

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

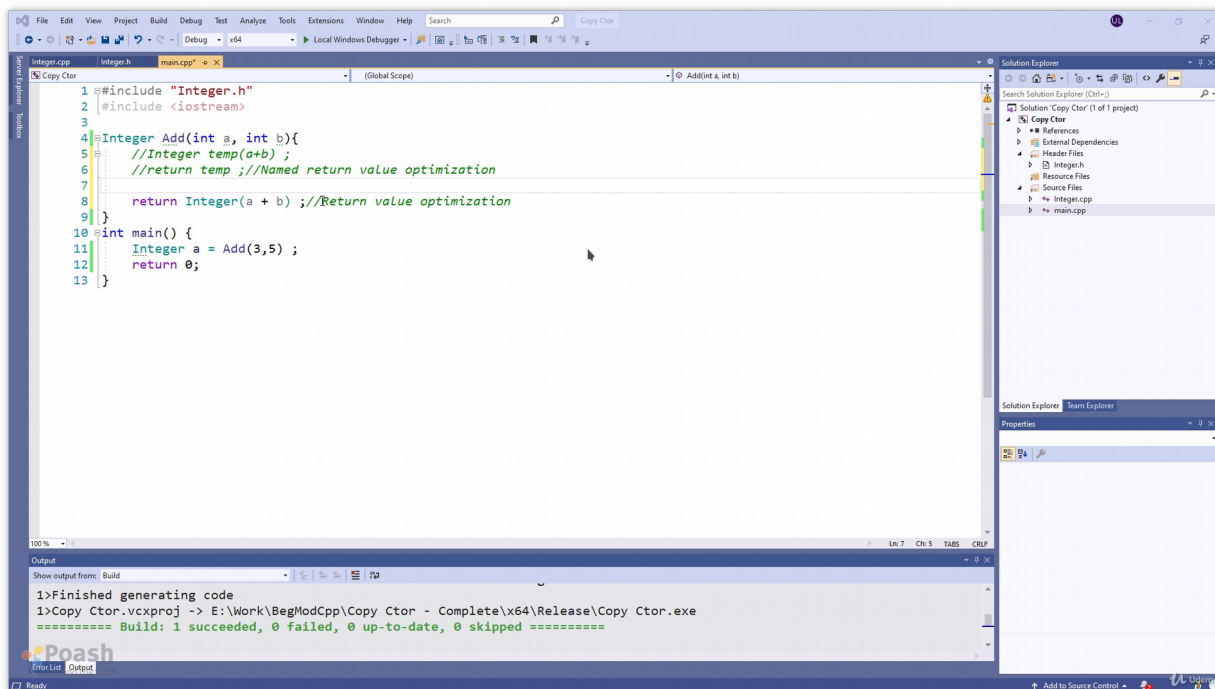
Rule of 5

- 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**

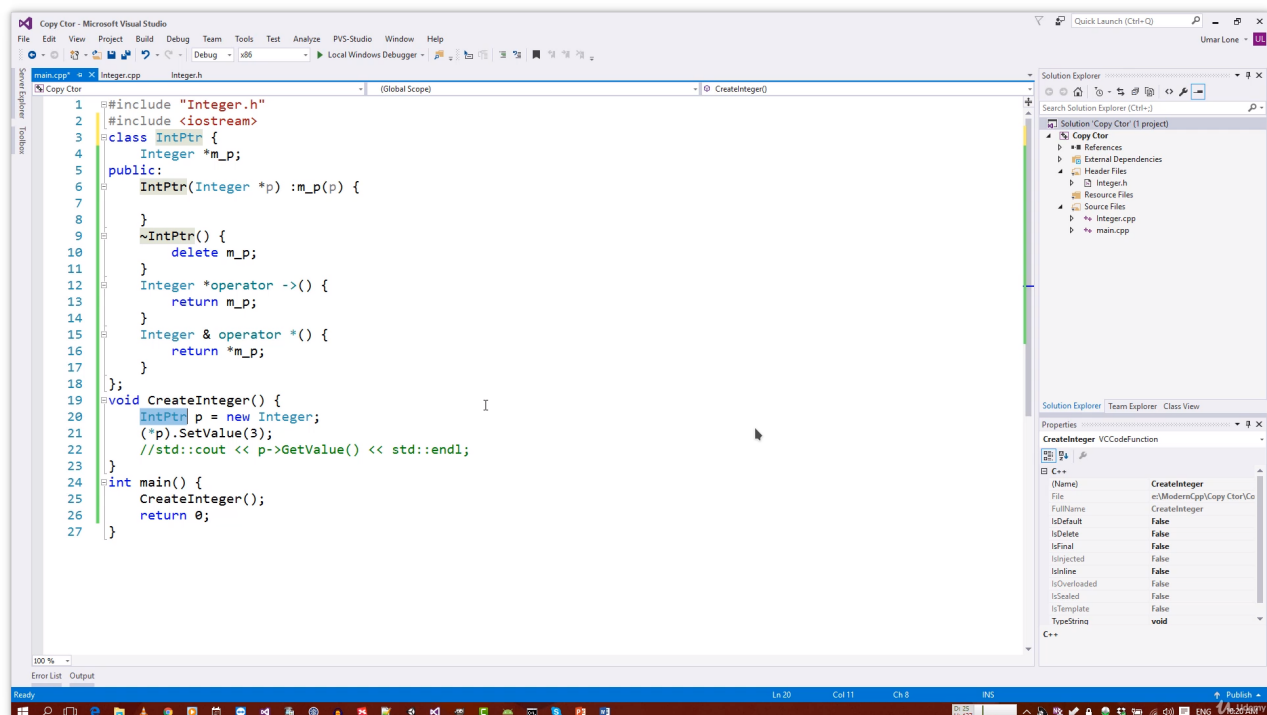
Poash® Technologies

Custom	Copy Constructor	Copy Assignment	Move Constructor	Move Assignment	Destructor
Copy constructor	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.

The screenshot displays a Visual Studio IDE with a C++ project named 'TypeConversion'. The main source file, 'Integer.cpp', contains the following code:

```
1 #include <iostream>
2 #include "Integer.h"
3 class Product {
4     Integer m_Id;
5 public:
6     Product(const Integer &id):m_Id(id) {
7         std::cout << "Product(const Integer &)" << std::endl;
8         //m_Id = id;
9     }
10    ~Product() {
11        std::cout << "~Product()" << std::endl;
12    }
13 };
14 int main() {
15     Product p(5);
16     return 0;
17 }
18
19 Integer(int)
20 Integer()
21 Product(const Integer &)
22 operator=(const Integer&)
23 ~Integer()
24 ~Product()
25 ~Integer()
```

The 'Output' window shows the following text:

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

The 'Solution Explorer' on the right shows the project structure:

- Solution 'TypeConversion' (1 project)
 - References
 - External Dependencies
 - Header Files
 - Integer.h
 - Resource Files
 - Source Files
 - Integer.cpp
 - Source.cpp

