

**UNIVERSIDAD NACIONAL DE SAN AGUSTÍN DE AREQUIPA**  
**FACULTAD DE INGENIERIA DE PRODUCCION Y SERVICIOS**  
**Escuela profesional de Ciencias de la Computación**



**CIENCIAS DE LA COMPUTACION II**

**Docente:** MAMANI ALIAGA, ALVARO

HUANCA PARQUI , Elizabeth Yasmin

# CONCEPTOS EN LA SEMANTICA DE MOVIMIENTO

## ¿QUE ES LA SEMANTICA DEL MOVIMIENTO?

Se añadió al lenguaje con el estándar c++ 11

```
T f(T o) { return o; }
```

```
T b = f(a);
```

Esta es una función sin semántica de movimiento.

```
T f(T o) { return o; }
```

Nuevo objeto construido

```
T b = f(a);
```

Nuevo objeto construido

Esta función toma un objeto de tipo T y devuelve un objeto del mismo tipo T  
esta función utiliza `Col Val value` que significa que cuando se llama a esta  
función se tiene que construir un objeto para que lo utilice la función como la  
función también devuelve por valor se construye otro objeto nuevo para el valor

de retorno, en este punto se han construido dos nuevos objetos. Uno de ellos es un objeto temporal que solo se utiliza mientras la función está operativa.

```
T f(T o) { return o; }
```



Objeto temporal destruido

```
T b = f(a);
```



Constructor de copia utilizado

Cuando se crea el Nuevo objeto de retorno se llama al constructor de copia para copiar los contenidos de los objetos temporales al nuevo objeto b después que la función haya terminado el objeto temporal que se usa en la función queda fuera de ámbito y se destruye.

## ¿Que hace un constructor de copia?

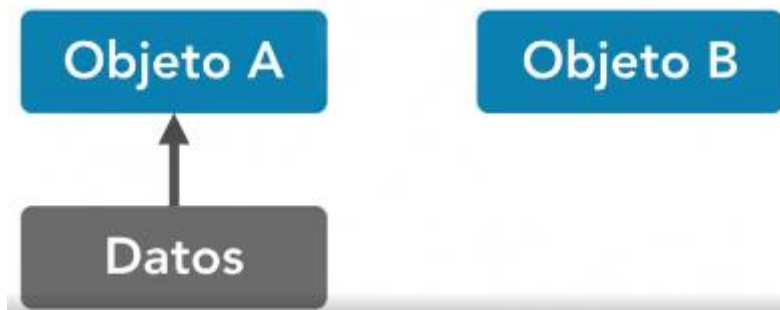
Primero tiene que *inicializar* el objeto y después *copiar* la *información* relevante del *antiguo* objeto al *nuevo* objeto , dependiendo de la clase puede que sea un *contenedor* con muchos datos, esto puede suponer una cantidad importante de tiempo y de uso de memoria.

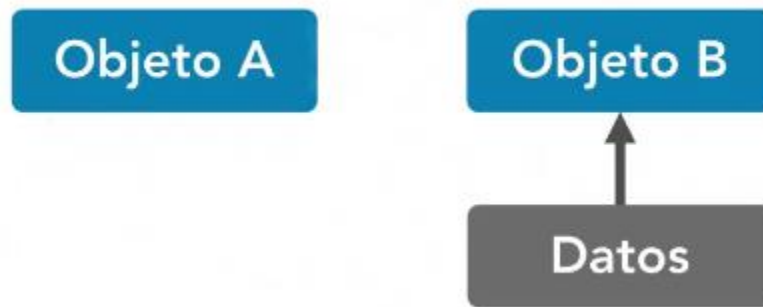
```
// constructor de copia
T::T( T & old ) {
    reset();
    copy_data(_a, old._a);
    copy_data(_b, old._b);
    copy_data(_c, old._c);
    copy_data(_d, old._d);
}
```

Con la *semántica* de *movimiento* se puede mitigar gran parte de ese trabajo simplemente *moviendo* los datos en vez de *copiarlos*.

```
// constructor de movimiento
T::T( T && old ) {
    reset();
    _a = std::move(old._a);
    _b = std::move(old._b);
    _c = std::move(old._c);
    _d = std::move(old._d);
    old.reset();
}
```

Mover los datos solo *implica* re asociar los datos con el *nuevo* objeto, *no* se hace *ninguna* copia, esto se hace con algo denominado *referencia rvalue*.





