## Memory Management - Part II

A screenshot of a computer program

Description automatically generated

/\* Create object from class integer \*/

* Writing code like this in modern C++ is not recommended Instead, we should use smart pointers.

**Note:**

Read code comments in VS code “App1”

# Smart pointers.

**discussing the usage of raw pointers and the benefits of using smart pointers instead**

1. **Raw Pointers**: we written code using raw pointers to dynamically allocate memory and manage objects manually.
2. **Functionality**: we defined functions like **getPointer** to create instances of objects dynamically and **display** to display their values.
3. **Manual Memory Management**: we mentioned the importance of handling memory deallocation manually using **delete**, and the need to assign **nullptr** after deletion to prevent dangling pointers.
4. **Code Complexity**: While the code works, you've noted that it can become hard to read and maintain, and there's a risk of memory leaks if deallocation is missed.
5. **Recommendation for Smart Pointers**: highlighted that writing code like this is not recommended in modern C++, and instead, smart pointers should be used for memory management.

# Unique pointer

1. **Usage of Unique Smart Pointer**: Unique smart pointers are used when the underlying pointer does not need to be shared with other parts of the code. They ensure exclusive ownership of the pointed-to object.
2. **Including the Header File**: To use unique pointers, the **<memory>** header file needs to be included in the code.
3. **Namespace and Class Template**: Unique pointers are part of the **std** namespace and are implemented as a class template. This means you need to specify the type information when creating an instance of a unique pointer.
4. **Automatic Destruction**: A unique pointer points to an object just like a regular pointer, but it is an object itself. At the end of the scope, a unique pointer will be automatically destroyed, and the pointed-to object will be deallocated.
5. **No Copying, Only Moving**: Unique pointers cannot be copied, but they can be moved. This ensures that ownership remains unique and prevents multiple pointers from pointing to the same object.
6. **No Sharing**: Unique pointers do not allow sharing of ownership. Each unique pointer has sole ownership of the pointed-to object.
7. **Passing by Reference**: While unique pointers cannot be copied, they can be passed by reference, allowing functions to access and manipulate the pointed-to object.
8. **Introduction to std::unique\_ptr**:
   * **std::unique\_ptr** is a smart pointer provided by the C++ Standard Library.
   * It's used when the underlying pointer doesn't need to be shared with other parts of the code.
   * It ensures automatic destruction of the pointer when it goes out of scope, preventing memory leaks.
9. **Including <memory> Header**:
   * To use **std::unique\_ptr**, you need to include the **<memory>** header in your code.
10. **Class Template**:
    * **std::unique\_ptr** is a class template, meaning you must specify the type information when declaring an instance of it.
11. **Direct Initialization**:
    * Unlike assignment, you need to use direct initialization to create a **std::unique\_ptr**.
    * Direct initialization involves specifying the type name without the asterisk.
12. **Checking for Null Pointer**:
    * Similar to raw pointers, it's essential to check if the **std::unique\_ptr** is null before dereferencing it to avoid runtime errors.
13. **Resetting the Pointer**:
    * Instead of assigning a new pointer directly, you should use the **reset** function of **std::unique\_ptr**.
    * The **reset** function deallocates the existing pointer, if any, and takes ownership of the new pointer.
14. **Accessing the Underlying Pointer**:
    * You can use the **get** function of **std::unique\_ptr** to access the underlying raw pointer if necessary.
    * This function returns the raw pointer managed by the **std::unique\_ptr**.
15. **No Need for Manual Deallocation**:
    * Since **std::unique\_ptr** manages memory automatically, there's no need to call **delete** explicitly.
    * This eliminates the risk of memory leaks and simplifies memory management.
16. **Copying and Moving**:
    * **std::unique\_ptr** cannot be copied because it represents exclusive ownership of a pointer.
    * However, it can be moved using **std::move**, which transfers ownership of the pointer from one **std::unique\_ptr** to another.
    * Passing **std::unique\_ptr** by reference is a suitable alternative to avoid copying and maintain ownership semantics.
17. **Passing by Reference**:
    * Passing **std::unique\_ptr** by reference allows using the smart pointer even after calling a function that accepts it.
    * This approach ensures that ownership remains with the original **std::unique\_ptr**, providing flexibility and avoiding unnecessary copying.
18. **Reuse after Move**:
    * After moving a **std::unique\_ptr** using **std::move**, it releases ownership of the pointer, allowing reuse or assignment of a new value.
    * This facilitates resource reuse and efficient memory management in complex codebases.

A computer code with text

Description automatically generated with medium confidence

A computer code with text

Description automatically generated

**.reset():** Reset will do two things if the smart pointer object holds an existing pointer, that will be deleted first and then it'll take ownership of this new pointer.

**.get():** get will return the underlying pointer so you can pass this pointer in any function that does not accept the unique pointer.

**P = nullptr: (delete + NULL)** Unique point has provided an overload of assignment operator that accepts nullptr as a parameter, that implementation of the assignment operator simply deletes the underlying pointer and makes the pointer variable null.

So this statement is just like calling delete on the real pointer and also assigning null to it.

A screenshot of a computer program

Description automatically generated

Top of Form

# Shared pointer

1. **Usage of std::shared\_ptr**:
   * **std::shared\_ptr** is used when the underlying pointer needs to be shared among multiple objects.
2. **Copyability**:
   * Unlike **std::unique\_ptr**, **std::shared\_ptr** allows copying, enabling multiple objects to share ownership of the same resource.
3. **Reference Counting**:
   * A reference count is maintained inside the **std::shared\_ptr** to keep track of the number of references to the underlying pointer.
   * All copies of a shared pointer share the same reference count.
4. **Accessing Reference Count**:
   * The reference count can be checked using the **use\_count** function, although it's primarily used for debugging purposes.
5. **Automatic Deallocation**:
   * The underlying pointer is automatically deleted when the reference count reaches zero, indicating that no more shared pointers are referencing the resource.
6. **Decrementing Reference Count**:
   * Upon destruction of a **std::shared\_ptr**, the reference count is decremented by one.
   * If the reference count doesn't reach zero, the destructor takes no further action.
   * If the reference count becomes zero, the destructor deletes the underlying pointer.
7. **Comparison with NULL**:
   * Similar to **std::unique\_ptr**, **std::shared\_ptr** can be compared with **NULL** to check if it's null or not.
8. **Usage of .reset()**:
   * The **.reset()** function can be used to reset a **std::shared\_ptr**, releasing ownership of the underlying pointer and decrementing the reference count.
9. **Control Block**:
   * **std::shared\_ptr** creates a control block that contains information about the allocated memory, including the reference count.

