**MEMBER POINTERS**

Data member pointer: Referencing to an offset inside a class

Member function pointer: Referencing to a (possible virtual) member function of a class

I have object, inside the object I have data member

I am referring to that data member and making this assignment

obj.\*dmptr = …;

ptr->\*dmptr = …;

Inside the object, I have this function and call them with 2 parameters.

(obj.\*fmptr)(par1, par2);

(ptr->\*fmptr)(par1, par2);

#include <iostream>  
class Date  
{  
public:  
 void set (int y, int m, int d);  
 int getYear() const { return \_year; }  
 int getMonth() const { return \_month; }  
 int getDay() const { return \_day; }

void print(std::ostream& os) const;  
 void hu();  
 void us();  
private:  
 int \_year;  
 int \_month;  
 int \_day;

int Date::\*p1;  
 int Date::\*p2;  
 int Date::\*p3;  
 char sep;  
};

These are integer of class Date pointers.

So for example p1 can point to an integer inside class Date.

So p1 can point to \_year, \_month, and \_day.

Compiler knows that p1 points to a integer of the class Date.

void Date::set(int y, int m, int d)  
{  
 \_year = y;  
 \_month = m;  
 \_day = d;  
}  
void Date::print(std::ostream& os) const  
{  
 os << this->\*p1 << sep << this->\*p2  
 << sep << this->\*p3;  
}  
std::ostream& operator<<(  
 std::ostream& os, const Date& d)  
{  
 d.print(os);  
 return os;  
}

//2017.4.20  
//4/20/2017

void Date::hu()  
{  
 sep = '.';  
 p1 = &Date::\_year;  
 p2 = &Date::\_month;  
 p3 = &Date::\_day;  
}  
void Date::us()  
{  
 sep = '/';  
 p1 = &Date::\_month;  
 p2 = &Date::\_day;  
 p3 = &Date::\_year;  
}  
int main()  
{  
 Date d;  
 d.set(2017,4,20);  
 d.hu();  
 std::cout << d << std::endl;  
 d.us();  
 std::cout << d << std::endl;  
}

void Date::set(int y, int m, int d)  
{  
 \_year = y;  
 \_month = m;  
 \_day = d;  
}  
void Date::print(std::ostream& os) const  
{  
 os << (this->\*g1)() << sep  
 << (this->\*g2)() << sep  
 << (this->\*p3)();  
}  
std::ostream& operator<<(  
 std::ostream& os, const Date& d)  
{  
 d.print(os);  
 return os;  
}

//2017.4.20  
//4/20/2017

int (Date::\*g1)() const;  
 int (Date::\*g2)() const;  
 int (Date::\*g3)() const;  
 /\*g1, g2, g3 are function pointers. These are functions inside the class of date. These functions don’t take any parameters and return integers and they are constant.\*/

};  
void Date::hu()  
{  
 sep = '.';  
 g1 = &Date::getYear;  
 g2 = &Date::getMonth;  
 g3 = &Date::getDay;  
}  
void Date::us()  
{  
 sep = '/';  
 g1 = &Date::getYear;  
 g2 = &Date::getMonth;  
 g3 = &Date::getDay;  
}  
int main()  
{  
 Date d;  
 d.set(2017,4,20);  
 d.hu();  
 std::cout << d << std::endl;  
 d.us();  
 std::cout << d << std::endl;  
}

lvalues : variables that have storage back in them.

rvalues : temporary values

string& name : if function parameter is like this, you can only send lvalues

const string& name : if function parameter is like this, you can send both lvalues and rvalues

string&& name : if function parameter is like this, you can only send rvalues

**LEFT vs RIGHT VALUE**

Assignment in earlier languages work the following way:

<variable> = <expression>, like x=a+5

In C/C++ however it can be:

<expression> = <expression>, like \*++ptr = \*++qtr;

But not all expressions are valid, like a+5 = x;

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.

int i = 42;  
int &j = i;  
int \*p = &i;

i = 99;  
j = 88;  
\*p = 77;

int\* fp() { return &i; } // returns pointer to i: lvalue  
int& fr() { return i; } // returns reference to i: lvalue

\*fp() = 66; // i = 66  
fr() = 55; // i = 55

// rvalues:  
int f() { int k = i; return k; } // returns rvalue

i = f(); // ok  
p = &f(); // bad: can't take address of rvalue  
f() = i; // bad: can't use rvalue on left-hand-side

**VALUE SEMANTICS**

Suppose s are strings:

s = s1 + s2 + s3 + s4 + s5;

returns temporary than add to s3 …

What happens to temporaries?

