Variadic templates

Variadic templates

- Template heading
  - Allows variable number of type arguments

```
template< typename ... TList>
class C { /* ... */ };
```

  - *typename* ... declares named ***template parameter pack***
  - may be combined with regular type/constant arguments

```
template< typename T1, int c2, typename ... TList>
class D { /* ... */ };
```

  - also in partial template specializations

```
template< typename T1, typename ... TList>
class C< T1, TList ...> { /* ... */ };
```

Variadic templates

```
template< typename ... TList>
```

- template parameter pack - a list of types
- may be referenced inside the template:
  - always using the suffix ...
  - as type arguments to another template:

```
X< TList ...>

Y< int, TList ..., double>
```

  - in argument list of a function declaration:

```
void f( TList ... plist);

double g( int a, double c, TList ... b);
```

  - this creates a named function parameter pack
  - in several less frequent cases, including
    - base-class list:

```
class E : public TList ...
```

    - number of elements in the parameter pack:

```
sizeof...(TList)
```

Variadic templates

```
template< typename ... TList>
  void f( TList ... plist);
```

- named ***function parameter pack***
- may be referenced inside the function:
  - always using the suffix ...
  - as parameters in a function call or object creation:

```
g( plist ...)
new T( a, plist ..., 7)
T v( b, plist ..., 8);
```

  - constructor initialization section (when variadic base-class list is used)

```
E( TList ... plist)
: TList( plist) ...
{
}
```

  - other infrequent cases

Variadic templates

```
template< typename ... TList>
  void f( TList ... plist);
```

- parameter packs may be wrapped into a type construction/expression
  - the suffix ... works as compile-time "for_each"
  - parameter pack name denotes the place where every member will be placed
    - more than one pack name may be used inside the same ... (same length required)

- the result is
  - a list of types (in a template instantiation or a function parameter pack declaration)

```
X< std::pair< int, TList *> ...>
class E : public U< TList> ...
void f( const TList & ... plist);
```

  - a list of expressions (in a function call or object initialization)

```
g( make_pair( 1, & plist) ...);
h( static_cast< TList *>( plist) ...);      // two pack names in one ...
m( sizeof( TList) ...);                     // different from sizeof...( TList)
```

  - other infrequent cases

- fold expressions - variadic templates
  - C++14 – recursion

C++14

```
auto old_sum(){ return 0; }
template<typename T1, typename... T>
auto old_sum(T1 s, T... ts){ return s + old_sum(ts...); }
```

  - C++17 - simplification

```
template<typename... T> ????( T... pack)
( pack op ... )                 // P1 op (P2 op ... (Pn-1 op Pn))
( ... op pack )                 // ((P1 op P2) op ... Pn-1) op Pn
( pack op ... op init )         // P1 op (P2 op ... (Pn op init))
( init op ... op pack )         // ((init op P1) op ... Pn-1) op Pn
```

```
template<typename... T>
auto fold_sum(T... s){
    return (... + s);
}
```

C++17

```
template<typename... T>
auto fold_sum1(T... s){
    return (0 + ... + s);
}
```

```
template<typename ...Args> void prt(Args&&... args)
{ (cout << ... << args) << '\n';  }
```

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

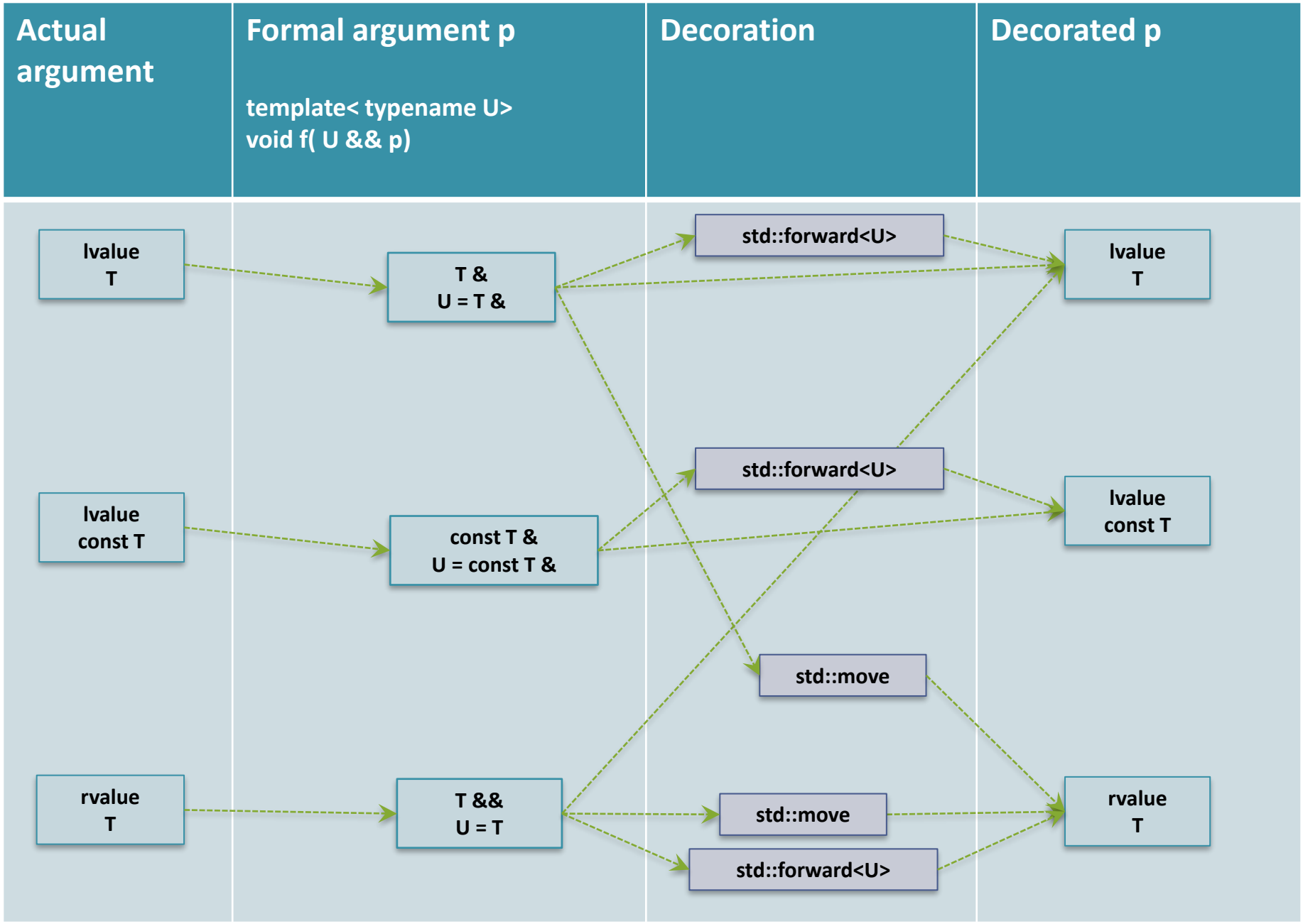
- A correct implementation of emplace

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