C++ Course 12 : Move semantics.

2020 by Oleksiy Grechnyev

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

Copy operations:

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

Move operations:

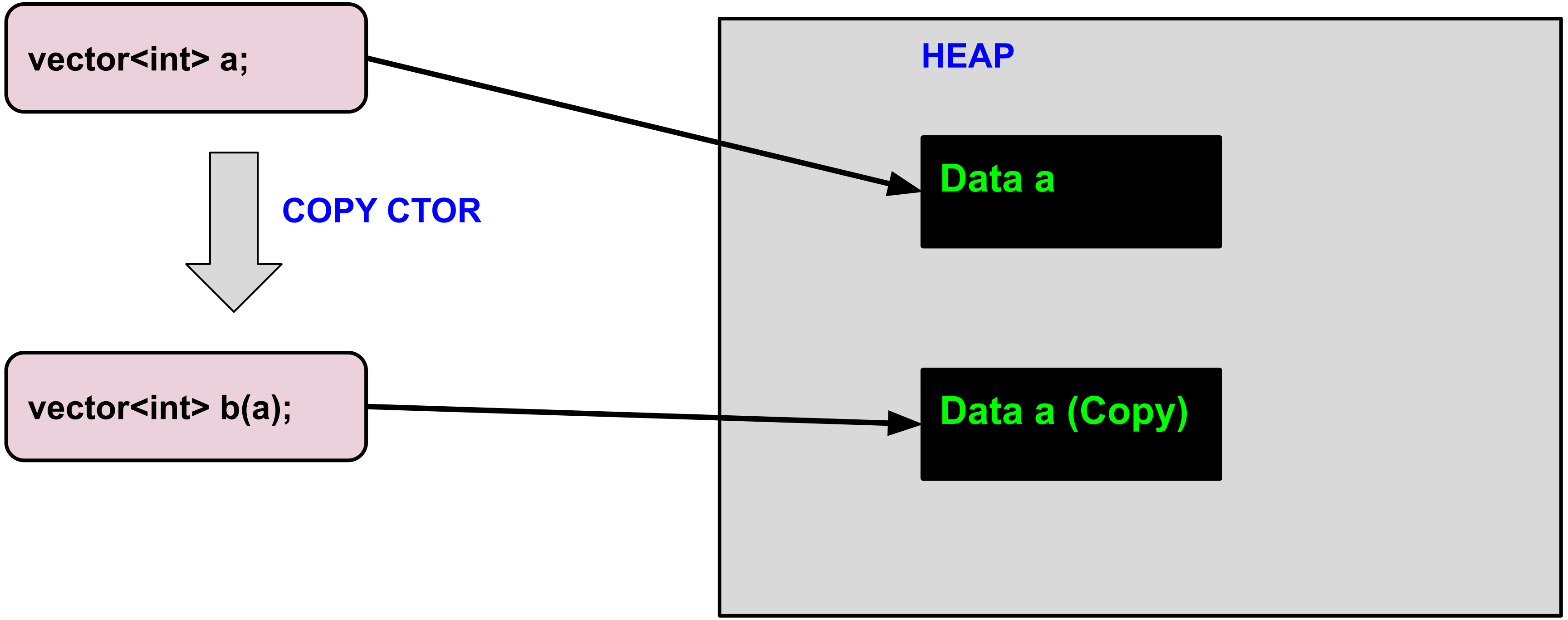
```
b = string("Cinderella");    // Temporary: Move assignment in C++ 11+ (copy in C++ 98)
c = move(b);               // Move assignment in C++ 11+, b is now empty string !
string d(move(a));      // Move ctor in C++ 11+, a is now empty string !
```

Copy is easy to understand: we clone our object.

But what is MOVE ?

