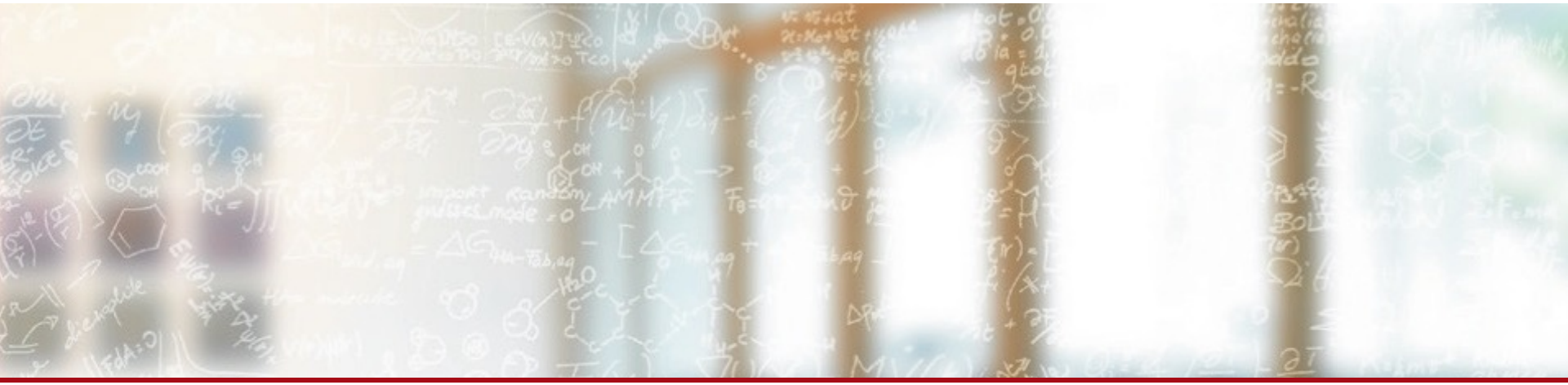
CSCS
Centro Svizzero di Calcolo Scientifico
Swiss National Supercomputing Centre

ETH zürich



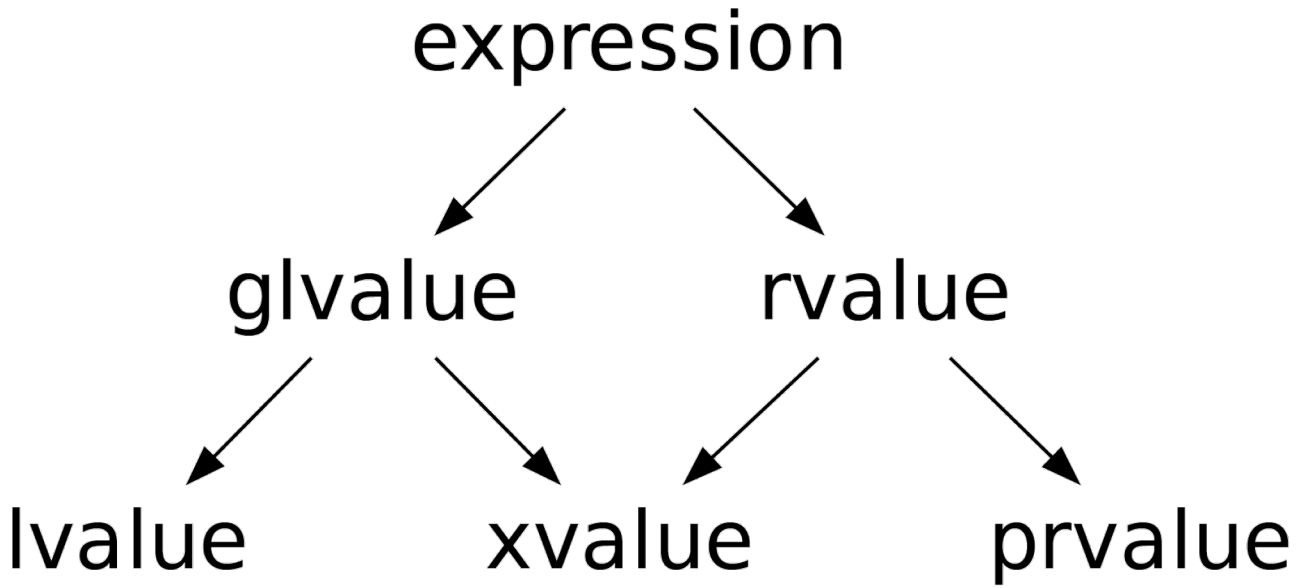
Who's Afraid of Move Semantics?

