Advanced Programming for Scientific Computing (PACS)
Lecture title: Move semantic

Luca Formaggia

MOX
Dipartimento di Matematica
Politecnico di Milano

A.A. 2022/2023

Introduction

In this lecture we illustrate an important feature of modern C++ that goes under the name of Move semanting.

But it is also an occasion to go deeper into important aspects of the language: value categories and reference bindings.

The original problem

One of the problems of C++11 is that often objects can be dynamic objects of big size (think of a matrix for instance).

Thus, we should avoid to make useless copies. Unfortunately, copies may happen in different situations.

Let's for instance look at this piece of code that swaps two dynamic matrices.

Swap may be costly

Let's consider this function that swaps the arguments:

```
void swap (MyMat0 & a, MtMat0 & b){
 MyMat0 tmp{a}; // make a copy of a
 a = b; // copy—assign b to a
 b = tmp; // copy assign tmp to b
} // tmp is destroyed on exit.
```

If *a* and *b* are of big size this function is very inefficient.

- Memory inefficient: we have to store tmp
- Computationally inefficient: copy operations impy copying all matrix elements.

The optimal swap

A dynamic matrix typically contains (directly or indirectly) a pointer to the dynamically allocated data. Let's assume it is a pointer to double: **double** * data. Thus, we would like to have instead an algorithm of this sort, operating with the pointers:

- Copy `a.data` into `tmp`;
- Copy `b.data` onto `a.data`;
- Copy `tmp` onto `b.data`.

Also here we have a temporary, but it is just a pointer!

The optimal swap before C++11

Let's assume that MyMat0 stores dynamic data for its elements as a **double** * data (we will see that it is better use a standard vector, but it is not relevant here). Before the introduction of move semantic I could have solved the problem by writing a special method or a friend function. For instance:

```
void swapWithMove(MyMat0 & a, MtMat0 & b){
 ...//swap number of rows and columns
 double * tmp=a.data; //save the pointer
 a.data=b.data; // copy the pointer
 b.data=tmp; // copy the saved ointer
}
```

This way I just swap the pointers, saving memory and operations, but only for this specific situation. It is not generalizable: I cannot write a function template swapWithMove<T> because I need to know how data is stored in T, for each case.

Questions to be addressed

To implement a move semantic 3 questions have to be addressed:

1. How can we identify objects that can be safely "moved" instead of copied, so that the compiler may perform the move automatically, whenever possible?
2. How can I actually implement the move in a uniform and general way?
3. How can I specify that I want to "move", instead of copying?

Let's give the answer one question at a time. For the first one, we need to introduce value categories.

Categories of values

In C++ a value is characterised by its type, and its category, which expresses how the value can be used.

In C++ we have 4 categories for values: glvalue, prvalue, xvalue and lvalue (Aagh!). Moreover, they can be const or non-const...

To simplify matters (without losing important information), we will only use 2 categories: lvalue (which also includes glvalue) and rvalue (which includes prvalue and xvalue).

rvalues and lvalues

The original definition of lvalues and rvalues from the earliest days of C is as follows: an lvalue is an expression that may appear on the left and on the right-hand-side of an assignment, whereas an rvalue is an expression that can only appear on the right hand side of an assignment.

```
double fun();// a function returning a double
3.14=a; // WRONG a literal expression is a rvalue!
fun()=5;// WRONG returning an object generates an rvalue
```

L- and Rvalues in C++

User defined types, **const** and operator overloading makes the definition of rvalues/lvalues rather complicated in C++. We avoid the formal definition contained in the standard (very technical): We give a simple definition, correct in most of cases:

An lvalue is an expression that refers to a memory location and allows us to take its address via the & operator. An rvalue is an expression that is not an lvalue.

For this reason lvalue is nowadays interpreted as locator-value and no more left-value. It is still true that an (non-const) rvalue can only be at the right-hand side of an assignment.

Examples of lvalues

The value held in a variable (i.e. a value with a name) is always an lvalue, I repeat, always. Even if it is const, or a **constexpr**, since we can take its address

```
double a;
int const b=10;
double * pa = &a;// address of a
int const * pb=&b;// address of b
```

If a function returns a lvalue reference (&), the returned value is an lvalue:

```
double & f(double & x){x*=3; return x;}
...
double y=8.0;
double * px=&(f(y));// it's the address of y
```

Examples of rvalues

The value returned by a function is an rvalue:

```
double fun(double x){....}
```

Here, &fun is a pointer to the function not to the returned value. I cannot take the address of the returned value, it's a temporary object.