Overall, **std::shared\_ptr** provides a mechanism for managing resources that need to be shared among multiple objects, ensuring proper memory management and avoiding resource leaks through reference counting and automatic deallocation.

**Note:**

Read code comments in VS code “employees”

Top of Form

**a summary of the example demonstrating the motivation for using std::shared\_ptr:**

1. **Scenario Description**:
   * Consider developing software for a company to manage employee and project information.
   * Multiple employees can work on a single project.
2. **Class Implementation**:
   * Create a **Project** class to represent projects, containing a name and potentially a list of employees.
   * Implement an **Employee** class to represent employees, containing a pointer to their associated project.
   * The **Employee** class allows dynamic assignment of projects, reflecting real-world scenarios where employees may switch projects.
3. **Pointer Management**:
   * **Project** instances may be shared among multiple **Employee** instances.
   * Deleting **Employee** instances should not delete the associated **Project** instances due to shared ownership.
4. **Usage and Output**:
   * Instantiate **Project** and **Employee** objects.
   * Utilize functions to display project and employee details.
   * Ensure proper memory management by manually deleting dynamically allocated instances.
5. **Ownership Clarification**:
   * Employee instances do not own the project pointer; therefore, their destruction should not delete the associated project.

This example illustrates the need for shared ownership of pointers, which is facilitated by **std::shared\_ptr**. It emphasizes dynamic relationships between objects and the importance of proper memory management to prevent memory leaks.

# Note

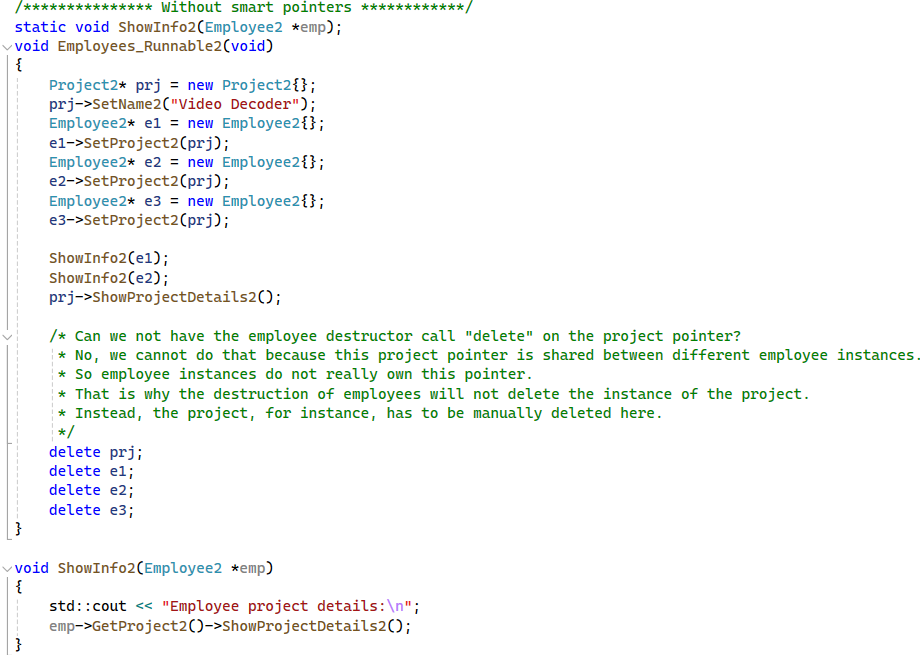
the approach for choosing between **std::unique\_ptr** and **std::shared\_ptr**:

1. **Start with std::unique\_ptr**:
   * If you're unsure whether the underlying pointer needs to be shared, begin by using **std::unique\_ptr**.
   * If the pointer is being shared anywhere in the code, the compiler will detect it during compilation.
2. **Compiler Detection**:
   * Attempting to create a copy of a **std::unique\_ptr** will result in a compilation error.
   * The compiler will point out the locations in the code where such attempts are made.
3. **Indication to Switch**:
   * Compilation errors indicating an attempt to copy a **std::unique\_ptr** serve as a clear indication that a **std::shared\_ptr** might be more appropriate.
   * Replace instances of **std::unique\_ptr** with **std::shared\_ptr** in scenarios where sharing of the underlying pointer is necessary.

Starting with **std::unique\_ptr** provides clarity on whether sharing is needed, and the compiler's feedback guides you towards the appropriate choice.

A screenshot of a computer program

Description automatically generated



explains why the employee destructor cannot call **delete** on the project pointer:

1. **Shared Ownership**:
   * The project pointer is shared among different employee instances, implying shared ownership of the project resource.
2. **Ownership Responsibility**:
   * Since the employee instances do not exclusively own the project pointer, they cannot be responsible for its deletion.
3. **Manual Deletion**:
   * Consequently, the responsibility falls on external management or the entity that originally allocated the project resource to manually delete it when appropriate.
4. **Avoiding Double Deletion**:
   * If the employee destructor were to delete the project pointer, it could lead to double deletion issues since other employee instances may still be using the same project resource.
5. **Clearing Ownership Boundaries**:
   * By not allowing the employee destructor to delete the project pointer, ownership boundaries are maintained, ensuring proper resource management and avoiding memory leaks or undefined behavior.