Since they are regular string objects, they allocate memory, they keep their data in that memory and when they die they release that memory. Why don’t we use that memory over and over again to make it more efficient? Why am I just deleting the allocated memory of that object instead of reusing again?

That is what make C++ slower than C.

In C you can reuse already allocated memory bc there are no destructors and constructors, I can do whatever I want with my memory.

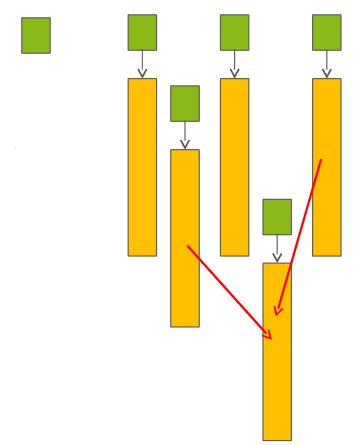
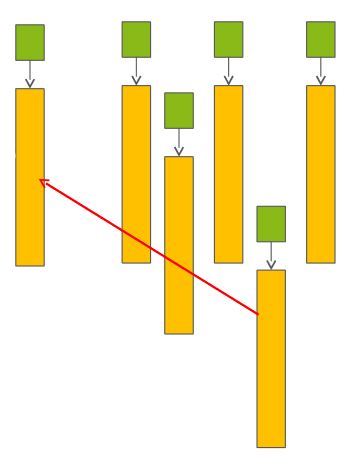
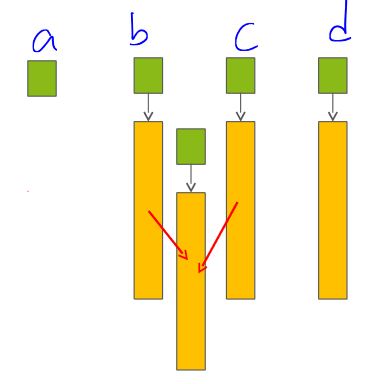
In C++ you have to create objects and delete them and if the object is temporary, you have to do this over and over again.

C++ has value semantics:

* Clear separation of memory areas
* Significant performance loss when copying large objects
* This can lead to improper use of (smart) pointers

class Array  
{  
public:  
 Array (const Array&);  
 Array& operator=(const Array&);  
 ~ Array ();  
private:  
 double \*val;  
};  
Array operator+(const Array& left,const Array& right)  
{  
 Array res = left;  
 res += right;  
 return res; //After return, res will be deleted  
}

void f()  
{  
 Array b, c, d;  
 ...  
 Array a = b + c + d;  
}

Chart

Description automatically generated



**temporary**



**temporary**

**temporary**

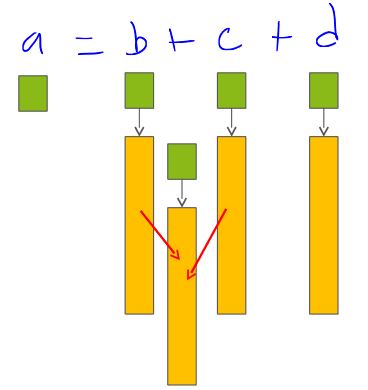
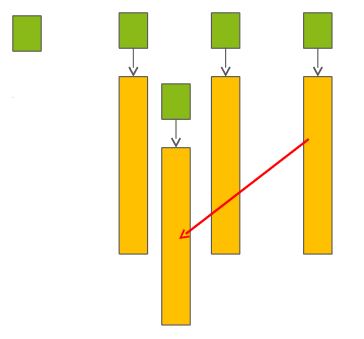
**temporary**

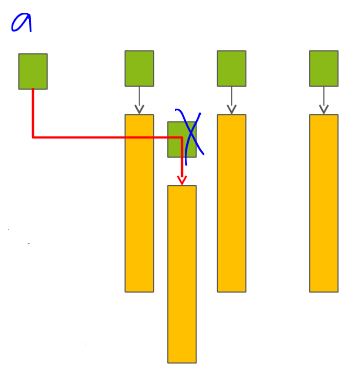
**temporary**

The thing is that I don’t have to make these 2 temporaries.

I could have done that without making temporaries and put everything inside result a.

Idea is when you make temporary, don’t let it die. Especially dynamically allocated part of it. Reuse it.