Una referencia rvalue funciona de manera muy similar a una referencia lvalue con una diferencia importante una referencia rvalue se puede mover, una referencia lvalue NO.

Semántica del movimiento->GANAR EN EFICACIA Y EN USO DE MEMORIA

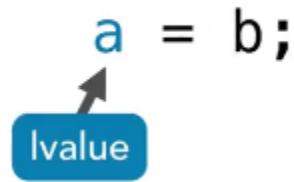
T & x Referencia lvalue

T && y Referencia rvalue

# ¿QUE SON LOS LVALUE -RVALUE?

Esta es una sencilla expresión de asignación

a = b;



EN TERMINOS MUY SIMPLISTAS CUALQUIER EXPRESION QUE PUEDA APARECER EN LA PARTE IZQUIERDA DE UNA ASIGNACION ES UN LVALUE

UNA EXPRESION QUE SOLO PUEDE APARECER EN LA PARTE DERECHA DE UNA ASIGNACION ES UN RVALUE.

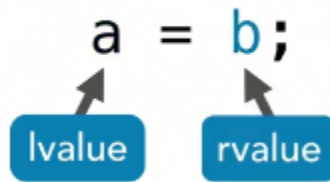
Observa la diferencia un lvalue puede aparecer en la parte izquierda de una asignación, también puede aparecer en la parte derecha, pero sigue siendo un lvalue porque sería válido que apareciese en la izquierda.

El rvalue se llama así porque normalmente aparece en el lado derecho de una asignación.

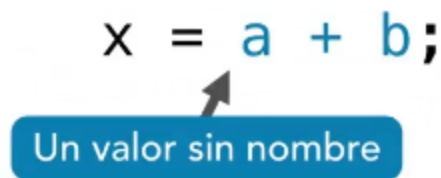
Así que fundamental la "l" viene de lefth ->izquierda; "r" de rigth ->derecha

Aquí la distinción más importante, es que un rvalue se puede mover.

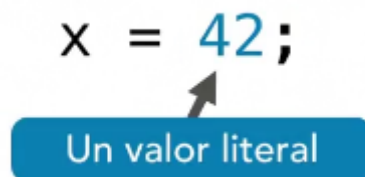
RVALUE es un valor temporal que está listo para caducar.



Esto también se llama un xvalue o un valor que caduca. Normalmente este es un valor sin nombre como el resultado de una expresión.



Un valor literal recibe a veces el nombre de rvalue puro o prvalue, normalmente esta categoría incluye valores literales y cualquier cosa que devuelve una función que no sea una referencia. Lo único en común que tienen estas categorías es que se pueden mover, esta es una diferencia importante.

