Move Semantics

1. PreRequisites:
a. Copy constructor
b. Shallow and deep copy
c. lvalue and rvalue references

a. Copy Constructor

->It is a overloaded constructor used to declare and initialize an object from another object.

-> 2 types: Default copy constructor and user defined copy constructor.

-> Syntax of copy constructor: Class_name(const class_name &oldobject);

```
   Eg: class A{
               A(A &x)
                 {
                    } }
```

-> Copy constructor can be called in 2 ways:

```
    A a2(a1) or A a2=a1;
```

-> Example:

```
#include <iostream>
using namespace std;
class A
{
public:
int x;
A(int a) //parameterized constructor.
{
x=a;
}
A(A &i) //copy constructor
{
x=i.x;
}
};
int main()
{
A a1(20);/Calling the parameterized constructor.
A a2(a1);// Calling the copy constructor.
cout<<a2.x;
return 0;
}
```

Output: 20

-> From the above example, we can observe that copy constructor is called when we initialize the object with existing object of same class type: Example: A a2=a1. Here 2 steps take place: Object a2 is created which is of class type A and then contents of object a1 is copied to object a2.

-> Copy constructor is called when object of the same class type is passed by value as an argument.

-> Copy constructor is called when function returns the object of same class type by value.

b. Shallow copy and deep copy

b.i. Shallow copy

-> It is a process of creating the copy of an object by copying the data of member variables
-> Default copy constructor can only produce shallow copy.
-> Both the objects point to the same memory location.

Example 1: Assume you and your friend are located in 2 different places and connected to network and both of you are editing the same excel sheet. If a person updates the excel sheet other person also can see the change.

Example 2:

```
#include<iostream>
```

```
using namespace std;

class Demo
{
int a;
int b;
int *p;
public:
```

```
Demo()
      {
          p=new int;
        }
        void setdata(int x,int y,int z)
        {
          a=x;
          b=y;
          *p=z;
        }
        void showdata()
        {
          std::cout << "value of a is : " <<a<< std::endl;
          std::cout << "value of b is : " <<b<< std::endl;
          std::cout << "value of *p is : " <<*p<< std::endl;
        }
      };
      int main()
      {
       Demo d1;
       d1.setdata(4,5,7);
       Demo d2 = d1;
       d2.showdata();
        return 0;
      }
```

-> From the above example we can observe that we do not give any copy constructor. So compiler gives default copy constructor, so shallow copy takes place. Objects d1 and d2 both are pointing to same memory location. So changes made to object d1 is reflected to object d2.

b.ii. Deep copy

-> Deep copy allocates memory for the copy and then copies the contents from actual value. So both source and copy will have distinct memory locations. In this way, source and destination will not share the same memory location

-> Deep copy requires us to write user defined copy constructor.

-> Example 1: Assume you and your friend are given assignment to write. You take assignment from your friend and make changes as per your requirement. Here you and your friend have different copies. Changes made by you in your copy doesnt change the contents in your friends copy.

Example 2:

```
#include <iostream>
using namespace std;
class Demo
{
   public:
   int a;
   int b;
   int *p;

   Demo()
          {
             p=new int;
          }
          Demo(Demo &d)
          {
             a = d.a;
             b = d.b;
             p = new int;
             *p = *(d.p);
          }
          void setdata(int x,int y,int z)
          {
             a=x;
             b=y;
             *p=z;
          }
          void showdata()
          {
             std::cout << "value of a is : " <<a<< std::endl;
             std::cout << "value of b is : " <<b<< std::endl;
             std::cout << "value of *p is : " <<*p<< std::endl;
          }
       };
       int main()
       {
        Demo d1;
        d1.setdata(4,5,7);
        Demo d2 = d1;
        d2.showdata();
       return 0;
       }
```

Output:

```
  Value of a is : 4
   Value of b is: 5
   Value of *p is:7
```

-> From the above example, we can observe that 2 objects d1 and d2 are created and then we can see we have explicitly defined copy constructor, so deep copy takes place where object d1 and d2 are assigned different memory locations, so changes made to d1 will not be reflected in d2.

c. l value and r value references

c.i l value and r value:

-> l value is the one which has location in memory, r value does not have any location until it is assigned

->lvalue has two components 1) value 2) address.
rvalue has only 1) value
-> rvalue is allowed only on right hand side on assignment only lvalue is allowed on left hand side of assignment lvalue is allowed on right hand side of assignment.

