C++ Course 14 : Move semantics.

By Oleksiy Grechnyev

Copy operations (C++ 98), Move operations (C++ 11+)

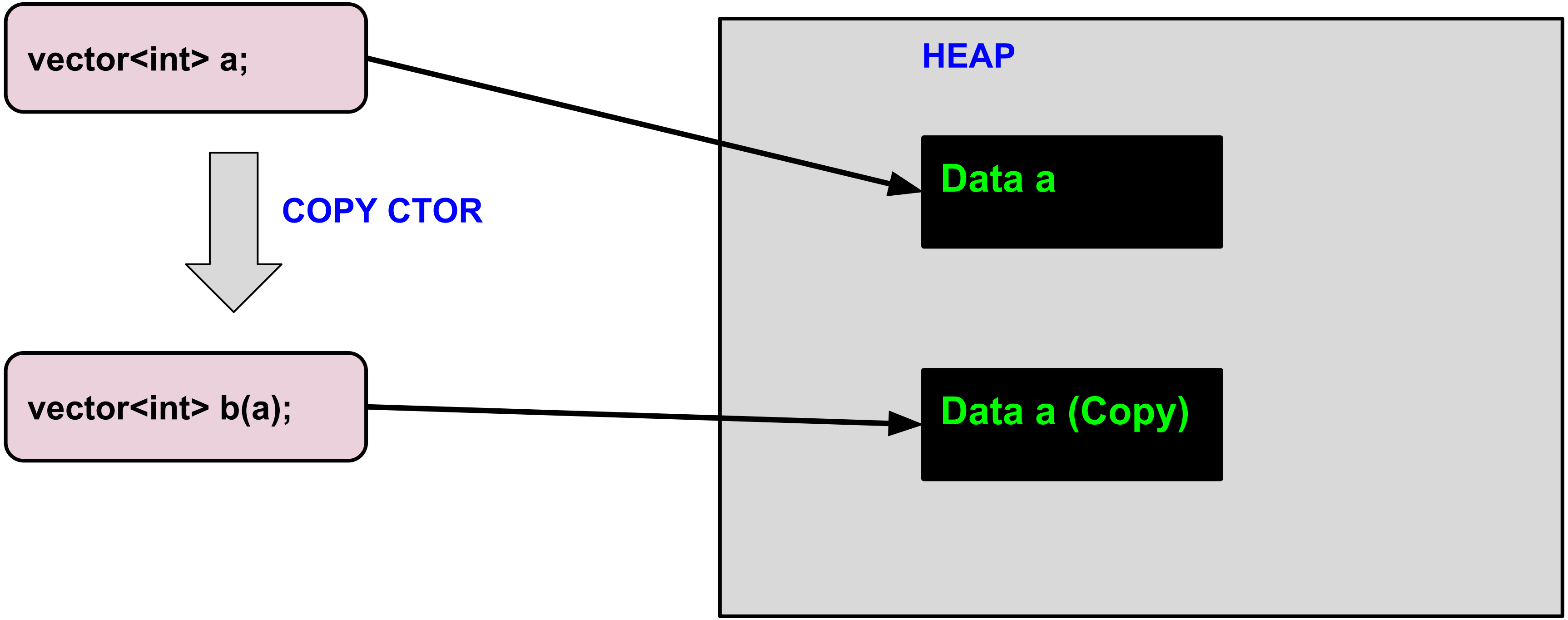
```
String a("Mickey mouse");
String b(a);        // Copy Ctor
String c;
String c = a;       // Copy Assignment

b = string("Cinderella");    // Copy assignment in C++ 98, Move assignment in C++ 11+
c = move(b);                 // Move assignment in C++ 11+, b is now empty string !
String d(move(a));           // Move ctor in C++ 11+, a is now empty string !
```

Copy is easy to understand: we clone our object.

