```cpp
//////// Lecture 78: Circular References
// Forward declaration of a class
class Employee;
class Project{
public: // keep everything public.
    // List of employees
    Employee *m_emp;
    // Implement constructors and destructors.
    Project(){
        std::cout << "Project()" << std::endl;
    }
    ~Project(){
        std::cout << "~Project()" << std::endl;
    }
};
class Employee{
public: // keep everything public.
    Project *m_prj;
    // implement the constructoss and destructors.
    Employee(){
        std::cout << "Employee()" << std::endl;
    }
    ~Employee(){
        std::cout << "~Employee()" << std::endl;
    }
};

int main()
{
    // create instances for an employee
    Employee *emp = new Employee{};
    Project *prj = new Project{};
    // Initialise the pointers inside both objects.
    emp->m_prj = prj;
    prj->m_emp = emp;

    delete emp; emp = nullptr;
    delete prj; prj = nullptr;
}

// This compiles fine, and you can see that
  constructors and destructors are being called for
```

```
    both the objects. This indicates there are no
    memory leaks, and the objects are properly
    destroyed. We would like to replace the raw
    pointers with smart pointers.
// And because we are sharing the pointers we would
    like to use shared_ptr.



/////// Implementation using shared_ptrs
class Employee;
class Project{
public: // keep everything public.
    // List of employees
    std::shared_ptr<Employee> m_emp;
    // Implement constructors and destructors.
    Project(){
        std::cout << "Project()" << std::endl;
    }
    ~Project(){
        std::cout << "~Project()" << std::endl;
    }
};
class Employee{
public: // keep everything public.
    std::shared_ptr<Project> m_prj;
    // implement the constructoss and destructors.
    Employee(){
        std::cout << "Employee()" << std::endl;
    }
    ~Employee(){
        std::cout << "~Employee()" << std::endl;
    }
};

int main()
{
    // create instances for an employee
    std::shared_ptr<Employee> emp {new Employee{}};
    std::shared_ptr<Project> prj {new Project{}};
    // Initialise the pointers inside both objects.
    emp->m_prj = prj;
```

```
79          prj->m_emp = emp;
80          // No need for delete methods - they should be
              released automatically.
81      }
82
83      // And as you can see when you compiled this, the
          destructors of the objects are NOT INVOKED.
84      // This indicates the memory is not released and is
          therefore LEAKED.
85
86      ////// CIRCULAR REFERENCE INTUITION
87      /*
88      This is called circular reference. This means that
          there are 2 objects and both objects point to each
          other. When you are using shared_ptr with circular
          references, then the underlying memory is not
          released.
89      [SEE SLIDES]
90      */
91      // We may use weak pointer on both of the classes if
          you wish, but let's use the weak pointer only in 1.
92
93      class Employee;
94      class Project{
95      public: // keep everything public.
96          // List of employees
97          std::shared_ptr<Employee> m_emp;
98          // Implement constructors and destructors.
99          Project(){
100             std::cout << "Project()" << std::endl;
101         }
102         ~Project(){
103             std::cout << "~Project()" << std::endl;
104         }
105     };
106     class Employee{
107     public: // keep everything public.
108         std::weak_ptr<Project> m_prj; // CHANGED TO WEAK
              PTR!
109         // implement the constructoss and destructors.
110         Employee(){
111             std::cout << "Employee()" << std::endl;
```

```
112         }
113         ~Employee(){
114             std::cout << "~Employee()" << std::endl;
115         }
116     };
117
118     int main()
119     {
120         // create instances for an employee
121         std::shared_ptr<Employee> emp {new Employee{}};
122         std::shared_ptr<Project> prj {new Project{}};
123         // Initialise the pointers inside both objects.
124         emp->m_prj = prj; // the weak pointer will be
    •       initialised with the control block of this
    •       shared_ptr.
125         prj->m_emp = emp;
126         // No need for delete methods - they should be
    •       released automatically.
127     }
128
```