

lvalue/rvalue

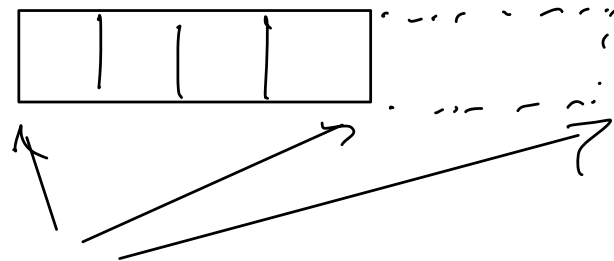
Perfect forwarding - motivation

- ▶ 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 ...);

    /* ... */
}
```



< -> tohle ještě zpet místo pointeru vrátí value_type *↳ pointers type T**

operator new (size_t s, ...) *define *p = new (...) char [n * sizeof(T)]*

redefinice operátoru new, který může brát vlastní argumenty

▶ 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; }
```

podvod, že první krok je zbytečný
neudělá nic, než že ten pointer vrátím

Perfect forwarding - motivation

tedy můžeme přímo libovolně modifikovat, zatímco u $f(x)$ to kromě prototypu konstanty: $\rightarrow$ tedy užijme, že funkce
 template< typename ... TList> **VOJTE** se ale přeloží v jednom $g(\text{string} \&\& r);$ to může modifikovat
 iterator `emplace( const_iterator p, TList && ... plist)` **případě.** a může to už jedno.
 { $g(x);$ **won't compile (I val)**

```
void * q = /* the space for the new element */;
```

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

$$/* \dots */$$
$$G(x);$$
 $f(x):$ ~~int ll $x = N_j$~~

Abbildung $\ell \ell \times = 1 + "$ $\ell \ell$ "

$$G(\text{string} \& n)$$

tedy to ale můžu vyhnout

$g(\text{move}(x));$ ✓

to val

κ_{val}

$\angle > r - \text{val mi}$

man generovat, že existuje jen jeden přístup k objektu
the constructor?

Diagram illustrating a sequence of boxes and arrows:

- Top box: k
- Arrow pointing down to box: k_{n-1}
- From k_{n-1} , two arrows point down to boxes: x (left) and n (right)
- Arrows from x and n point up to box: k_{n-2}
- Text knp is written to the right of the k_{n-1} box.

► 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.

~~being 22 pop()~~

U návrhových hodnot je důležité
kolik a na čem.

string & more (string & p)

{ return static_card

$$\langle \text{string } \ell \rangle (\mu)_i \} \quad \{$$

lval

h val

- jinné potoky musí nékde ten
- objekt nékde žít a sloupě ▶ Ho
- nevím, kdy ho pak zabíjet

Defneta je velmi vyjimecne, aby funkce vmeela r-value, protoze to riká, ze ten objekt, co vmeí, dlouho neprežije...

- Použíá se u move(...), forward(...) a pár jiných.

To se může hodit, pokud vím,
ze to je poslední volání funkce transform.

Perfect forwarding - rules

- Reference collapsing rules

- Applied only when template inference is involved

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

Model: transform (a.begin(), a.end(),
b.begin(), b)