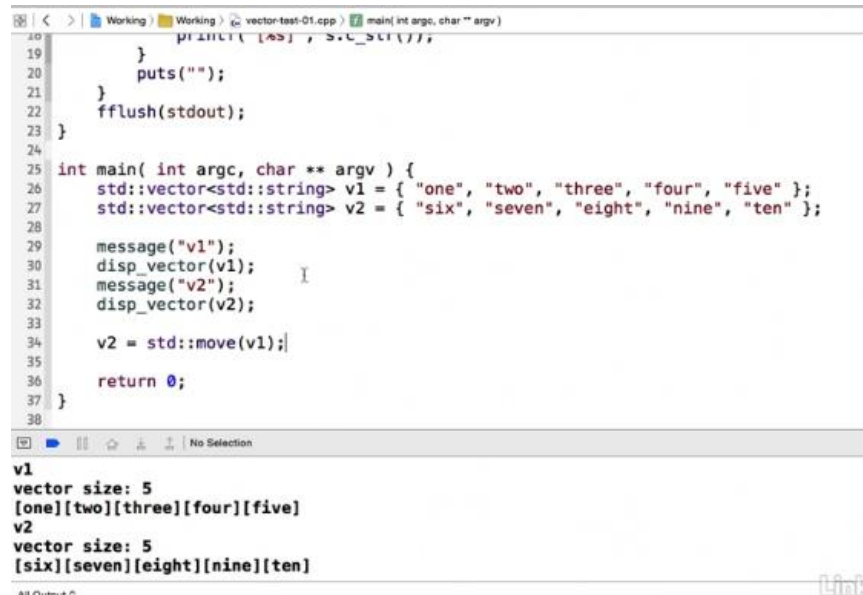


# CÓMO USAR LA SEMANTICA DEL MOVIMIENTO

## USO DE STD::MOVE

La biblioteca de C++ proporciona una función plantilla denominada move, el move estándar funciona en realidad como un conversor de tipo, se utiliza para decirle al compilador que utilice un objeto como si fuera un rvalue, lo que significa que se puede mover

### VECTOR-TEST-01.CPP



```
20 }
21 puts("");
22 }
23 fflush(stdout);
24 }
25 int main( int argc, char ** argv ) {
26     std::vector<std::string> v1 = { "one", "two", "three", "four", "five" };
27     std::vector<std::string> v2 = { "six", "seven", "eight", "nine", "ten" };
28
29     message("v1");
30     disp_vector(v1);
31     message("v2");
32     disp_vector(v2);
33
34     v2 = std::move(v1);
35
36     return 0;
37 }
38
```

v1  
vector size: 5  
[one][two][three][four][five]  
v2  
vector size: 5  
[six][seven][eight][nine][ten]

Quiero mover v1 a v2 y para eso v2 es igual al move estándar v1, lo que hace esto es utilizar esta función plantilla de move que en realidad es solo un conversor de tipo que únicamente dice, se puede mover esto, solo funciona si el valor se puede convertir en un rvalue, es el caso de v1.

El otro lado del signo sigue igual, a donde se va a mover tiene que admitir una copia de movimiento en este caso v2 lo hace.



Aquí se ve en el resultado que se movieron.

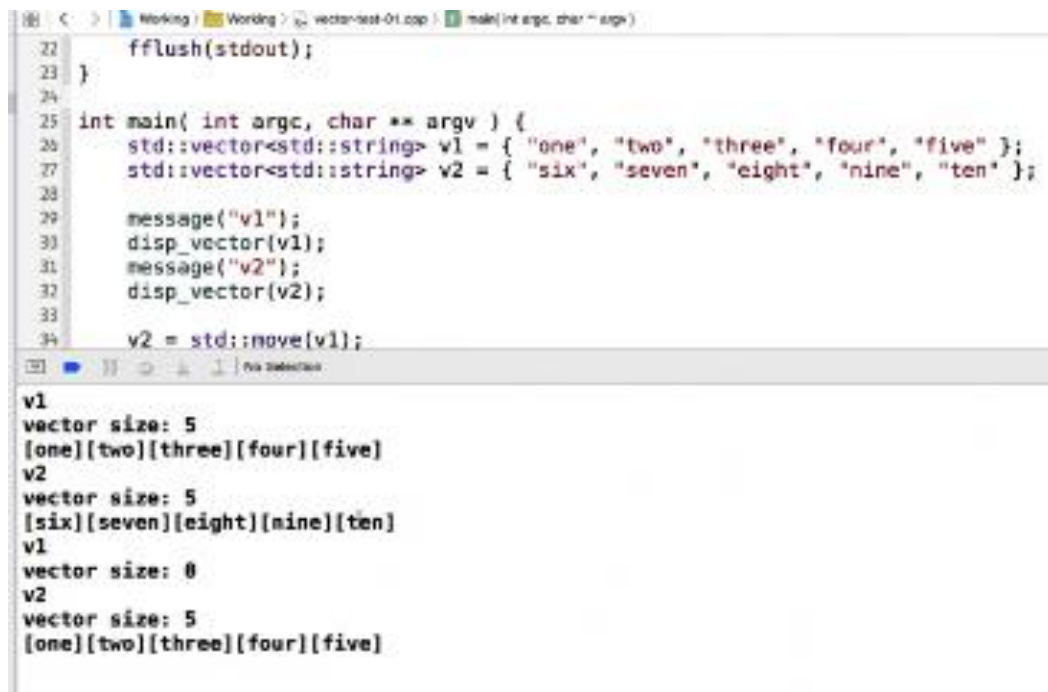
```
int main( int argc, char ** argv ) {
    std::vector<std::string> v1 = { "one", "two", "three", "four", "five" };
    std::vector<std::string> v2 = { "six", "seven", "eight", "nine", "ten" };

    message("v1");
    disp_vector(v1);
    message("v2");
    disp_vector(v2);

    v2 = std::move(v1);
    message("v1");
    disp_vector(v1);
    message("v2");
    disp_vector(v2);

    return 0;
}
```