**MOVE SEMANTICS**



To achieve this, we implement following page.

class Array  
{  
public:  
 Array (const Array&); 🡪 copy constructor for lvalue objects  
 Array& operator=(const Array&);  
 ~ Array ();  
 Array& operator=(Array&&); 🡪 MOVE ASSIGNMENT (I know parameter will be rvalue)  
 Array (Array&&); 🡪 MOVE CONSTRUCTOR (copy constructor for rvalue objects)  
private:  
 double \*val;  
};

Array operator+(const Array& left,const Array& right)  
{  
 Array res = left;  
 res += right;  
 return res; //After return, res will be deleted  
}

rvalue (i.e. a+b) lvalue (i.e. a)  
Array operator+(Array&& left, const Array& right)  
{  
 left += right;  
 return left; //returns rvalue  
}

//Instead of deleting what we return, I am using its memory to store more data to it and I am returning it later.

//f() + b 🡪 OK if f returns a rvalue

void f()  
{  
 Array b, c, d;  
 ...  
 Array a = b + c + d;  
}

Move constructor is used to copy a temporary object’s or rvalue object’s data to my object. I know that whatever I am copying from is going to die soon. So I can take its memory position.

Move assignment works same. If I do this (a is Array and f returns an Array object): “a = f();”. This is move assignment bc f returns an rvalue.

a=b; 🡪 normal assignment operator

**Right Value Reference**

For overloading, we need a new type

* Reference type for performance reasons
* Overload resolution should prefer this new type on rvalue objects

void f(X& arg\_) // lvalue reference parameter  
void f(X&& arg\_) // rvalue reference parameter  
void f(const X& arg\_) // const lvalue reference parameter

X x; 🡪 object x of class X  
X g(); 🡪 function returns an X

f(x); // lvalue argument --> f(X&)  
f(g()); // rvalue argument --> f(X&&)

int a; a = 1; 🡪 here, a is an lvalue

int x;

int& getRef() {return x; }

You can also have lvalues that aren’t variables.

getRef() = 4; 🡪 getRef returns an lvalue

int x;

int getVal(){return x;}

getVal returns rvalue bc you cannot assign a value to it.

getVal();

string getName(){return “Alex”;}

getName returns a rvalue.

string name = getName();

* const string& name = getName(); 🡪 OK
* string& name = getName(); 🡪 NOT OK

**YOU CANNOT KEEP A REFERENCE TO RVALUE.**

* const string&& name = getName(); 🡪 OK
* string&& name = getName(); 🡪 ALSO OK (name is a rvalue reference)

**WE NEVER USE REFERENCES THIS WAY**

printReference(const String& str){ cout << str; } 🡪 Takes reference to a lvalue

printReference(String&& str) { cout << str; } 🡪 Takes reference to a rvalue

string me(“alex”);

printReference(me); 🡪 Calls first printReference (me is a lvalue)

printReference(getName()); 🡪Calls 2nd printReference(getName returns rvalue)

Move semantics allow us to move objects around.

We use move semantics in cases which we don’t really need to copy an object from one place to another.

class ArrayWrapper{

public:

ArrayWrapper (int n)

: \_p\_vals(new int[n]), \_size(n) { }

//copy constructor ArrayWrapper (const ArrayWrapper& other) :

\_p\_vals(new int[other.\_size],

\_size(other.\_size)

{

for (int i = 0; i < \_size; ++i){

\_p\_vals[i] = other.\_p\_vals[i];

}

~ArrayWrapper(){

if (\_p\_vals != nullptr)

delete [] \_p\_vals;

}

private:

int \*\_p\_vals;

int \_size;

};

//Change the class so that it uses temp object

class ArrayWrapper{

public:

ArrayWrapper() : \_p\_vals(new int[64]),

\_size(64) { }

ArrayWrapper(int n)

: \_p\_vals(new int[n]), \_size(n) { }

//move constructor

//copy obj that can be rvalue

ArrayWrapper(ArrayWrapper&& other) :

\_p\_vals(other.\_p\_vals),

\_size(other.\_size)

{

other.\_p\_vals = NULL;

other.\_size = 0;

}

//copy constructor

//copy obj that can be lvalue

ArrayWrapper (const ArrayWrapper& other) :

\_p\_vals(new int[other.\_size],

\_size(other.\_size)

{

for (int i = 0; i < \_size; ++i){

\_p\_vals[i] = other.\_p\_vals[i];

}

~ArrayWrapper(){

if (\_p\_vals != nullptr)

delete [] \_p\_vals;

}

private:

int \*\_p\_vals;

int \_size;

};

**MOVE SEMANTICS:**

* Instead of copying, steal the resources of other objects
* Leave the other object in a destructible state
  + Other object is still a valid object but it can be destroyed without taking away of my resources.
* Rule of 3 becomes rule of five (BIG FIVE)
  + If you don’t implement big three when it is needed, you are in trouble.
  + If you don’t implement big five, all you are going to lose is some performance.
* All standard library components were extended

**Reverse compatibility:**

* If we implement the old-style member functions with lvalue reference (BIG THREE) but do not implement the rvalue reference overloading versions we keep the old behaviour. If you like you can gradually move to move semantics.
* If we implement only rvalue operations (BIG TWO) we cannot call these on lvalues: no default copy ctor or operator= will be generated.
* So when you start coding BIG TWO, you have to code all of them.