A screenshot of a computer program

Description automatically generated

A screenshot of a computer program

Description automatically generated

A screenshot of a computer program

Description automatically generated

the transition from manual memory management to using smart pointers, specifically unique pointers, and the challenges encountered during the process:

1. **Smart Pointer Adoption**:
   * The switch from manual memory management to smart pointers is initiated to improve memory handling and avoid potential issues such as memory leaks.
2. **Usage of Unique Pointers**:
   * Raw pointers are replaced with unique pointers for managing the project resource.
   * Unique pointers are preferred due to their exclusive ownership semantics, preventing accidental copies and ensuring proper resource cleanup.
3. **Errors and Challenges**:
   * Upon compilation, numerous errors are encountered, indicating issues with copying unique pointers.
   * Unique pointers cannot be copied; they can only be moved, necessitating adjustments in code logic and function signatures.
4. **Debugging and Analysis**:
   * The code is run and debugged to identify the root cause of the issues.
   * The debugger reveals that while project information is correctly set within employee objects, the project smart pointer itself remains empty.
5. **Inability to Share Pointers**:
   * The fundamental issue arises from the inability to share the project smart pointer among different employee instances.
   * As soon as the project pointer is assigned to one employee, its state is moved, rendering it inaccessible to other employees.
6. **Conclusion and Resolution**:
   * It's concluded that using unique pointers for this scenario is not viable due to the need for shared ownership.
   * Alternative smart pointers, such as shared pointers, may be considered to enable shared ownership of the project resource among multiple employees.

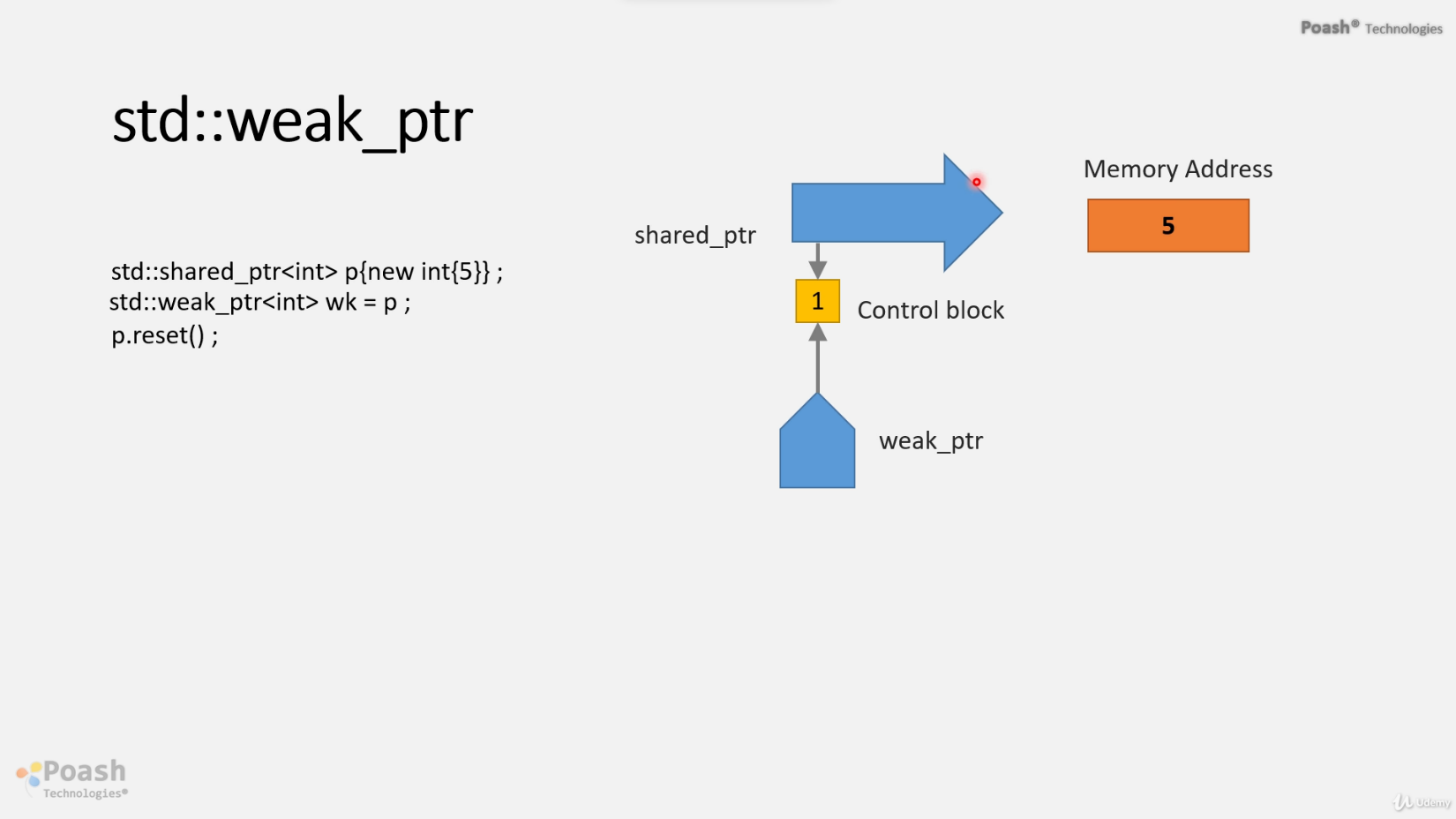
Overall, we highlights the importance of selecting appropriate smart pointers based on ownership requirements and addresses challenges encountered during the transition from manual memory management to smart pointer usage.