Resultado v1 esta vacio. Se han copiado sin que tuviera que ver una copia



```
22     fflush(stdout);
23 }
24
25 int main( int argc, char ** argv ) {
26     std::vector<std::string> v1 = { "one", "two", "three", "four", "five" };
27     std::vector<std::string> v2 = { "six", "seven", "eight", "nine", "ten" };
28
29     message("v1");
30     disp_vector(v1);
31     message("v2");
32     disp_vector(v2);
33
34     v2 = std::move(v1);
```

Output:

```
v1
vector size: 5
[one][two][three][four][five]
v2
vector size: 5
[six][seven][eight][nine][ten]
v1
vector size: 0
v2
vector size: 5
[one][two][three][four][five]
```

Aquí se cambia el código y lo que hace es que se copiarían los valores.

```

24
25 int main( int argc, char ** argv ) {
26     std::vector<std::string> v1 = { "one", "two", "three", "four", "five" };
27     std::vector<std::string> v2 = { "six", "seven", "eight", "nine", "ten" };
28
29     message("v1");
30     disp_vector(v1);
31     message("v2");
32     disp_vector(v2);
33
34     v2 = v1;
35     message("v1");
36     disp_vector(v1);
37     message("v2");
38     disp_vector(v2);
39
40     return 0;
41 }
42

```

Y el resultado es diferente. V1 se copió en V2.

```

24
25 int main( int argc, char ** argv ) {
26     std::vector<std::string> v1 = { "one", "two", "three", "four", "five" };
27     std::vector<std::string> v2 = { "six", "seven", "eight", "nine", "ten" };
28
29     message("v1");
30     disp_vector(v1);
31     message("v2");
32     disp_vector(v2);
33
34     v2 = v1;
35     message("v1");
36     disp_vector(v1);
37     message("v2");
38     disp_vector(v2);
39
40     return 0;
41 }
42

```

vector size: 5  
[six][seven][eight][nine][ten]  
v1  
vector size: 5  
[one][two][three][four][five]  
v2  
vector size: 5  
[one][two][three][four][five]

Este move estándar también se puede utilizar para crear una función de intercambio .

Va ser una plantilla , voy a teclearlo en vez de usar aquí un atajo Vamos a usar nuestra función "message" para mostrar que estamos en esta función de aquí. Tecleamos "swap" y después hacemos el "move". Creamos un objeto

temporal y movemos el contenido de "a" al objeto temporal. Después movemos "b" a "a" y volvemos a mover el valor temporal a "b". De esta forma el objeto temporal se destruye. Por supuesto, después de quedar fuera de ámbito. Creamos un objeto temporal y movemos el contenido de "a" al objeto temporal, después movemos b al a y volvemos a mover el valor temporal a "b", de esta forma

```
24
25 template <typename T>
26 void swap(T & a, T & b) {
27     message("swap()");
28     T tmp(std::move(a));
29     a = std::move(b);
30     b = std::move(tmp);
31 }
32
33 int main( int argc, char ** argv ) {
34     std::vector<std::string> v1 = { "one", "two", "three", "f
35     std::vector<std::string> v2 = { "six", "seven", "eight",
36
37     message("v1");
38     disp_vector(v1);
39     message("v2");
40     disp_vector(v2);
41
42     v2 = std::move(v1);
43
vector size: 5
[six][seven][eight][nine][ten]
v1
vector size: 0
v2
vector size: 5
[one][two][three][four][five]
```

Puedes ver que aquí pone "swap". Es así porque en realidad está llamando a mi función "swap". Hay una función "swap" en el espacio estándar de nombres que viene con la biblioteca estándar.