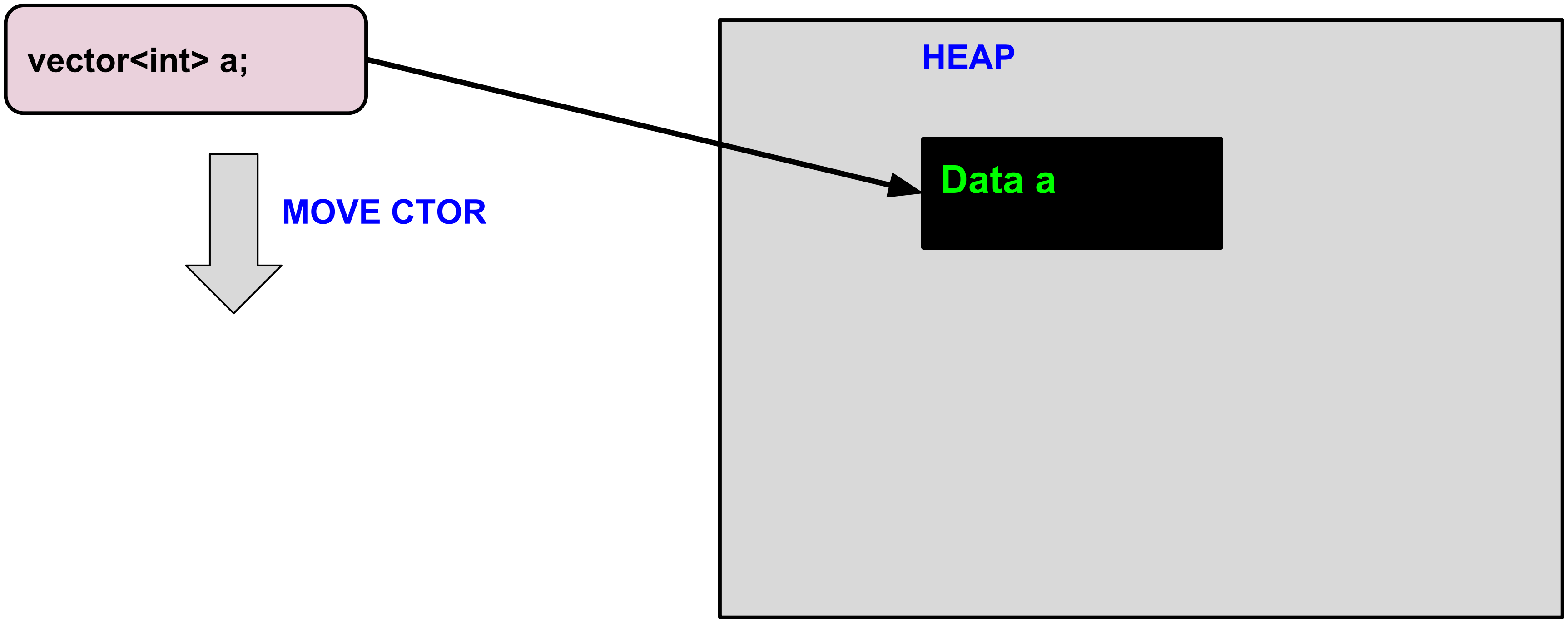
But what is MOVE ?