**unique pointers to shared pointers:**

1. **Requirement for Shared Ownership**:
   * The specific requirement necessitates sharing the project pointer among multiple objects.
2. **Drawback of Unique Pointers**:
   * Unique pointers, due to their exclusive ownership semantics, do not allow sharing of the underlying pointer.
3. **Solution: Shared Pointers**:
   * Shared pointers are introduced as they enable shared ownership of the underlying pointer.
   * Shared pointers facilitate sharing resources among multiple objects by maintaining a reference count.
4. **Implementation Changes**:
   * Replace unique pointers with shared pointers to allow the desired sharing behavior.
   * Adjust function signatures and return types to use shared pointers where appropriate.
   * Utilize the const qualifier for parameters that are not modified within functions.
5. **Resource Management with Shared Pointers**:
   * Shared pointers automatically manage the deletion of the underlying pointer when the reference count reaches zero.
   * The destructor of the shared pointer decrements the reference count, and if it becomes zero, the underlying pointer is deleted.
6. **Debugging Tools**:
   * Use the **use\_count** function for debugging purposes to check the reference count of shared pointers.
   * This helps in understanding when the underlying pointer will be deleted based on the reference count.
7. **Comparisons and Operations**:
   * Shared pointers can be compared with null or used in boolean expressions to check for the validity of the contained pointer.
   * The **reset** method allows reassigning the shared pointer to a new object, managing the reference count accordingly.
8. **Benefits of Shared Pointers**:
   * Shared pointers offer flexibility in managing resources by allowing multiple objects to share ownership.
   * They simplify resource management tasks, such as avoiding memory leaks and ensuring proper deletion of resources.
9. **Conclusion**:
   * Shared pointers prove to be a suitable solution for scenarios requiring shared ownership of resources among multiple objects.
   * They provide efficient memory management and facilitate resource sharing without the risk of memory leaks or dangling pointers.

By transitioning from unique pointers to shared pointers, the code achieves the desired behavior of sharing the project pointer among different objects while ensuring proper resource management and avoiding memory-related issues.

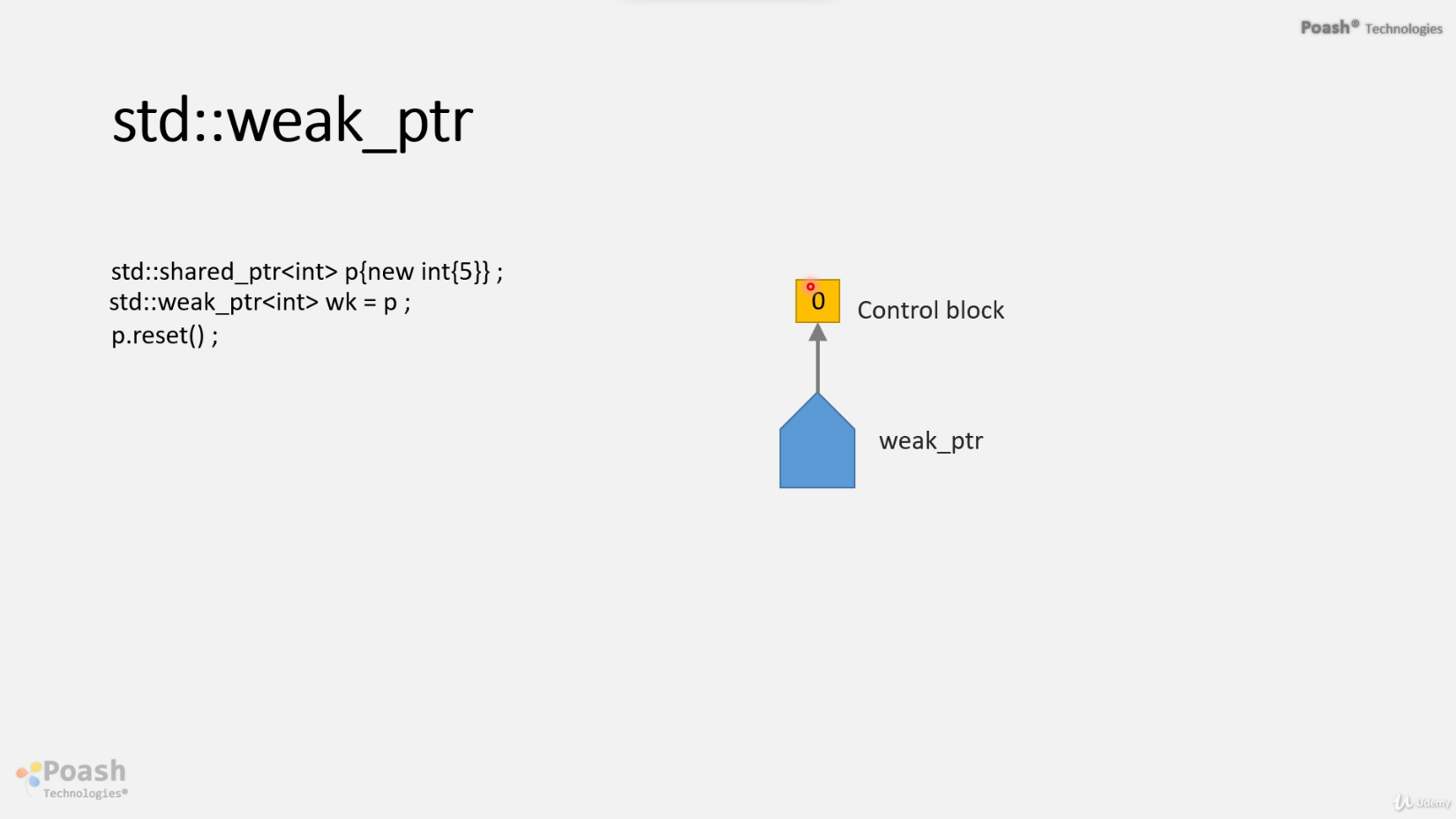
# Weak pointer



Memory allocated

the shared pointer object

* Week pointer is always initialized with Shared pointer and will point to the control block.
* When reset on the shared pointer the memory will be released and the shared pointer will be destroyed.
* Reference cout will be Zero



* So the reference count becomes zero, but the weak pointer will still continue pointing to the control block
* you can have multiple weak pointers pointing to the same shared point, multiple weak pointers can point to the same control block.

