# Programmazione Avanzata per il Calcolo Scientifico
# Advanced Programming for Scientific Computing
# Lecture title: Move semantic

Luca Formaggia

MOX
Dipartimento di Matematica
Politecnico di Milano

A.A. 2014/2015

# Introduction

One of the problems of C++11 is that often objects can be of big size (think of a matrix for instance). Thus, we should avoid to make useless copies. Unfortunately, copies may happen in different places. Let's for instance look at this piece of software that swaps two matrices.

```
Matrix a,b;
... //some work with the matrices
Matrix temp(a);
a=b;
b=temp;
```

This is memory inefficient, we do not really need to have at the same time temp, a and ı!b!. We just want to swap the contents of the two Matrix objects!

This lecture is largely based on the material found in thbecker.net/articles/rvalue_reference and chapter 5 of the book Effective modern C++ by H. Sutter.

# Move semantic and perfect forwarding

The new standard has taken a decisive step to solve this problems with the introduction of rvalue references, which allow to implement:

Move semantics: makes it possible for compilers to replace expensive copying operations with less expensive moves. Move constructors and move assignment operators offer control over the semantics of moving.

Perfect forwarding makes it possible to write function templates that take arbitrary arguments and forward them to other functions such that the target functions receive exactly the same arguments as were passed to the forwarding functions.

However before dealing with move semantic it is better to revise some important concepts.

# types and categories

In C++ expressions and variables are characterized by two different properties: type and category.

The type identify an expression in terms of its size and the functionality it provides. A category is instead linked to the context in which the expression or variable is used.

# Type

A type may itself be subdivided in basic type, qualifiers and "adornment".
More than giving definitions, it is simpler to give examples

```
int a; // basic type is int
const int a; //basic type is int, qualified as const
int& c=a; // basic type is int. Type is reference to int
```

& is an adornment that specifies that the type is a (lvalue) reference, i.e. an alias to another object.
The two most important qualifiers are **const** and **volatile**. **const** indicates that the variable cannot be changed, **volatile** that it can change between different accesses, even if it does not appear to be modified. **mutable** is another qualifier.

# Categories

There are in fact three primary categories in C++11: lvalue, prvalue and xvalue.

But, for the sake of simplicity we identify the last two with their more common name: rvalue.

# 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 or 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
```

# Rvalues in C++

User defined types and operator overloading makes the definition of rvalues/lvalues rather complicated in C++. We avoid the formal definition contained in the standard (very technical), which in fact distinguishes rvalues in prvalues and xvalues. We give a simple definition, correct in the 99.8 per cent of cases:

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

# What does the "l" in lvalue mean?

An rvalue cannot be used to initialize a non-const lvalue reference (while if can be used to initialize const references)
That is, an rvalue cannot be converted to an lvalue, but when an lvalue is used in a context where an rvalue is expected, the lvalue is implicitly converted to an rvalue.

```cpp
// lvalues:
 int i = 42;
 i = 43; // ok, i is an lvalue
 int* p = &i; // ok, i is an lvalue
 const & g=42;//
 int& foo();
 foo() = 42; // ok, foo() is an lvalue
 int* p1 = &foo(); // ok, foo() is an lvalue
  // rvalues:
 int foobar();
 int j = 0;
 j = foobar(); // ok, foobar() is an rvalue
 int* p2 = &foobar(); // error
    // cannot take the address of an rvalue
 j = 42; // ok, 42 is an rvalue
```

# Another example

```
struct UserType
{
  double member_function();
}
int var = 0;
var = 1 + 2; // ok, var is an lvalue here
var + 1 = 2 + 3; // error, var + 1 is an rvalue
int* p1 = &var; // ok, var is an lvalue
int* p2 = &(var + 1); // error, var + 1 is an rvalue
const double & a=UserType().member_function(); // ok, calling a
        // member function of the class returns a rvalue
       // and a rvalue may bind to a const lvalue reference
       // THE LIFESPAN OF THE RETURNED OBJECT IS EXTENDED!
```

# Reference binding

The process by which an expression is associated to a reference is called binding. The rules for ordinary (lvalue) references are rather straightforward. The interplay between parameter types and binding is used for function overloading.