Copy constructor : std::vector copies data in the heap

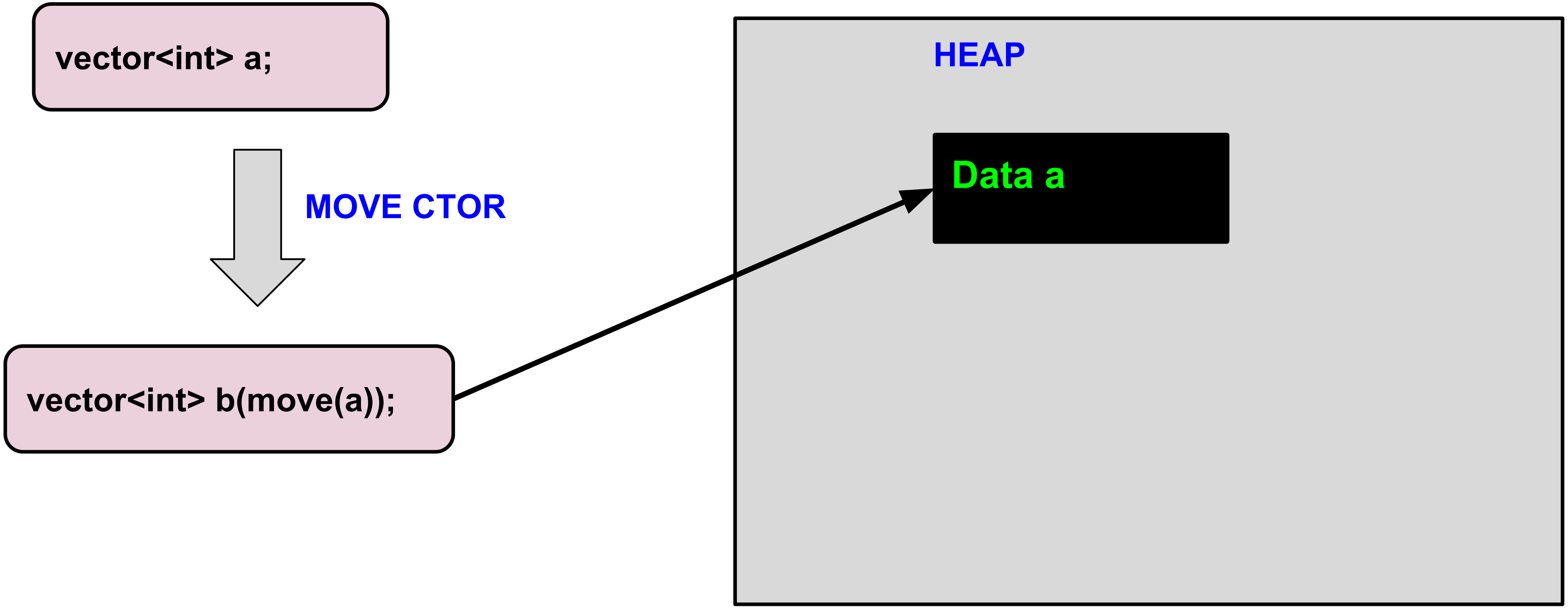


A new vector **b** is created. The data of vector **a** is COPIED in the heap to **b**.
Copying class objects is often EXPENSIVE.

