Static Members
- Variables – not part of the object, belong to the class, only one copy exist
- Functions – required only in declaration, belong to the class, do not receive this pointer, cannot access non static member functions, can be invoked directly through the class name

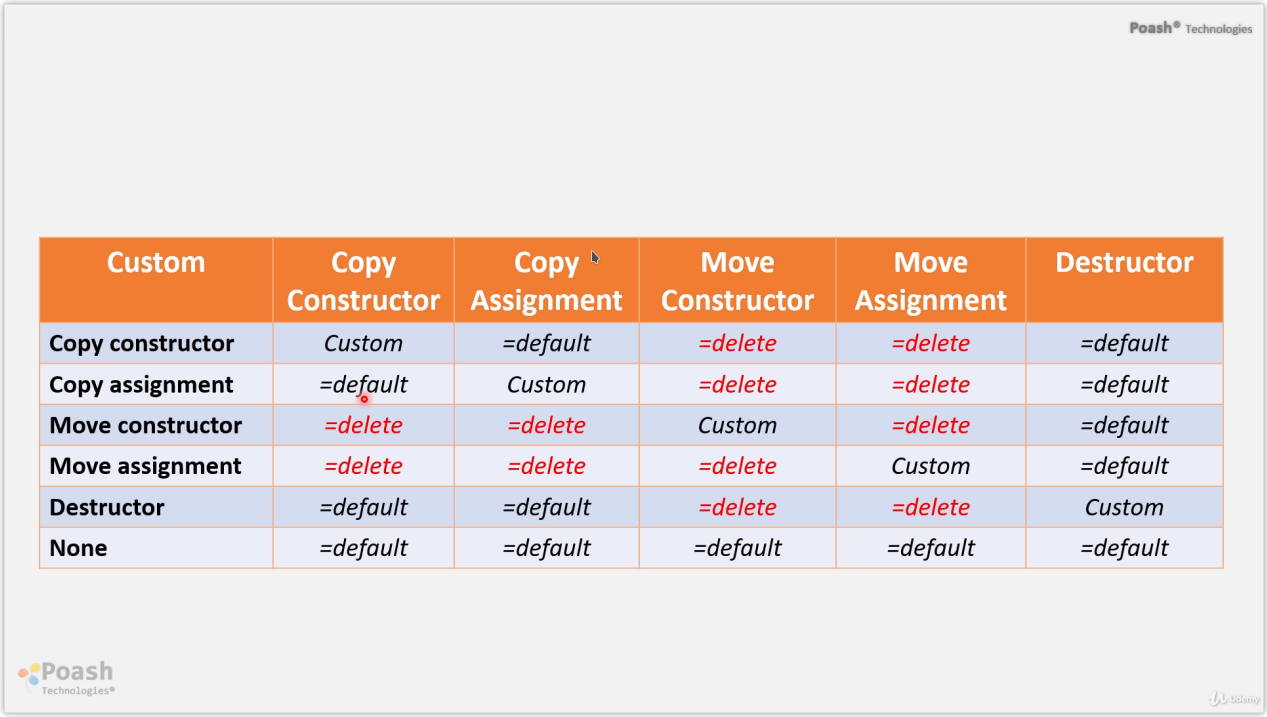
Constructors
- **Rule of 3** – All should be defined in a user implements any of them – **Destructor, Copy Constructor, Copy Assignment operator**
- **Delegating constructor –** constructor that invokes other constructor/constructors of the class

Copy of object
- **Shallow copy –** copy the address of pointer only
- **Deep copy –** allocate new address and then copy the value of the pointer

Rule of 5
- If class has ownership semantics, then you must provide a user-defined:
- **Destructor, Copy constructor, Copy assignment operator, Move constructor and Move assignment operator**