Non-string literals are rvalues:

```
double * pd = &(10.5);// Error (it doesn't make sense)
```

compilers are free not to store them in memory, so no address may be taken (and it does not make sense taking it). Strings, however, are lvalues.

Answer to the first question

Non-const rvalues are eligible for "automatic moving". Indeed, if we cannot take the address it means that they exists only to be put somewhere.

So we have the answer to the first question: rvalues are movable. In particular values returned by a function are movable.

The second question

To answer the second question, let's look at how references bind according to the category of the bound values.
We consider ordinary references first, from now on called lvalue references. A non-const lvalue reference cannot bind to rvalues, while both lvalues and rvalues can be bound to const lvalue references:

```
double & pi=3.14;// wrong! A literal expr. is a rvalue
double const & pi=3.14;// Ok!
```

Other examples:

```
int pippo(); // returns a rvalue
int & pluto (int & a); //returns reference thus an lvalue
int micky (int const & a); // returns a rvalue
....
auto p=pippo();//ok p is an int
int & c=pluto(p);// ok function returns a lvalue here!
int & d=pluto(3);// NO! 3 is an rvalue cannot be bound to
                 // to a (lvalue) reference
auto & x=pluto(pippo())// NO! as above.
cont int & a = micky(pippo()) // Ok a rvalue binds to const lvalue ref.
```

Reference binding in overloaded functions

The interplay between reference types and binding is clear (and important) when looking at function overloading.

```
void foo(int & a);
void foo(const int & a);
void goo(const int & a);
void zoo(int & a);
...
foo(5); //calls foo(const int &)
int g;
foo (g);// calls foo(int &)
goo (g);// goo(const int &);
const int b=10;
foo (b);// calls foo(const int &)
goo (b);// goo(const int &);
zoo (b);// ERROR!!
```

The compiler chooses the best match. An lvalue (g) is a better binding for a non-const lvalue reference, while a literal (5) or a const lvalue (b) can only match a const lvalue reference. Const lvalue reference may bind to both rvalues and lvalues.

Conclusion on lvalue reference binding

- A non-const lvalue reference can bind only, and preferably, to non const lvalues;
- A const lvalue reference binds both to lvalues and rvalues, const and non-const alike.

Here, preferably means that it will be chosen in case there is the choice.

This before C++11. Well, in fact it is still true if we just use lvalue references,

The consequence is that with just lvalue references we cannot distinguish lvalues from rvalues.

Relation with moving

Let's look at this piece of code

```
MyMat0 foo(); // a function returning a big object
...
MyMat0 a;
a= foo();
```

The return value of foo could be moved into a safely! (indeed the RVO does already do that for constructors).

It would be nice to have an "adornment" that acts like a reference, while however it can bind only and preferably to rvalues! In this way we may overload the assignment operator:

```
Matrix & operator =(Matrix const & a);// ordinary copy
Matrix & operator =(Matrix "new adorn" a);//Move!
```

rvalue references

Indeed, C++11 has introduced a new kind of adornment, called rvalue reference, indicated by &&.

It exclusively and preferably binds to rvalues. Preferably means that, if given the choice, an rvalue binds to an rvalue reference.

An important things to remember it that rvalue references love rvalues and only rvalues. And are rather jealous: they will not share them with anybody else.

Categories of values

We resume some rules:

- If a function returns a value that value is considered an rvalue.
- If a function returns a lvalue reference (const or non-const) that value is considered an lvalue.
- It a function returns a rvalue reference, that value is an rvalue.
- A (named) variable is always an lvalue.

This is fundamental for move semantic.

What about const rvalues and const rvalue references?

Well, indeed a fuller picture of reference bindings should include also the possibility of having a const rvalue (a non-string literal is a const rvalue) and const rvalue references, like **const double**&&.

To understand things better let's look at the simple example in Bindings/main.cpp, where you have all possible overloads with references.
Play with it by commenting some and see what happens!

Indeed, you never use all the overloads of that examples (see the last slides), it is just an example to show the binding rules.

How is move semantic implemented?

We are now able to answer the second question. The key is the move constructor and the move assignment operators.
This is the standard signature of move operations for a class named `Foo`:

```
Foo(Foo&&); // move constructor
Foo & operator=(Foo&&);// move assignment
```

Remember that, unless you have defined some other constructors or the copy assignment, the compiler provides a syntethic move constructor and move assignement operator automatically, which apply the corresponding moving operation on the non-static data members of the class.

