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

`wp1.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> wp1 = sp1
⇓
creating a wkptr

Now

```
class A
{
    weak_ptr<A> wp1;
}

sp1(new A)
sp2(new A)

    sp1->setAdj(sp2)
    sp2->setAdj(sp1)
```

— There will not be cyclic reference and Object will be deleted.

wp1->print() X Not work first get shared_ptr from it and then call

`wp1->lock()->print()`

why custom Deleter?

```
//legacy code

    class A
{
        ...
        void release () {...}
        ~A() {..}
    };

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

(why we need to call more fn before deleting the ptr

→ legacy code so we do not force to change it

what is the problem here?
If we use smart pointer here then this release method will ~~not~~ never get call it will call only default destr

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 on A object delete obj; It will call A destructor and release() 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->release(); // we Need to
          delete ptr;
        }       explicitly call
}
        release method in the destror.
```

custom deleter

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

what

custom deleter using lambda exp.

```
shared_ptr<A> spA(new A(),
      [](A* ptr) { cout<<"Deleting"<<e
                  nd;
       if(ptr)
       { ptr->release();
            delete ptr;
       }
       });
```

Custom deleter using functor (320)

```
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
int &b= a
```

b is a reference of a

```
int &a; X
int &j= 5 X
int &i= (j+k) X
```

```
int b
int &a = b
```

~~int&~~ const int &j=5

~~Not~~

exp computation in temp which is kind of ~~refere~~ 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 refering to same loc

Ref

- we cannot Have a ~~reference~~ unassigned reference, or ref to literal or ref to exp.
- We can a const ref to library and ~~ex~~expression.

ex. 2

```
int ref_const (const int &x)
{    ++x ✗
}    but return (x+1) allowed
```

return by value

```
int returnbyval( int &x)
{.
     return (x)
}
int main()
{.   int a=10
     int b= return byval (a); ✓
     ~~int~~ &b= return byval(a) ✗
```

since while returning value x is getting copied to temporary and and it is express which is constant. and we can not have a reference to exp so make (b) as const ref.

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

(323) return by referenc

int & return_by_ref(int &x)
{
return(x) → It is not returning the value of x but reference of of x i.e add 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

324

ex.2

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

vaui

* We should always return ref of live variable like

① heap

ⓑ static, global

c

int a= 100, &b=a' ✓

&b=c X

* In c++ ~~retu~~ reference stores the add ~~by~~ but there is no way to catch a hold of that add.

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