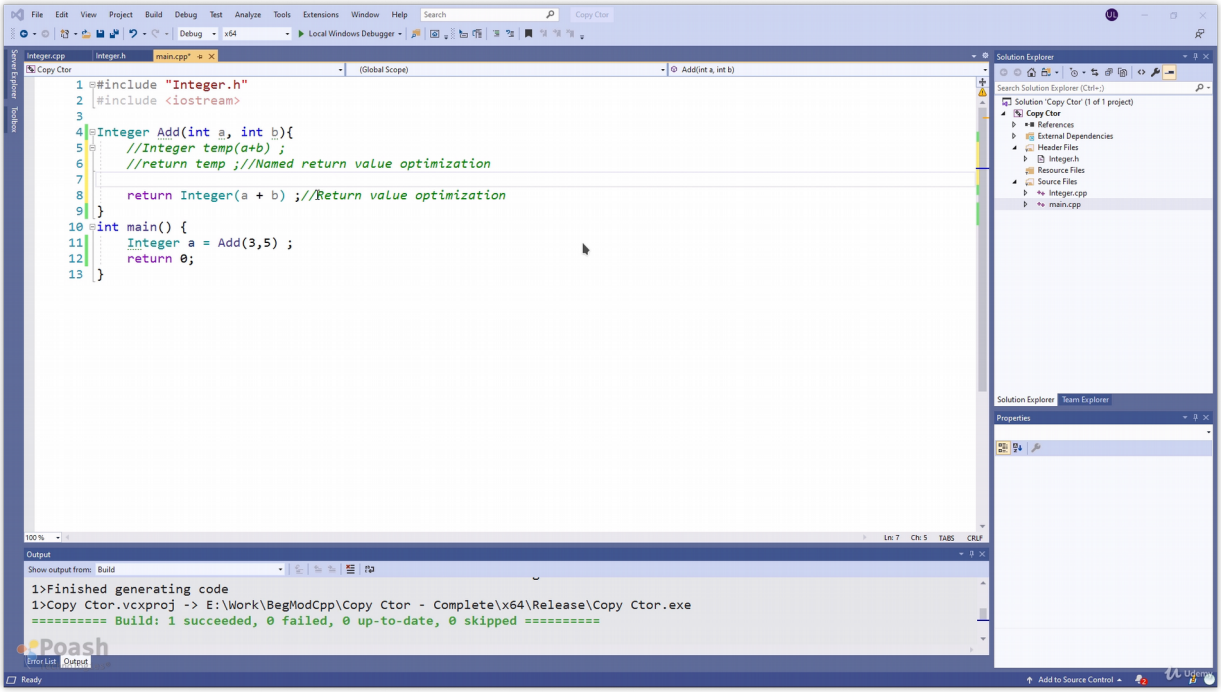
- If any copy constructor/copy assignment operator or destructor is defined – move operations will not be created implicitly.

- if any move operation is defined – all other move operations and copy operations will be deleted