Move semantic for MyMat0

Let's go back to MyMat0. Assume that MyMat0 stores the data as a pointer to double. A possible copy-constructor and copy-assignment take the form

```
MyMat0 &(MyMat0 const & rhs):nr{rhs.nr}, nc{rhs.nc},
 data{new double[nr*nc]}
{
// make a deep copy
for (i=0;i<rhs.nr*rhs.nc;++i)data[i]=rhs.data[i];}

MyMat0 & operator=(MyMat0 const & rhs){
// release the resource
delete[] this->data;
// Get new data buffer
data=new double[rhs.nr*rhs.nc];
// make a deep copy
for (i=0;i<rhs.nr*rhs.nc;++i)data[i]=rhs.data[i];}
```

The move operators

The corresponding move operator could be

```
MyMat0(MyMat0 && rhs):nr{rhs.nr},nc{rhs.nc},
data{rhs.data}{
// fix rhs so it is a valid empty matrix
rhs.data=nullptr;
rhs.nc=rhs.nr=0; // zero n. col an n. row
}
MyMat0 & operator=(MyMat0 && rhs){
delete[] this->data; // release the resource
data=rhs.data; //shallow copy
// fix rhs so it is a valid empty matrix
rhs.data=nullptr;
rhs.nc=rhs.nr=0;}
```

I just grab the resource and leave an empty matrix!
It is important to ensure that the moved object can be deleted correctly!. Since the destructor of MyMat0 calls **delete**[] on `data`, I set the latter to the `nullprt`.

The consequence

```
MyMat0 foo();
...
MyMat0 a;
a=foo();// move assignement is called
```

We say that a class implements move semantic if the move operators are defined (even if the synthetic ones).

`std::string`, and **all standard containers** implement move semantic. `std::unique_ptr` has move operators, but deleted copy operators.

Move semantic and perfect forwarding

Now the third question (how can I specify that I want to "move", instead of copying?), which in fact I divide in two parts:

Move: how to tell explicitly to the compiler to replace a copying operation with a move if move semantic is implemented (maybe with the synthesized move operators!)

Perfect forwarding: How to write function templates that take arbitrary arguments and forward them to other functions such that the target functions receive the values with the same category they were passed to the forwarding function.

Forcing a move: std::move

Well, first of all std::move doesn't move anything. They have chosen a wrong name, they should have called it std::movable instead. But we have to live with it.

std::move(expr) unconditionally casts expr to a rvalue. So it makes it available to be moved.

You use it to indicate to the compiler that you want something to be moved, even if it is an lvalue. It is actually moved if move semantic has been implemented for that type. If not, it will be copied.

```
// A poor man move()
template <typename T>
T&& move(T& t){return t;}
```

(Don't write your own,use std::move())

A new (generic) version of swap

Now we are able to write our swap, and in a generic way!

```
template<class T>
void swap(T& a, T& b) {
T tmp{std::move(a)};// move construct
a = std::move(b);   // move assign
b = std::move(tmp); // move assign
}
```

It type T implements move semantic the swap is made using the move operators and (if they are implemented correctly) with less memory requirement! If not, we have the usual copy.

Note: Use std::swap, which does exactly that. All containers have also a swap() method, which performs the swap intelligently.

I repeat: variables are always lvalues

Named variables are always lvalues! Even if they are declared as rvalue references. Indeed you can take their address!

In particular function parameters (of any function, also constructors) are lvalues, even if their type is an rvalue reference.

Inside the scope of this function

```
void f(Matrix&& m){
....
```

`m` is an lvalue. Remember that the terms *rvalue* and *lvalue* refer to categories, not types. You can take the address of `m` (i.e. you can write `Matrix * pm=&m` inside function `f`), so it is an lvalue.

I repeat: variables are always lvalues

For instance, let's suppose that class Foo is composed with a MyMat0 and takes it with the constructor. We may wish to have a version of the constructor that moves, instead of copy. I can do

```
class Foo{
public:
Foo(MyMat0 && m):MyM{m}{}
...
private:
MyMat0 MyM;
};
```

but THIS IS WRONG (though syntactically correct). MyM{m} calls the copy constructor since m is an lvalue.

The solution

You have to force the move!

```
class Foo{
public:
Foo(MyMat0 && m):MyM{std::move(m)}{}
...
private:
MyMat0 MyM;
};
```

Now MyM{std::move(m)} calls the move constructor and m is moved into MyM.

Another solution is to use forwarding references (see a next slide).

I like to move it. Move it!

All std containers support move semantic and all std algorithms are written so that if the contained type implements move semantic the creation of unnecessary temporaries can be avoided.

For instance, std::sort() (which does a lot of swaps) is much more efficient on dynamic big objects if move semantic is implemented.