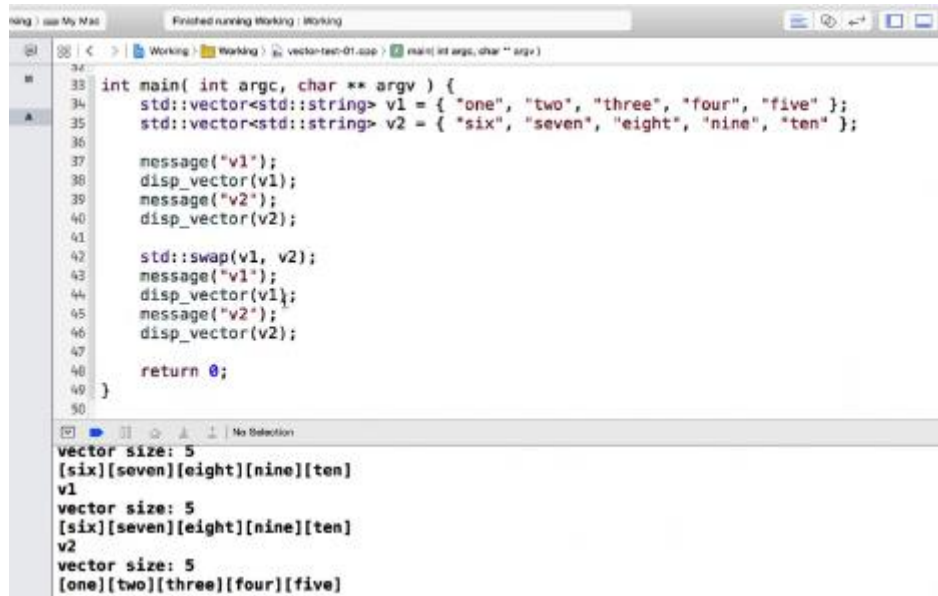
```
31 }
32
33 int main( int argc, char ** argv ) {
34     std::vector<std::string> v1 = { "one", "two", "three", "four",
35     std::vector<std::string> v2 = { "six", "seven", "eight", "nine"
36
37     message("v1");
38     disp_vector(v1);
39     message("v2");
40     disp_vector(v2);
41
42     ::swap(v1, v2);
43     message("v1");
44     disp_vector(v1);
45     message("v2");
46     disp_vector(v2);
47
48     return 0;
}

swap()
v1
vector size: 5
[six][seven][eight][nine][ten]
v2
vector size: 5
[one][two][three][four][five]
```

Hay una función "swap" en el espacio estándar de nombres que viene con la biblioteca estándar. Si aquí pongo estándar y compilo y ejecuto verás que obtenemos el mismo resultado Pero aquí no vemos la palabra "swap" porque no estamos llamando a nuestra función "swap" en el espacio principal de "main". En su lugar estamos llamando a la función "swap" estándar, en el espacio estándar.

Sólo quería enseñarte como se hace con el "move". Que fácil es hacer una función incluso con una función plantilla y que no se utiliza ninguna operación de copia. Se hace todo con la semántica de movimiento. Las plantillas estándar de "move" y "swap" son unas herramientas fundamentales para usar con la

semántica de movimiento.



```
33 int main( int argc, char ** argv ) {
34     std::vector<std::string> v1 = { "one", "two", "three", "four", "five" };
35     std::vector<std::string> v2 = { "six", "seven", "eight", "nine", "ten" };
36
37     message("v1");
38     disp_vector(v1);
39     message("v2");
40     disp_vector(v2);
41
42     std::swap(v1, v2);
43     message("v1");
44     disp_vector(v1);
45     message("v2");
46     disp_vector(v2);
47
48     return 0;
49 }
50
```