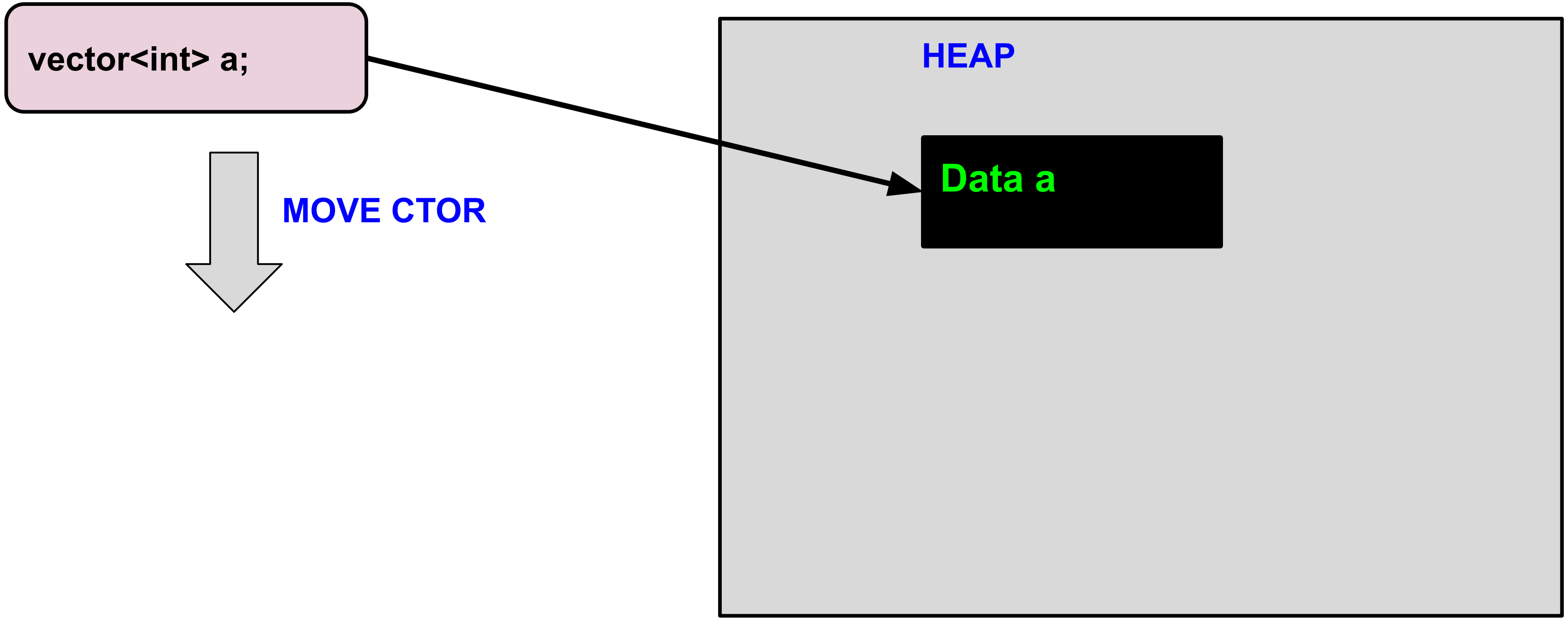
DATA is moved, NOT OBJECT !!!

