

```

1  // Lecture 73: Sharing Pointers
2  // 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.
3  // so we'll create a class for the project and a
   • class for the employees.
4
5          ProjectA { ARRAY OR LIST of employees}
6
7
8  Employee_1--> ProjectA      Employee#2-->
   • projectA      Employee#3--> ProjectA
9
10 class ProjectA{
11     std::string m_Name;
12 public:
13     void SetName(const std::string&name)
14     {
15         m_Name = name;
16     }
17     void ShowProjectDetails() const {
18         std::cout << "[project name]" << m_Name <<
   •         "\n";
19     }
20 };
21
22 // For the employee
23 class Employee{
24     // So each employee will have a pointer or a
   • reference to its Projecr.
25     // Ideally, it will be a pointer.
26     Project *m_pProject{};
27 public:
28     void SetProject(Project *prj)
29     {
30         m_project = prj;
31     }
32     const Project* GetProject()const{
33         return m_pProject;
34     }
35 }; // Each employee will also have the information
   • about the project that it is working on.

```

```

36
37 void ShowInfo(Employee *emp)
38 {
39     emp->GetProject()->ShowProjectDetails();
40 }
41 int main()
42 {
43     // Instance for project
44     ProjectA *prj = new Project{};
45     prj->SetName("Video Decoder"); // initialise a
    • m_Name
46     Employee *e1 = new Employee{};
47     e1->SetProject(prj);
48     Employee *e2 = new Employee{};
49     e2->SetProject(prj);
50     Employee *e2 = new Employee{};
51     e2->SetProject(prj);
52     ShowInfo(e1);
53
54     // Now we need to call the delete methods on the
    • heap
55     // 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.
56     // So employee instances do not really own this
    • pointer .
57     // That is why the destructor of Employee cannot
    • delete the instance of the project.
58     delete prj; //
59     delete e1;
60     delete e2;
61     delete e3; // everything works now and we don't
    • have any memory leaks.
62
63 }
64
65 // Lecture 74: Sharing std::unique_ptr
66 // Instead of performing manual memory management, I
    • want to use smart pointers. So I will replace all
    • the raw pointers to smart pointers.
67 class Project{

```

```

68         std::string m_Name;
69     public:
70         void SetName(const std::string&name)
71         {
72             m_Name = name;
73         }
74         void ShowProjectDetails() const {
75             std::cout << "[project name]" << m_Name <<
76             •         "\n";
77         }
78     };
79     // For the employee
80     class Employee{
81         // So each employee will have a pointer or a
82         •         reference to its Projecr.
83         // Ideally, it will be a pointer.
84         std::unique_ptr<Project> m_pProject{}; // First
85         •         thing to change.
86     public:
87         void SetProject(std::unique_ptr<Project> prj)
88         {
89             m_project = prj
90         }
91         const std::unique_ptr<Project> GetProject()const{
92             return m_pProject;
93         }
94     }; // Each employee will also have the information
95     •         about the project that it is working on.
96
97     void ShowInfo(std::unique_ptr<Employee> emp)
98     {
99         emp->GetProject()->ShowProjectDetails();
100     }
101
102     int main()
103     {
104         // Instance for project
105         std::unique_ptr<Project> prj {new Project{}};
106         prj->SetName("Video Decoder"); // initialise a
107         •         m_Name
108         std::unique_ptr<Employee> e1 = {new Employee{}};
109         e1->SetProject(prj);

```

```

105     std::unique_ptr<Employee> e2 = {new Employee{}};
106     e2->SetProject(prj);
107     std::unique_ptr<Employee> e3 = {new Employee{}};
108     e3->SetProject(prj);
109     ShowInfo(e1);
110
111     // No need to call delete, because we're not
    •     using raw_pointers
112     // anywhere.
113     // But there are many errors in the code:
114     // The first error is : "Attempting to reference
    •     a deleted function"
115     // This is because we are trying to copy assign a
    •     unique_ptr.
116     // In the previous lesson, we learnt that a
    •     unique_ptr cannot be copied,
117     // it can be only moved. So you cannot use copy
    •     constructors or copy assignments.
118     // But you can move a unique_ptr using a move
    •     constructor or a move assignment.
119 }
120
121 //==== Second Attempt
122 class Project{
123     std::string m_Name;
124 public:
125     void SetName(const std::string&name)
126     {
127         m_Name = name;
128     }
129     void ShowProjectDetails() const {
130         std::cout << "[project name]" << m_Name <<
    •         "\n";
131     }
132 };
133
134 // For the employee
135 class Employee{
136     // So each employee will have a pointer or a
    •     reference to its Projecr.
137     // Ideally, it will be a pointer.
138     std::unique_ptr<Project> m_pProject{}: // First

