```cpp
//////// Lecture 73: Sharing Pointers
// Let's suppose we want to develop software for a company that will store information about its employees, and the projects that they're working on.
// so we'll create a class for the project and a class for the employees.

                ProjectA { ARRAY OR LIST of employees}


Employee_1--> ProjectA     Employee#2--> projectA        Employee#3--> ProjectA

class ProjectA{
    std::string m_Name;
public:
    void SetName(const std::string&name)
    {
        m_Name = name;
    }
    void ShowProjectDetails() const {
        std::cout << "[project name]" << m_Name << "\n";
    }
};

// For the employee
class Employee{
    // So each employee will have a pointer or a reference to its Projecr.
    // Ideally, it will be a pointer.
    Project *m_pProject{};
public:
    void SetProject(Project *prj)
    {
        m_project = prj;
    }
    const Project* GetProject()const{
        return m_pProject;
    }
}; // Each employee will also have the information about the project that it is working on.
```

```
void ShowInfo(Employee *emp)
{
    emp->GetProject()->ShowProjectDetails();
}
int main()
{
    // Instance for project
    ProjectA *prj = new Project{};
    prj->SetName("Video Decoder"); // initialise a m_Name
    Employee *e1 = new Employee{};
    e1->SetProject(prj);
    Employee *e2 = new Employee{};
    e2->SetProject(prj);
    Employee *e2 = new Employee{};
    e2->SetProject(prj);
    ShowInfo(e1);

    // Now we need to call the delete methods on the heap
    // WAIT - can we not have the employee destructor call delete on the project pointer? No we can't. This project pointer is shared between different Employee instances.
    // So employee instances do not really own this pointer .
    // That is why the destructor of Employee cannot delete the instance of the project.
    delete prj; //
    delete e1;
    delete e2;
    delete e3; // everything works now and we don't have any memory leaks.

}

/////////// Lecture 74: Sharing std::unique_ptr
// Instead of performing manual memory management, I want to use smart pointers. So I will replace all the raw pointers to smart pointers.
class Project{
```

```
    std::string m_Name;
public:
    void SetName(const std::string&name)
    {
        m_Name = name;
    }
    void ShowProjectDetails() const {
        std::cout << "[project name]" << m_Name << "\n";
    }
};

// For the employee
class Employee{
    // So each employee will have a pointer or a reference to its Projecr.
    // Ideally, it will be a pointer.
    std::unique_ptr<Project> m_pProject{}; // First thing to change.
public:
    void SetProject(std::unique_ptr<Project> prj)
    {
        m_project = prj
    }
    const std::unique_ptr<Project> GetProject()const{
        return m_pProject;
    }
}; // Each employee will also have the information about the project that it is working on.

void ShowInfo(std::unique_ptr<Employee> emp)
{
    emp->GetProject()->ShowProjectDetails();
}
int main()
{
    // Instance for project
    std::unique_ptr<Project> prj {new Project{}};
    prj->SetName("Video Decoder"); // initialise a m_Name
    std::unique_ptr<Employee> e1 = {new Employee{}};
    e1->SetProject(prj);
```

```cpp
    std::unique_ptr<Employee> e2 = {new Employee{}};
    e2->SetProject(prj);
    std::unique_ptr<Employee> e3 = {new Employee{}};
    e3->SetProject(prj);
    ShowInfo(e1);

    // No need to call delete, because we're not using raw_pointers
    // anywhere.
    // But there are many errors in the code:
    // The first error is : "Attempting to reference a deleted function"
    // This is because we are trying to copy assign a unique_ptr.
    // In the previous lesson, we learnt that a unique_ptr cannot be copied,
    // it can be only moved. So you cannot use copy constructors or copy assignments.
    // But you can move a unique_ptr using a move constructor or a move assignment.
}

///// Second Attempt
class Project{
    std::string m_Name;
public:
    void SetName(const std::string&name)
    {
        m_Name = name;
    }
    void ShowProjectDetails() const {
        std::cout << "[project name]" << m_Name << "\n";
    }
};

// For the employee
class Employee{
    // So each employee will have a pointer or a reference to its Projecr.
    // Ideally, it will be a pointer.
    std::unique_ptr<Project> m_pProject{}; // First
```

```
    std::unique_ptr<Project> m_project{}; // First thing to change.
public:
    void SetProject(std::unique_ptr<Project> &prj) // in order to prevent another copy being created at the pass by value, we pass by reference instead.
    {
        m_project = std::move(prj); // Move the unique_ptr into this function.
        // will be deleted once out of this scope.
    }
    const std::unique_ptr<Project>& GetProject()const{
        return m_pProject; // Here this is being returned by value,
        // so it will create a copy,
        // to avoid that we can return this object by reference.
    }
}; // Each employee will also have the information about the project that it is working on.

// In the same way, we'll pass this unique_ptr by reference.
void ShowInfo(std::unique_ptr<Employee>& emp) const
{
    emp->GetProject()->ShowProjectDetails(); // since we don't have to modify the class's state within the function, we can qualify it with const.
