

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

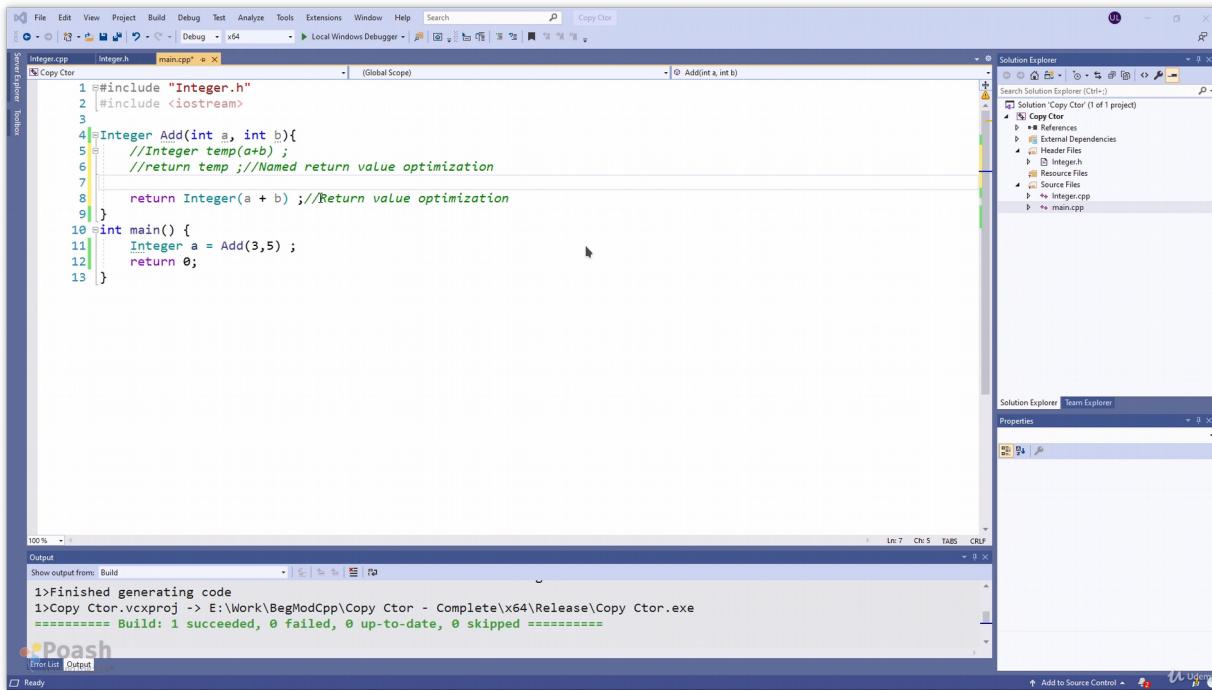
- If class has ownership semantics, then you must provide a user-defined:
 - **Destructor, Copy constructor, Copy assignment operator, Move constructor and Move assignment operator**
 - **If any copy constructor/copy assignment operator or destructor is defined – move operations will not be created implicitly.**
 - **if any move operation is defined – all other move operations and copy operations will be deleted**

Poash® Technologies					
Custom	Copy Constructor	Copy Assignment	Move Constructor	Move Assignment	Destructor
Copy constructor	<i>Custom</i>	<i>=default</i>	<i>=delete</i>	<i>=delete</i>	<i>=default</i>
Copy assignment	<i>=default</i>	<i>Custom</i>	<i>=delete</i>	<i>=delete</i>	<i>=default</i>
Move constructor	<i>=delete</i>	<i>=delete</i>	<i>Custom</i>	<i>=delete</i>	<i>=default</i>
Move assignment	<i>=delete</i>	<i>=delete</i>	<i>=delete</i>	<i>Custom</i>	<i>=default</i>
Destructor	<i>=default</i>	<i>=default</i>	<i>=delete</i>	<i>=delete</i>	<i>Custom</i>
None	<i>=default</i>	<i>=default</i>	<i>=default</i>	<i>=default</i>	<i>=default</i>