# Weak pointer

* Weak pointers are often used in languages like C++ to avoid circular references and memory leaks, particularly in situations where shared ownership is involved.
* **Control Block Destruction:** When the last strong (or shared) pointer pointing to a ‎dynamically allocated object is destroyed, the control block associated with that ‎object (which manages reference counting for shared pointers) may be ‎destroyed. This, in turn, may invalidate any weak pointers associated with that ‎object.‎
* **Indirect Access:** Weak pointers typically don't allow direct access to the underlying ‎pointer to which they point. Instead, you need to use member functions like ‎expired() to check if the object being weakly referenced still exists, and lock() to ‎obtain a shared pointer to the object if it does exist. This prevents accessing an ‎object that may have been deleted.‎
* You can have multiple weak pointers pointing to the same shared point, multiple weak pointers can point to the same control block.
* Week pointer is always initialized with a Shared pointer and will point to the control block.
* This shared pointer maintains ownership of the underlying resource, while weak pointers provide non-owning, observing references to it.
* Either when the reference count becomes zero, the weak pointer will continue pointing to the control block.
* **Weak Pointer Behavior on Zero Reference Count:** When the reference count of the shared pointer becomes zero (meaning there are no more strong references to the resource), the weak pointer will continue to point to the control block. This allows you to check if the resource has been deallocated.
* Control Block will destroyed when the weak pointer destroyed
* Can not access the underlying pointer to the weak pointer directly. Instead, call a member function of the weak pointer **expired**.

# Control Block Destruction on Weak Pointer Destruction

* When the last shared pointer is destroyed, the reference count of the control block reaches zero.
* If there are no weak pointers associated with the control block, it can be safely destroyed, and the memory associated with the object can be deallocated.
* If there are still weak pointers associated with the control block, it indicates that there might still be observers interested in the object's state. In this case, the control block may not be destroyed immediately; **it may only be destroyed when all weak pointers are destroyed as well.**
* This mechanism ensures that resources are deallocated only when there are no more active observers interested in their state, whether they are owning (via shared pointers) or observing (via weak pointers) those resources.

1. **Class Definition**:
   * Define a class called **printer** with a member variable of type integer pointer.
   * Add member functions:
     + **setValue**: to initialize the integer pointer.
     + **print**: to print the value at the address held by the pointer.
2. **Initialization and Memory Allocation**:
   * Initialize a **printer** object.
   * Read an integer value from the console.
   * Allocate memory for the integer pointer.
3. **Memory Management**:
   * Set the value of the integer pointer using the **setValue** function.
   * Print the value using the **print** function.
   * Delete the integer pointer at the end of the scope.
4. **Problem Identification**:
   * Introduce a conditional statement to check if the value is greater than 10.
   * Delete the pointer and assign it to a null pointer to avoid a double delete situation.
   * Encounter unexpected behavior where the printed value becomes junk after deletion.
5. **Understanding the Issue**:
   * The **printer** object retains a pointer to memory that may have been deallocated.
   * Manual checks for the validity of the pointer within the **print** function are needed.
6. **Transition to Smart Pointers**:
   * Replace raw pointers with **shared\_ptr** to manage memory automatically.
   * Use **shared\_ptr** for both member variables and function parameters.
7. **Addressing Memory Release**:
   * Investigate the issue of memory not being released despite setting the pointer to null.
   * Understand that copies of **shared\_ptr** maintain the reference count, delaying memory release.
8. **Introduction of Weak Pointers**:
   * Introduce **weak\_ptr** to maintain a non-owning reference to the shared resource.
   * Use **weak\_ptr** alongside **shared\_ptr** to check the validity of the underlying pointer.
9. **Final Testing**:
   * Run the program with different input values to ensure proper memory management and printing.

In summary, the transition from raw pointers to smart pointers, specifically **shared\_ptr** and **weak\_ptr**, resolves issues related to manual memory management and ensures safe memory usage, preventing potential crashes and memory leaks.

Top of Form

The internal workings of **shared\_ptr** and **weak\_ptr** involve the use of control blocks to manage memory and references. Here's how they function:

1. **Memory Allocation**:
   * When a **shared\_ptr** is initialized, memory is allocated for both the shared resource and a control block.
   * The control block stores metadata about the shared resource, including a reference count initialized to 1.
2. **Control Block**:
   * The control block contains information about the shared resource, such as its reference count.
   * For each **shared\_ptr**, there's a corresponding control block.
3. **Reference Count**:
   * The reference count in the control block increments when a new **shared\_ptr** is created.
   * It decrements when a **shared\_ptr** is destroyed or goes out of scope.
4. **Weak Pointer Relationship**:
   * **weak\_ptr** internally points to the control block of the associated **shared\_ptr**.
   * It does not affect the reference count directly.
5. **Memory Release**:
   * When the reference count of a **shared\_ptr** becomes zero, indicating no more references to the shared resource, the memory is released.
   * The control block is destroyed only when all associated **shared\_ptr** and **weak\_ptr** instances are gone.
6. **Accessing Shared Resource**:
   * To access the shared resource via a **weak\_ptr**, you first check if it's expired using the **expired()** method.
   * If not expired, you call the **lock()** method on the **weak\_ptr**.
   * **lock()** increments the reference count, returns a **shared\_ptr**, and ensures the resource remains valid until all references are released.
7. **Handling Expired Pointers**:
   * If **expired()** returns true, it indicates the shared resource is no longer available.
   * In such cases, attempting to **lock()** the **weak\_ptr** will return an empty **shared\_ptr**.
8. **Error Handling**:
   * Proper error handling is crucial when dealing with weak pointers to avoid accessing invalid memory.
   * Checking expiration and using **lock()** helps ensure safe resource access.
9. **Advantages**:
   * **weak\_ptr** allows referencing a shared resource without affecting its lifetime.
   * It helps prevent circular dependencies and potential memory leaks.
10. **Real Pointer Comparison**:
    * Implementing similar functionality with raw pointers would require manual reference counting and careful memory management, prone to errors like dangling pointers and memory leaks.