Example 1: int i=10 //valid
  Here 'i' is a l value and 10 is a r value.

```
                int *y = &x   //ok
```

Here y is variable(l value) and on right side an r value produced by the address of operator.

```
                  666 = y; //error because r values cant be assigned
```

Example 2: Functions returning l values and r values.

```
//Example 2.1
Int SetValue()
{
  return 6;
}



main()
```

```
{
setValue()=3//error because setvalue returns r value which cannot be left
operand of assignment.
}

//Example 2.2

int global = 100;

int& setGlobal()
{
return global;
}

main()
{
setGlobal() = 400; //perfectly valid
}
```

-> Above example we can see that setglobal() returns reference unlike value in previous example. Reference is something that points to the exisitng memory location(global variable), which is an l value so it can be assigned. Here & is not address-of operator, it defines type of whats returned.

<u>c.ii l value to r value conversion:</u>

-> Let us understand it with some example:

Example:

```
    int x = 1;
    int y=3;
    int z=x+y;//ok
```

-> We know that x and y are l values, but when it is used with addition(+) operator, x and y have undergone implict lvalue-to-rvalue conversion.

<u>c.iii l value references:</u>

-> Let us understand the concept with some examples:

Example 1:

```
int y=10;
int& yref = y;
 yref++; //y is now 11
```

-> In above example, yref is a reference of type int&. It is a l value reference which points to y. It is possible to change value of y through yref.

Example 2:

```
int &yref = 10; // Error
```

-> It doesnt work because &yref is a reference (l value reference) which should point to existing object. But 10 is a numeric constant (without specific memory address).
-> ***Conversion from r value to l value is forbidden.*** Volatile numeric constant(rvalue) should become l value to be referenced to. If it is allowed you could alter the value of the numeric constant through its reference which is meaningless.

Example 3:

```
void fnc(int& x){
                  }
 int main()
 {
        fnc(10);// Doesnt work
  }
```

-> The above example fails because temporary r value(10) is passed to function which takes reference as argument. Invalid r value to l value conversion.

-> Fix for the above problem is we can create a temporary variable to store a r value and then pass it to function which is compilicated.

```
Eg:    int x=10
        fnc(x);
```

<u>c.iv const lvalue references:</u>

-> The above 2 example gives error as follows:

```
   error: invalid initialization of non-const reference of type
'int&' from an rvalue of type 'int'
```

-> The above error says reference not being const, namely constant.
According to language specifications, **it is allowed to bind a const l value to an r value.**

-> *const int& ref=10; // works perfectly!!*

-> Example 1:

```
    void fnc(const int& x)
      {
       }

      int main()
        {
         fnc(10); //ok
         }
```

-> The literal constant 10 is volatile and would expire, so reference to it would be meaningless, so it is better to make reference itself a constant so that value the reference points doesnt get modified.

-> The following statement:

```
      const int& ref = 10;
```

-> Compiler converts the above statement into :

```
        int __internal_unique_name = 10;
        const int& ref = __internal_unique_name;
```

c.iv lvalue references and r value references:

-> Let us understand l value and r references with an example:

```
     void printName(const string &name)//function 1(l value reference)
      {
        cout<<"[lvalue]"<<name;
       }

void printName( string &&name)//function 2(r value reference)
      {
        cout<<"[rvalue]"<<name;
       }

int main()
```

```
{
 string firstName = “Ramesh”;
string secondName = “Aggarwal”;
string fullName = firstName + lastName;
printName(fullName);// l value( calls function 1)
printName(irstName + lastName);// r value(calls function 2)
}
```

-> From the above example, we can observe that PrintName is overloaded, one is l value reference and other is r value reference. So when printName(firstName) is passed, then function 1 is called and when printName(firstName + lastName) is passed, then function 2 is called.

