→ Smart pointers in C++ :-

The problem with normal pointers is that that the memory allocated by normal pointers are not deallocated automatically but you have to explicitly dellocate them.

If you forget to do so, you get memory leak, which may cause programm to crash.

for now we make a class for this smart pointer but these are inbuilt class for this, which we will learn about later.

In this class • pointer is created and automatically destroyed.

ex.-
```
#include<iostream>
using namespace std;

class SP{
    int *ptr;
Public:
    SP(int *p = NULL){ptr=p;}
    ~SP(){delete ptr;}
```

```
    int & operator *()
    {
        return *ptr;
    }
};

int main()
{
    SP p(new int());
    *p = 20;
    cout << *p;
    return 0;
}
```

We can also use template for this.

NOTE:- To call a destructor for a dynamically allocated object you need to call delete, so if you have a dynamically allocated object of class object with destructor, but if object go out of scope destructor will not invoked automatically you need to use delete keyword for that object to invoke destructor, or use

this method.

Wrong method destructor will not called:-

```
class test{
    public:
        int x, y;
        test(int a=, int b=0)
        {
            x = a;
            y = b;
            Cout << " constructor called \n";
        }
        ~test()
        {
            Cout << "Destructor called \n";
        }
};
int main()
{
    Cout << "main begins \n";
    {
        test * P = new test(10, 20);
    }
    Cout << " main ends";
    return 0;
}
```

Output:- main begins
constructor called
main ends

Right method destructor called:-

```
#include<iostream>
Templet <class T>
class SP{
   T * ptr;
   Public:
    SP(T * P = NULL){ Ptr =P;}
   ~SP(){ delete. ptr;}

     T & operator *(){ return *ptr;}
     T * operator ->(){ return ptr;}
};

class test{
   Public:
    int x, y;
    test(int a=0, int b=0){
            x = a;
            y = b;
    }

   ~Test(){
        cout<<" destructor called";
    }
   void show()
   {
      cout<< given values are"<< a
          <<"and"<< b <<endl;
};
```

```
int main()
{
    SP(new Test(3,10));
    P->show();
    return 0;
}
```

But there is another problem with this self made smart pointer, if we have two objects of this smart pointer class pointing to same address and if one of them erased/deleted address associated with both pointers will be deleted as both are pointing to same object, to solve this problem we have check if other pointer are not pointing to same address then only delete pointer, we can do so by keeping reference count or use some built in smart pointers of C++.

here is an example of problem caused by using self made smart pointer class:-

```
int main()
{
    int *ptr1 = new int(10);
    {
        int *ptr2 = ptr1;
        sp<int> sp(ptr2);
    }
    cout << *ptr1;
    return 0;
}
```

In this example as you can see ptr1 is normal pointer and ptr2 is smart pointer and both are pointing to same address of integer type. as scope of sp object end ptr2 deleted and it also deallocate the memory pointed by ptr1 also. So, if we want to access ptr1 we get error.

So, we have to use other methode.

→ Unique_ptr, Shared_pointer and Weak_ptr :-

They are built in smart pointer class in C++.

→ Unique_ptr :-

If you have a unique pointer pointing to an object, then other pointer can't point to that object/address, so, there can be only one unique ptr pointing to same/one address.

But we can move it to some other unique pointer, to point to same address but more than one can't point to that simultaneously, so first one will stop pointing to that.

NOTE :- 9+ is recommended to use unique_ptr over shared_ptr and weak_ptr.

Syntax:-

unique_ptr<classname> P = make_unique<class>(arguments of objects);

OR

unique_ptr<class> P(new {{class}}(arguments));

From both method first one is recommed as it is easy to read than second one.

To make second unique pointer to point to already pointing address of another unique pointer we use `move` keyword.

unique_ptr<class> P1 = make_unique<class>(arguments);

unique_ptr<class> P2;
// here second unique pointer is made
// but it is not pointing to any
// object, so, no constructor invoked.

P2 = move(P1);

→ Shared_ptr :-

By using shared_ptr you can have more than one shared_ptr pointing to same address. Shared_ptr class keep count of reference count (i.e. how many pointers pointing to that add.), so, pointers will destroy only if reference count become zero.

You can check the reference count by using use_count() method.

Syntax:-

```
Shared_ptr <class> p= make_shared <class>
                      (arguments);
```

Or,

```
shared_ptr <class> p=(new {class}(arguments));
```

To make second shared pointer to point to same add. you can simply use =.

```
Shared_ptr <class> P2;
    P2 = P;
```

→ Weak pointers :- / weak_ptr

weak pointers can only be make for shared_ptr objects. They can't itself own any objects. They are made to point to already existed shared_ptr object.

We used weak_ptr when we want temporary pointer to that object or we want a pointer to that object but don't want to increase the reference count. As, weak_ptr are not counted in reference count.

We make weak_ptr and assign it to shared pointer. So, there is no make_weak like function.

```
weak_ptr <class> P1;
shared_ptr<class> P2 = make_shared <class> (arg);
P1 = P2;
```

NOTE :- As we know weak_ptr doesn't increase reference count, so if reference

Count of shared pointer at some object becomes zero, it destroyed even if weak ptr is ~~pointi~~ still pointing.

There is another function associated with weak_ptr, it checks whether object pointed by weak_ptr is still valid or destroyed, this function is 'expired();'

```
weak_ptr <class> P1;
{
shared_ptr<class> P2 = make_shared<class>(10);
  P1 = P2;
cout<<P1.use_count()<<endl;
}
cout<< P1.expired();
```

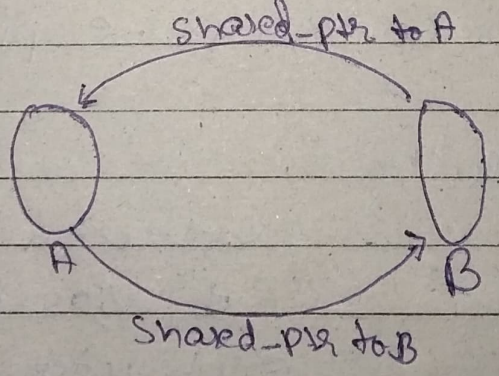
Output:-

```
constructor called //by P2
1 //as only P2 is counted not P1
destructor called //as P2 is dest so.
                  //reference count=0
1 //as P2 is destroyed/expired
```

Weak_ptr is used to create temporary pointer and they are always used with shared pointer.

Another work which weak_ptr serve is, if we have two objects with shared pointer both pointing to each other, now here problem is, we can't destruct any one them as, if one of them is destroyed both will destroy as other one has the hold of first one.

To solve this problem if we can use weak_ptr to point to one of these object.



both of use_count = 2, so can't delet any one them.

NOTE :- While using copy constructor always write your own copy constructor as default copy constructor only do sallow copy which causes problem when your class has pointer variable.

ex -

```
#include <iostream>
using namespace std;

class Test{
    int *ptr;
public:
    Test(int x){
        ptr = new int(x);
    }
    Test(const Test &t){
        int val = *(t.ptr);
        *ptr = new int(val);
    }
    void set(int x){
        *ptr = x;
    }
    void print(){ cout << *ptr << "\n"; }
};
```

```
int main()
{
    Test t1(10);
    Test t2(t1);
    t2.set(20);
    t1.print();
    t2.print();
    return 0;
}
```

You write this programme twice, one with self written copy constructor once without you can see the difference.