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

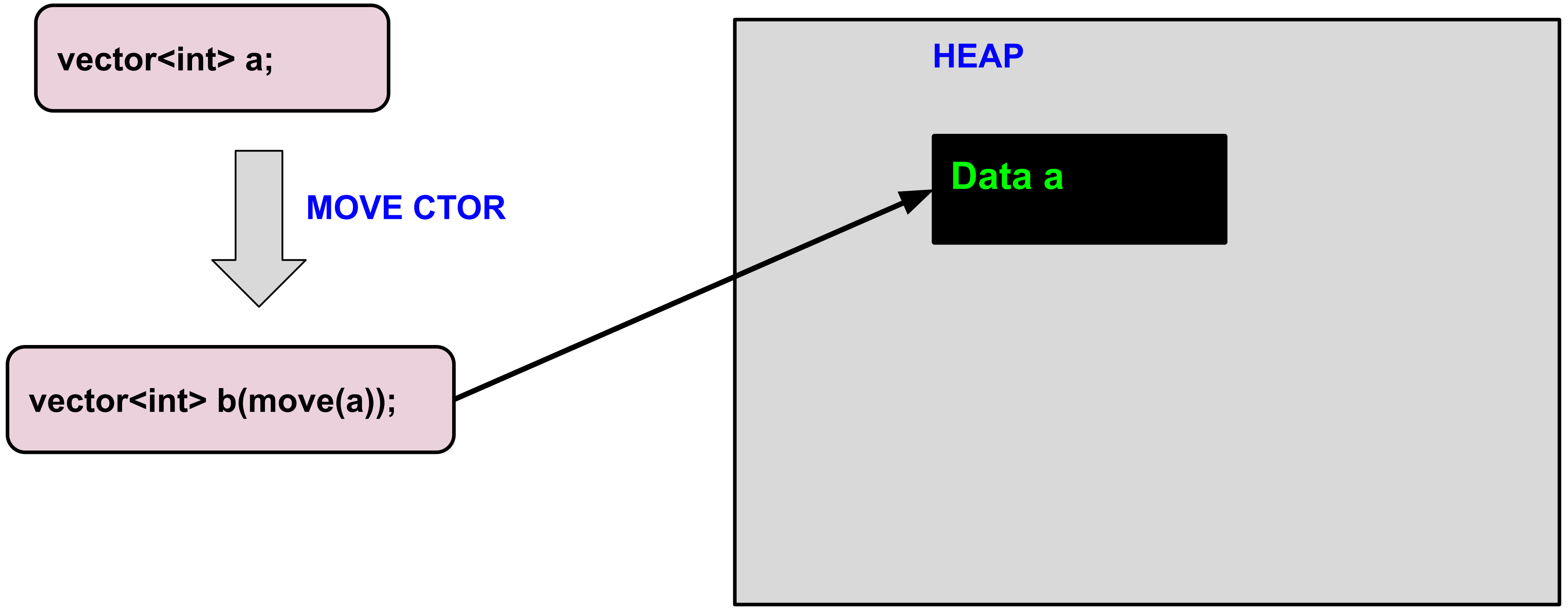


A new vector **b** is created. The data of vector **a** is COPIED in the heap to **b**.
Copying class objects is often EXPENSIVE.

