

Move Semantics

TPV 2
Samir Genaim

Lvalue y Rvalue
Rvalue reference
Move Semantics
Perfect Forwarding

Lvalue y Rvalue

¿Qué son y Porque Necesito Saberlo?

- ◆ Cada expresión en C++ se clasifica como **lvalue** o **rvalue** (otras categorías: **xvalue**, **gvalue**, **prvalue**)
- ◆ Saber distinguir entre estos tipos de expresiones es muy importante:
 - ✓ para entender el paso de parámetros
 - ✓ para aprovechar de lo que ofrece el lenguaje/compilador C++ (en términos de código eficiente) como move semantics, perfect forwarding, etc.
 - ✓ para entender errores y warnings del compilador
 - ✓ para entender como el compilador infiere los tipos en function templates (y auto)
 - ✓ ...

Definición de Lvalue y Rvalue

- ◆ lvalue: un objeto que ocupa una posición identifiable en la memoria
- ◆ rvalue: un objeto que no sea lvalue
- ◆ la palabra “objeto” no refiere a una instancia de una clase, sino a una expresión c++ (o lo que devuelve una expresión)
- ◆ Es una definición intuitiva, pero no la más precisa

Ejemplos de Lvalue y Rvalue

```
int i;  
int* p = &i;  
i = 2
```

i es un lvalue

Tiene posición identificable en la memoria (uso & para consultarla)

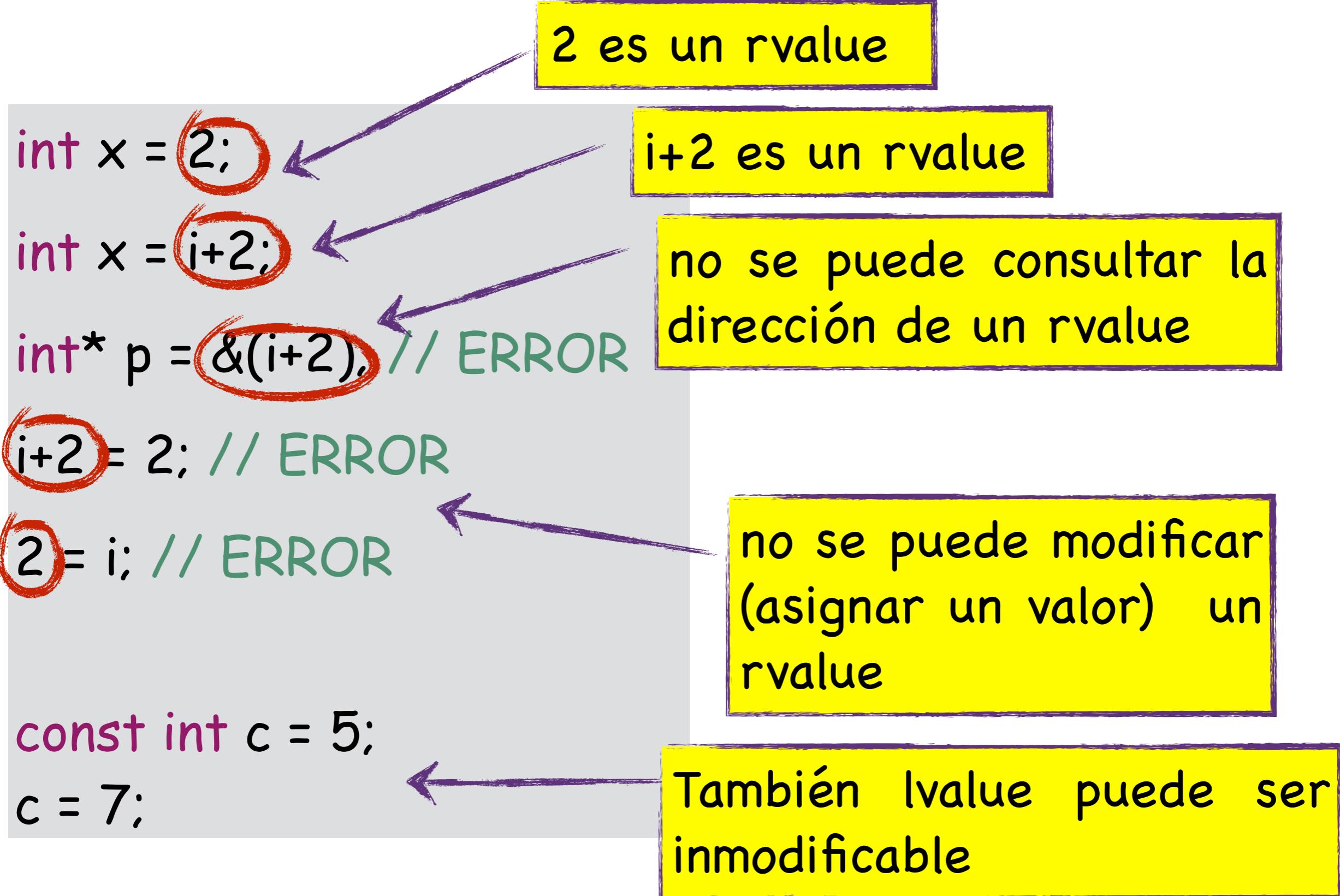
se puede modificar una expresión lvalue (asignarle un valor).

```
class Dog {};  
Dog d;  
Dog* p = &d;
```