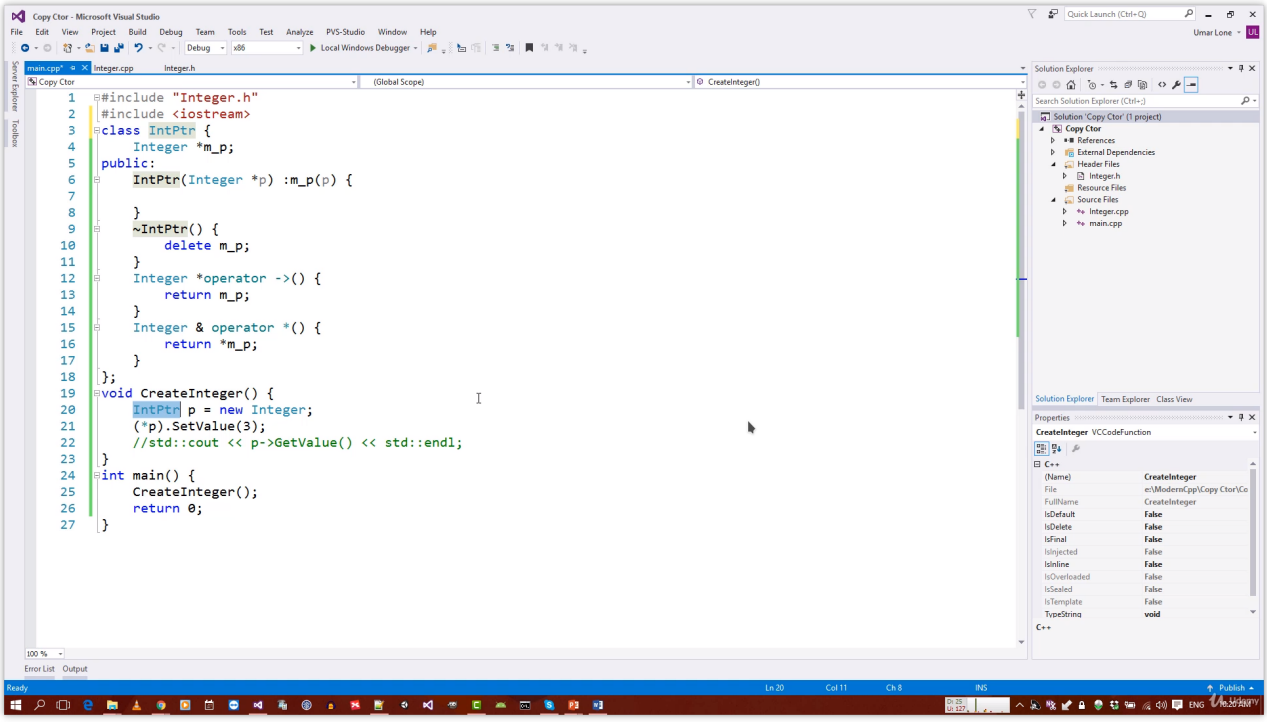


Custom	Copy Constructor	Copy Assignment	Move Constructor	Move Assignment	Destructor
Copy constructor	*Custom*	*=default*	*=delete*	*=delete*	*=default*
Copy assignment	*=default*	*Custom*	*=delete*	*=delete*	*=default*
Move constructor	*=delete*	*=delete*	*Custom*	*=delete*	*=default*
Move assignment	*=delete*	*=delete*	*=delete*	*Custom*	*=default*
Destructor	*=default*	*=default*	*=delete*	*=delete*	*Custom*
None	*=default*	*=default*	*=default*	*=default*	*=default*

Copy/move elision



RAII principle – **resource acquisition is initialization –** The basic idea is to represent a resource by local object, so that the local object constructor will allocate the resource, and the local object destructor will release the resource. The problem is solved by smart pointers.