Move semantic also makes a few (but not all!) template metaprogramming techniques now used in some libraries like the Eigen to avoid large size temporaries unnecessary. *Since version (3.3.9)Eigen matrices implement move semantic.*

Modern swap

So if you want to swap your matrices do

```
Matrix A;
Matrix B;
....
std::swap(A,B);
```

IMPORTANT NOTE

If your class stores its dynamic and potentially big data in standard containers, you just need the synthetic move operators (which means that you have move semantic for free!). Another good reason of using standard containers.

Move iterators

A little utility is provided if you want to move the contents of a container.

```
std::set<std::vector<double> > s1;
// s1 is filled with vectors
...// move the set's elements into a vector
std::vector<std::vector<double> > v1{
std::make_move_iterator(s.begin())
std::make_move_iterator(s.end())
};// move into v1 the vectors stored in s1
// s1 is now filled with empty vectors!
```

If I used the standard begin and end iterators I'd copy the elements, not move them.

Move constructor and RVO (copy elision)

Move constructor and copy elision (return value optimization) are two different things. If I do

```
MyMato m{Hilbert(3000)};
```

the compiler may apply (and it normally does, if a few conditions are met) copy elision, neither the copy nor the move constructors are used: `m` is built directly with the `MyMat0` value that `Hilbert` returns.

If you want to know more about copy elision you may go to cppreference.com. But it is very technical. It is sufficient to know, that this construct may be beneficial in terms of efficiency (but not in terms of debugging...).

Two examples

In MoveSemantic_simple you have a simple program that show how move semantic operates on a class composed with a std::vector.

In MoveSemantic an example of a class implementing a full matrix where we have defined move constructor and move assignment operator.

After compiling it do `make test` and see the result of the memory usage of the version of the code where move semantic is activated and the one where is deactivated (by defining the preprocessor macro NOMOVE).

You need to have `valgrind` installed, the modules provided with the course have it.

The perfect forwarding problem

There is another problem that rvalue references can solve. Let's look at this function that implements the object factory design pattern passing a object to the constructor

```
template<typename Arg>
unique_ptr<Polygon> factory(Arg const & arg,std::string switch)
{
  if(switch==''Triangle'')
  return std::make_unique<Triangle>(arg);
  ... etc}
```

This is fine, but if the first argument in the call of factory is an rvalue, it is transformed to a lvalue (arg is a variable), and I do not take advantage of move semantic. Yet, I cannot use std::move because it will move arg even if I do not want to. Actually, it will give a compiler error since arg is const.

The perfect forwarding problem

What I want is something that if I do,

```
std::vector<Point> t;
...
auto p=factory(t,"Triangle")
```

the arg passed to make_unique be a lvalue (copy constructor is used). While, if I do, for instance

```
std::vector<Point> t;
...
auto p=factory(std::move(t),"Triangle")
```

arg should be an rvalue.

A possibility is to make an overload of factory with Arg&&, which will be called on rvalues arguments. But this way I replicate code uselessly.

There is another, cleaner solution, but first we need to introduce universal references, also called forwarding references.

Universal/forwarding references

First of all there is no mention of universal or forwarding references in the C++ standard. It is a term invented by H. Sutter to explain in simple terms a particular behavior of rvalue references when used as template dependent parameters.

This behavior is linked to the so called reference collapsing rule, by which T&&&=T&, but if you understand universal references you can avoid understanding reference collapsing. Universal references appear when you have constructs of the type

```
template <class T>
double fun(T&& x);
```

How do universal references work?

The combination of template parameter deduction and collapsing rule causes parameter x to be able to bind both to lvalues and rvalues!

```
double randomValue(); // a function
double a{5.0};
double const & ra=a;
...
x=fun(6.7);// fun(double&&) (T=double)
x=fun(a);// fun(double&) (T=double&)
x=fun(randomValue());// fun(double&&) (T=double)
x=fun(ra);// fun(const double&) (T=const double&)
```

That's why they are called universal!. Note that whenever the argument is an rvalue, T is a simple type. Otherwise, it is a reference (maybe const).

BEWARE

A rvalue reference behaves as universal reference only if its type is deduced at the moment of the instance of the function!

```
template<class P>
void fun(P && x);// Universal reference
template <class T>
double fvector(std::vector<T>&& x); //NOT UNIVERSAL
```

In the second function the rvalue reference is not a universal reference (i.e. it just binds to rvalues).

In other words, universal references takes (almost) just the form

```
template
<class T> f(T&& x)
```

(of course the template parameter may have a different name and you may have more than one parameter...).