Move constructor : data is moved to another vector object



Move constructor : data is moved to another vector object

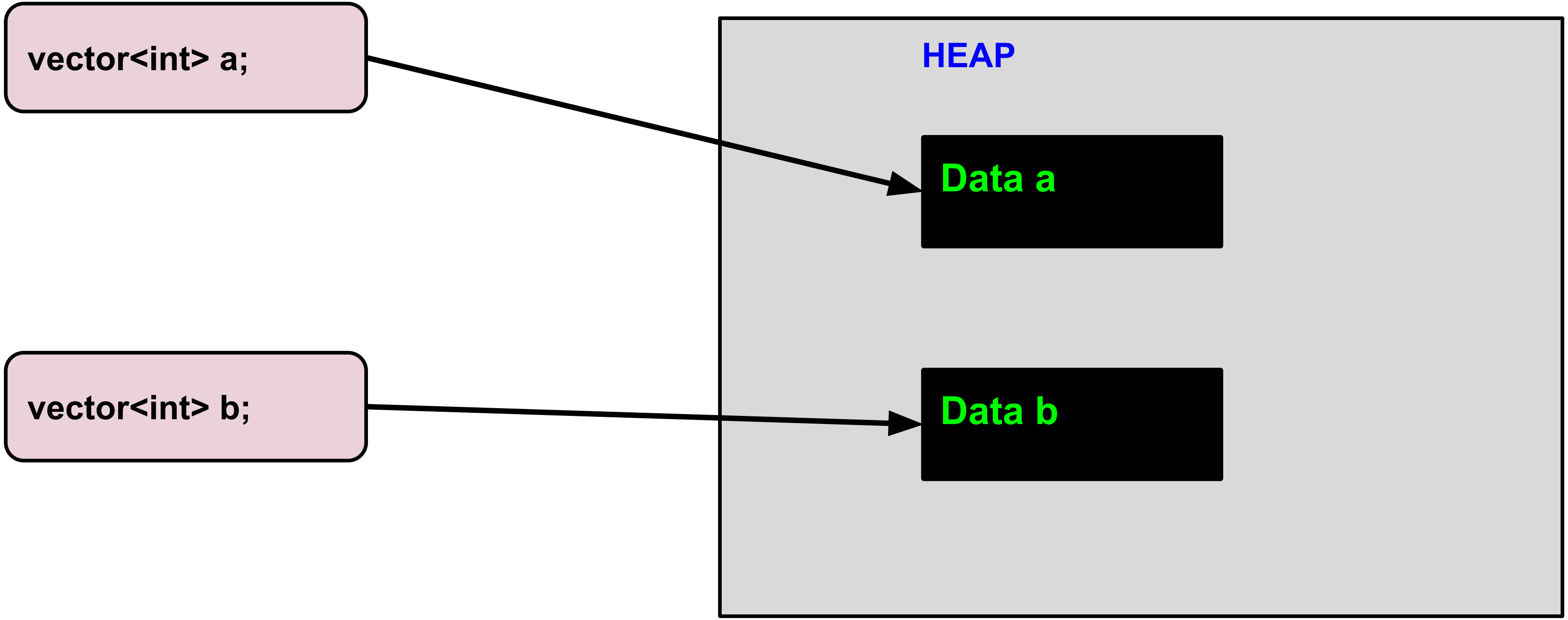


The data is moved from object **a** to the new vector **b**.

Vector **a** is now empty (**size == 0, capacity == 0**), but NOT destroyed !

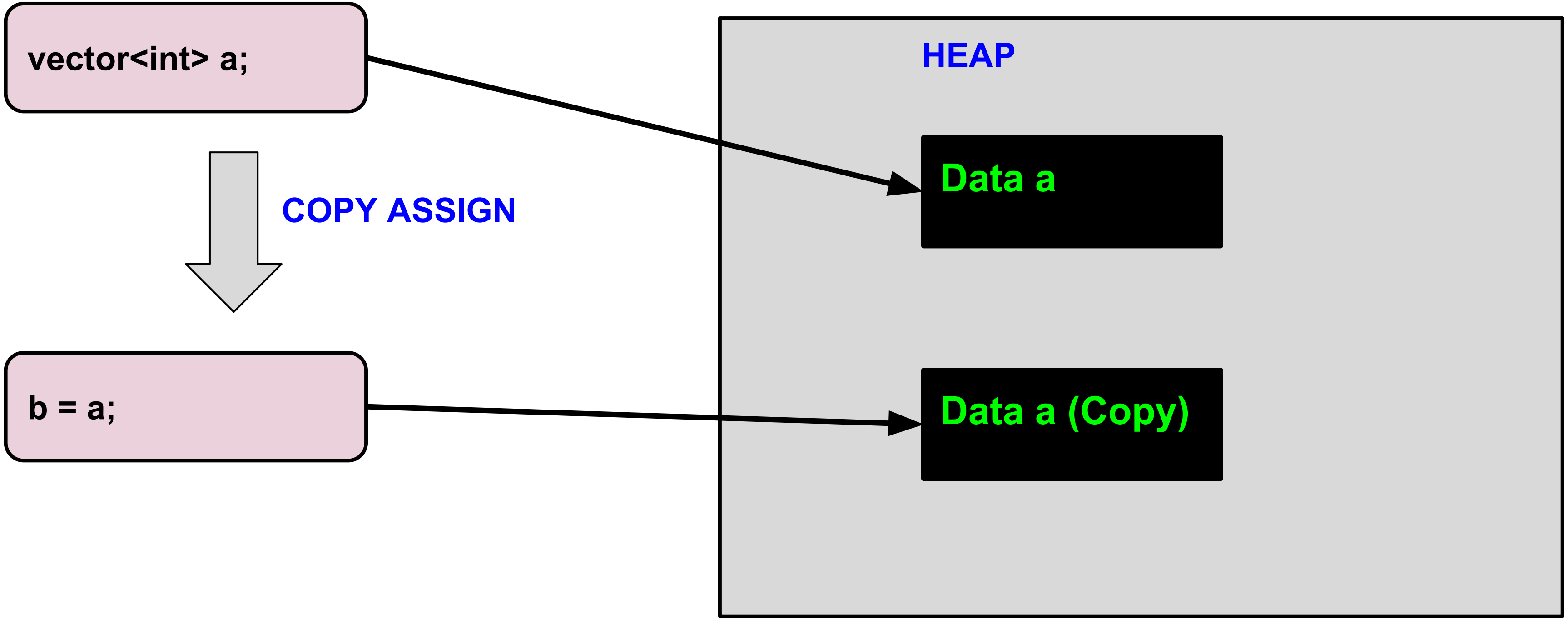
Data is moved, not objects !