-> This one is very useful in move semantics which will be learnt in next section.

2. Move Semantics

-> It is a new way of moving resources around in optimal way by avoiding unnecessary copies of temporary objects based on rvalue references.

-> When we want to pass value to function, compiler copies the value and then creates a temporary object copies the content to the object and returns the temporary object to main function which is a costly affair.

-> When you want to manage memory yourself you should follow so called ***Rule of Three.***

-> Rule of three states if your class defines one or more of the folllowing methods, it should probably explicitly define all three:

2. i destructor
2.ii Copy constructor;
2.iii Copy assignment operator

-> Now let us understand Rule of three first. Rule of five will be explained in further section which is extension of Rule of three along with move assignment operator and move constructor.

2.ii Implementing Copy Constructor

-> Copy constructor is normally used to create new object from existing object and copy contents of existing object into new object

-> Example: Looks of copy constructor

```
Holder(const Holder& other)
{
  m_data = new int[other.m_size];  // (1)
  std::copy(other.m_data, other.m_data + other.m_size, m_data);  // (2)
  m_size = other.m_size;
}

main(){

Holder h1(10000); // regular constructor

Holder h2 = h1;   // copy constructor

Holder h3(h1);    // copy constructor (alternate syntax)

}
```

-> From above example, we observe that first object h1 is created by invoking a constructor. When h2=h1 statement comes, copy constructor is invoked, which creates h2 object out of existing object which is passed as reference to copy constructor. Then it then copies the data from h1 into h2(i.e from other.m_data to m_data).

2.iii Implementing assignment operator

-> Assignment operator is used to replace existing object with another existing object unlike copy constructor where new object is created and then contents of old object are copied to new object.

-> Example 1: Looks of assignment operator

```
   Holder& operator=(const Holder& other)
{
  if(this == &other) return *this;  // (1)
  delete[] m_data;  // (2)
  m_data = new int[other.m_size];
  std::copy(other.m_data, other.m_data + other.m_size, m_data);
  m_size = other.m_size;
  return *this;  // (3)
}

main()
{
Holder h1(10000);  // regular constructor
Holder h2(60000);  // regular constructor
h1 = h2;           // assignment operator

}
```

-> Basic difference we can observe between code of copy constructor and assignment is line 2, where we are using delete to wipe out current data so that it can be replaced with data from other object.

Limitations of current class design

-> Let us now analyze the limitations of current class design:

Example 2:

```
 Holder createHolder(int size)
   {
return Holder(size);
    }

  int main(){

Holder h = createHolder(1000);

    }
```

-> From the above example, we can observe that createHolder returns the Holder object by value, so compiler creates temporary object (rvalue)

-> The flow for the above program is as follows: In the main function, object h is created by invoking parameterized constructor. The object is then passed to the copy constructor, where it creates new object, allocates its own m_data pointer by copying the data from the temporary object.

-> 2 expensive operations take place: creation of temporary object and copying content from temporary object into other object.

-> Same copy assignment procedure occurs within assignment operator which wipes out memory and reallocates it from scratch by copying data from temporary object.

Eg: Holder h = createHolder(1000); //Copy constructor

h = createHolder(500); // Assignment operator

-> We can see its leading to too many expensive copies, there is a short lived object which is returned by createHolder, which is a rvalue that fades away.

-> *Why dont we steal or move the allocated data which is inside temporary object instead of making expensive copy out it? This will be discussed in next section, this is where move semantics comes to rescue!!*

Implementing move semantics with r value references

-> We have to add new versions of copy constructor and assignment operator so that they can take a temporary object in input to steal data from. To steal data means to modify object the data belongs.

-> Data can be modified using r references!!