In summary, **shared\_ptr** and **weak\_ptr** provide automatic memory management and reference tracking, ensuring safe and efficient resource sharing in C++ programs. They help prevent memory leaks, dangling pointers, and other common pitfalls associated with manual memory management.

# Indirect Access

* **Expired() Functionality: Expired checks:** This reference count, if this reference count is zero, then expired returns number. And that indicates the shared pointer and the underlying memory is no longer available.

But if the reference count is not zero, then expired will return false. This indicates a sharepoint that still exists.

* **Lock:** first, increment the reference count by one and return a shared pointer.
* **Lock() Functionality: The lock()** function attempts to create a shared pointer to the resource that the weak pointer is referencing. If the resource still exists (i.e., the reference count is not zero), it returns a valid shared pointer. Otherwise, it returns an empty shared pointer.
* **Usage of Lock:** When a weak pointer needs to access the resource it refers to, you can call lock() to obtain a shared pointer. This shared pointer allows safe access to the resource and ensures that it is still valid.
* **Weak Pointer has helped us in pointing weakly to a resource that may or may not be available.**
* **If it is available, then we simply apply a lock, which will create a shared pointer and then access the resource using the Share Pointer.**

# Purpose of Weak Pointers:

1. **Referencing Resources that May or May Not Be Available:** Weak pointers provide a non-owning reference to an object that is managed by a shared pointer. Unlike a shared pointer, a weak pointer does **not affect the reference count of the object it points to**. This means that **the object can be deallocated even if weak pointers are still referring to it.**
2. **Observing Resources without Affecting Lifetime:** Weak pointers are particularly useful in scenarios where you want to observe the state of an object without extending its lifetime. For example, in a scenario where an object is owned by a shared pointer and you need to access it conditionally, you can use a weak pointer to check if the object is still valid without preventing its destruction.
3. **Preventing Circular References:** In systems where objects have mutual references to each other (forming a circular reference), memory leaks can occur because objects will never be deallocated due to cyclic dependencies. Weak pointers help break these cycles because **they do not contribute to the reference count.** This means that even if circular references exist, objects can still be deallocated when they are no longer reachable via strong references (shared pointers).

In summary, weak pointers are **a mechanism for observing shared resources without affecting their lifetime**. They provide a way to safely handle situations where objects may or may not be available and help prevent memory leaks by breaking circular dependencies in the ownership graph.

**Note:** Read code comments in VS code “Print”

A screenshot of a computer program

Description automatically generated

A screenshot of a computer program

Description automatically generated

A computer screen shot of a computer code

Description automatically generated

A computer code with colorful text

Description automatically generated with medium confidence

# Circle of reference.

* This means two objects are pointing at each other.
* And the underlying memory is not released when you're using a shared pointer with circular references.



* The issue of circular references, where two objects reference each other directly or indirectly, can lead to memory leaks when using shared pointers. This is because the reference count of the shared pointers never reaches zero, preventing the objects from being properly destroyed. Weak pointers offer a solution to this problem by breaking the circular reference cycle.
* In the example provided, there are two classes: **Employee** and **Project**, each containing a shared pointer to the other class. When using raw pointers or shared pointers, the destructors of these classes may not be called due to the circular references, leading to memory leaks.
* By replacing one of the shared pointers with a weak pointer, the circular reference cycle is broken. Weak pointers do not affect the reference count directly, allowing the objects to be properly destroyed when no other shared pointers reference them.
* In summary, weak pointers help prevent memory leaks in scenarios involving circular references by breaking the reference cycle and allowing objects to be properly destroyed when they are no longer needed. This ensures efficient memory management and prevents resource leaks

the process of creating instances of **employee** and **project** objects and managing their lifetimes using smart pointers is described in detail. Let's break down the explanation line by line:

1. **Creating employee instance and storing it in a shared pointer (emp)**:
   * An instance of the **employee** class is created.
   * This instance pointer is stored inside a shared pointer named **emp**.
   * The reference count of **emp** becomes 1.
2. **Creating project instance and storing it in a shared pointer (prj)**:
   * An instance of the **project** class is created.
   * This instance pointer is stored inside a shared pointer named **prj**.
   * The reference count of **prj** becomes 1.
3. **Assigning prj shared pointer to the m\_prj member of emp**:
   * The **m\_prj** member of the **employee** object (**emp**) is assigned the **prj** shared pointer.
   * This increases the reference count of **prj** by 1.
   * The reference count of **prj** becomes 2.
4. **Assigning emp shared pointer to the m\_emp member of prj**:
   * The **m\_emp** member of the **project** object (**prj**) is assigned the **emp** shared pointer.
   * This increases the reference count of **emp** by 1.
   * The reference count of **emp** becomes 2.
5. **Destruction of objects**:
   * When the scope ends, destructors are invoked for the smart pointers.
   * The **emp** shared pointer is destroyed first, reducing its reference count to 1.
   * Since the reference count is still greater than 0, the **emp** object is not destroyed.
   * Next, the **prj** shared pointer is destroyed, reducing its reference count to 1.
   * Since the reference count is still greater than 0, the **prj** object is not destroyed.
   * As a result, both **employee** and **project** objects are not destroyed, leading to a memory leak.
6. **Solution using weak\_ptr**:
   * By using **weak\_ptr** for one of the member variables (**m\_prj** or **m\_emp**), a circular reference is avoided.
   * When one of the objects is destroyed, the reference count of the other object becomes 0, leading to its destruction.
   * This prevents memory leaks.