C++ for HPC course

Mauro Bianco, CSCS

Value Types in C++

- rvalues and lvalues were not enough



References

- Not an object:
  - [No arrays of | no pointers to | no references to] references
- T&: lvalue reference
  - Used to alias an pre-existing object
- T&&: rvalue reference
  - Used to extend the lifetime of temporary objects
    - const T& x = 10; x++; // Error
    - T&& x = 10; x++; // Ok!
- The distinction is useful for overload resolution

Overload resolution with & and &&

```
int f(int& x) {return x;}
int f(int&& x) {return x;}

int main() {
    int x=10;
    f(x);
    f(10);
}
```

```
struct A {
    void callme() &{}
    void callme() &&{}
};

int main() {
    A a;
    a.callme();
    A{}.callme();
}
```

- Allows the implementation of move constructor/assignment
- Allows also move-aware functions, like push_back

A movable container from scratch

```
struct movable {
    int * pv = nullptr;    int s = 0;

    movable(int s) : pv{new int[s]}          {s} {}
    movable(movable&& other) {
        pv = other.pv;          other.pv=nullptr;
        s=other.s;              other.s=0;
    }
    movable(movable const & other) {
        if (pv) { delete[] pv; }
        s=other.s;              pv = new int[s];
        std::copy(other.pv, other.pv+other.s, pv);
    }
};
int main() {
    movable z(movable(200));
    movable u(std::move(z));
}
```

Overload with && defines the move constructor

Move semantics means "stealing the guts" while leaving the other object consistent

