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, 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

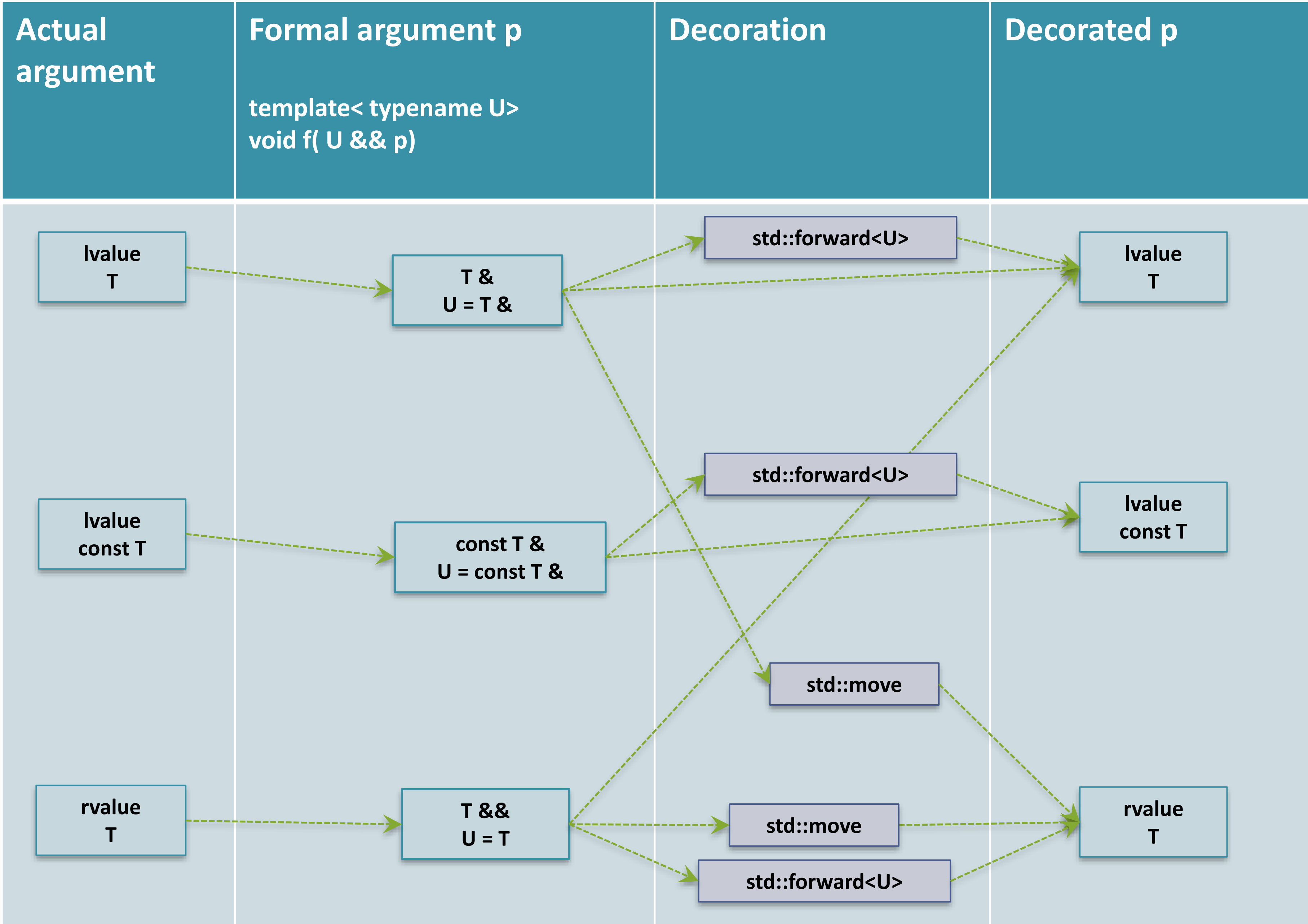
- A correct implementation of emplace

```
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));
  }
};
```