How to fool the compiler

Also in this case the type is not deduced at the moment of instance of the method, so the rvalue reference is not universal

```
template<class T>
class toto{
public:
void foo(T&& x); // T is not deduced
...
```

But you can fool the compiler!

```
template<class T>
class toto{
public:
template <class D=T>
void foo(D&& x); // D now is deduced
...
```

Perfect forwarding

What the use of forward references?? Well, we can introduce the magic of std::forward<T>() to solve our problem:

```
template<typename Arg>
unique_ptr<Polygon> factory(Arg&& arg,std::string switch){
  if(switch==''Triangle'')
  return std::make_unique<Triangle>(std::forward<Arg>(arg));
  ... etc}
```

Thanks to the "universal binding" of universal references and the magic of std::forward<T>(), this version passes to make_unique the argument bound to arg preserving its category.

What does std::forward<Arg>(arg) do?

It forwards lvalues as either lvalues or as rvalues, depending on Arg For instance, if Arg is a simple type, it converts arg to a rvalue, if Arg is an lvalue reference arg is returned as an lvalue.

It allows to forwards the argument to another function with the value category it had when passed to the calling function.

You do not need to know the details, but the advantage of its usage (now becoming very common in advanced C++ programming).

A universal constructor

```
class Foo
{
  public:
  // I can pass anything convertible to vector
  template<typename T>
  Foo(T && v):toto{std::forward<T>(v)}{}
  private:
  std::vector<double> toto;
}
```

If v can be moved it will be. If not, it is just copied. I do not need to use overloading.

```
std:: vector<double> randomVector(int);
...
vector<double> b;
...
Foo foo(randomVector(30));// MOVED
Foo goo(b);// b is copied
```

Why not going variadic?

```
template<typename... Arg>
unique_ptr<Polygon> make_trianglePtr(Arg&&... arg)
{
  return std::make_unique<Triangle>(std::forward<Arg>(arg)...));
}
```

Now make_unique may use any available constructors of Triangle:

```
auto a = make_trianglePtr();// uses default constructor
auto b = make_trianglePtr(a);// uses copy constructor
auto c = make_trianglePtr(std::move(a)); //uses move constructor
//uses constructor that takes 3 points:
auto d = make_tianglePtr(p1,p2,p3);
//point 3 is now moved
auto e = make_trianglePtr(p1,p2,std::move(p3));
```

Powerful, isn't it? But let's relax a bit. Variadic templates are complex stuff (but fun!). By the way, make_unique uses the same technique!

H. Sutter docet

- Use std::move() if you want to move lvalues, but
- always use std::forward<T> on forwarding references!

Use universal references and std::forward<> to make functions able to operate both on lvalues and rvalues and to apply move semantic on rvalues without the need of overloading.

Remember that after a move the moved object should be a valid "empty" object!

Note: To use std::move and std::forward you need the <utility> header.

General guidelines for function arguments I

If the argument is modified by the function (it is also an output of the function) you have only one choice: using an lvalue-reference

```
void fun(Myclass & m)
{
 // Modify m;
}
```

General guidelines for function arguments II

The argument is just used in the function, i.e. accessed "read-only". Use a constant lvalue-reference or, if copying is cheap, a value:

```
void fun(const Myclass & m)
{
// Using m;
}

void gun(int m)
{
// Using m;
}
```

General guidelines for function arguments III

The argument is meant to be copied inside the function/method before being used. If copy is expensive, pass by value and move, or use a forwarding reference.

```
void fun(Myclass m)
{
MyClass x =std::move(m);
}

// or (more complex but better)
template< class M>
void gun(M&& m)
{
 MyClass x =std::forward<M>(m);
}
```

This will use move semantic depending on the value category of the argument. The first solution will do two moves if argument is an rvalue.

General guidelines for function arguments IV

The argument is passed to another function and you want to keep the value category (to eventually exploit move semantic). Use forwarding reference:

```
class Foo
{ // a constructor taking lvalues and rvalues
template<class M>
Foo(M&&):m_{std::forward<M>(m)}{...}
private:
MyClass m_;
}

// A function adapter
template<class M>
decltype(auto) fun(M&& m)
{
 return goo(std::forward<M>(m),25.0);
}
```

General guidelines for function arguments V

You need to distinguish the behavior on non-const rvalues and the rest (this is what you do in copy/move constructors). Use overloading with const lvalue and rvalue references:

```
void fun(MyClass&& m)
{
// operates on non-const rvalues
}
void fun(const MyClass& m)
{
// operates on lvalues
}
```

Other overloading are possible (as you have seen them in the example), but seldomly used!