Move constructor : data is moved to another vector object



Move constructor : data is moved to another vector object

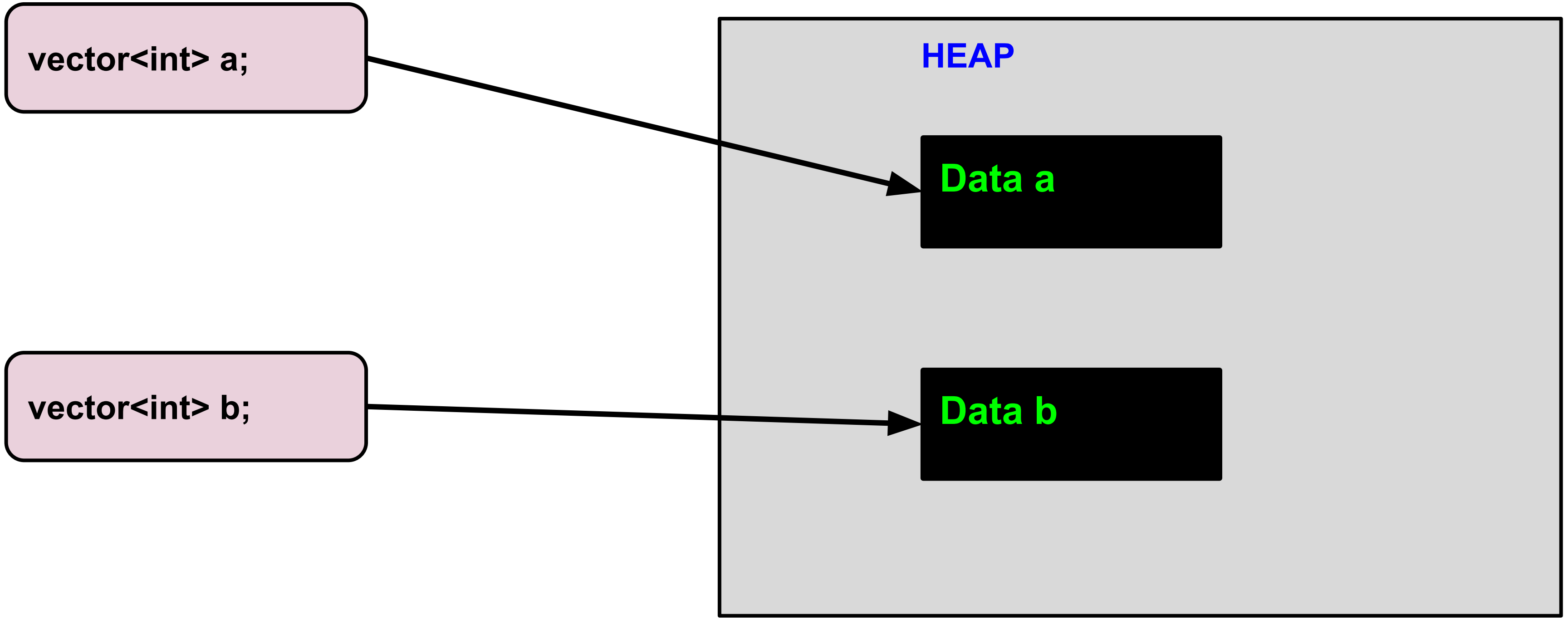


The data is moved from object **a** to the new vector **b**.

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

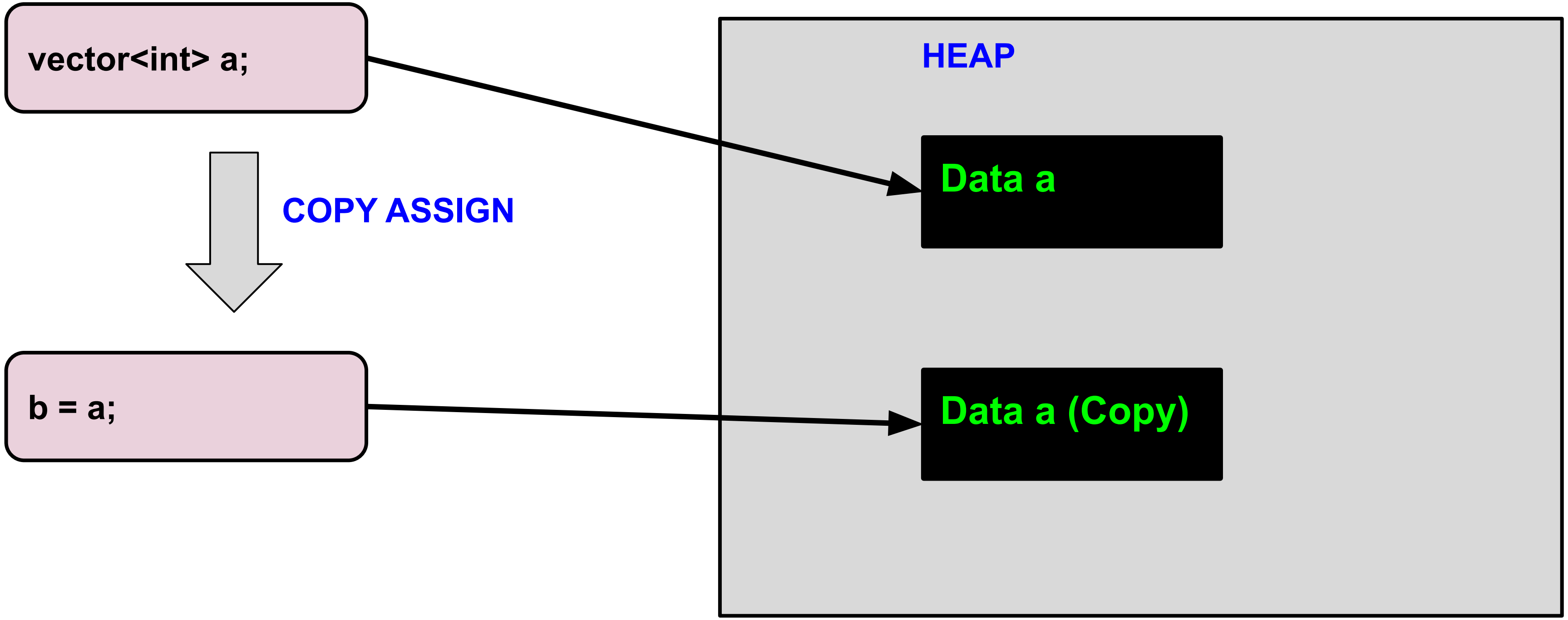
Data is moved, not objects ! This is efficient compared to the copy constructor!