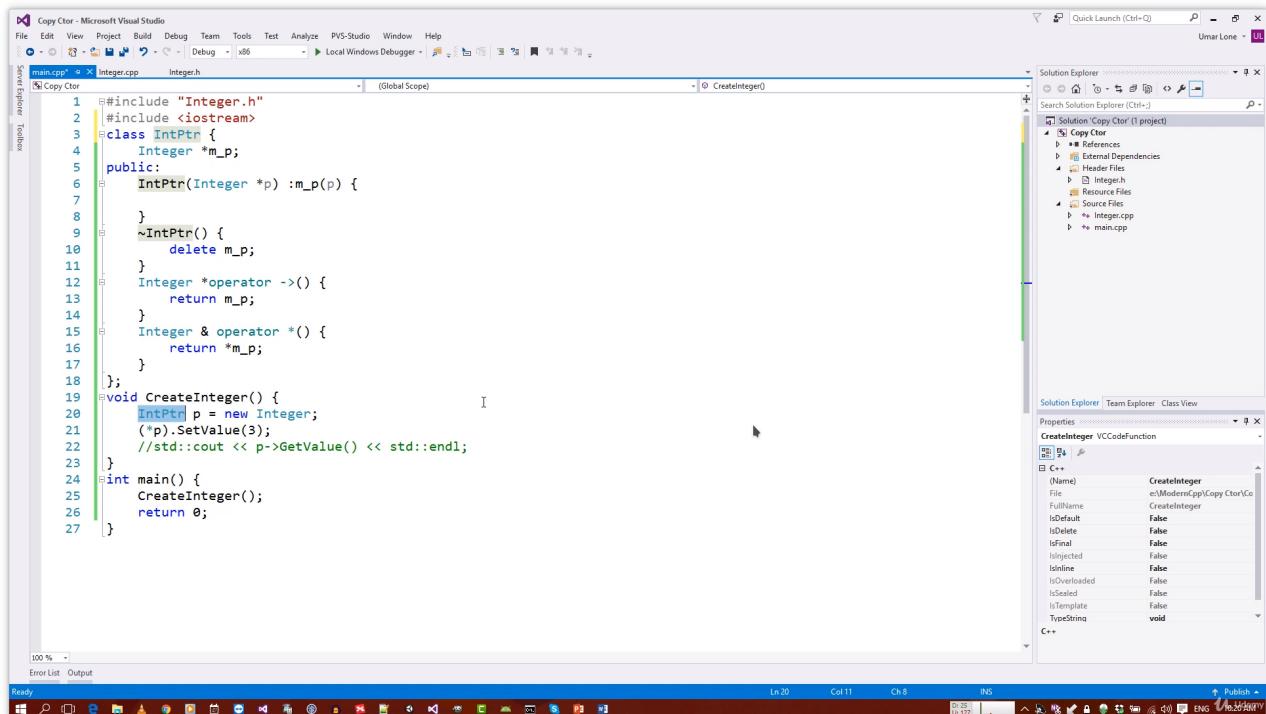
 Poash Technologies®



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.