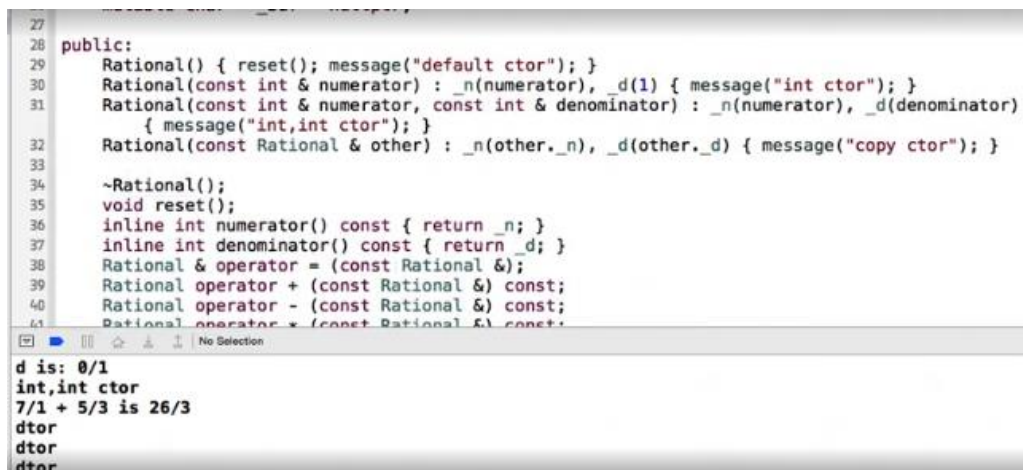
vector size: 5  
[six][seven][eight][nine][ten]  
v1  
vector size: 5  
[six][seven][eight][nine][ten]  
v2  
vector size: 5  
[one][two][three][four][five]

## ¿COMO CREAR UN CONSTRUCTOR DE MOVIMIENTO

Para poder aprovechar las ventajas de la semántica del movimiento en una de tus propias clases necesitas crear un constructor de movimiento

RATIONAL01.CPP clase muy sencilla para trabajar con números racionales

Esta clase incluye instrucciones "print" muy útiles, para mostrar donde se llama a su constructores y destructores.



```
27
28 public:
29     Rational() { reset(); message("default ctor"); }
30     Rational(const int & numerator) : _n(numerator), _d(1) { message("int ctor"); }
31     Rational(const int & numerator, const int & denominator) : _n(numerator), _d(denominator)
32     { message("int,int ctor"); }
33     Rational(const Rational & other) : _n(other._n), _d(other._d) { message("copy ctor"); }
34
35     ~Rational();
36     void reset();
37     inline int numerator() const { return _n; }
38     inline int denominator() const { return _d; }
39     Rational & operator = (const Rational &);
40     Rational operator + (const Rational &) const;
41     Rational operator - (const Rational &) const;
42     Rational operator * (const Rational &) const;
43
```

d is: 0/1  
int,int ctor  
7/1 + 5/3 is 26/3  
dctor  
dctor  
dctor

Aquí puedes ver el "doble ampersand". Eso quiere decir que el argumento va a ser un "rvalue". Esto es lo que hace que sea un constructor de movimiento. También puedes ver que aquí pongo "no except", así. Esto en realidad es una macro.

Normalmente sólo sería esta palabra clave: "no except", así. Por desgracia, Microsoft visual C++ no admite todavía esa palabra clave. Lo hará en la siguiente versión.

```
5 #include <memory>
6
7 // MSC standards compliance issues
8 #ifdef _MSC_VER
9 #include "bw_msposix.h"
10 #else
11 #define _NOEXCEPT noexcept
12 #endif
13
14 void message(const char * s) {
15     puts(s);
16     fflush(stdout);
17 }
```

Estoy usando la versión 18 que es la última en estos momentos. Necesito decidir aquí esta macro. Eso hará que funcione con el mismo código tanto en Microsoft, como en otros sistemas operativos.