Before Copy or Move assignment



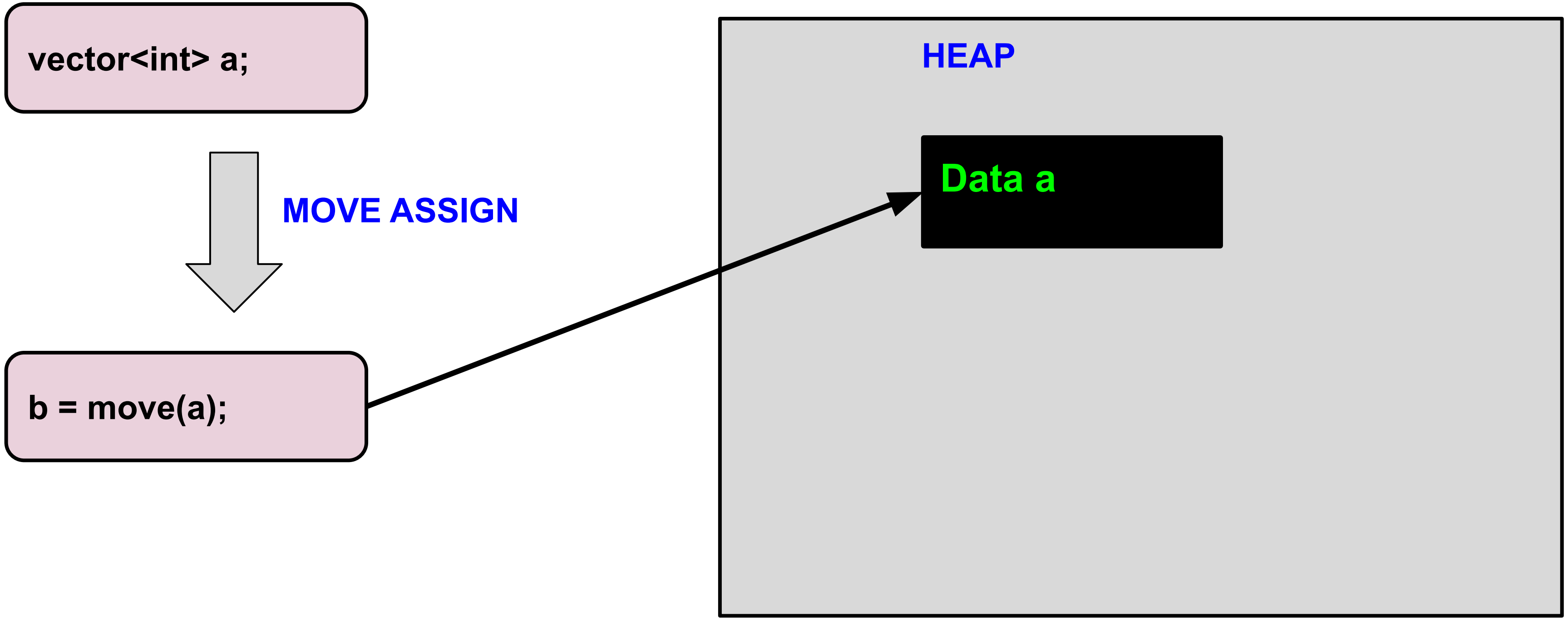
Vectors **a** and **b** both exist and keep their own heap data.

