Item 18 :- Make Interface easy to use correctly & hard to use incorrectly.

Date:

→ Each interface is a means by which clients interact with your code.

```
Investment *createInv();
```

→ Client can perform 2 types of errors
① failure to delete a ptr
② deletion of same ptr more than once

```
shared_ptr<Investment> createInv();
```

→ This forces clients to store the return value in shared_ptr eliminating possibility of forgetting to delete the Investment object.

We have seen before that shared_ptr allows a resource release functn called "deleter" (auto_ptr does not have such capability).

what if clients want to pass that pointer to a function getRidOfInvestment.

instead of using delete.

New error will occur.

Date:

Shared-ptr constructor takes 2 arguments deleter to be called when reference count goes to zero, with getRidofInv as its deleter.

```
tr1::shared_ptr<Investment>
    pInv(0, getRidofInv);
```

// 0 is not a pointer, won't compile.

```
tr1::shared_ptr<Investment> pInv(static_cast
    <Investment*>0, getRidofInv);
```

=> Shared-ptr uses it per-pointer deleter to avoid cross DLL problem

This problem happens when an object is created using one dynamically linked library (DLL) but its deleted in different DLL.

Shared-ptr avoid this, since its default deleter uses delete from same DLL where it is created.

Date:

Example- if Stock class is derived from Investment.

```
tr1::shared_ptr<Investment> createInv()
    { return tr1::shared_ptr<Investment>(new Stock);
    }
```

Thus, using Shared_ptr can eliminate many client errors.

=> BOOST shared_ptr is twice the size of raw pointer, uses dynamically allocated memory for bookkeeping and deleter specific data, uses virtual function call when invoking its deleter.

Note =>

Ways to prevent errors include creating new types, restricting operations on types, constraining object values & eliminating client resource mgmt responsibilities.

Shared_ptr uses custom deleters prevents cross DLL problem.