d es un lvalue

Tiene posición identificable en la memoria (uso & para consultarla)

Ejemplos de Lvalue y Rvalue



Ejemplos de Lvalue y Rvalue

```
class Dog {};  
Dog d = Dog();
```

Dog() es un rvalue, su dirección no es identifiable, por otro lado d es lvalue

```
Dog() = ... // OK!  
Dog().foo()
```

se puede “modificar” rvalue (?!?!), pero no directamente – p.e.j., si foo modifica el estado o el operator =

```
int sum(int x, int y) { return x+y; }  
int i = sum(3,4);
```

sum(3,4) es un rvalue, su dirección no es identifiable, por otro lado i es lvalue

Lvalue puede crear Rvalue y vice versa

```
int i = 1;  
int x = i + 1;
```

se puede usar lvalue (i) para
crear rvalue (i+1)

```
int v[3];  
*(v + 2) = 3;
```

se puede usar rvalue (v+2)
para crear lvalue *(v+2)

Parámetro Lvalue

también esto es (parámetro) lvalue

```
int square(int x){return x*x;}
```

```
int i = 10;
```

```
square(i);
```

```
square(10);
```

se puede pasar tanto lvalue
como rvalue a un parámetro
lvalue

Lvalue Reference

Ivalue reference es una referencia a lvalue (hasta ahora le llamamos simplemente reference)

i es un lvalue, 2 es un rvalue

```
int i = 2;
```

&x es un lvalue reference

```
int &x = i;
```

no se puede asignarle rvalue a lvalue reference, porque rvalue no tiene dirección identificable ...

```
int &y = 5; // ERROR
```

```
const int &z = 5;
```

Caso excepcional: si la referencia es const, se puede asignar rvalue. El compilador crea una variable auxiliar w con valor 5 y asigna w a z (más o menos)

Parámetro Lvalue Reference

```
int square(int& x){return x*x;}
```

```
int i = 10;  
square(i);  
square(10); // ERROR
```

```
int cube(const int& x){return x*x*x;}  
cube(i);  
cube(10);
```

Esto es un parámetro lvalue reference. No hay que confundirlo con x (x es lvalue porque tiene dirección de memoria identifiable)

a un parámetro lvalue reference se puede pasarle lvalue pero no rvalue.

usando const, a lvalue reference se puede pasarle tanto lvalue como rvalue – caso excepcional

Métodos pueden devolver lvalue

```
int sum(int x, int y) { return x+y; }  
int x = sum(3,4);
```

sum devuelve rvalue

```
int global;  
int& foo() {return global;}  
foo() = 5;
```

foo devuelve lvalue (la salida es lvalue
referente pero lo que devuelve se
trata como lvalue)

operator[] devuelve lvalue (usually)

```
class A {  
    int x[10];  
public:  
    int& operator[](int index) {  
        return x[index];  
    }  
};  
  
A a;  
cout << a[1];  
a[1] = 2;
```

devuelve lvalue, cambiando `int&` por `int` devolvería rvalue

cambiando `int&` por `int` esto compila

cambiando `int&` por `int` esto sería un error, no compila porque `a[1]` no es lvalue