Before Copy or Move assignment



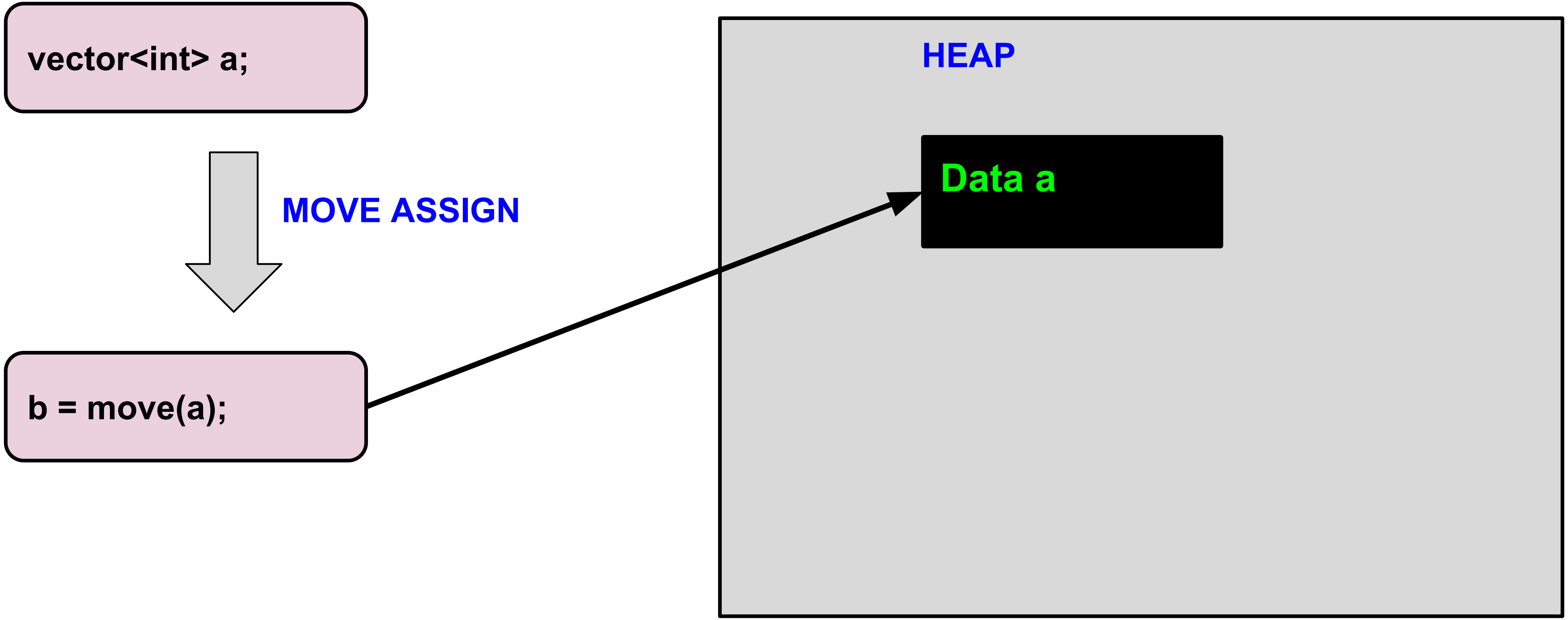
Vectors **a** and **b** both exist and keep their own heap data.

Copy assignment



Data **b** is lost (and destroyed on the heap), Data **a** is copied in the heap to vector **b**.

Move assignment



Data **b** is lost (and destroyed on the heap), Data **a** is moved to vector **b**.

Vector **a** is now empty (**size == 0, capacity == 0**), but NOT destroyed !

Data is moved, not objects !

In a move operation the victim is NOT destroyed !!!

Move is a robbery, not murder ! MOVE -- это ограбление, а не убийство!

```
{
    string a("Sword");
    string b(move(a));
    // Now b == "Sword", a == "" (Empty, Not destroyed !)
    cout << "a = " << a << endl;  // OK, Empty
    cout << "b = " << b << endl;  // OK, Sword
    a = "Spear";    // a can be assigned again
}  // Now a, b run out of scope and are destroyed !
```

Move victim is never destroyed only robbed of data (becomes empty/null).