Aquí puedes ver que tenemos el "no except". Este modificador "no except", en realidad se necesita en el reconstructor de movimiento. Evita las excepciones que se generan al dejar el objeto en un estado desconocido. Tu compilador podría ignorar en realidad el constructor de movimiento si no se declara con "no except". Así que necesitamos tenerlo.

```
18 public:
19     Rational() { reset(); message("default ctor"); }
20     Rational(const int & numerator) : _n(numerator), _d(1) { message("int ctor"); }
21     Rational(const int & numerator, const int & denominator) : _n(numerator), _d(denominator)
22     { message("int,int ctor"); }
23     Rational(const Rational & other) : _n(other._n), _d(other._d) { message("copy ctor"); }
24     Rational(Rational &&) _NOEXCEPT;
25     ~Rational();
26     void reset();
27     inline int numerator() const { return _n; }
28     inline int denominator() const { return _d; }
29     Rational & operator = (const Rational &);
30     Rational operator + (const Rational &) const;
31     Rational operator - (const Rational &) const;
32     Rational operator * (const Rational &) const;
33     Rational operator / (const Rational &) const;
```



```
};

Rational::Rational(Rational && rhs) _NOEXCEPT {
    _n = std::move(rhs._n);
    _d = std::move(rhs._d);
    rhs.reset();
    message("move ctor");
}

Rational::~Rational() {
    message("dtor");
    reset();
}
```

Puedes ver que todavía no se ha llamado al nuestro constructor de movimiento. Necesitamos hacer algo realmente sobre un "rvalue" para que se haga la llamada. Así que voy a tomar este "d", y voy a llamar al "move" estándar pasándole "c".

Eso debería ser nuestro constructor de movimiento. Ahora puedes ver que se llama a nuestro constructor de movimiento y que "c" se deja vacío, "cero partido por uno".

Obviamente no puedo hacerlo "cero partido por cero" porque no sería un número válido, ¿no?. Esa es la razón por la que tengo mi pequeño "reset" aquí, que lo deja en un estado lógico. Un denominador "cero" no es válido porque no puedes dividir por cero. Así que se puede llamar automáticamente al constructor de movimiento en algunas circunstancias.

```
int main( int argc, char ** argv ) {

    Rational a = 7;    // 7/1
    Rational b(5, 3);  // 5/3
    Rational c = b;    // copy ctor
    Rational d = std::move(c); // move ctor

    printf("a is: %s\n", a.c_str());
    printf("b is: %s\n", b.c_str());
    printf("c is: %s\n", c.c_str());
    printf("d is: %s\n", d.c_str());

    printf("%s + %s is %s\n", a.c_str(), b.c_str(), Rational(
    return 0;
}
```

Por ejemplo si tengo una función que toma un objeto racional y devuelve ese objeto racional. Después voy aquí abajo y pongo "f" y entre paréntesis "c". Compilo y ejecuto. Puedes ver que tiene "d". Se está utilizando un constructor

de copia y uno de movimiento. Así que cuando se llama a "f" se utiliza el constructor de copia para crear ese nuevo objeto. Pero porque el objeto que se devuelve es un objeto temporal sin ningún nombre real. Así que cuando la función no devuelve aquí en esta asignación ese es un objeto temporal sin nombre. Por lo que es automáticamente un "rvalue". Así que se llama al constructor de movimiento. Tenemos un constructor de movimiento "d" que en realidad es una copia de "c". Porque se ha llamado el constructor de copia cuando se llama a la función. Y así "c" no se deja vacío. Pero lo que ha pasado es que hemos guardado una copia. Ahora hay una copia menos de las que habría habido si no hubiésemos implementado nuestro constructor de movimiento.

```
Rational f(Rational o) {
    return o;
}

int main( int argc, char ** argv ) {
    Rational a = 7;    // 7/1
    Rational b(5, 3);  // 5/3
    Rational c = b;    // copy ctor
    Rational d = f(c);

    printf("a is: %s\n", a.c_str());
    printf("b is: %s\n", b.c_str());
    printf("c is: %s\n", c.c_str());
    printf("d is: %s\n", d.c_str());
}
```

re ctor  
is: 7/1  
is: 5/3  
is: 0/1  
is: 5/3  
int ctor

Por eso, tener un constructor de movimiento es un factor importante al admitir la semántica de movimiento.

¿Cómo crear un operador de asignación de movimiento?

Junto con el constructor de movimiento, el operador de asignación de movimiento también es una parte importante de la compatibilidad con la semántica de movimiento en tu código.