Serious performance gain except some rare RVO situations.

All standard template library functions and classes (like vector) implement move semantics.

class X

{

public:

X(const X& rhs);

X(X&& rhs);

X& operator=(const X& rhs); // = default or = delete

X& operator=(X&& rhs);

private:

// ...

};

X& X::operator=(const X& rhs)

{

// free old resources then allocate and copy resource from rhs

return \*this;

}

X& X::operator=(X&& rhs) // draft version, will be revised

{

// free old resources then move resource from rhs

// leave rhs in a valid, destructable state

// I am gonna steal resources from rhs bc rhs will die

// This is gonna be fast for “a = f();” kinda cases where f is

// returning object of X. Whatever f returns will die soon.

// Before it dies, I take its resources using “=” operator.

return \*this;

}

*Default meaning of move constructor is making the shallow copy.*

**GENERATION OF SPECIAL MEMBERFUNC.**

1. The two copy operations (copy constructor and copy assignment) are independent. Declaring copy constructor does not prevent compiler to generate copy assignment (and vice versa). (same as in C++98)

2. Move operations are not independent. Declare either prevents the compiler to generate the other.

3. If any of the copy operation is declared, then none of the move operation will be generated.

4. If any of the move operation is declared, then none of the copy operation will be generated. This is the opposite rule of (3).

5. If a destructor is declared, than none of the move operation will be generated. Copy operations are still generated for reverse compatibility with C++98.

6. Default constructor generated only no constructor is declared. (same as in C++98)

**MOVE OPERATIONS**

For reverse compatibility, move operations are generated only when

– No copy operations are declared

– No move operations are declared

– No destructor is declared

Function templates do not considered here

– Templated copy constructor, assignment does not prevent move operation generations

– Same rule since C++98 with copy operations

#include <iostream>

#include <vector>

struct S

{

S() { a = ++cnt; }

int a;

static int cnt;

};

int S::cnt = 0;

int main()

{

std::vector<S> sv(5); //5 constructors will be called here

sv.push\_back(S()); //1 more constructor will be called

for (std::size\_t i = 0; i< sv.size(); ++i)

std::cout << sv[i].a << “ “;

std::cout << std::endl;

} 🡪 1 2 3 4 5 6

**std::move**

struct S

{

S() { a = ++cnt; std::cout << “S() ”; }

S(const S& rhs) { a = rhs.a; std::cout << “copyCtr ”; }

S(S&& rhs) { a = rhs.a; std::cout << “moveCtr ”; }

S& operator=(const S& rhs) { a = rhs.a; std::cout << “copy= ”;

return \*this; }

S& operator=(S&& rhs) { a = rhs.a; std::cout << “move= ”;

return \*this; }

int a;

static int cnt;

};

int S::cnt = 0;

template<class T>

void swap(T& a, T& b)

{

T tmp(a);

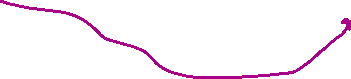


a = b;

s() s() copyCtr copy= copy=

b = tmp;

}



int main()

{

S a, b;

swap( a, b);

}

In main 2 no parameter constructor will be called 🡪 S() S()

Parameters of swap function are lvalue references so no copy constructor will be called.

At “T tmp(a);” line, copy constructor will be called bc a is lvalue.

At “a = b;” line, copy assignment will be called bc b is lvalue.

At “b = tmp;” line, copy assignment will be called bc tmp is lvalue.