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 ! This is efficient compared to the copy assignment!

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

Move is a robbery, not murder !

```
{
        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?

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

How does move work? 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]**, pointer deref. ***ptr** etc.
RVALUE (right value) cannot be assigned. Temporary: literal, expression result etc.

However, in the 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 !

BOOK: Scott Meyers, Effective Modern C++, chapter 5

Rvalue vs lvalue: Modern definition

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

string s("Mouse");

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

string * pS2 = & string("Rat"); // ERROR, **string("Rat")** is 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"), "Coypu"s, vector<int>{3, 10, 17}**
3. Expression result: **2*a + b**
4. Function return by value : **cos(alpha)**
5. Function return by RVALUE ref : **move(s)**

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;
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 its argument to *RVALUE* !

Copy Ctor/assignment is selected for LVALUEs.

Move Ctor/assignment is selected for RVALUEs.

!!! **move(a)** will not work if **a** is **const** (including **const** ref)! **const** RVALUE ref == useless !!!

Source (factory) : function which 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 (but not guaranteed until C++ 17) !

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 (hopefully) !

Note: Alternatively, a source can return **unique_ptr<Tjej>** .

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 have types "RVALUE ref", but they are not RVALUES !
        Tjej t(rrT);    // Copy, NOT MOVE !
        string s(rrS);     // Copy, NOT MOVE !
        ...
}
```

All variables and function parameters are LVALUES by definition !

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

```
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
```

Why make an extra move operation? Better to pass by value !

```
void sink1(Tjej t) {        // Param by value (move), a local that dies at the closing '}' !
        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
```

"Parameter by value" is not always COPY, can be MOVE too (if called with RVALUE argument)

Writing a sink (consumer)

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

```
void sink1(Tjej t) {        // Param by value (move), a local that dies at the closing '}' !
        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, you should 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 Ctor:

```
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= moves the DATA (not object !)!

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.

8. **const** RVALUE is a stupid thing. Copy operation is selected! Don't use **move()** on **const** objects!

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 with **&&** 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
```

Linear algebra libraries in C++

- BLAS/LAPACK: classical Fortran libraries
- Eigen: header-only OR optionally can use BLAS/LAPACK for speed
- GNU Scientific Library (GSL): Copyleft, uses BLAS
- Armadillo: Nice C++ classes, uses BLAS/LAPACK
- OpenCV : Not really linear algebra, efficient linear algebra only with BLAS/LAPACK

Other languages:

- Python: numpy/scipy : uses BLAS/LAPACK
- Matlab : uses BLAS/LAPACK

Do you see any common theme here?

Library of the day: CBLAS + LAPACKE (or BLAS/LAPACK)

BLAS (Basic Linear Algebra Subprograms) :

Very low-level operations (up to matrix*matrix)

Original *Reference BLAS* : FORTRAN, slow

Modern BLAS is super-optimized for modern CPUs (SIMD etc.)

It is very important to use recent versions for modern CPUs !

OpenBlas : various architectures (Intel, AMD, ARM), open source

Intel MKL : Intel CPUs only, very slow on AMD, proprietary

LAPACK (Linear Algebra Package) :

Higher-level operations (eigenvalues etc.)

Usually generic FORTRAN LAPACK is used

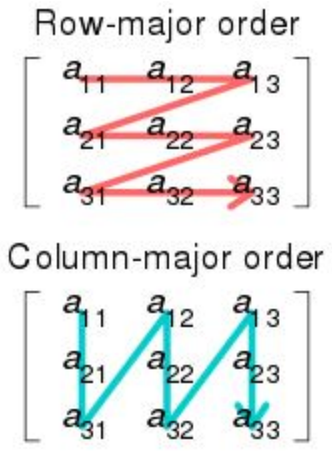
Intel MKL has an own modified LAPACK version

Fortran : BLAS + LAPACK

C/C++ wrappers : CBLAS+LAPACKE (or you can use fortran libraries directly)

BLAS+LAPACK

- Blas level 1: Vector-vector operations (vec+vec, vec.vec)
- Blas level 2: Matrix-vector operations (mat*vec)
- Blas level 3: Matrix-matrix operations (mat*mat)
- Lapack *computational routines*: QR-factorization etc.
- Lapack *driver routines*: Highest levels (eigenvalues+eigenvectors)



BLAS+LAPACK facts:

- BLAS-LAPACK uses *column-major* matrices (Opposite to C/C++)!
- Support for special matrices (symmetric, band, triangular, sparse)
- Lapack has different algorithms to choose from for some operations
- Vectors and matrices can have strides ("increment" or "leading dimension")
- Uses low-level C (or Fortran), no C++ classes

Other libraries: FFTW (Fastest Fourier Transform in the West), ScaLapack (multi-CPU)
Documentation : https://software.intel.com/en-us/mkl-developer-reference-c

Blas levels 1, 2 :

Blas level 1 example: vector-vector: dot product of two vectors (Note : C API, uses *pointers*)

