Goal: Hand-made functor corresponding to the following lambda

```
[p](T & x){ x += p; }
```

- Naive approach

```
template< typename T> class ftor {
public:
  ftor( T && p) : p_( std::forward< T>( p)) {}
  void operator()( T & x) { x += p_; }
private:
  T p_;
};
```

  - Does not work well

```
auto f1 = ftor< std::string>( “Hello”);        // works, passed by moving
std::string s = “Hello”;
auto f2 = ftor< std::string>( s);     // does not work: can’t bind rvalue reference p
```

Example – storing values of any type

- A better implementation

```
template< typename T> class ftor {
public:
  template< typename T2> ftor( T2 && p) : p_( std::forward< T2>( p)) {}
  void operator()( T & x) { x += p_; }
private:
  T p_;
};
```

  - Everything works

```
auto f1 = ftor< std::string>( “Hello”);       // passed as const char * && to conversion
std::string s = “Hello”;
auto f2 = ftor< std::string>( s);    // passed as std::string & to copy-ctor
auto f3 = ftor< std::string>( std::move( s)); // passed as std::string && to move-ctor
```

  - But why the user needs to specify std::string explicitly?

```
auto f2 = make_ftor( s);
```

Example – storing values of any type

- A class

```
template< typename T> class ftor { /*...*/ T p_; };
```

- and its wrapper function (naive attempt)

```
template< typename T>
inline ftor<T> make_ftor( T && p)    // must use forwarding reference
{ return ftor<T>( std::forward< T>(p));
}
```

  - This implementation is wrong and dangerous!

```
std::string s = "Hello";
std::for_each( b, e, make_ftor( s)); // stores std::string & - works faster, but...


std::vector< std::string> v = { "Hello" };
auto f = make_ftor( v.back());                  // stores std::string &
v.pop_back();
std::for_each( b, e, f);                        // crash!!!
```

  - Such implementation may be useful, but users must know that it may store by reference

Example – storing values of any type

- A class

```
template< typename T> class ftor { /*...*/ T p_; };
```

- and its correct wrapper function

```
template< typename T>
inline ftor<std::remove_cvref_t<T>> make_ftor( T && p)
{ return ftor<std::remove_cvref_t<T>>( std::forward< T>(p));
}
```

  - Shorter syntax

```
template< typename T>
inline ftor<std::remove_cvref_t<T>> make_ftor( T && p)
{ return { std::forward< T>(p)};                    // if ctor is not explicit, {} may be omitted
}
```

  - C++14: auto with return values

```
template< typename T>
inline auto make_ftor( T && p)
{ return ftor<std::remove_cvref_t<T>>( std::forward< T>(p));
}
```

Useful standard library type traits

```
#include <type_traits>
```

- Type properties
  - is_void_v, is_enum_v, is_pointer_v, is_const_v, is_abstract_v, is_copy_assignable_v, …
  - Logically: compile-time functions returning bool parametrized by a type
  - Technically: constexpr bool variable templates parametrized by a type
    - xxx_v<T> is a shortcut for xxx<T>::value
  - Usage:

```
template< typename T> struct example {
  static constexpr bool v = std::is_reference_v<T>;
};
```

- Type transformations
  - remove_reference_t, remove_cv_t, remove_cvref_t [C++20]
  - Logically: compile-time functions returning type parametrized by a type
  - Technically: type alias (using) templates parametrized by a type
    - xxx_t<T> is a shortcut for typename xxx<T>::type
  - Usage:

```
template< typename T> struct example {
  using U = std::remove_reference_t<T>;
};
```

- More complex functionality
  - is_same_v, is_convertible_v, make_signed_t, conditional_t …

Type traits – possible implementation

- Type traits – master definition

```
template< typename T> struct is_reference {
  static constexpr bool value = false;
};
```

- Type traits – partial specializations

```
template< typename U> struct is_reference< U &> {
  static constexpr bool value = true;
};
template< typename U> struct is_reference< U &&> {
  static constexpr bool value = true;
};
```

- Global constexpr variable template

```
template< typename T>
inline constexpr bool is_reference_v = is_reference<T>::value;
```

Type traits – possible implementation

- Type traits – master definition

```
template< typename T> struct remove_reference {
  using type = T;
};
```

- Type traits – partial specializations

```
template< typename U> struct remove_reference< U &> {
  using type = U;
};
template< typename U> struct remove_reference< U &&> {
  using type = U;
};
```

- Global type alias template

```
template< typename T>
using remove_reference_t = typename remove_reference<T>::type;
```

Type traits – possible implementation

- Frequently used type alias templates in <type_traits>

```
template< typename T>
using remove_reference_t = typename remove_reference<T>::type;

template< typename T>
using remove_cv_t = typename remove_cv<T>::type;

template< typename T>
using remove_cvref_t = remove_cv_t<remove_reference_t<T>>;
```

  - Usage:

```
template< typename T> inline void example(T && v) {
   std::remove_cvref_t<T> a_copy_of_v = std::forward<T>(v);
   ++a_copy_of_v; std::cout << a_copy_of_v;
}
void test_example(const std::string & x) {
   example(x[0]);  // T = const char &
}
```

Example – storing values of any type

```
template< typename T> class ftor { public:
  template< typename T2> ftor( T2 && p);
};
```

- C++17: deduction guides

```
template< typename T2>
ftor( T2 && p) -> ftor< std::remove_cvref_t< T2>>;
```

  - Allows use of this syntax:

```
std::string s = “hello”;
ftor x( s);
auto y = ftor( s);
```

  - Wrapper functions no longer needed

```
std::pair p(i,d);
```

    - instead of

```
std::pair<int,double> p(i,d);
auto p = std::make_pair(i,d);
```

Example – storing values of any type

- A class

```
template< typename T> class ftor {
public:
  template< typename T2> ftor( T2 && p) : p_( std::forward< T2>( p)) {}
  void operator()( T & x) { x += p_; }
private:
  T p_;
};
```

- and its wrapper function

```
template< typename T> auto make_ftor( T && p)
{ return ftor<std::remove_cvref_t<T>>( std::forward< T>(p));
}
```

- still does not work

```
std::vector< std::string> v;
std::for_each( v.begin(), v.end(), make_ftor( “Hello”));
```

  - ftor<const char *>::operator() requires const char * &

Example – storing values of any type

- The correct implementation is

```
template< typename T1> class ftor {
public:
  template< typename T2> ftor( T2 && p) : p_( std::forward< T2>( p)) {}
  template< typename T3> void operator()( T3 && x) { x += p_; }
private:
  T1 p_;
};
template< typename T4>
ftor( T4 && p) -> ftor< std::remove_cvref_t< T4>>;
```

  - Note: always use forwarding reference T3 && instead of lvalue reference T3 &
    - this allows functionality on containers producing fake references (like vector< bool>)
  - Note: why we did not hide the std::remove_cvref_t inside ftor?

```
template< typename T> class ftor { /*...*/ std::remove_cvref_t<T> p_; };
```

    - Because ftor(x) and ftor(x+1) would produce different instantiations of ftor