}
int main()
{
    // Instance for project
    std::unique_ptr<Project> prj {new Project{}};
    prj->SetName("Video Decoder"); // initialise a m_Name
    std::unique_ptr<Employee> e1 = {new Employee{}};
    e1->SetProject(prj);
    std::unique_ptr<Employee> e2 = {new Employee{}};
    e2->SetProject(prj);
    std::unique_ptr<Employee> e3 = {new Employee{}};
    e3->SetProject(prj);
```

```
        e3->SetProject(prj);
        ShowInfo(e1);
    }
    // We can build it again, and it compiles. We can run it once, while printing the details of the second employee, the code has crashed.

    // We can analyse the code in the again
    // at this line of code:
        e1->SetProject(prj);
    // the employee object has the project information, but now the Project smart pointer itelf is EMPTY. Why is that?

    // This is because when we set the Project insdie the first employeee, its state gets moved,  and this project instance (prj) becomes empty.

    // So there's no point in setting it and other employee objects, it has become a nullptr.

    // How do we take care of this problem? It is difficult to get around this, because as soon as you move the project ptr, the state of this unique_ptr gets moved into the project smart pointer. So there's nothing we can do here!

    // The bottom line is we cannot share this project smart pointer with different employees, this is ecause as soon as it is given to 1 employee, it loses its state. The state gets moved into the employee, so this project pointer can never be used. Even if the project contained only 1 employee, this is semantically not right for our OOP purposes.

    // We can no longer use this Project object after setting it inside the Employee, and the code will inevitably crash.

    // So in this case, we cannot use the unique_ptr, because we have to share the underlying smart pointer
```

```
  pointer.


////////// Lecture 75: std::shared_ptr


// The requirement here is the Project ptr has to be shared with other objects. Unique_ptr does not allow sharing, and we discussed this in the previous lecture.
// So what should we do here? The solution is to use the shared_ptr, the shared _ptr allows sharing of the underlying pointer with other objects.

// To make it work, all we need to do is to replace the unique_ptr with the shared _ptr.

// For the employee
class Employee{
    std::shared_ptr<Project> m_pProject{}; // First thing to change.
public:
    void SetProject(std::shared_ptr<Project> &prj)const
    {
        //m_project = std::move(prj); // shared_ptr supports copy.
        // So there is no need to use move() here.
        m_project = prj; // and its not a good idea to use move() anyway, because the source shared_ptr will be empty!
        // We can also qualify it with const.
    }
    const std::shared_ptr<Project>& GetProject()const{
        return m_pProject;
    }
};

void ShowInfo(std::shared_ptr<Employee>& emp) const
{
    emp->GetProject()->ShowProjectDetails();
int main()
{
```

```
{
    // Instance for project
    std::shared_ptr<Project> prj {new Project{}};
    prj->SetName("Video Decoder"); // initialise a m_Name
    std::shared_ptr<Employee> e1 = {new Employee{}};
    e1->SetProject(prj);
    std::shared_ptr<Employee> e2 = {new Employee{}};
    e2->SetProject(prj);
    std::shared_ptr<Employee> e3 = {new Employee{}};
    e3->SetProject(prj);
    prj->ShowProjectDetails();
}
// This compiles, and now we are able to share the Project smart pointer with other objects.

// So if you're not sure of what kind of smart pointer you need to use in your code, start with the unique_ptr. If the unique_ptr is being shared anywhere,  you'll immediately know that after compiling it, the code will not compile and in fact the compiler will point out places where you are attempting to create a copy of the unique_ptr.
// That is an indication you cannot use unique_ptr so you can replace the unique_ptr with a shared_ptr.

// how does shared_ptr keep track of its copies? Inside the shared_ptr, a reference count is maintained and that reference count is shared between ALL the copies of the std::shared_ptr<T> of type T.


// Now in this case, when the Project object is given to the employee, it is a reference count will be incremented. By the time the control reaches
        prj->ShowProjectDetails();
// 4 copies of the shared_ptr and this information is available to all the shared_pointers.


////// in fact, we can check the reference count through a function called use_count(). But this is
```

```
through a function called use_count(). But this is meant to be used only for debugging.
        Syntax:
            prj.use_count();

/// When will the underlying pointer get deleted? This happenns when the reference count becomes zero. When will the reference count become zero? the reference count becomes 0 when all the shared_pointers are destroyed. When the reference count is 0, then the compiler will delete the underlying pointer.


/// What if one of the shared_ptrs is destroyed?
// for example an empolyee object is destroyed, then it will internally it will automatically destroy the shared_ptr.

// In the destructor of the shared_ptr, the reference count will be decremented by 1. If the result is not zerom, then the compiler will not do anything else.

/// Shared_ptr contains the same methods as that of unique_ptr.

// 1. To check if the shared_ptr contains valid _ptr
if (ptr == nullptr){}; //OR
if(!ptr)

```