317

weak_ptr must be converted to shared ptr inorder to access the referenced Object using lock() method.

wpl.lock() return a shared pointer if the Object is valid or empty shared pointer if Object is not valid

This like

expired() ? shared_ptr<T>() : shared_ptr (*this)

eg:

weak_ptr<myclass> wpl = spl

↓

creating a wkptr

Now

class A

{ weak_ptr<A> wpl;

}

sp1(new A)

sp2(new A)

sp1->setAdj(sp2)

sp2->setAdj(sp1) } — There will not be cyclic reference and Object will be deleted.

wpl->ptr() X NOT work first get shared_ptr from it and then call

wpl->lock()->ptr()

why custom deleter?

```
// legacy code
class A
{
  ...
  void release() {...}
  ~A() {...}
};

A *aptr = new A();
...
aptr->release();
delete aptr;
```

why we need to call release before deleting the ptr

legacy code so we do not have access to change it

what is the problem here?
[If we use smart pointer here then this release method will never get called it will call only default destructor

Custom deleter for shared_ptr

↓

specified in the constructor of smart pointer

```
shared_ptr<A> spA(new A(), &deleteA);
```

If we not provide our own deleteA then it will call default delete operator which will A destructor and release() method will not be called.

deleter can be

- a fn pointer
- A lambda fn
- functor

(319)

Ex. deleter function using for pointer

```
void deleteA(A* ptr)
{
    cout<<"Deleting"<<endl;
    if(ptr)
    {
        ptr->released; // we Need to
        delete ptr;
    }
}
```

explicitly call release method in the destr.

custom deleter

```
shared_ptr<A> sp(new A(),
                 &deleteA);
```

custom deleter using lambda exp.

```
shared_ptr<A> sp(new A(),
    [](A* ptr) { cout<<"Deleting"<<en
                 d;
    if(ptr)
    {
        ptr->released;
        delete ptr;
    }
});
```

Custom deleter using functor

```
class Del
{ public:
    void operator()(A* ptr)
    { cout << "Deleting" << endl;
      if(ptr)
      { ptr->release();
        delete ptr;
      }
    }
};

shared_ptr<A> spA(new A(), Del());
```

custom deleter for unique_ptr

specified in the constructor. deleter type will be part of unique_ptr type

```
unique_ptr<A, function<void(A*)>>
p(new A(), [](A* ptr) { cout <<
"Deleting" << endl;
    delete p;
});
```

(321)

Reference → it is like alias of b

```
int a=10                      b is a reference of a
int &b= a

                              int b
int &a; X                     int &a = b

int &j = 5 X                  const int &j=5

int &i = (j+k) X              not
```

exp computation in temp which is kind of const.

```
const int &i= j+k;
```

Call by ref.

```
void call_by_val( int &b, int c)
{
    cout<<b<<&b<<endl;
    cout<<c<<c<<endl;
}

int main()
{   int a=10, b=20
    call_by_val( a, b)
```

value of b is copied to value of c

b and a will start referring to same location

Ref

* We cannot have a assigned reference, or ref to literal or ref to exp.
* We can a const ref to literal and expression.

Ex. 2

```
int ret_const (const int &x)
{
    ++x   X
    int return (x+1) allowed
}
```

return by value

```
int returnbyval(int &x)
{
    return (x)
}
int main()
{
    int a = 10
    int b = returnbyval (a) ✓
    int &b = return byval(a) X
```

since while returing value x is getting copied to temporary and and it is expression which is constant and we cannot have a reference to exp so make (b) as const ref.

```
const int &b = returnbyval
```

(323) return by reference

int & return_by_ref(int &x)
{
    return(x) → It is not returning the value of x but reference of it i.e address of x i.e ref of x which is an alias of x.
}

int a=10

const int &b= return_by_ref(a);

int &b= return_by_ref(a)

b= return_by_ref(a)

so b now becomes an alias of x.

* while return by ref we should not return ref of local variable it is unpredictable

ex.2

int & return_ref(int &x) { return(x); }

int a=10, b

b= return_ref(a)

return_ref(a)=3

a= cout a=

O/P 3

Ex. 2

```
int & return_ref( int &x)
{
    int t = a
    ++t;
    return (t);
}
int a=10, b;
b= return_ref(a)
return_ref(a)=3
```

⊗ unpredictable value

We should always return ref or live variable like

(1) heap
(b) static, global

```
int a = 10, &b=a;
&b=c
```

In C++ reference stores the add but there is no way to access hold of that add.

The reference in Java and C++ has a different.