```
void foo(int & a);
void foo(const int & a);
void goo(const int & a);
...
foo(5); //calls foo(const int &)
int g;
foo (g);// calls foo(const int &)
goo (g);// goo(const int &);
```

In the call of foo the compiler chooses the best match. In the first case an rvalue const reference (a literal) is bound to a constant reference, in the second a (non const) **int** is bound to a **const int**&. In the call of goo a **int** is bound to a **const int**& since there is no better match.

# The need of moving objects

Let's consider a simple example. A function that swaps the arguments

```
template<class T>
void swap (T & a, T& b){
 T tmp{a}; // make a copy of a
 a = b; // copy-assign b to
 b = tmp; // copy assign tmp to b
} // tmp is destroyed on exit.
```

If *a* and *b* are objects of big size this function is memory-inefficient. We have to store tmp.

# The optimal swap

We would like to have instead an algorithm of this sort:

- Move `a` into `tmp`, leaving `a` empty;
- Fill `a` by moving `b`, leaving `b` empty;
- Fill `b` by moving `tmp`, leaving `tmp` empty.

It is now possible thanks to the move semantic.

# rvalue references

To implement move semantic C++11 has introduced a new type, or more precisely a new type decorator, called rvalue reference, indicated by &&.

Its main usage is in operator overloading: it allows to select operations that move objects instead of copying them.

This is possible because a non-constant (and thus movable) rvalue preferably binds to a rvalue reference. But they can bind also to constant lvalue references (so the old behavior is kept if move semantic is not implemented).

While, lvalues can only bind to lvalue (ordinary) references &.

# Categories of function return values

We also have the following important 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 (but there is normally no reason to do so!), that value is an rvalue.

This is fundamental for move semantic.

To understand things better let's look at the simple example in Bindings/main.cpp.

```
void foo(int & a);
void foo(const int & a);
int createFive();
void goo(int & a);
void goo(int && a);
void goo(const int & a);
```

I have two overloaded functions and a function returning an **int**.

# The main

```
foo(25);              //foo(const int &)
foo(a);               //foo(int &)
foo(b);               //foo(int &)
foo(createFive());    //foo(const int &)
goo(25);              //goo(int &&) NOTE!
goo(a);               //goo(int &)
goo(createFive());    //goo(int &&) NOTE!
goo(b);               //goo(int &)
```

# The general overloading rules

If you implement **void** foo(X&); but not **void** foo(X&&); the behavior is the usual C++98 one: `foo` can be called on lvalues, but not on rvalues.

If you implement **void** foo(X **const** &); but not **void** foo(X&&); then again, the behavior is the old one: `foo` can be called on lvalues and rvalues, but it is not possible to distinguish between them. That is possible only by implementing **void** foo(X&&); as well.

Finally, if you implement **void** foo(X&&); but neither one of **void** foo(X&); and **void** foo(X **const** &); then `foo` can be called on rvalues, but trying to call it on an lvalue will trigger a compile error.

# A first important note

Never ever write a function that returns a reference to a temporary.

```
Matrix & createMatrix(){
  Matrix tmp;
  ...
  return tmp;
}
```

This not only wrong, but useless! Temporaries must be always returned by values. Thanks to RVO (return value optimization) and the fact that the returned object is a rvalue returning by value is optimal!

```
Matrix  createMatrix(); // OK!
```

# Another important thing to remember

Named variables of rvalue rererence type are lvalues! 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) so it is an lvalue.

# How to implement move semantic?

You need to write a move constructor and a move assignment operator.

Beware that unless you have defined (even if with the keyword **default**) come other constructor or copy assignement (see next slide) the compiler provides a move constructor and move assignement operator automatically.

This is the standard signature of move operations for a class named `Foo`:

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

# The rules for the generation of automatic operators

compiler implicitly declares

| user declares | default constructor | destructor | copy constructor | copy assignment | move constructor | move assignment |
|---|---|---|---|---|---|---|
| Nothing | defaulted | defaulted | defaulted | defaulted | defaulted | defaulted |
| Any constructor | not declared | defaulted | defaulted | defaulted | defaulted | defaulted |
| default constructor | user declared | defaulted | defaulted | defaulted | defaulted | defaulted |
| destructor | defaulted | user declared | defaulted | defaulted | not declared | not declared |
| copy constructor | not declared | defaulted | user declared | defaulted | not declared | not declared |
| copy assignment | defaulted | defaulted | defaulted | user declared | not declared | not declared |
| move constructor | not declared | defaulted | deleted | deleted | user declared | not declared |
| move assignment | defaulted | defaulted | deleted | deleted | not declared | user declared |

Let us consider the version of MyMat0 class that holds the matrix data in the member **double** * data (not the best choice but good for this example).
A possible construcnto and copy-assignement will take the form

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

# The move operators

The corresponding move operator are

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

I just grab the resource!

it is important to ensure that the moved object can be deleted correctly!. Since the destructor of MyMat0 calls **delete**[] on data, I

# The consequence

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

# Forcing a move

If named variables, and consequently function parameters, are lvalues, how can I apply the move semantic on them? Well, two utilities of the standard library come to our rescue: std::move() and std::forward<T>()

std::move(expr) unconditionally casts expr to a rvalue reference
std::forward<T>(expr) casts expr to the lvalue reference T& if expr is an lvalue, and to the rvalue reference T&& if expr is an rvalue.

We will see the latter in the context of perfect forwarding and universal references (another feature of C++11). For the moment let's concentrate on move().

# A new version of swap

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

It type T implements move semantic the swap can be made with much less memory requirement!

# Some details

Suppose that a Base class has implemented