Data is moved, not objects !

When is MOVE useful? Когда MOVE полезен?

1. Classes with data in heap : **vector**, **string**, most other containers.
2. Classes which cannot be copied : **unique_ptr**, **ifstream**, **thread**, **future**.

Rvalue vs lvalue

Move operations are implemented with the help of RVALUE REFERENCES.

The terminology LVALUE, RVALUE comes from the early days of C.
The assignment statement:
LVALUE = RVALUE;
LVALUE (left value) can be assigned. Variable, array element **arr[i]**, etc.
RVALUE (right value) cannot be assigned. Temporary: literal, expression result etc.

However, in modern sense of the word:
const a = 17;
a is an LVALUE (every variable is LVALUE), but cannot be assigned!

Confusing!

We need a better definition of LVALUE, RVALUE !

Rvalue vs lvalue: Better definition

LVALUE is something you can take the address of with the **&** operator:

string s("Mouse");

string * pS1 = & s; // OK, **s** in an LVALUE

string * pS2 = & string("Rat"); // ERROR, **string("Rat")** in an RVALUE

LVALUE examples:

1. Variable, or constant variable: **int a = 17;**
2. Function/method return by LVALUE ref : **arr.at(j)**
3. Operator return by LVALUE ref : **arr[i], *ptr**

RVALUE examples:

1. Literal : **17, "Mouse"**
2. Temporary object : **string("Rat")**
3. Expression result: **2*a + b**
4. Function return by value : **cos(alpha)**

LVALUE vs RVALUE references

C++ 98 : Non-const (LVALUE) reference binds to LVALUE only.

string s("Guinea Pig");
string & lrS1 = s; // OK, ref to variable s
string & lrS2 = string("Marmot"); // ERROR !!!

C++ 98 : Const (LVALUE) reference binds to LVALUE, but also RVALUE.

string s("Guinea Pig");
const string & clrS1 = s; // OK, const ref to variable s
const string & clrS2 = string("Marmot"); // OK, const ref to a temporary

C++ 11+ : RVALUE reference binds to RVALUE (temporary object) only.

string s("Guinea Pig");
string && rrS1 = s; // ERROR rvalue ref to lvalue !
string && rrS2 = string("Marmot"); // OK, rvalue ref to a temporary

Note: The lifetime of TEMPORARY is extended until **}** for **clrS2** and **rrS2** !

Copy and move constructors and assignment operators

```
Tjej(const Tjej & rhs) noexcept : name(rhs.name){}               // Copy Ctor

Tjej(Tjej && rhs) noexcept : name(std::move(rhs.name)){}         // Move Ctor

Tjej & operator= (const Tjej & rhs) noexcept{                 // Copy assignment
     if (this != &rhs)     // Check for self-assignment
          name = rhs.name;
     return *this;
}

Tjej & operator= (Tjej && rhs) noexcept{                     // Move assignment
     if (this != &rhs)     // Check for self-assignment
          name = std::move(rhs.name);
     return *this;
}
```

Tjej && is an *RVALUE* reference.**const Tjej &** is a const *LVALUE* reference.
The actual MOVE operation is implemented in move Ctor and Assignment.
move() does not move anything, it selects the *RVALUE* overload.

Copy (LVALUEs) and Move (RVALUEs)

How are COPY or MOVE overloads selected ?

```
String a("Mickey mouse");
String b(a);        // Copy Ctor, a is a LVALUE
String c;
String c = a;       // Copy Assignment, a is a LVALUE

b = string("Cinderella");    // Move assignment, string("Cinderella")  is a RVALUE
c = static_cast<string &&>(b);           // Move assignment, cast to RVALUE !
c = move(b);              // The same
String d(static_cast<string &&>(a));     // Move ctor, cast to RVALUE !
String d(move(a));      // The same
```

move() does not move anything, it casts to *RVALUE* !

Copy Ctor/assignment is selected for LVALUEs.

Move Ctor/assignment is selected for RVALUEs.

Source (factory) : creates a new object

Return by value, no MOVE !

```
Tjej source1(const string & s){
    Tjej t(s);
    return t;
}
...
Tjej t1 = source1("Karen Koenig");
```

Return Value Optimization : No copy or move !