Rvalue Reference

Rvalue reference

- ◆ Ya sabemos que un rvalue no se puede pasar por referencia, excepto si el parámetro es **const**.
- ◆ En C++11 esto ha cambiado, han introducido algo que se llama **rvalue reference**.

(1) `void foo(int& x) { }`

lvalue reference

(2) `void foo(int&& x) { }`

rvalue reference

`int i=10;`

`foo(i);`

llama a (1) porque i es lvalue

`foo(10);`

llama a (2) porque 10 es rvalue

Rvalue reference

(1) `void foo(int& x) { }`

lvalue reference

(2) `void foo(int&& x) { }`

rvalue reference

```
int i=10;  
foo(i);  
foo(10);  
foo((int&&)i);  
foo(std::move(i));  
foo((int&)10); // ERROR
```

llama a (1)

llama a (2)

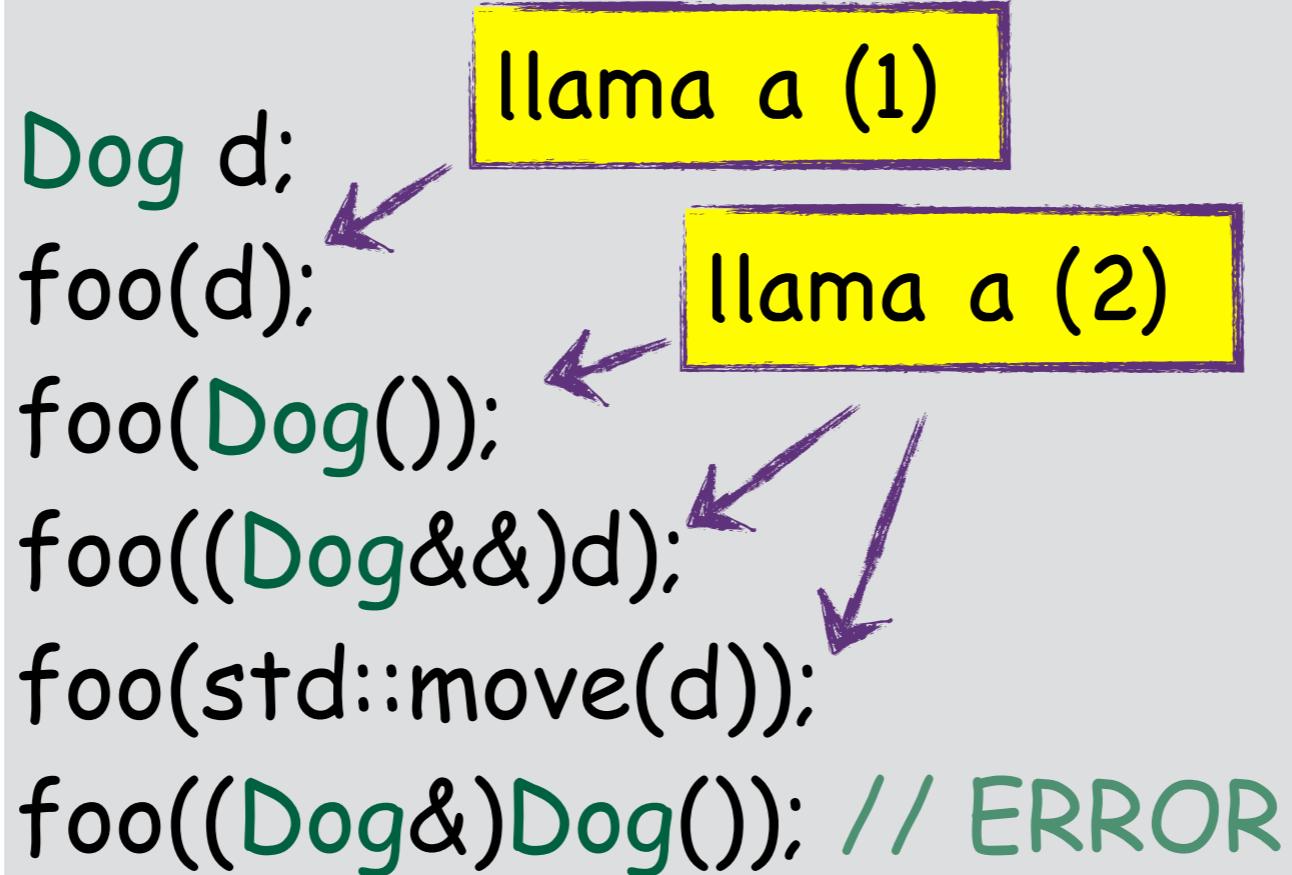
llama a (2) - el casting dice
al compilador que interprete i
como rvalue