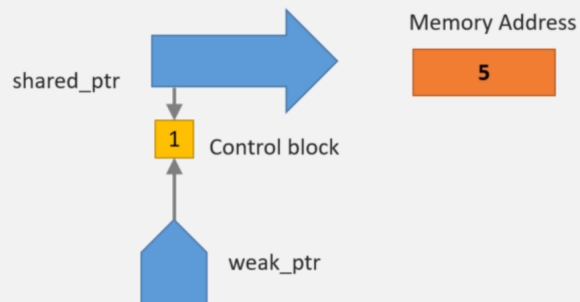
weak_ptr

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();
```



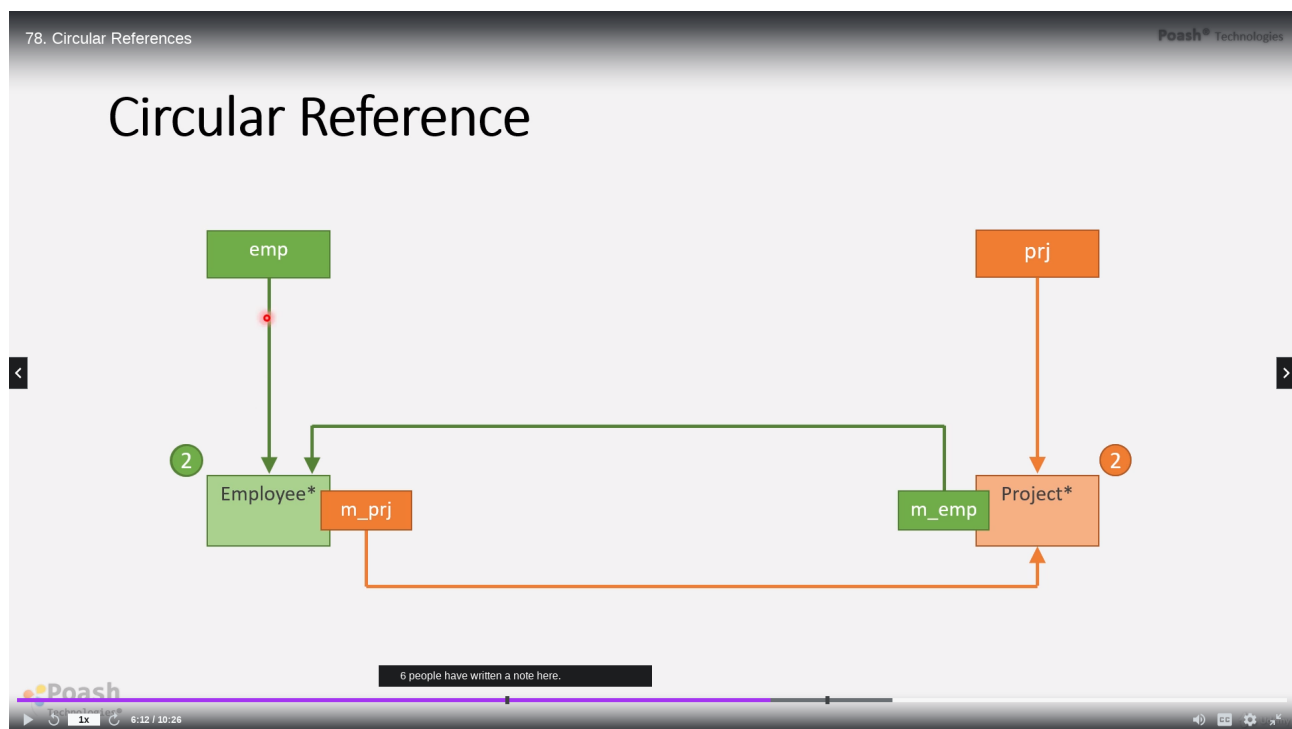
Poash®
Technologies®

```
WeakPtr - Microsoft Visual Studio
File Edit View Project Build Debug Team Tools Test ReSharper Analyze Window Help
Debug - x86 - Local Windows Debugger -
Source.cpp - WeakPtr (Global Scope) - @ main()
1 #include <iostream>
2 class Printer {
3     std::weak_ptr<int> m_pValue{};
4 public:
5     void SetValue(std::weak_ptr<int> p) {
6         m_pValue = p;
7     }
8     void Print() {
9         if(m_pValue.expired()) {
10             std::cout << "Resource is no longer available" << std::endl;
11             return;
12         }
13         auto sp = m_pValue.lock();
14         std::cout << "Value is:" << *sp << std::endl;
15         std::cout << "Ref count:" << sp.use_count() << std::endl;
16     }
17 };
18 int main() {
19     Printer prn;
20     int num{};
21     std::cin >> num;
22     std::shared_ptr<int> p{new int{num}};
23     prn.SetValue(p);
24     if("p > 10") {
25         p = nullptr;
26     }
27     prn.Print();
28 }
29 }

Solution Explorer
Search Solution Explorer (Ctrl-J)
Solution 'WeakPtr' (1 project)
  References
  External Dependencies
  Header Files
  Resource Files
  Source Files
  Source.cpp

Solution Explorer | Team Explorer
Properties
Error List | Output
Ready 1:28 / 1:29
```

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.



78. Circular References

```
#include <iostream>
class Employee;
class Project {
public:
    std::shared_ptr<Employee> m_emp;
    Project() {
        std::cout << "Project()" << std::endl;
    }
    ~Project() {
        std::cout << "~Project()" << std::endl;
    }
};
class Employee {
public:
    std::shared_ptr<Project> m_prj;
    Employee() {
        std::cout << "Employee()" << std::endl;
    }
    ~Employee() {
        std::cout << "~Employee()" << std::endl;
    }
};

int main() {
    std::shared_ptr<Employee> emp(new Employee());
    std::shared_ptr<Project> prj(new Project());
    emp->m_prj = prj;
    prj->m_emp = emp;
}
```

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);
```