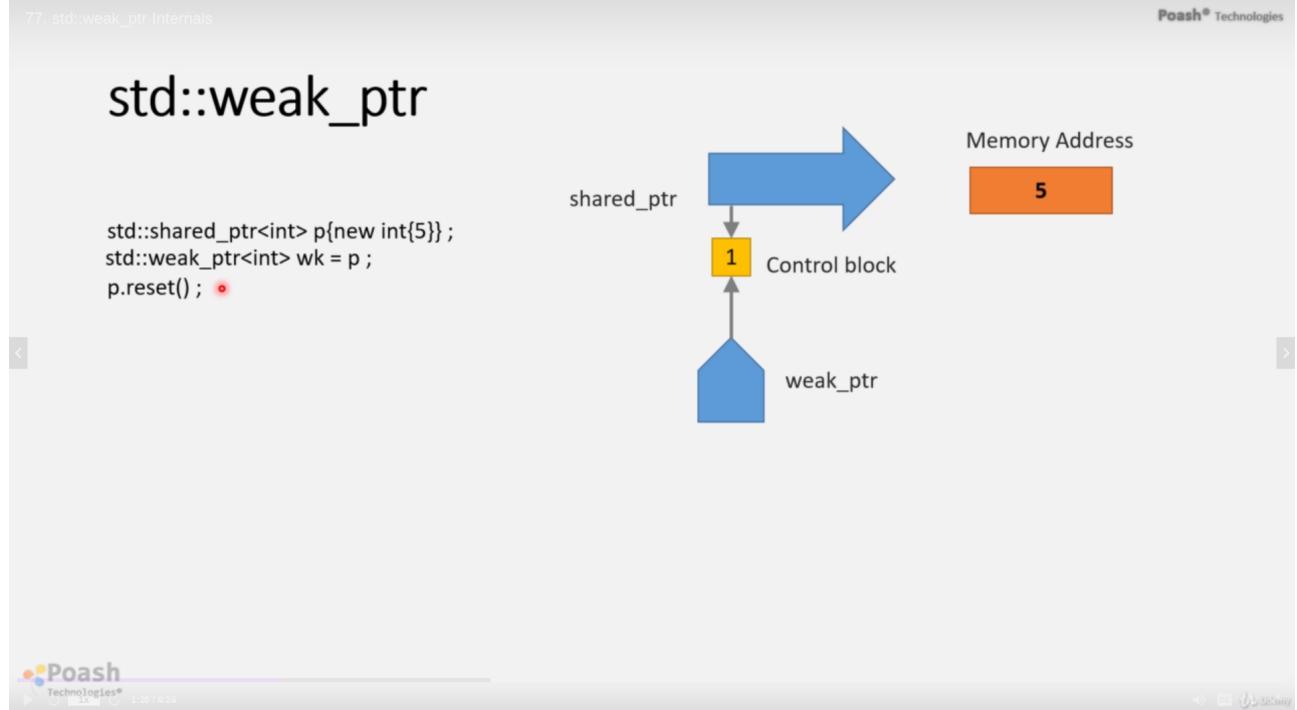
- Always prefer to use std::initializer_list to initialize members of the class. It allow initialization of member variables without assignment.

The screenshot shows the Microsoft Visual Studio IDE interface. On the left, the Solution Explorer displays a project named 'TypeConversion' containing files 'Integer.h' and 'Source.cpp'. The Source Explorer pane shows the contents of 'Source.cpp'. The code defines a class 'Product' with a constructor taking a const Integer& parameter and an implicit conversion operator from Integer to Product. The main function creates a Product object with an integer value of 5. On the right, the Output window shows the command-line interface (cmd.exe) output. It lists several constructor and conversion functions for 'Integer' and 'Product' classes. The output ends with a prompt: 'Press any key to continue . . .'. The status bar at the bottom indicates a build succeeded at 4:23:51:18.

```
#include <iostream>
#include "Integer.h"
class Product {
    Integer m_Id;
public:
    Product(const Integer &id):m_Id(id) {
        std::cout << "Product(const Integer &)" << std::endl;
    }
    ~Product() {
        std::cout << "~Product()" << std::endl;
    }
};
int main() {
    Product p(5);
    return 0;
}
```

```
Integer(int)
Integer()
Product(const Integer &)
operator=(const Integer&)
Integer()
~Integer()
~Product()
~Integer()
Press any key to continue . . .
```