We can even make source within source:

```
Tjej source2(const string & s){
    return source1(s);
}
Tjej t2 = source2("Alice Elliot");
```

Still no copy or move, only 1 object created !

Is RVALUE reference VARIABLE an RVALUE? NO !!!

```
void doSomething(Tjej && rrT){       // rrS is an RVALUE parameter
    string && rrS = string("Maria Traydor");       // rrS is an RVALUE variable
    // rrT and rrS are RVALUE refs, but not RVALUES !
    Tjej t(rrT);    // Copy, NOT MOVE !
    string s(rrS);     // Copy, NOT MOVE !
    ...
}
```

All variables and parameters are LVALUES by definition !

If we want move, we must cast them to rvalue refs as usual :

```
void doSomething2(Tjej && rrT){
    string && rrS = string("Maria Traydor");
    Tjej t(move(rrT));    // MOVE !
    string s(move(rrS));     // MOVE !
    ...
}
```

Writing a sink (consumer)

A sink consumes an object by MOVE and lets it die.

```
void sink(Tjej && t0) {        // By rvalue ref
        Tjej t(move(t0));          // Move to a local var
        cout << "sink : " << t.getName() << endl;
}  // t dies here
```

A copy of a ref parameter? Better to pass by value !

```
void sink1(Tjej t) {        // By value (move), local that dies !
        cout << "sink1 : " << t.getName() << endl;
}  // t dies here
...
Tjej t3("Lucia");
sink1(move(t3));  // 1 move, 0 copy, Move ctor is selected for the parameter t
```

Writing a sink (consumer)

A copy of a ref parameter? Better to pass by value !

```
void sink1(Tjej t) {        // By value (move), local that dies !
        cout << "sink1 : " << t.getName() << endl;
}  // t dies here
...
Tjej t3("Lucia");
sink1(move(t3));  // 1 move, 0 copy, Move ctor is selected for the parameter t
```

Chain sinks :

```
void sink2(Tjej t) {  // By value (move), local that dies !
    sink1(move(t));    // The correct way, move to die !
} // t dies here
...
Tjej t4("Margarete Gertrude Zelle");
sink2(move(t4));  //  2 moves, 0 copy, Move ctor is selected for the parameter t
```

In Move Ctor/assign use move() for :

Class fields:

```
Tjej(Tjej && rhs) noexcept : name(std::move(rhs.name)){}     // Move Ctor
Tjej & operator= (Tjej && rhs) noexcept{              // Move assignment
        if (this != &rhs)    // Check for self-assignment
                name= std::move(rhs.name);
        return *this;
}
```

Parent class:

```
class LilTjej : public Tjej{
        ...
        // Move Ctor
        LilTjej(LilTjej && rhs) noexcept : Tjej(std::move(rhs)){}
};
```

Remember ! Move ctor must be **noexcept** to work in **std::vector** !

Move summary

1. MOVE operations are only useful for classes with heap memory (containers) and the ones which cannot be copied (**unique_ptr**, **thread**, **ostream**).

2. C++ does not move anything, the move ctor/operator= does!

3. Move ctor/assign is selected for RVALUES, copy ctor/assign is selected for LVALUES.

4. **std::move()** does not move anything, it casts to RVALUE.

5. Sources return by value, sinks get parameter by value.

6. RVALUE reference is itself LVALUE, not RVALUE ! Use **std::move()** to move it!

7. The victim of MOVE is not destroyed, it becomes empty.

Universal references

Universal ref can bind to both LVALUES and RVALUES:

```
template <typename T>
void fun(T && t){ ...}    // Universal REF !

auto && t = ...;     // Universal REF !
```

Only **T &&** or **auto &&** is universal ref, everything else is RVALUE ref:

```
template <typename T>
void fun(vector<T> && t){ ...}    // Rvalue REF !

template <typename T>
void fun(T::type && t){ ...}    // Rvalue REF !

void fun(string && t){ ...}    // Rvalue REF !
```

How does it work ???

Normally *REFERENCE to REFERENCE* is forbidden in C++:

int & & a = i;   // ERROR !

But in template type deduction and **auto** the rules are different.

Reference collapse rules :

& &   -> &

&& &   -> &

& &&   -> &

&& && -> &&

In reality THERE IS NO UNIVERSAL REFERENCES.

They are special RVALUE references.