-> Now let us discuss *Rule of Five*, which is a extension of Rule of Three seen before. New additions in Rule of Five are: *move constructor and move assignment operator.*

->Rule of three states if your class defines one or more of the folllowing methods, it should probably explicitly define all three:

i destructor
ii Copy constructor;
iii Copy assignment operator
iv. Move constructor
v. Move assignment operator

-> First three have been explained in rule of three in previous section, now we will discuss last 2 rules: Move constructor and move assignment operator.

iv. Implementing move constructor

-> Let us implement move constructor below:

```
Holder(Holder&& other)     // <-- rvalue reference in input
{
  m_data = other.m_data;   // (1)
  m_size = other.m_size;
  other.m_data = nullptr;  // (2)
  other.m_size = 0;
}
```

-> From the above example, we can observe that move constructor takes input an r value reference to another Holder object.

-> Being an r value reference, we can modify it.

-> In (1), we steal the data first and then set it to null (2). No deep copies, just we have moved resources around.

-. In (2), we set rvalue reference data to some valid state to prevent it from being accidentally deleted when temporary object dies(Holder destructor calls delete[] m_data).

v. Implementing move assignment operator

-> Let us implement move assignment operator below:

```
Holder& operator=(Holder&& other)     // <-- rvalue reference in input
{
  if (this == &other) return *this;

  delete[] m_data;         // (1)

  m_data = other.m_data;   // (2)
  m_size = other.m_size;

  other.m_data = nullptr;  // (3)
  other.m_size = 0;

  return *this;
}

int main()

{

  Holder h1(1000);              // regular constructor
  Holder h2(h1);                // copy constructor (lvalue in input)
  Holder h3 = createHolder(2000); // move constructor (rvalue in input) (1)

  h2 = h3;                    // assignment operator (lvalue in input)
  h2 = createHolder(500);       // move assignment operator (rvalue in input)
}
```

-> In the above example, (2) we steal data from other object which is coming as rvalue reference, after a cleanup of exisiting resources (1). Then we put temporary objects to some valid state (3) as we did in move constructor.

-> By observing main function, we can see that when rvalue is passed to object. Move constructor is called and when lvalue is passed to object Copy constructor is called.

-> One more point to observe in above example is that we are not able to move l values. That can also be done by using **std::move**, which is used to convert an lvalue into an rvalue.

-> Example:

```
int main()
        {
  Holder h1(1000);          // h1 is an lvalue
  Holder h2(std::move(h1));  // move-constructor invoked (because of rvalue in input)
}
```

-> Here std::move has converted lvalue h1 into rvalue. Compiler sees such rvalue and triggers move constructor. Object h2 will steal data from h1 during its construction stage.

-> We saw in previous section that we used *other.m_data=nullptr* for object h1. By doing this we can reuse h1 object.

Move Operations in STL Containers

-> Let us take a small example below:

```
std::vector<MyString> sarr(5);
MyString s1("abcdxyz");
sarr.push_back(std::move(s1));
```

-> We can observe that std::vector provides overloaded push_back operations which takes r-value references of instantiated type which prevents additional copy of resources in source object.

-> s1 need not maintain the buffer anymore once a copy of it(which is moved) is pushed on to vector

Universal References

->Let us understand universal references with small example:

```
        template<typename T>
void foo( T && t )
{
   // "T &&" is a UNIVERSAL REFERENCE
}

int i = 42;

foo( i );  // lvalue, "T &&" deduced to be "int &"
foo( 42 ); // rvalue, "T &&" deduced to be "int &&"
```

-> In above code, we can see T&& is universal reference. In c++ there are lvalue and rvalue references to distinguish between references to named and unnamed objects. T&& can bind to anything, lvalue or rvalue.

Perfect Forwarding

-> If a function templates forward its arguments without changing its lvalue or rvalue characteristics, we call it perfect forwarding.

-> Reduces excessive copying and simplifies code by reducing the need to write overloads to handle lvalues and rvalues seperately.

-> Note: The function the arguments are forwarded to can be a normal function, another template function or a constructor.

```
->  template<typename T>
void OuterFunction(T& param)
{
   InnerFunction(param);
}
```

-> In the above example, outerfunction accepts lvalue reference, so we can only pass lvalues, we cannot rvalues.

-> Following code will not compile:

OuterFunction(5); // Wont work trying to pass an rvalue to lvalue reference

-> To fix this, we could make outer function accept const lvalue reference to allow us to pass rvalues but inner function wont modify its argument.

void OuterFunction(T&& param)

-> *std::forward* does the following:

* If passed argument isnt lvalue reference. Eg: int& val, then it will return a rvalue reference.