In summary, the problem arises due to circular references created by assigning shared pointers to each other's members, which prevents the objects from being destroyed when they are no longer needed. Using **weak\_ptr** breaks the circular reference and ensures proper destruction of objects, thus avoiding memory leaks.

A screenshot of a computer program

Description automatically generated

A computer code with text

Description automatically generated

**m\_prj** and **m\_emp** are member variables in the **employee** and **project** classes, respectively. The reference count of **emp** and **prj** is not directly incremented by assigning these member variables. Instead, the reference count increments because these member variables are of type **std::shared\_ptr**, and when you assign one **std::shared\_ptr** to another, the reference count increases.

Let's break it down:

1. **emp->m\_prj = prj;**: Here, you're assigning the **prj** shared pointer to the **m\_prj** member variable of the **emp** object. This assignment means that **m\_prj** now shares ownership of the **prj** object along with the original **prj** shared pointer. As a result, the reference count of **prj** increases by 1.
2. **prj->m\_emp = emp;**: Similarly, here you're assigning the **emp** shared pointer to the **m\_emp** member variable of the **prj** object. This assignment means that **m\_emp** now shares ownership of the **emp** object along with the original **emp** shared pointer. As a result, the reference count of **emp** increases by 1.

So, by assigning these member variables, you're effectively increasing the reference count of both **emp** and **prj** shared pointers, leading to a circular reference and potentially preventing the objects from being destroyed when they're no longer needed, causing a memory leak.

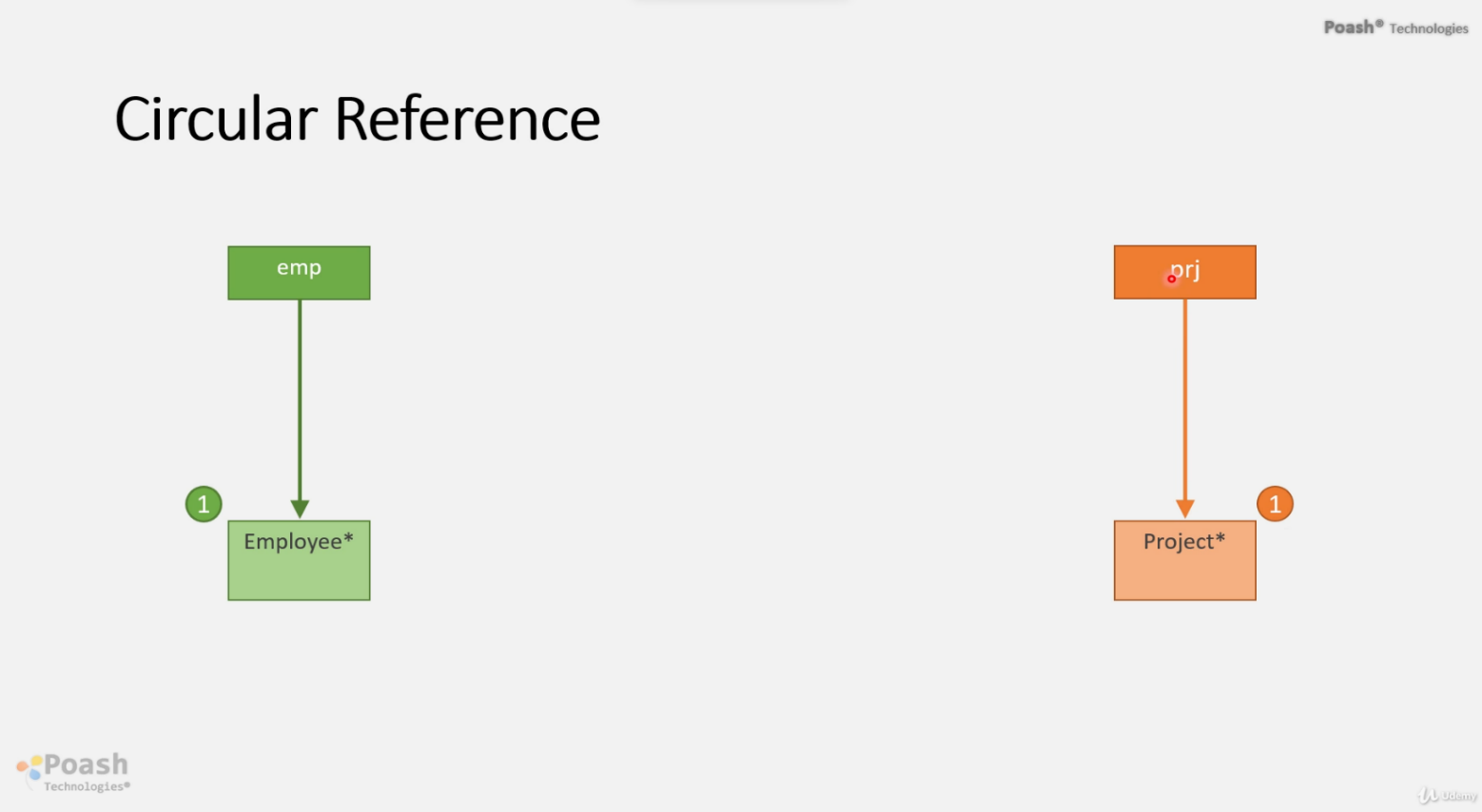
To break this circular reference and avoid the memory leak, one of the member variables (**m\_prj** or **m\_emp**) should be of type **std::weak\_ptr** instead of **std::shared\_ptr**. This ensures that the ownership relationship does not prevent the objects from being destructed when they're no longer needed.



When you assign a **shared\_ptr** to a **weak\_ptr**, it does not increment the reference count of the object being pointed to. Instead, it creates a relationship where the **weak\_ptr** observes the state of the **shared\_ptr** without participating in ownership.