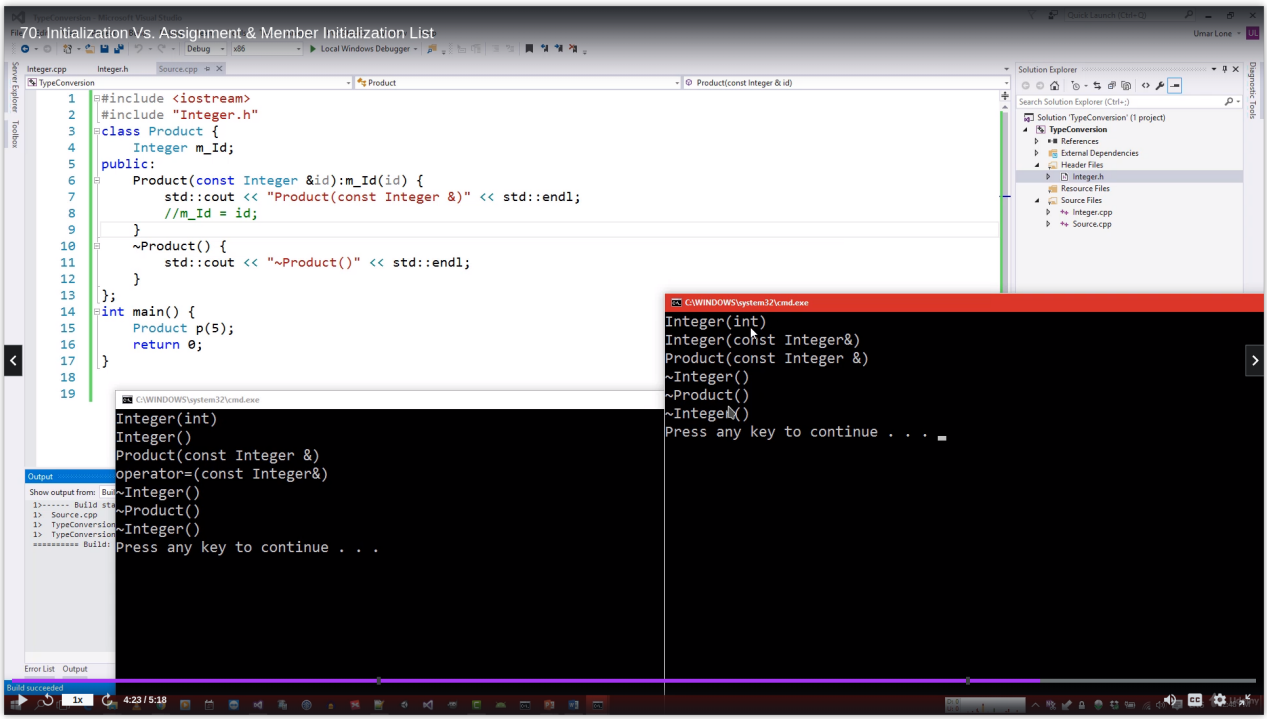


Type conversions

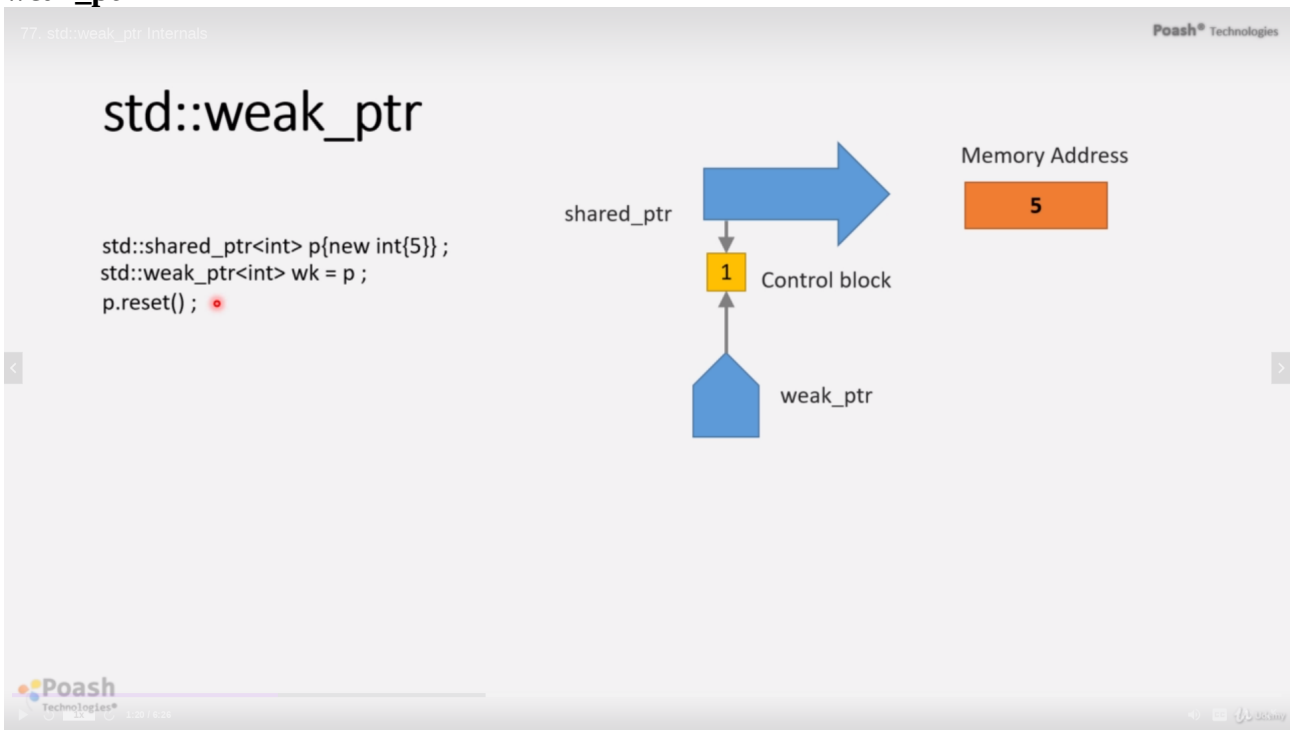
– Always prefer static_cast(c++ cast) over C-cast, because **C-cast do not check for the validness of the cast**.