We are gonna tell the compiler that make this tmp an rvalue object. For that, we use std::move.

struct S

{

S() { a = ++cnt; std::cout << “S() ”; }

S(const S& rhs) { a = rhs.a; std::cout << “copyCtr ”; }

S(S&& rhs) { a = rhs.a; std::cout << “moveCtr ”; }

S& operator=(const S& rhs) { a = rhs.a; std::cout << “copy= ”;

return \*this; }

S& operator=(S&& rhs) { a = rhs.a; std::cout << “move= ”;

return \*this; }

int a ;

static int cnt;

};

int S::cnt = 0;

template<class T>

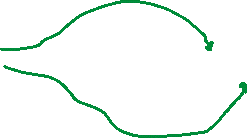
void swap(T& a, T& b)

{

T tmp(std::move(a));

a = std::move(b);

b = std::move(tmp);



s() s() moveCtr move= move=

}

int main()

{

S a, b;

swap( a, b);

}

T tmp(std::move(a)); 🡪 //treat a as an rvalue object

**std::move(x)**

* Right value reference cast
* Usually has positive effect of performance
  + Many standard lib function utilize right-value references
* Sometimes we have to use it
  + Movable non-copyable classes
  + std::unique\_ptr, std::fstream, std::thread
* Might be dangerous
  + A variable with name left with unspecified value

For example you have vector : “vector<int> v(100);” and another v2. Then you want to push back v to v2 and you know that you are not gonna use v after that, it’s gonna die: “v2.push\_back(move(v));”. Treat v as a rvalue, so you can steal its resources bc you are not gonna use v after this point.

Very dangerous bc after that, if you keep using v, you’ll be in trouble. BUT v2 will not make any copies of it. v2 will use it as original.

#include <iostream>

#include <vector>

struct S

{

S() { a = ++cnt; std::cout << “S() ”; }

S(const S& rhs) { a = rhs.a; std::cout << “copyCtr ”; }

S(S&& rhs) { a = rhs.a; std::cout << “moveCtr ”; }

S& operator=(const S& rhs) { a = rhs.a; std::cout << “copy= ”;

return \*this; }

S& operator=(S&& rhs) { a = rhs.a; std::cout << “move= ”;

return \*this; }

int a;

static int cnt;

};

int S::cnt = 0;

S() S() S() S() S() S() moveCtr

int main()

{

copyCtr copyCtr copyCtr copyCtr copyCtr

*copyCtrs are because it looks like inside vector, it is moving things around.*

std::vector<S> sv(5);

sv.push\_back(S());

for (std::size\_t i = 0; i < sv.size(); ++i)

std::cout << sv[i].a << “ “;

std::cout << std::endl;

}

S() returns an rvalue and it goes away. Since push\_back has to make a new object, it makes that object using that rvalue. That is where moveCtr comes from.

void push\_back (const value\_type& val);

void push\_back (value\_type&& val); 🡪 steals the internals of whatever is

pushed, takes rvalue reference

I expected that inside vector, it would use moveCtr, why copyCtr?

If you promise your move constructor doesn’t throw an exceptions, you can

satisfy this need by:

S(S&& rhs) noexcept { a = rhs.a; std::cout << “moveCtr ”; }

Now you get:

S() S() S() S() S() S() moveCtr moveCtr moveCtr moveCtr moveCtr moveCtr

If you have class (ArrayWrapper) and 2 members : \_p\_vals (int pointer) and \_metadata (another class object). Your move constructor is like this:

ArrayWrapper(ArrayWrapper&& other)

: \_p\_vals(other.\_p\_vals)

, \_metadata(other.\_metadata)

{

other.\_p\_vals = NULL;

}

Problem is this: I am gonna steal other object’s pointer (just copy it) but when I am stealing the other object’s metadata, which is a large object, I am doing a copy (copyCtr is called for \_metadata). That is not efficient. Meaning of move constructor is supposed to be fast and I know that other is gonna die. So I just move it just like this:

ArrayWrapper(ArrayWrapper&& other)

: \_p\_vals(other.\_p\_vals)

, \_metadata(std::move(other.\_metadata))

{

other.\_p\_vals = NULL;

}

Now my move constructor for metadata will work which is:

MetaData (MetaData&& other)

: \_name( std::move(other.\_name) )

, \_size( other.\_size )

{ }

*//CONCEPT IS IMPORTANT, NOT FOCUS ON DETAILS*

How does std::move work?

It is just typecasting stuff. It is not expensive. It is just telling the compiler that believe me this is an rvalue. If you are doing it right, then you should be fine.

STL automatically use std::move for advantage.

For your functions, if you are getting some objects through reference, you can overload your functions with the move overloads like push\_back.

**RVO**

std::vector<double> fill();

int main()

{

std::vector<double> vd = fill();

// ...

}

std::vector<double> fill()

{

std::vector<double> local;

// fill the elements

return local; // return or move?

}

If I make a local object and return it, you don’t wanna kill local object inside the stack, this is gonna create some problems for you. So be careful about this kind of stuffs: “Should you return or move?”. This is called RVO problem.

If object is not killed after we’ve done with move assignment or move constructor, we would be in trouble.