```

```

---
    • thing to change.
139 public:
140     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.
141     {
142         m_project = std::move(prj); // Move the
    • unique_ptr into this function.
143         // will be deleted once out of this scope.
144     }
145     const std::unique_ptr<Project>& GetProject() const {
146         return m_pProject; // Here this is being
    • returned by value,
147         // so it will create a copy,
148         // to avoid that we can return this object by
    • reference.
149     }
150 }; // Each employee will also have the information
    • about the project that it is working on.
151
152 // In the same way, we'll pass this unique_ptr by
    • reference.
153 void ShowInfo(std::unique_ptr<Employee>& emp) const
154 {
155     emp->GetProject()->ShowProjectDetails(); // since
    • we don't have to modify the class's state
    • within the function, we can qualify it with
    • const.
156 }
157 int main()
158 {
159     // Instance for project
160     std::unique_ptr<Project> prj {new Project{}};
161     prj->SetName("Video Decoder"); // initialise a
    • m_Name
162     std::unique_ptr<Employee> e1 = {new Employee{}};
163     e1->SetProject(prj);
164     std::unique_ptr<Employee> e2 = {new Employee{}};
165     e2->SetProject(prj);
166     std::unique_ptr<Employee> e3 = {new Employee{}};
167     e3->SetProject(prj);

```

```

157         e1->SetProject(prj);
168         ShowInfo(e1);
169     }
170     // 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.
171
172     // We can analyse the code in the again
173     // at this line of code:
174     e1->SetProject(prj);
175     // the employee object has the project information,
    • but now the Project smart pointer itself is EMPTY.
    • Why is that?
176
177     // This is because when we set the Project inside the
    • first employee, its state gets moved, and this
    • project instance (prj) becomes empty.
178
179     // So there's no point in setting it and other
    • employee objects, it has become a nullptr.
180
181     // 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!
182
183     // 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.
184
185     // We can no longer use this Project object after
    • setting it inside the Employee, and the code will
    • inevitably crash.
186
187     // So in this case, we cannot use the unique_ptr,
    • because we have to share the underlying smart
    • pointer

```

```

188         pointer.
189
190         // Lecture 75: std::shared_ptr
191
192
193         // The requirement here is the Project ptr has to be
194         • shared with other objects. Unique_ptr does not
195         • allow sharing, and we discussed this in the
196         • previous lecture.
197
198         // So what should we do here? The solution is to use
199         • the shared_ptr, the shared_ptr allows sharing of
200         • the underlying pointer with other objects.
201
202         // To make it work, all we need to do is to replace
203         • the unique_ptr with the shared_ptr.
204
205         // For the employee
206         class Employee{
207             std::shared_ptr<Project> m_pProject{}; // First
208             • thing to change.
209         public:
210             void SetProject(std::shared_ptr<Project>
211             • &prj) const
212             {
213                 //m_project = std::move(prj); // shared_ptr
214                 • supports copy.
215                 // So there is no need to use move() here.
216                 m_project = prj; // and its not a good idea
217                 • to use move() anyway, because the source
218                 • shared_ptr will be empty!
219                 // We can also qualify it with const.
220             }
221             const std::shared_ptr<Project>& GetProject() const{
222                 return m_pProject;
223             }
224         };
225
226         void ShowInfo(std::shared_ptr<Employee>& emp) const
227         {
228             emp->GetProject()->ShowProjectDetails();
229         }
230         int main()
231         {

```

```

210 1
219 // Instance for project
220 std::shared_ptr<Project> prj {new Project{}};
221 prj->SetName("Video Decoder"); // initialise a
    • m_Name
222 std::shared_ptr<Employee> e1 = {new Employee{}};
223 e1->SetProject(prj);
224 std::shared_ptr<Employee> e2 = {new Employee{}};
225 e2->SetProject(prj);
226 std::shared_ptr<Employee> e3 = {new Employee{}};
227 e3->SetProject(prj);
228 prj->ShowProjectDetails();
229 }
230 // This compiles, and now we are able to share the
    • Project smart pointer with other objects.
231
232 // 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.
233 // That is an indication you cannot use unique_ptr so
    • you can replace the unique_ptr with a shared_ptr.
234
235 // 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.
236
237
238 // 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
239 prj->ShowProjectDetails();
240 // 4 copies of the shared_ptr and this information is
    • available to all the shared_pointers.
241
242
243 // 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.

```

244         Syntax:
245             prj.use_count();
246
247     /// When will the underlying pointer get deleted?
    

- This happens 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.


248
249
250     /// What if one of the shared_ptrs is destroyed?
251     // for example an employee object is destroyed, then
    

- it will internally it will automatically destroy
- the shared_ptr.


252
253     // In the destructor of the shared_ptr, the reference
    

- count will be decremented by 1. If the result is
- not zero, then the compiler will not do anything
- else.


254
255     /// Shared_ptr contains the same methods as that of
    

- unique_ptr.


256
257     // 1. To check if the shared_ptr contains valid _ptr
258     if (ptr == nullptr){}; //OR
259     if(!ptr)
260

```