std::move “convierte” lvalue a
rvalue - usar siempre en
lugar del casting

no se puede interpretar rvalue
como lvalue reference

Rvalue reference

- (1) `void foo(Dog& x) { }`
- (2) `void foo(Dog&& x) { }`



Al llamar a (2) el parámetro actual (en la llamada) se pasa por referencia, no copia

Rvalue reference: ambigüedad

- (1) `void foo(int x) { }`
- (2) `void foo(int&& x) { }`

```
int i=10;  
foo(i);
```

llama a (1) porque es el único que acepta lvalue

```
foo(10);  
foo(std::move(i));
```

Ambiguo porque (1) y (2) aceptan rvalue

Rvalue reference: acepta lvalue!

(2)

```
void foo(int&& x) { }
```

```
int i=10;
```

```
foo(i);
```

```
foo(10);
```

```
foo(std::move(i));
```

Acepta tanto lvalue como rvalue, porque no hay otra definición con `int& x`

Todos llaman a (2) – si no tenemos otra definición para foo

Dentro la función foo, no podemos estar seguros de que el parámetro actual (en la llamada) es rvalue. Si tenemos otra función

(1) `void foo(int& x)`

sí podemos estar seguros porque si el parámetro actual es lvalue llamaría a (1) y si es rvalue llamaría a (2)

Move Semantics

(el uso de rvalue reference)

Implementación de un Vector

```
template<typename T>
class MyVector {
    T** elem_;
    int capacity_;
    int size_;
public:
    MyVector() {
        capacity_ = 10;
        size_ = 0;
        elem_ = new T*[capacity_]();
    }
    virtual ~MyVector() {
        for(int i=0; i<size_; i++) delete elem_[i];
        delete [] elem_;
    }
    void push_back(const T& e){
        ...
        elem_[size_++] = new T(e);
    }
};
```

Implementación de un vector (parecida a lo que hace std::vector)

push_back recibe e por referencia para no hacer una copia innecesaria. El const es para permitir el paso de rvalue

Para añadir e al array, hacemos una copia ...

Cuando copiar no es necesario ...

```
class A {  
public:  
    A(int x, double z) { /* add x and z to data */ }  
    A(const A& o) { /* deep copy of o.data to data */ }  
    virtual ~A() { /* delete data */ }  
    void addData(int x, double z) { /* add to data */ }  
private:  
    LinkedList* data;  
};
```

```
MyVector<A> v;  
  
void f()  
{  
    ...  
    v.push_back(A(1,2,3));  
    ...  
}
```

- ◆ push_back hace una copia de este rvalue, hace "deep copy" del atributo data.
- ◆ pero el rvalue no se usa después de la llamada a push_back, sería mejor **mover** data del rvalue y no copiarlo.
- ◆ **mover** significa hacer **data=o.data** y después **o.data=nullptr**
- ◆ pero esto no se puede hacer en la constructora de copia, porque para otras situaciones necesitamos hacer "deep copy" de data

