Copy & Move Semantics

Copy semantics are required for the classes that have ownership semantic.
↓
memory, file, socket, etc.

Following policies can be implemented for copy ownership:

① no ~~cop~~ copying policy:
- class objects cannot be copied.
- done by private copy ctor & assign operator in classic C++.
- in C++'11 it is recommeded to use "=delete" function.

In C++98, file stream classes follow no copying policy.

② exclusive ownership policy / move:
- extension of no copy policy.
- transfer ownership to another object.
- Also referred as "move semantics".
- implemented with r-value reference
- e.g. C++'11, thread, fstream classes.

Note: if not implemented, compiler synthesizes default copy ctor & assignment operator, which does shallow copy. For ownership semantics, this ~~can be~~ is not desirable.

```
class Integer {
    int * m_pInt = new int {};
public:
    Integer(const Integer &)=delete.
    Integer& op=(const Integer &)=delete;
    Integer (Integer && other) {
        m_pInt = other.m_pInt;
        other.mpInt = nullptr;
    }
    Integer& op=(const Integer && other) {
        if (this == &other)
            return *this;
        delete m_p Int;
        m_pInt = other.m_pInt;
        other.m_pInt = nullptr;
    }
    //...
};
```

move semantics

Remove resource from source object so that its destructor doesn't

③ deep copy policy:

Target object copies values & resource from source object. Both objects are independent of each other and resources are deleted in respective ~~de~~ destructor.

e.g. string, vector and stl containers.

Need impl of copy ctor & copy assignment operator.

NILESH GHULE

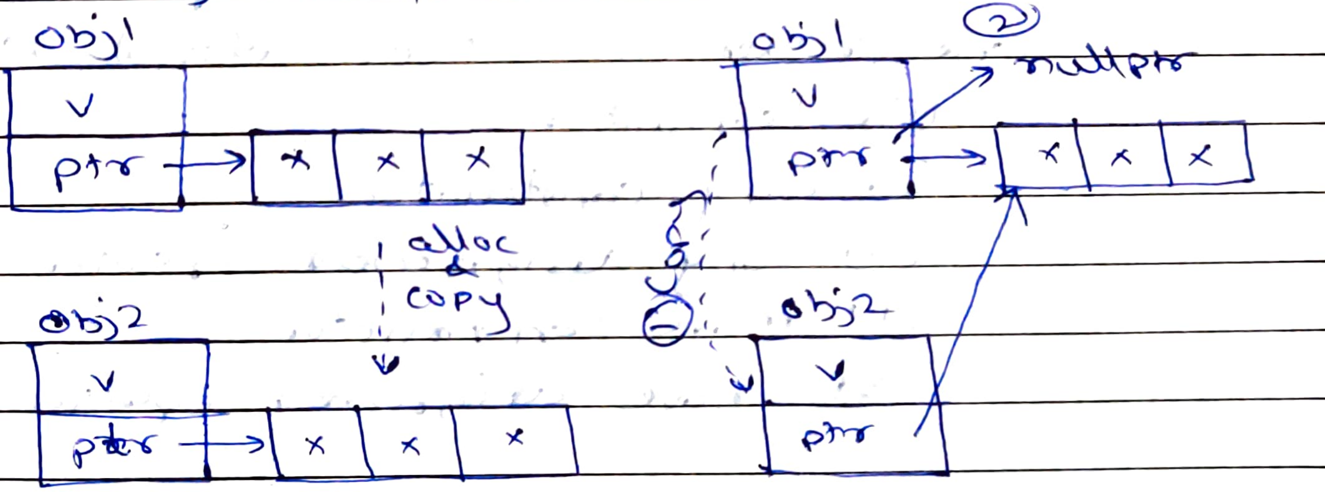
Page No. | Date

(4) shared ownership policy:

copy the object members but do not copy resources. The resource is shared among multiple objects. The resource is released when all objects are deleted.

e.g. shared_ptr<> - smart pointer.

Deep Copy Semantics	vs	Move Semantics
* copy state/resource		* move state/resource



obj1 & obj2 destructor delete their resources.	obj2 destructor will release the resource obj1 destructor ignore delete op. (due to null ptr).
slower process.	faster process.

NILESH GHULE

Rule of 3 :

If class implement any of following 3 fns, it should implement other 2 as well.

1. destructor
2. copy constructor
3. copy assignment operator

Rule of 5 :

If class has ownership semantics, one must write user-defined fns:

1. destructor
2. copy constructor
3. copy assignment operator
4. move constructor
5. move assignment operator

Rule of 0 :

If class do not have ownership semantics, do not implement any of above. Compiler will synthecise all of them.

However if we implement few of them compiler may not synthecise all of them.

Custom impl	copy ctor	copy assign	move ctor	move assign	destructor
copy ctor	custom	=def	=del	=del	=def
copy assign	=def	custom	=del	=del	=def
move ctor	=del	=del	custom	=del	=def
move assign	=del	=del	=del	custom	=def
destru	=def	=def	=def	=def	custom

Summary:

① if not provided any of these fns, compiler synthecise all of them.

② if any copy semantic is implemented, move semantic are not synthecised; but other copy semantic is synthecized.

③ if any move semantic is implemented no other copy and move semantic is implemented.

④ destructor is always synthecised.

Programmer may choose to implement other required fns or may ask compiler to synthecise forcibly using "=default"

Copy Elision

Compiler may choose to avoid copy of temp objects. This is called as copy elision.

```
Integer Add (Integer &a, Integer &b) {
    Integer temp {a.get() + b.get()};
    return temp;
}
```

In main ():

```
    Integer a{2}, b{3}, c;
    c= ~~a~~ Add (a, b);
```

step1: "temp" returned from Add() is copied into temporary obj using copy ctor.

step 2: temporary obj is assigned to "c" - using move assign.

step3: temporary obj is destroyed

In visual studio this is default behavior in debug mode. However in release mode, compiler optimise and avoid temp object.

step1: returned "temp" is assigned to "c" directly with move assign op.

Note that "temp" destructor is not called.

In a simple example:

    Integer x = 2;

Compiler does optimization and directly call param ctor for x → avoid temporary ↓
~~However~~ do copy elision.

However, it should be expanded as follows.

    Integer x = new Integer(2);

step 1: Create Integer obj (temporary) using param ctor.

step 2: Copy temporary into x using copy ctor.

step 3: destroy temporary obj

In visual studio there is no way to disable this optimization.

In Linux g++ we can disable this optimization using gcc flag.

```
g++ -fno-elide-constructors main.cpp
```

Move Semantic - std::move().

std::move() → utility header.

- always used with lvalues.
- forces compile to use move fn instead of copy.
- using with primitive type is redaundent. It does copy (not move).

```
Integer a{2};
Integer b{a}; → copy ctor
Integer c{(Integer&&) a}; → move ctor
```

- But this is not readable - casting.

```
Integer d{ std::move(a) }; → move ctor
```

- Does casting - but more readable.

To be used when object is no more needed, once desired use is done.

```
void print(Integer x){
    cout << x.get();
}
```

In main():

```
Integer a(2);
a.set(4);
Print(a); → a will be copied into x.
Print(std::move(a)); → a is moved into x.
```

- a is no more needed now.
- it will be destroyed in fn print.

Most used with unique_ptr<>.
For non-copiable classes like fstream or thread