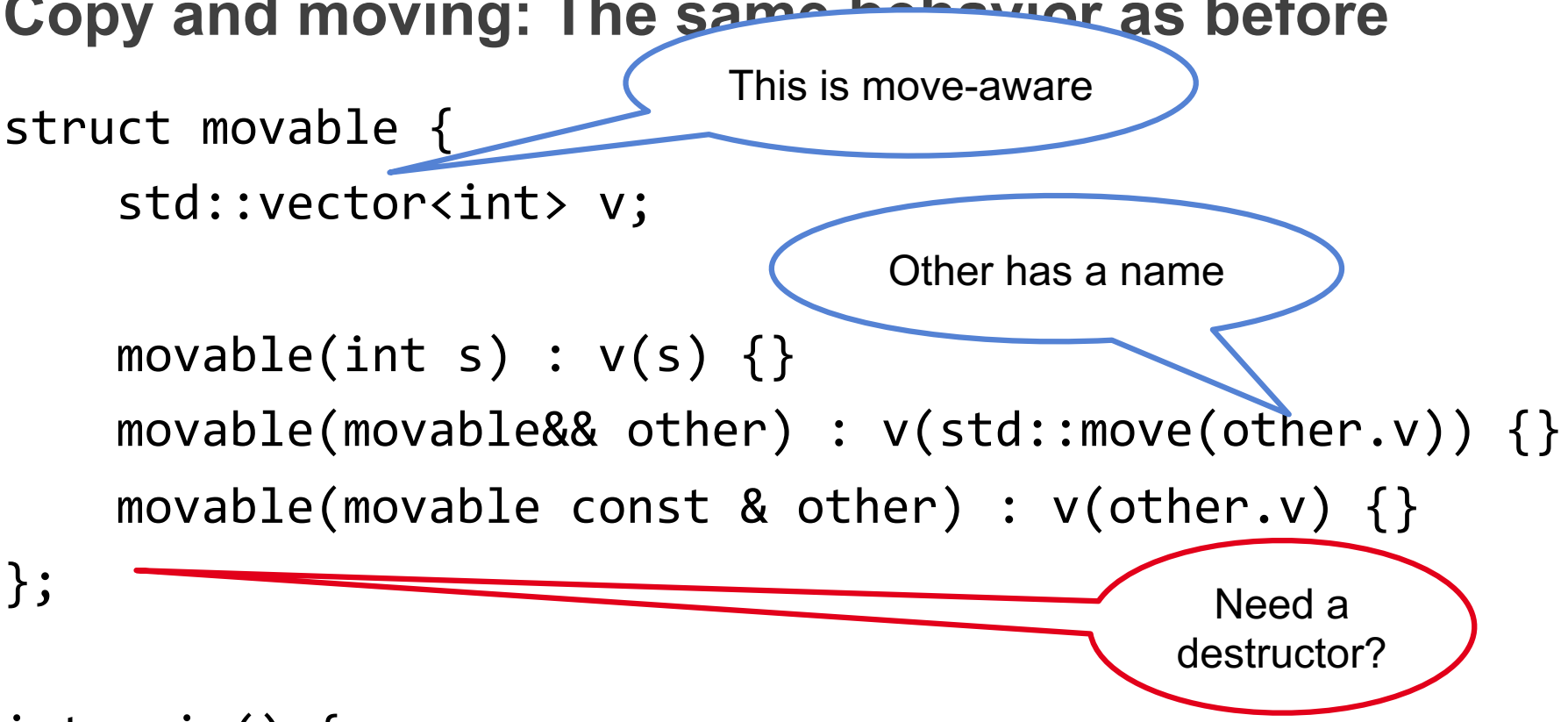
Compare performance

Need a destructor?

temporary unnamed object

Z needs to be transformed into an rvalue ref

Copy and moving: The same behavior as before




```
struct movable {
    std::vector<int> v;

    movable(int s) : v(s) {}
    movable(movable&& other) : v(std::move(other.v)) {}
    movable(movable const & other) : v(other.v) {}
};

int main() {
    movable z(movable(200)); // move constructor
    movable u(std::move(z)); // move constructor
}
```

Move only

```
struct movable {
    std::vector<int> v;

    movable(int s) : v(s) {cout << “Construct\n”;}
    movable(movable&& other) : v(std::move(other.v)) {cout << “Move\n”;}
    movable(movable const & other) = delete;
};

int main() {
    movable z(movable(200));
    SHOW(z.v.size());
    movable u(std::move(z));
    SHOW(z.v.size());
    SHOW(u.v.size());
}
```

```
Construct
z.v.size() :  200
Move
z.v.size() :  0
u.v.size() :  200
```

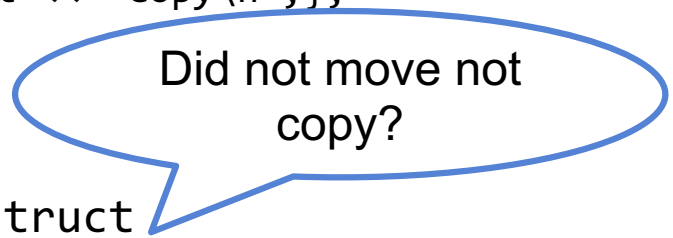
When is move triggered?

```
struct movable {
    std::vector<int> v;



    movable(int s) : v(s) {cout << “Construct\n”;}
};

int main() {
    movable z(movable(200));
    SHOW(z.v.size());
    movable w(z);
    movable u(std::move(z));
    SHOW(z.v.size());
    SHOW(u.v.size());
}
```

```
Construct
z.v.size() :  200
z.v.size() :  0
u.v.size() :  200
```

Move & Copy

```
struct movable {
    std::vector<int> v;

    movable(int s) : v(s) {cout << “Construct\n”;}
    movable(movable&& other) : v(std::move(other.v)) {cout << “Move\n”;}
    movable(movable const & other) : v(other.v) {cout << “Copy\n”;};
};

int main() {
    movable z(movable(200));
    SHOW(z.v.size());
    movable w(z);
    movable u(std::move(z));
    SHOW(z.v.size());
    SHOW(u.v.size());
}
```




```
Construct
z.v.size() :  200
Copy
Move
z.v.size() :  0
u.v.size() :  200
```

Unmovable

```
struct movable {
    std::vector<int> v;

    movable(int s) : v(s) {cout << “Construct\n”;}
    movable(movable&& other) = delete;
    movable(movable const & other) = delete;
};

int main() {
    movable z(movable(200));
    SHOW(z.v.size());
    //movabl
    // movab
    // SHOW(
    // SHOW(
}
```

```
move_only_fail.cpp: In function ‘int main()’:
move_only_fail.cpp:15:27: error: use of deleted function
                                 ‘movable::movable(movable&&)’
```

Almost a Real-World Example

```
while (input != "end") {
    std::string input = read_inpuit();
    std::cout << "inserting \"" << input << "\"\n";
    vector.push_back(std::move(input));
}
```

- push_back has two signatures
  - `void push_back( const T& value );`
  - `void push_back( T&& value );`

Subtleties of moving

```
struct movable {
    std::vector<int> v;
    int s = 0; // stores the size
    movable(int s) : v(s), s{s}
    { std::cout << "Default\n"; }
};
```