$T\&u = *ib;$

$T\&u = *ib.begin();$ TEMP

môžeme
přeložit

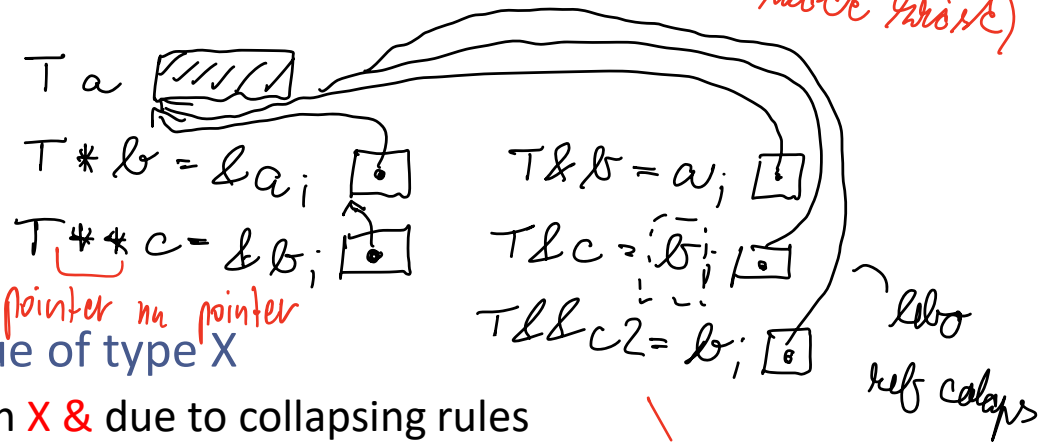
ne je val
a b kuck
môže být

- "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));`

l-value vyhovuje nad r-value

- When the actual argument is an rvalue of type X

- Compiler uses $T = X$, type of p is $X \&\&$

v případě referencí...
b je typu T& and
to míří na a

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`

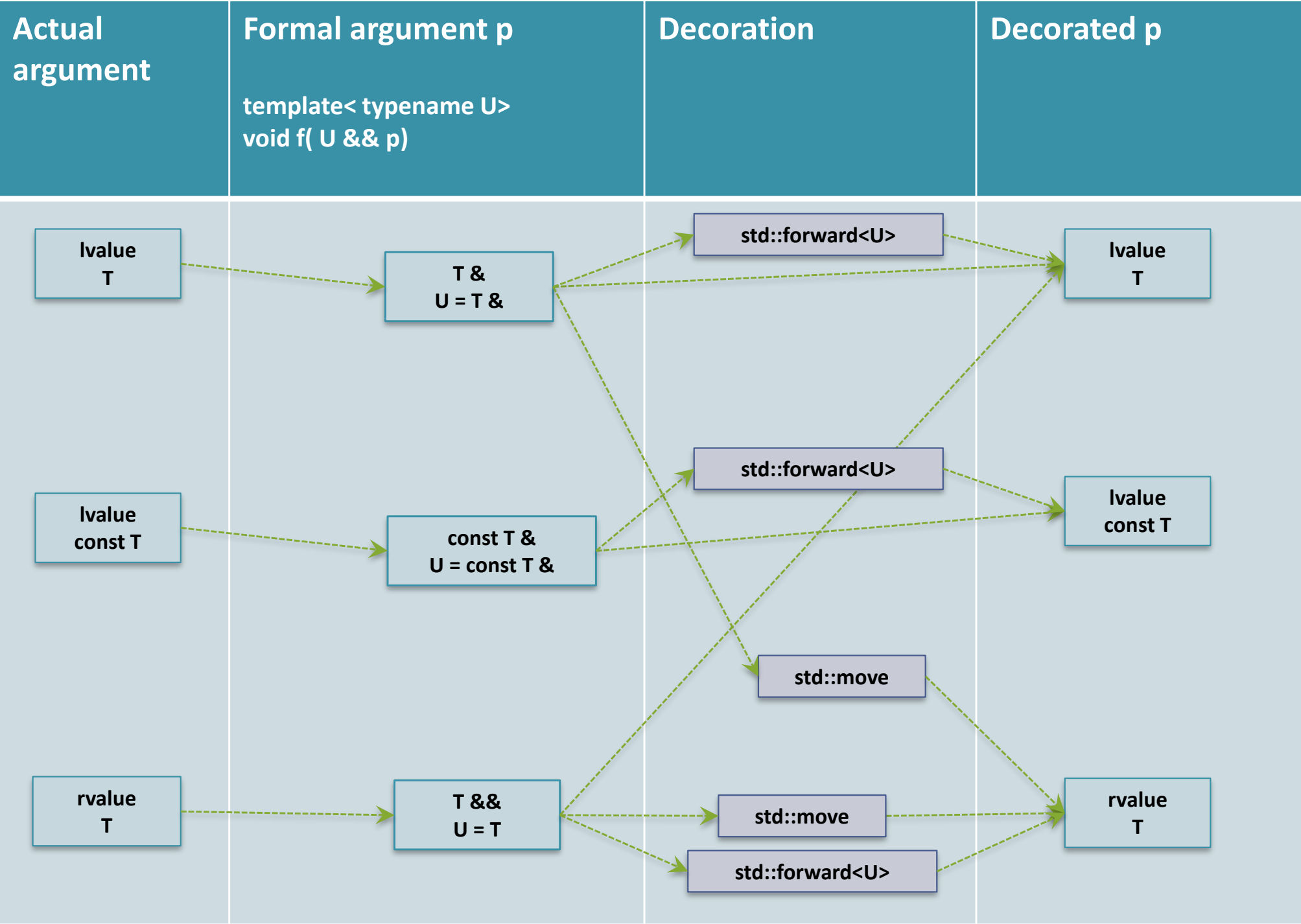
```
template< typename ... TList>
iterator emplace( const_iterator p, TList && ... plist)
{
    void * q = /* the space for the new element */;

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

    /* ... */
}
```

ah zabudnem, medostanem likat,
in pomaljš
program

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));
```

```
    }
```

```
};
```