weak_ptr



WeakPtr - Microsoft Visual Studio

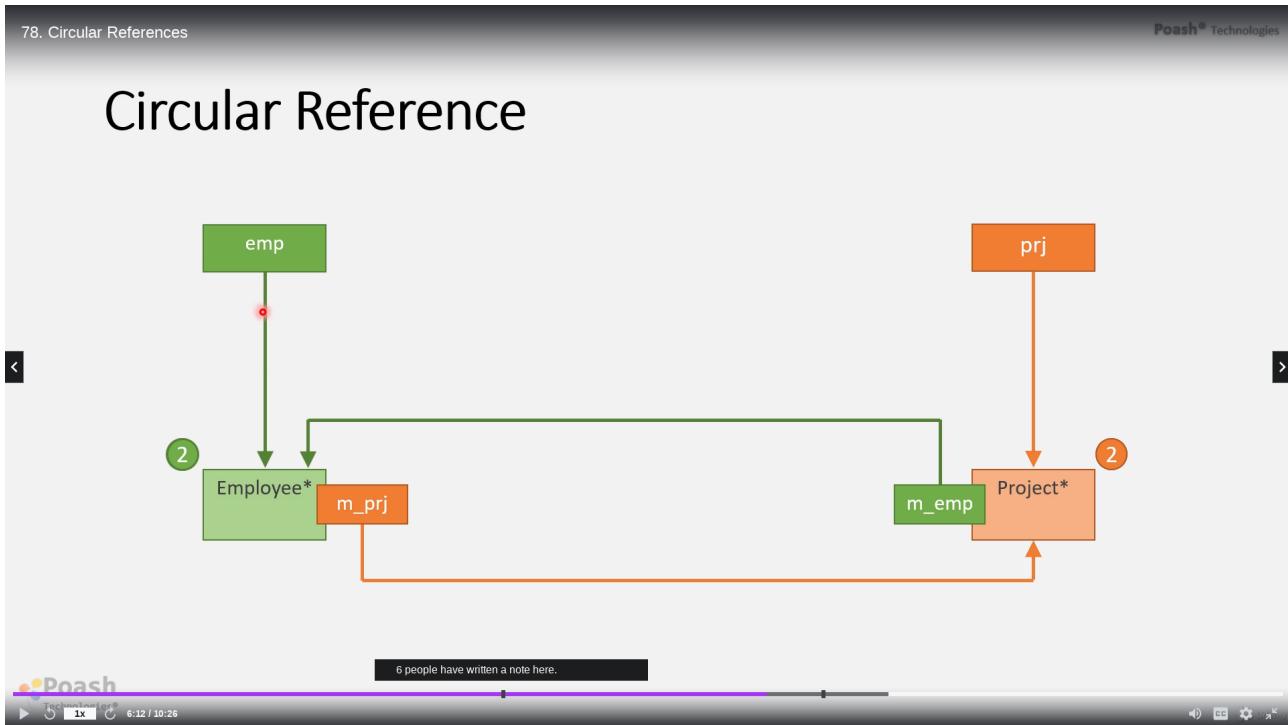
Source.cpp

```
#include <iostream>
class Printer {
public:
    std::weak_ptr<int> m_pValue{ } ;
    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
Search Solution Explorer (Ctrl+Shift+F)
Solution Explorer WeakPtr (1 project)
Source Files
Source.cpp

Properties

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



Screenshot of Microsoft Visual Studio showing the code for **CircularReference** and the Solution Explorer.

```

78. Circular References Team Tools Test ReSharper Analyze Window Help
File CircularReference Employee ~Employee
1 #include <iostream>
2 class Employee ;
3 class Project {
4 public:
5     std::shared_ptr<Employee> m_emp ;
6     Project() {
7         std::cout << "Project()" << std::endl ;
8     }
9     ~Project() {
10        std::cout << "~Project()" << std::endl;
11    }
12 };
13 class Employee {
14 public:
15     std::shared_ptr<Project> m_prj ;
16     Employee() {
17         std::cout << "Employee()" << std::endl ;
18     }
19     ~Employee() {
20         std::cout << "~Employee()" << std::endl;
21     }
22 };
23
24 int main() {
25     std::shared_ptr<Employee> emp{new Employee{}};
26     std::shared_ptr<Project> prj{new Project{}};
27
28     emp->m_prj = prj ;           I
29     prj->m_emp = emp ;
30
31 }

```

The Solution Explorer shows the project structure:

- Solution 'CircularReference' (1 project)
 - Source Files
 - Source.cpp

A note at the bottom states: "17 people have written a note here."

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);
```

OOP Object creation

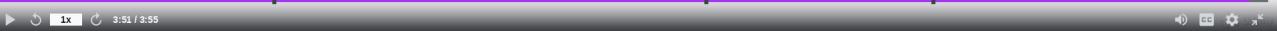
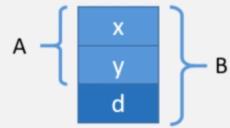
Object Construction

- Constructors execute from base to child
- Destructors execute from child to base
- Base data members will be part of child object

```
class A{  
private:  
    int x;  
    float y;  
//Member functions
```



```
class B : public A{  
private:  
    double d;  
//Member functions
```



Virtual table creation

When the class has virtual functions, the compiler creates **Vtable** and **Vptr**.

Vtable is an array of function pointers and contains function pointers to all virtual member functions.

The **Vptr** points to the first element of the **Vtable** array. It is implicitly created member of the class.

When class is inherited, it creates his own **Vtable** with all of his functions that overrides virtual members of the parent, plus parent's virtual members that are not overridden. It also inherits the **Vptr** from the base class and it points to his own **Vtable**.

The compiler decides at runtime which function to invoke.