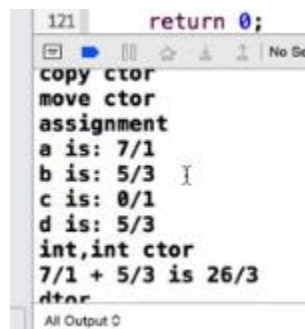


Este es nuestro operador de asignación

```
66 Rational & Rational::operator = ( const Rational & rhs ) {  
67     if( this != &rhs ) {  
68         _n = rhs.numerator();  
69         _d = rhs.denominator();  
70     }  
71     return *this;  
72 }  
73  
74 Rational Rational::operator + ( const Rational & rhs ) const {
```

```
int main( int argc, char ** argv ) {  
  
    Rational a = 7;      // 7/1  
    Rational b(5, 3);    // 5/3  
    Rational c = b;      // copy ctor  
    Rational d = std::move(c);  
  
    d = b;  
  
    printf("a is: %s\n", a.c_str());  
    printf("b is: %s\n", b.c_str());  
    printf("c is: %s\n", c.c_str());
```

Aquí `b=d`;



```
121 return 0;  
copy ctor  
move ctor  
assignment  
a is: 7/1  
b is: 5/3  
c is: 0/1  
d is: 5/3  
int,int ctor  
7/1 + 5/3 is 26/3  
dctor  
All Output
```

Cambiando nuestra asignación.

Si ahora cambiamos esta asignación para poner "move" de "b" y compilamos y ejecutamos, verás que seguimos llamando a ese operador de asignación porque no tenemos ningún operador de asignación de movimiento. Por tanto, aunque la plantilla de movimiento hace que "b" sea un "rvalue" como no admitimos ese operador de asignación de movimiento seguimos llamando al operador normal de asignación.

```

105 int main( int argc, char ** argv ) {
106
107     Rational a = 7;      // 7/1
108     Rational b(5, 3);    // 5/3
109     Rational c = b;      // copy ctor
110     Rational d = std::move(c);
111
112     d = std::move(b);
113
114     printf("a is: %s\n", a.c_str());
115     printf("b is: %s\n", b.c_str());
116     printf("c is: %s\n", c.c_str());
117     printf("d is: %s\n", d.c_str());
118
119     printf("%s + %s is %s\n", a.c_str(), b.c_str(),
120           std::move(a).c_str());
121     return 0;
}

```

int ctor  
 int,int ctor  
 copy ctor  
 move ctor  
 assignment  
 a is: 7/1  
 b is: 5/3  
 c is: 0/1  
 d is: 5/3

Creamos un operador de asignación de movimiento

Se coloca un segundo "ambersand" y el "no except". Se vuelve a usar esta macro para "no except" para que sea compatible con todas las versiones de Microsoft Visual C++. La actual que no tiene la palabra clave "no except" y los otros compiladores actuales como este.

```

3 Rational(Rational &&) _NOEXCEPT;
4 ~Rational();
5 void reset();
6 inline int numerator() const { return _n; }
7 inline int denominator() const { return _d; }
8 Rational & operator = (const Rational &);
9 Rational & operator = (const Rational &&) _NOEXCEPT;
10 Rational operator + (const Rational &) const;
11 Rational operator - (const Rational &) const;
12 Rational operator * (const Rational &) const;
13 Rational operator / (const Rational &) const;
14 operator std::string () const;
15 std::string string() const;
16 const char * c_str() const;
17 };

```

Este es nuestro operador de asignación de movimiento

```

Rational & Rational::operator = ( Rational && rhs ) _NOEXCEPT {
    message("move assignment");
    if( this != &rhs ) {
        _n = std::move(rhs.numerator());
        _d = std::move(rhs.denominator());
        rhs.reset();
    }
    return *this;
}

```

```

Rational a = 7;    // 7/1
Rational b(5, 3);  // 5/3
Rational c = b;    // copy ctor
Rational d = std::move(c);

d = std::move(b);

printf("a is: %s\n", a.c_str());
printf("b is: %s\n", b.c_str());
printf("c is: %s\n", c.c_str());
printf("d is: %s\n", d.c_str());

printf("%s + %s is %s\n", a.c_str(),

```

El operador de asignación es una parte fundamental para mantener la compatibilidad con la semántica de movimiento en tu clase. Es muy fácil de implementar. Siempre deberías mantener la compatibilidad cuando ofreces compatibilidad con un constructor de movimiento.