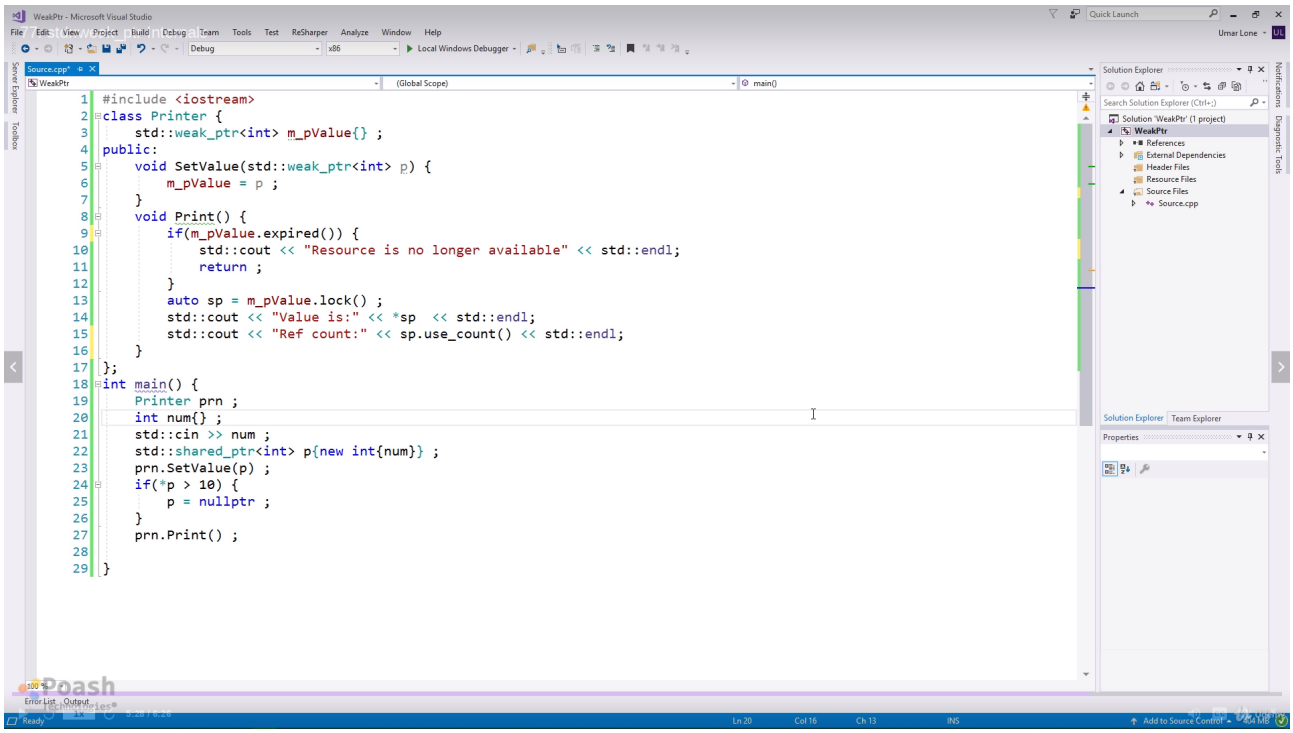
- If we do not want constructor to be invoked implicitly due to type conversion (Integer a = 5), we have to **mark the constructor with keyword explicit**.

- Always prefer to use std::initializer_list to initialize members of the class. It allow initialization of member variables without assignment.