```
vector<double> a{2., 3., 7., 2.}; // We can use vector
double b[] = {1., -1., 1., -2.};  // Or array
int n = 4;                        // Vector size
int inca = 1, incb = 1;           // Strides of a, b
double result = cblas_ddot(n, a.data(), inca, b, incb);   // Double DOT
```

Blas level 2 example: matrix-vector: operation **dgemv** **alpha*A*x + beta*y**

Double **GE**neral matrix **M**atrix-**V**ector product

```
double a[]{1, 2, 3, 6, 5, 4};
double x[]{1, -1};
double y[]{2, 1, -1};
int m = 3, n = 2;            // Dimensions of A
int incx = 1, incy = 1;      // Strides of x, y
int lda = m;                 // "Leading dimensions" (matrix strides)
double alpha = -1., beta = -2.;
cblas_dgemv(CblasColMajor, CblasNoTrans, m, n, alpha,
            a, lda, x, incx, beta, y, incy);
```

Blas levels 3 :

Blas level 3 example: matrix-matrix: operation **dgemm** (**alpha*A*B + beta*C**)

Double **GE**neral matrix **M**atrix-**M**atrix product

```
double a[]{1, 2, 3, 6, 5, 4};   // 3x2
double b[]{1, -1, 2, 0, 0, 2, 1, 1};   // 2x4
double c[]{0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5};  // 3x4

int m = 3, k = 2;                 // Dimensions of A
int n = 4;                        // Columns of B
int lda = m, ldb = k, ldc = m;    // "Leading dimensions" (matrix strides)
double alpha = 0.5, beta = 2.0;

cblas_dgemm(CblasColMajor, CblasNoTrans, CblasNoTrans, m, n, k,
            alpha, a, lda, b, ldb, beta, c, ldc);
```

Lapack

Lapack example: matrix-matrix: diagonalize a real symmetric matrix : operation **dsyevr**

Double **SY**mmetric **E**igen**V**alues with Relatively **R**obust Representations

This is a high-level driver routine!

```
double a[2][2] = {0., 1., 1, 0.};
int n = 2;                      // Dimension of a
int lda = 2;                    // Leading dimension of z
char jobz = 'V';                // Compute e-values + e-vectors
char range = 'A';               // Compute all e-values
char uplo = 'U';                // Upper triangular part of a

double z[2][2];                 // Output eigenvectors
int ldz = 2;                    // Leading dimension of z
double w[2] {-3.14, -3.14};     // Output eigenvalues
int m = -1;                     // Output: number of e-values found
int isuppz[4];                  // "Support output"

LAPACKE_dsyevr(LAPACK_COL_MAJOR, jobz, range, uplo, n, (double *)a, lda, 0, 0,
               0, 0, 0, &m, w, (double *)z, ldz, isuppz);
```

title

text

Thank you for your attention !

THE END