* If an argument passed in a lvalue reference, the function returns an lvalue reference, it does nothing to the argument.

-> This means that passing in an rvalue to a template function that accepts an rvalue reference will be able to forward that argument as an rvalue to any inner functions and if an lvalue is passed and we have an overload for our inner function that accepts lvalues then that function is called instead.

-> Example:

```
class A
{
public:
A(std::string b) : b(b) {}// Copy Constructor
A(const A& other) : b(b)
{
b = other.b;
std::cout << "Copy Constructor" << std::endl;
}// Move Constructor
A(A&& other)
{
b = std::move(other.b); std::cout << "Move Constructor" << std::endl;
}
private:
std::string b;
};
//And a template function
...
template<typename T>
void OuterFunction(T&& param)
{
A a(std::forward(param));
}
...
// Passing an lvalue
A a = A("Hello");
OuterFunction(a);

// Passing an rvalue
OuterFunction(A("World"));
```

-> Here an object of typeA has its constructor called within a template function, with the move constructor, or copy constructor being called dependent on the value passed to template function.

-> When we pass variable ‘a’ to OuterFunction which is lvalue, it means the type T is deduced to A&. Given that param now looks like A&& as universal reference rule states that if lvalue is passed T&& is treated as lvalue reference.

-> Finally, param is passed to std::forward and static cast of std::forward returns lvalue reference.

-> When we pass object A(“World”) an rvalue expression to OuterFunction results in T being deduced to A.

-> Given that param now looks like A && where the universal reference is treated as a rvalue reference, so we just get A&&.

-> Finally, param is passed to std::forward and the static cast of std::forward returns an rvalue reference due to the reference collapsing rules.

Value Categories:

Lvalue, Xvalue, Prvalue, Glvalue, Rvalue.

i. lvalue: Stands for locator value. It occupies memory location

Eg: int number=7

Value which is left to = sign is lvalue and value which is right to = is rvalue.

ii. rvalue: Anything that is not an lvalue.

Eg: Eg: int number=7

Value that is on right hand side of =operator is rvalue.

Iii. Xvalue: Value that is about to expire. It can be passed to functions that expect rvalue reference (&&). Eg: e.g. casting with std::move, functions returning rvalue ref, anonymous/temporary objects

iv. Glvalue: Stands for generalized value. It can be an lvalue or an xvalue.

v. prvalue: A *prvalue* is an *rvalue* that is not an *xvalue*. It represents direct value. Eg: literals, expressions like
var++, a + b , &var, functions not returning any ref etc.

Trivial Types:

i. Trivial

A class is said to be trivial type under following conditions
-> No virtual functions or virtual base class
-> Trivial or deleted default constructor
-> TriviallyCopyable type
-> Trivial or deleted copy constructor
-> Trivial or deleted assignment operator
-> Trivial non deleted destructor
-> Trivial or delete move constructor, move assignment operator (From C++11)
-> All data members & base classes are of Trivial types

Example:

```
   *Since there are no explicit constructors,
there exists a default constructor*/
struct Trivial {
   int i;

private:
   int j;
};

/* In Trivial2 structure, the presence of the
   Trivial2(int a, int b) constructor requires
   that you provide a default constructor. For
   the type to qualify as trivial, we must
   explicitly default that constructor.*/
struct Trivial2 {
   int i;
   Trivial2(int a, int b)
   {
       i = a;
   }
   Trivial2() = default;
};
```

-> Type traits check- std::is_trivial, std::is_trivially_copyable.

ii. Standard layout

A class is said to be of Standard Layout Type, if
->No virtual functions, virtual base class
->All non static data members under same access control
->No non static data member of reference type
->All non static data members & base classes are of standard layout type
-> Type traits Check: std::is_standard_layout.

iii. POD

A class is said to be of Plain Old Data(POD) Type, if
->Trivial type
->Standard Layout type
->All non static data members are of POD type
->Type Traits Check: std::is_pod