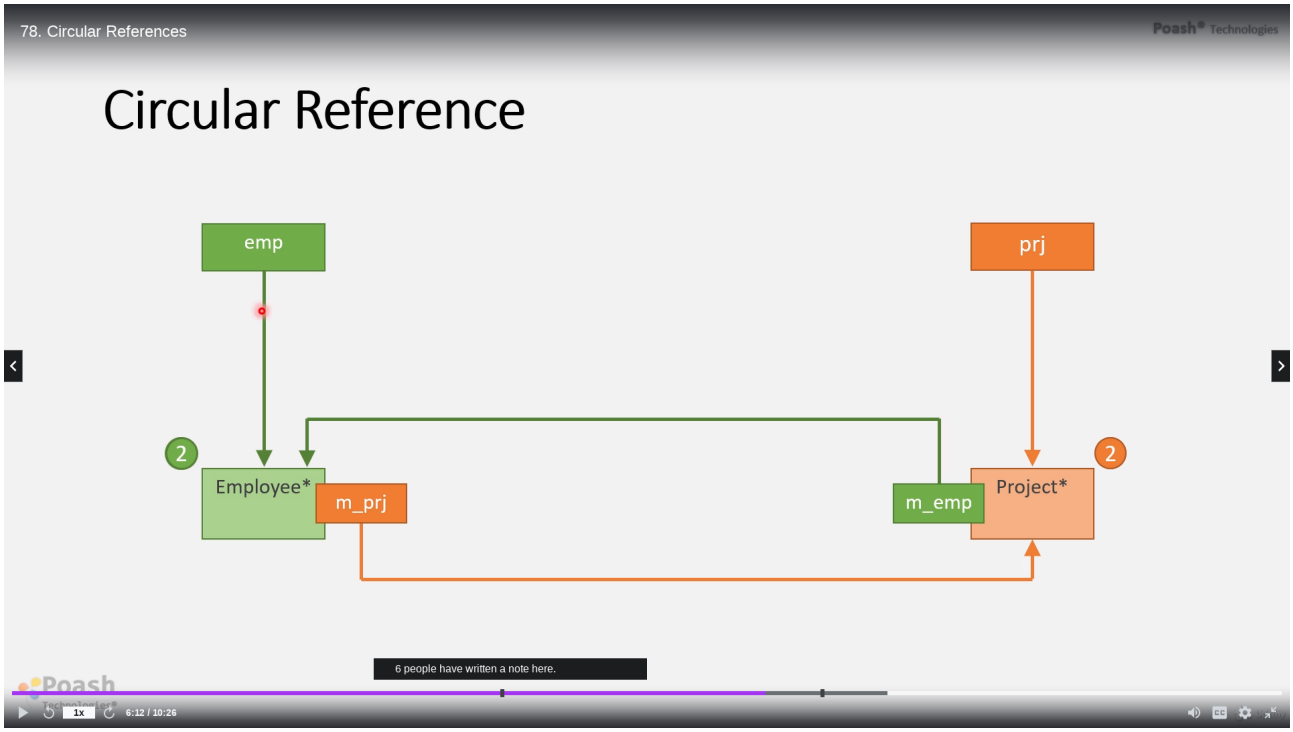


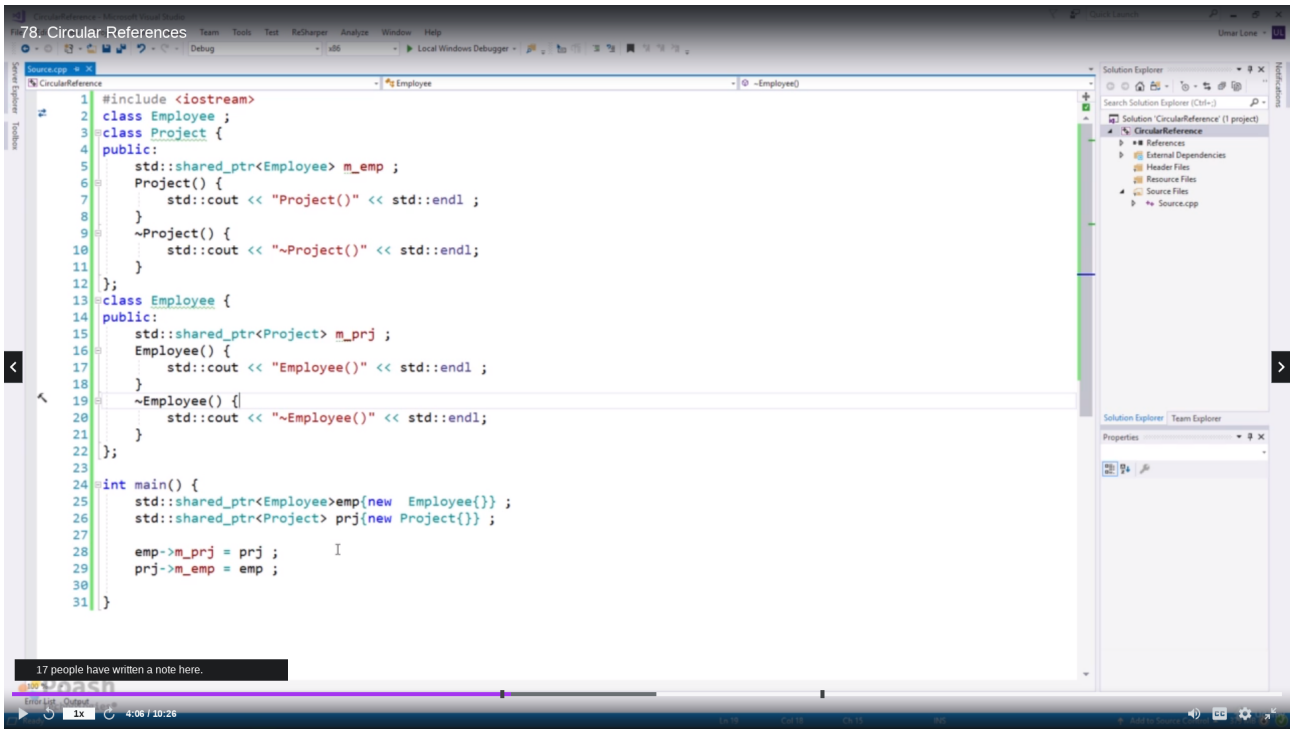
weak_ptr


77. std::weak_ptr Internals
Poash® Technologies
std::weak_ptr
std::shared_ptr<int> p{new int{5}} ;
std::weak_ptr<int> wk = p ;
p.reset() ;
shared_ptr
Memory Address
5
1
Control block
weak_ptr
Poash Technologies


WeakPtr - Microsoft Visual Studio
Source.cpp
#include <iostream>
class Printer {
std::weak_ptr<int> m_pValue{} ;
public:
void SetValue(std::weak_ptr<int> p) {
m_pValue = p ;
}
void Print() {
if(m_pValue.expired()) {
std::cout << "Resource is no longer available" << std::endl;
return ;
}
auto sp = m_pValue.lock() ;
std::cout << "Value is:" << *sp << std::endl;
std::cout << "Ref count:" << sp.use_count() << std::endl;
}
};
int main() {
Printer prn ;
int num{} ;
std::cin >> num ;
std::shared_ptr<int> p{new int{num}} ;
prn.SetValue(p) ;
if(*p > 10) {
p = nullptr ;
}
prn.Print() ;
}
Solution Explorer
Poash Technologies

Circular reference – when two object point to each other. When using shared_ptr this cause a memory leak. Insted you should use weak_ptr(in one of the objects at least) to avoid the problem.





Smart pointers – Deleter - Why and when would we need something like that?
- In C where, when you wrap FILE*, or some kind of a C style structure free(), custom deleters may be useful.
- In order to fully delete an object sometimes, we need to do some additional action. What if performing “delete” (that smart pointers do automatically)is not the only thing which needs to be done before fully destroying the owned object.

Example for shared_ptr using lambda for default deleter:

```
std::shared_ptr<MyType> sp(new int{10}, [](int *p) { delete p; });
```

Example for unique_ptr using lambda for default deleter:

```
auto deleter = [](MyType*){ ... }
std::unique_ptr<MyType, decltype(deleter)>> u1(new MyType(), deleter);
```