It is all about type deduction + reference collapsing.

How does it work ???

Template with a universal ref :

template <typename T>

void fun(T && t){ ...}   // Universal REF !

For LVALUE:

string s("Mickey Mouse");

fun(s);

T is deduced as **string &**, and we get : **fun(string & && s);**

Which means **fun(string & s);**

For RVALUE:

string s("Mickey Mouse");

fun(move(s));

T is deduced as **string** , and we get : **fun(string && s);**

std::move possible implementation

In C++ 11:

```
template<typename T>
typename remove_reference<T>::type && move(T  && param){
        using ReturnType = typename remove_reference<T>::type &&;
        return static_cast<ReturnType>(param);  // Cast to RVALUE ref
}
```

In C++ 14:

```
template<typename T>
decltype(auto) move(T  && param){
        using ReturnType = typename remove_reference<T>::type &&;
        return static_cast<ReturnType>(param);  // Cast to RVALUE ref
}
```

But how does it work ?

std::move for LVALUES

```
template<typename T>
typename remove_reference<T>::type && move(T  && param){
        using ReturnType = typename remove_reference<T>::type &&;
        return static_cast<ReturnType>(param);  // Cast to RVALUE ref
}
```

For string LVALUE: T = string & :

```
typename remove_reference<string &>::type && move(string & && param){
        using ReturnType = typename remove_reference<string &>::type &&;
        return static_cast<ReturnType>(param);  // Cast to RVALUE ref
}
```

or

```
string && move(string & param){
        using ReturnType = string &&;
        return static_cast<ReturnType>(param);  // Cast to RVALUE ref
}
```

std::move for RVALUES

```
template<typename T>
typename remove_reference<T>::type && move(T  && param){
        using ReturnType = typename remove_reference<T>::type &&;
        return static_cast<ReturnType>(param);  // Cast to RVALUE ref
}
```

For string RVALUE: T = string :

```
typename remove_reference<string>::type && move(string && param){
        using ReturnType = typename remove_reference<string>::type &&;
        return static_cast<ReturnType>(param);  // Cast to RVALUE ref
}
```

or

```
string && move(string && param){
        using ReturnType = string &&;
        return static_cast<ReturnType>(param);  // Cast to RVALUE ref
}
```

Perfect forwarding. Problem :

We want to write a template which adds element to a **vector** :

```
template<typename V, typename T>
void addToVec(V & v, const T & t){
    v.push_back(t);
}
```

Is it good ? Not exactly !

```
vector<Tjej> v;
v.reserve(10);
Tjej t1("Clair Lasbard");
addToVec(v, t1); // Add a LVALUE, 1 copy, OK

addToVec(v, Tjej("Nel Zelpher"));   // RVALUE is copied instead of MOVE !!!
```

Solution : Perfect forwarding + universal reference

We *perfect forward* the argument :

```
template<typename V, typename T>
void addToVec(V & v, T && t){
    v.push_back(forward<T>(t));
}
```

std::forward() always needs type argument : **forward<T>(...)**

Is it good ? YES !

```
vector<Tjej> v;
v.reserve(10);
Tjej t1("Clair Lasbard");
addToVec(v, t1); // Add a LVALUE, 1 copy, OK

addToVec(v, Tjej("Nel Zelpher"));   // Add a RVALUE, 1 move, OK
```

How does std::forward work ? Possible implementation.

```
template<typename T>
T && forward(typename remove_reference<T>::type & param){
    return static_cast<T &&>(param);
}
```

Type **T** is not deduced, but fixed by universal ref in e.g. **addToVec()**.

For **string** RVALUE: **T** **= string** : Similar to **move** :

```
string && forward(string & param){
    return static_cast<string  &&>(param);
}
```

For **string** LVALUE: **T** **= string &** :

```
string & forward(string & param){
    return static_cast<string  &>(param);
}
```

Perfect forwarding variadic version

We want to write a template which adds element to a **vector** using **emplace_back()**:

```
template <typename V, typename ... Args>
void addToVec2(V & v, Args && ... args){
	v.emplace_back(forward<Args>(args)...);
}
```

We perfect-forward all arguments to **emplace_back()**.

```
addToVec2(v, string("Maria Traydor"));      // Construct in-place
```

Thank you for your attention !

THE END

title

text