Trivially movable objects are copied!

```
int main() {
    movable z(200);
    movable u(std::move(z)); // move constructor
    SHOW(z.v.size()); // prints 0 (OK)
    SHOW(z.s); // prints 200 (!!)
}
```

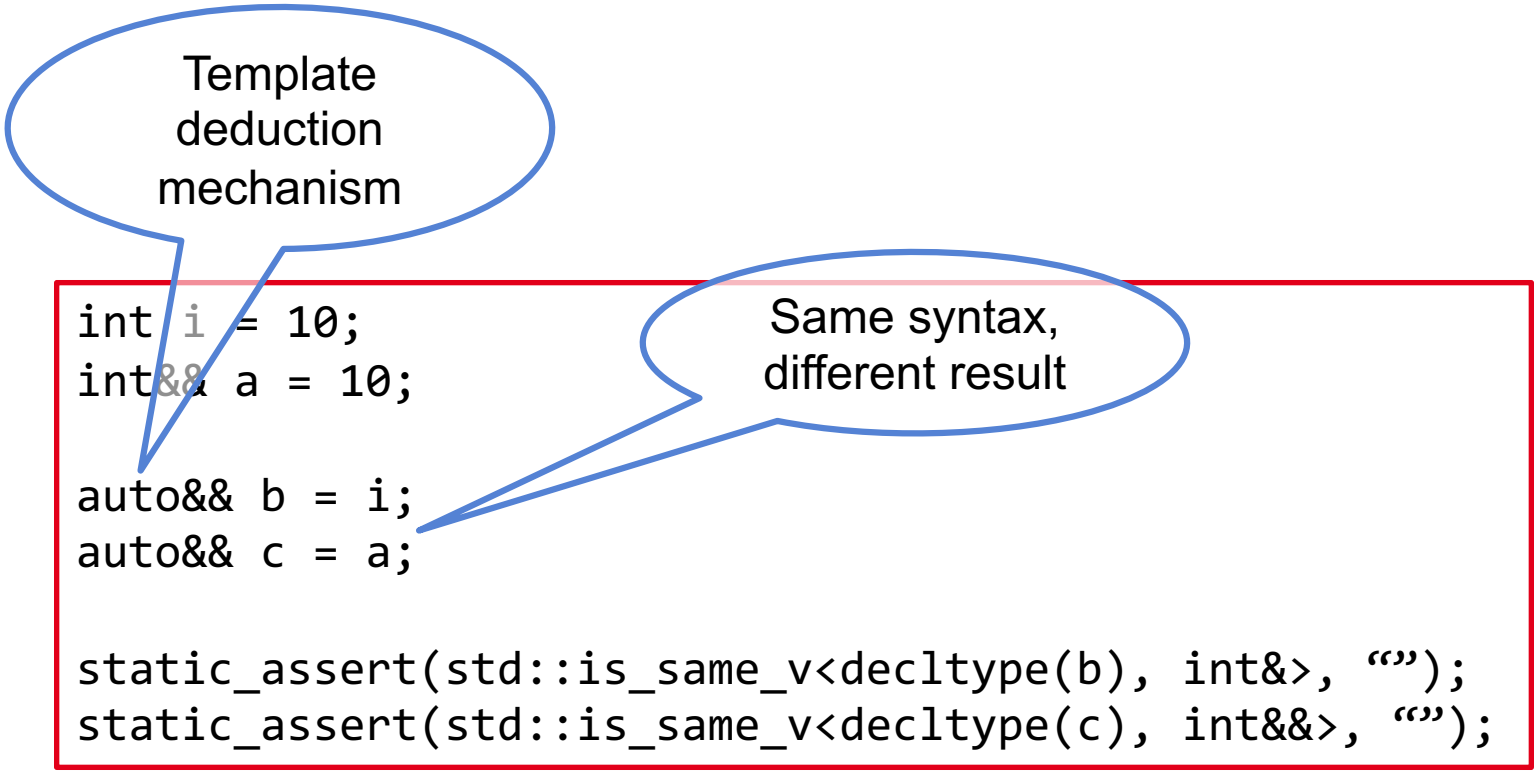
z is probably in a inconsistent state

Templates and T&&: Universal References

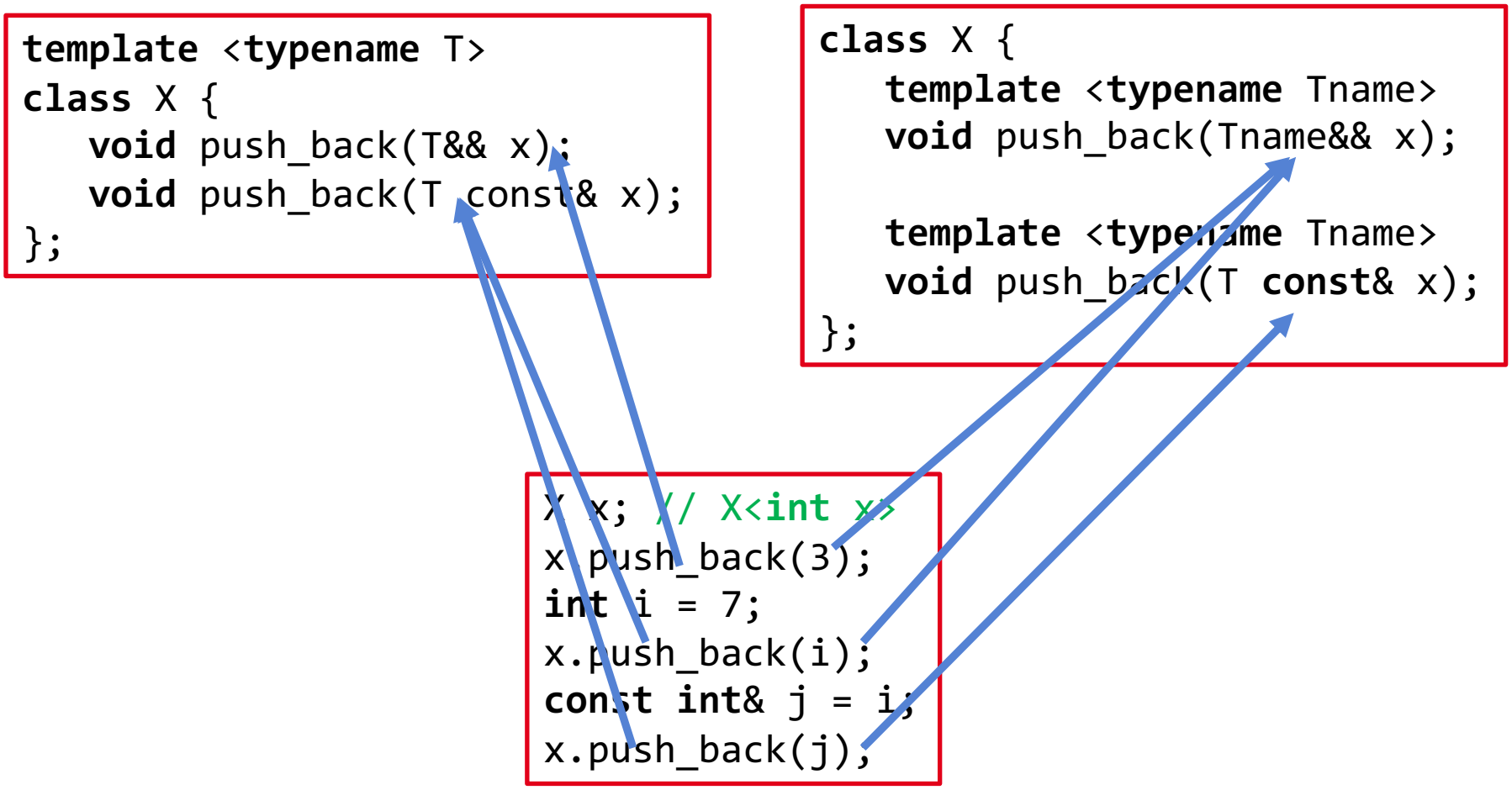
```
template <typename T>
void foo(T&& x) {
    static_assert(std::is_same<decltype(x), ???>::value, " ");
}

int main() {
    int i = 19;
    foo(i);
    foo<int&>(i);
    foo(42);
}
```

- T& && -> T&
- T&& && -> T&&
- T&& & -> T&
- T& & -> T&

Templates and T&&: Universal references



Templates and T&&: Universal references



std::forward

- Passing to the next function maintaining the value category

```
void bar(int& x) {x++;}
void bar(int&& x) {};

template <typename T>
void foo(T&& x) {
    bar(std::forward<T>(x));
}

int main() {
    int i = 19;
    foo(i);
    foo<int&>(i);
    foo(42);
}
```

If decltype(x) is T&& then x is a &&, otherwise it's a &

What if std::move?