Constructora de Movimiento

```
class A {  
public:  
    A(const A& o) { /* deep copy of o.data to data */ }  
    A(A&& o) { data=o.data; o.data=nullptr; }  
    ...  
private:  
    LinkedList* data;  
};
```

Constructora de movimientos a partir de C++11. Acepta rvalue

- ◆ Construir una instancia de A usando otra de categoría **rvalue**, llamaría a la constructora de movimiento en lugar de la constructora de copia.
- ◆ Se supone que ese **rvalue** se va a destruir (o no se va a usar más) al salir de la constructora, así que podemos mover sus recursos en lugar de copiarlos.
- ◆ Construir una instancia de A usando otra de categoría **lvalue** llamaría a la constructora de copia

Mover en lugar de copiar ...

```
class A {  
public:  
    A(A&& o){ data=o.data; o.data=nullptr; }  
    ...  
};
```

Constructora de movimiento.

```
private:  
    LinkedList* data;  
};
```

```
void push_back(T&& e) {  
    assert(size_<capacity_);  
    elem_[size_++] = new T(std::move(e));  
}
```

Añadimos una versión
de `push_back` para
rvalue reference

Invoca la constructora de movimiento. `std::move` es
necesario porque `e` no es rvalue es lvalue. Sin
`std::move` llamaría a la constructora de copia!

`v.push_back(A(1,2,3))` invoca `push_back(T&& e)` porque
el parámetro es rvalue

Se puede usar para mover lvalue ...

```
MyVector<A> v();  
  
void g() {  
    A x(1,2,3);  
    x.addData(2,4,5);  
    // add lots of things  
    // ...  
    v.push_back(std::move(x));  
    // ...  
}
```

Se supone que x no se va a usar después de la llamada a push_back, así que se puede mover en lugar de copiar

std::move es para “convertir” x de lvalue a rvalue, así llamaría a push_back(T&& e) y no a push_back(const T& e). En principio no vamos a usar x después de esta linea!

operator= de Movimiento

```
class A {  
public:  
    A& operator=(const A& o) { ← operator= de copia  
        // delete current list in data;  
        // deep copy o.data to data here;  
        return *this;  
    }  
    A& operator=(A&& o) { ← operator= de movimiento  
        // delete current list in data;  
        data = o.data;  
        o.data = nullptr;  
        return *this;  
    }  
    ...  
};
```

```
A x;  
A z;  
A w;  
x = A(1,2,4);      // move  
z = x;             // copy  
w = std::move(z); // move
```

Perfect Forwarding

Perfect Forwarding: el problema

```
void push_back(T&& e) {  
    assert(size_ < capacity_);  
    elem_[size_++] = new T(std::move(e));  
}
```

Como sabemos que en la llamada el parámetro actual es rvalue, y tenemos acceso al nombre del parámetro, podemos usar std::move para convertir e en rvalue

```
template<typename... Targs>  
void emplace_back(Targs... args) {  
    assert(size_ < capacity_);  
    elem_[size_++] = new T(args...);  
}
```

Al usar parameters pack es mas difícil hacerlo, porque no sabemos si los parámetros actuales (en la llamada) son lvalue o rvalue: args... se expande a parámetros ros lvalue

Perfect Forwarding: la solución

```
template<typename... Targs>
void emplace_back(Targs&& ...args) {
    assert(size_ < capacity_);
    elem_[size_++] = new T(std::forward(args)...);
}
```

El compilador mantenga las categorías (lvalue/rvalue) de los parámetros actuales (en la llamada a `emplace_back`) cuando se las pasa a la constructora. Esto se llama **perfect forwarding**

Sin Perfect Forwarding ...

```
class A {  
public:  
    A(int x, double & z) { } (1)  
    A(int x, double && z) { } (2)  
...  
private:  
    DataStructure* data;  
};
```

- ◆ `v.emplace_back(1,2.3)` tiene que añadir “`new A(1,2.3)`” al vector, es decir tiene que usar la constructora (2) para crear el objeto porque 2.3 es un rvalue
- ◆ sin `&&` y `std::forward` usaría (1), porque la instancia de `embalce_back` pasaría lvalue a la constructora (el segundo parámetro de `embalce_back`)