```
Base(Base const & rhs); // copy constr.
Base(Base&& rhs); // move constr.
```

and we want to use the move-constructor of Base in the move constructor of a Derived class. We need to use std::move

```
Derived(Derived&& rhs) : Base(std::move(rhs)){
  // Derived-specific stuff
}
```

This because rhs is an lvalue (it has a name!). Without move(), Base(rhs) would use the copy-constructor of Base !!

# I like to move it. Move it! 

All std containers support move semantic (since C++11) and all std algorithms have been rewritten so that if the contained type implements move semantic the creation of unnecessary temporaries can be avoided.

For instance, `std::swap()` does exactly what we have seen in a previous slide and `std::sort()` (which does a lot of swaps) will now be much more efficient on big objects.

Move semantic will also make a few (not all!) template metaprogramming techniques now used in some libraries (like the Eigen) to avoid large size temporaries unnecessary (with a considerable reduction of headaches).

# An example

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).

# A perfect forwarding

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

```
template<typename Arg>
unique_ptr<Base> factory(Arg arg, int switch){
  if(switch==1)
  return unique_ptr<Base>(new D1(arg));
  ... etc}
```

We are making a useless copy of the first argument into the parameter arg, while what we want is just to forward arg to the constructor of D1

One may think that the problem is solved by using references. But this solution has some other drawbacks, we will not detail them, see here for a discussion. Furthermore all "standard" solutions will block move semantic.

We do not want to go into too much detail, we just present the C++11 solution. But first a digression on universal references.

# Universal references

First of all there is no mention of universal references in the C++11 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, but if you understand universal references you can avoid the headache of understanding reference collapsing. Universal references appear when you have constructs of the type

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

# How does it 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!.

# 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>
void foo(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) since at the moment of the instance the type is known (well you have to think a little on how templates are instantiated to understand why....)

In other words, universal references takes the (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...).

# Perfect forwarding

Now we can we can use the std::forward<T>() function to solve the problem:

```
template<typename Arg>
unique_ptr<Base> factory(Arg&& arg, int switch){
  if(switch==1)
  return unique_ptr<Base>(
          new D1(std::forward<Arg>(arg)));
  ... etc}
```

Thanks to the “universal binding” of universal references and the magic of std::forward<T>(), this version works both if Arg is an lvalue or rvalue and it resolves rvalue references correctly: no useless temporaries if move semantic is implemented, all automagically.

# H. Sutter docet

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

But remember that after a move the moved object is "empty"!

# Some useful usage of universal refs

```
struct Foo{
 template <class T>
 Foo(T && x):M_x(std::forward<T>(x)){}
 private:
 Matrix M_x;
}
```

A constructor that takes any argument convertible to a Matrix. In one go we have both the copy and move version!!!

```
Matrix HilbertMatrix(); // a function
Matrix m;
Foo g(m); // copy
Foo z(HilbertMatrix());// moved!
```

# Cheating the compiler

```
template<class T>
struct Pippo{
 void f(T &&); // NO UNIVERSAL
}
```

It is not universal because when an object of type Pippo<T> is instantiated T assumes a value, so there is nothing to be deduced when calling Pippo<T>::f(). Workaround:

```
template<class T>
struct Pippo{
 template <class Q>
 void f(Q &&); // UNIVERSAL!!
}
```

# Conclusions

Move semantic is one of the most relevant features of C++11. It is not simple (but not overly complicated after all).

Do implement move semantic on class of large size. You just have to add the move constructor and the move assignment operator. Remember that the compiler may give you one automatically (but remember also that sometimes it does not). If the automatic move constructor is what you need and you want to be sure to have it, use the `default` keyword.