

# r-value references

P. Ronchese

Dipartimento di Fisica e Astronomia “G.Galilei”

Università di Padova

“Object oriented programming and C++” course

## “l-value” and “r-value” early definition

### C

- “l-value”: an expression that may appear on the left or on the right hand side of an assignment
- “r-value”: an expression that may appear only on the right hand side of an assignment

```
int a=12; // "a": lvalue, "12": rvalue
int b=a;  // "a": lvalue,
          //          may appear on the right
int c=a*b; // "a*b": rvalue (on the right)
c*b=15;    // WRONG! "c*b": rvalue,
          //          cannot appear on the left
```

## C++: new situations

References introduce some complication:

- The call of a function returning by value gives a r-value
- The call of a function returning by reference gives a l-value

```
int  vFunc(); // return by value
int& rFunc(); // return by reference
...
int a=vFunc(); // vFunc(): rvalue
rFunc()=43;    // rFunc(): lvalue
vFunc()=57;    // WRONG! rFunc(): rvalue,
                // cannot appear on the left
```

## “l-value” and “r-value” redefinition

### C++

- “l-value”: an expression that refers to a memory location so that the address of that memory location can be taken via the & operator
- “r-value”: an expression that’s not a l-value

```
int  vFunc(); // return by value
int& rFunc(); // return by reference
...
int* p=&rFunc(); // rFunc(): lvalue,
                // address can be taken
int* q=&vFunc(); // WRONG! vFunc(): rvalue,
                // address cannot be taken
int& r= vFunc(); // WRONG! vFunc(): rvalue,
                // reference cannot be taken
```

## r-value references

### C++11 only

r-value references give a solution to two issues:

- move semantics
- perfect forwarding

r-values references can be declared with a `&&`

```
void f(      X&  x); // called with non-const
                // lvalues only
void f(const X&  x); // called with
                // lvalues or rvalues
void f(      X&& x); // called with rvalues
```

## r-value references are r-values?

r-value references without a name are r-values,  
otherwise they're l-values.

```
void g(X&& x) {f(x); // call f(X& x)}
```

```
X&& g();
```

```
...
```

```
    f(g()); // call f(X&& x)}
```

## Universal references

### Template functions

Variables or parameters with type  $T\&\&$  where  $T$  is a deduced type are called “universal references”.

The actual type depends on a set of rules.

Type deduction for template parameters being universal references:

- l-values of type  $T$  are deduced to be of type  $T\&$ .
- r-values of type  $T$  are deduced to be of type  $T$ .

Reference-collapsing rules:

- A rv-reference to a rv-reference becomes a rv-reference.
- Any other reference to reference becomes a lv-reference.

## Moving objects

An object is “moved” when the resources it owns are simply taken by another object, or swapped, without being copied.

Move semantics allow performance gain in several situations:

- moving from objects just before their end-of-life:
  - temporary objects
  - objects in `std::vector` to reallocate
- object swap (e.g. in sorting)

## Move semantics

“move” constructor and assignment:  
constructor and assignment operators  
with a r-value reference argument

```
class X {  
    ...  
    X(X&& x) noexcept;  
    const X& operator=(X&& x) noexcept;  
    ...  
};
```

### Object lifetime

The objects owned by `x` are not deleted until the instance of `X` move-created or move-assigned from `x` is deleted:  
be careful with destructor side effects!

Move constructor and assignment are used in STL containers  
only when declared `noexcept`.

## Force move semantics

### Object swap

Object swap is usually done by round-copying objects through a temporary; usual assignment involves l-values. Move assignment can be forced transforming l-values to r-values by `std::move` ( in `<utility>` ).

```
template<class T>
void swap(T& l, T& r) {
    T tmp(std::move(l));
    l = std::move(r);
    r = std::move(tmp);
}
```

Simply declaring the arguments as r-value reference  
`template<class T> void swap(T&& l, T&& r)`  
is not sufficient.

## Perfect forwarding

### Function parameters forwarding

Function call inside another function,  
passing as parameters (part of) the parameters  
of the enclosing function, with exactly the same

```
template<class T>
void wrap( T&& x ) {
    ...
    func( std::forward<T>( x ) );
    ...
    return;
}
```