So, assigning a **shared\_ptr** to a **weak\_ptr** does not contribute to increasing the reference count. The reference count remains unaffected, and the **weak\_ptr** allows you to check the validity of the **shared\_ptr** it observes without affecting the object's lifetime.

In the context of breaking circular dependencies and avoiding memory leaks, using a **weak\_ptr** appropriately can help manage object lifetimes without causing reference count increases. This ensures that objects can be properly destroyed when they are no longer needed, even if they have dependencies on each other.



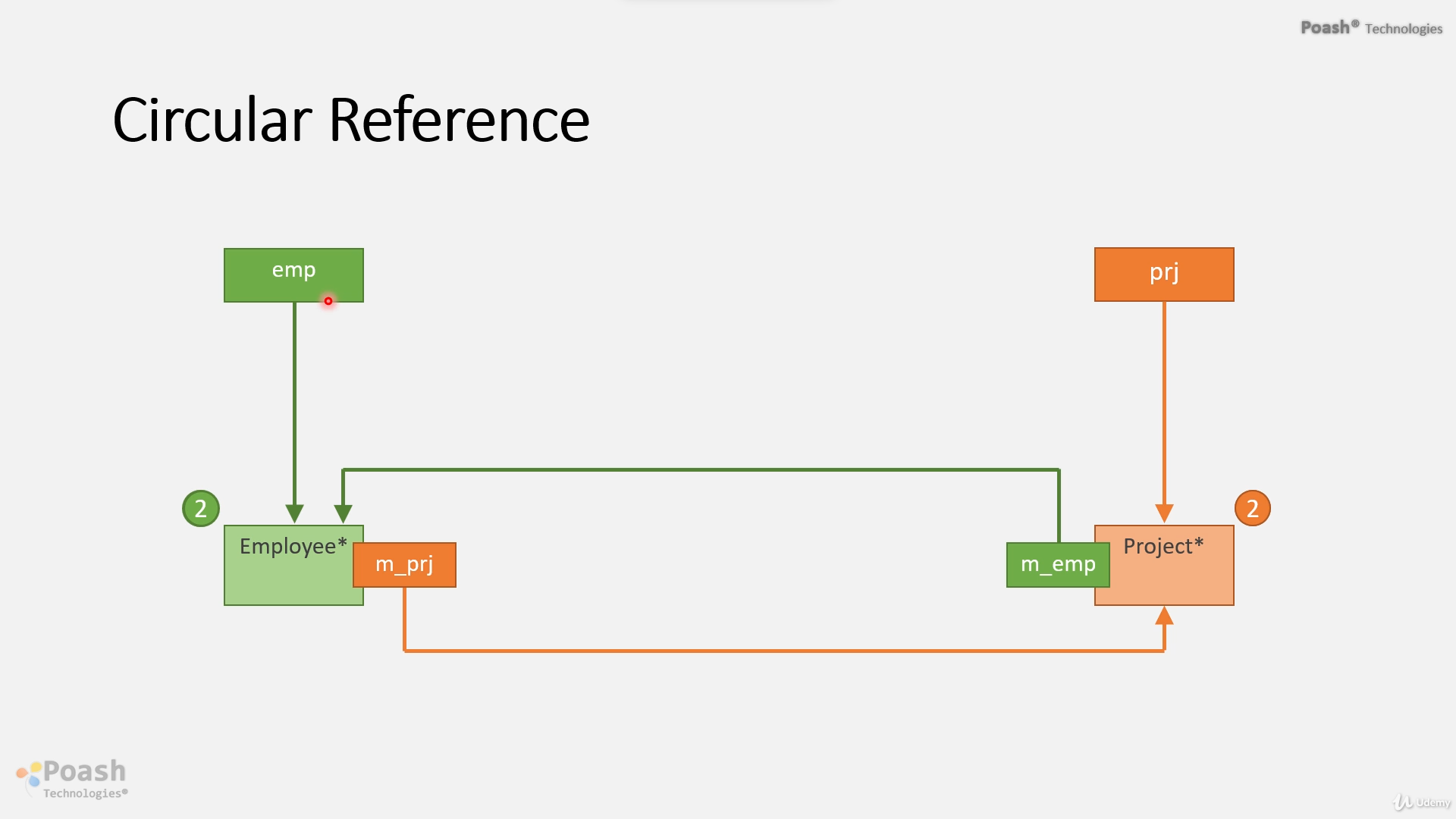
actual pointer

Shared Smart pointer

the employee pointer

Shared Smart pointer

Reference Count



A diagram of a reference

Description automatically generated

A diagram of a reference

Description automatically generated

A diagram of a reference

Description automatically generated

A diagram of a reference

Description automatically generated

A diagram of a reference

Description automatically generated

A diagram of a reference

Description automatically generated

A screenshot of a computer

Description automatically generated

1. Object Creation: Instances of employee and project are created using std::make\_shared. This ensures that memory is allocated for these objects, and their constructors are called.
2. Weak Pointer and Shared Pointer Assignment: The m\_prj member of the employee object is assigned a weak pointer to the project object, and the m\_emp member of the project object is assigned a shared pointer to the employee object.
3. No Ownership: Even though the project holds a shared pointer to the employee and the employee holds a weak pointer to the project, this does not create a circular dependency in ownership. The weak pointer in an employee does not prevent the project object from being deallocated if no other shared pointers to it exist.
4. Proper Deallocation: At the end of the employeesweak\_Runnable function, the shared pointers emp and prj go out of scope, and their destructors are called. Since the employee and project objects are managed by shared pointers, their memory is properly deallocated, and their destructors are invoked.
5. No Memory Leak: Using weak pointers breaks the circular reference, ensuring that the objects can be deallocated properly even though they refer to each other.

Top of Form

# We have legacy code and want to use smart pointers with it

A computer screen shot of a program code

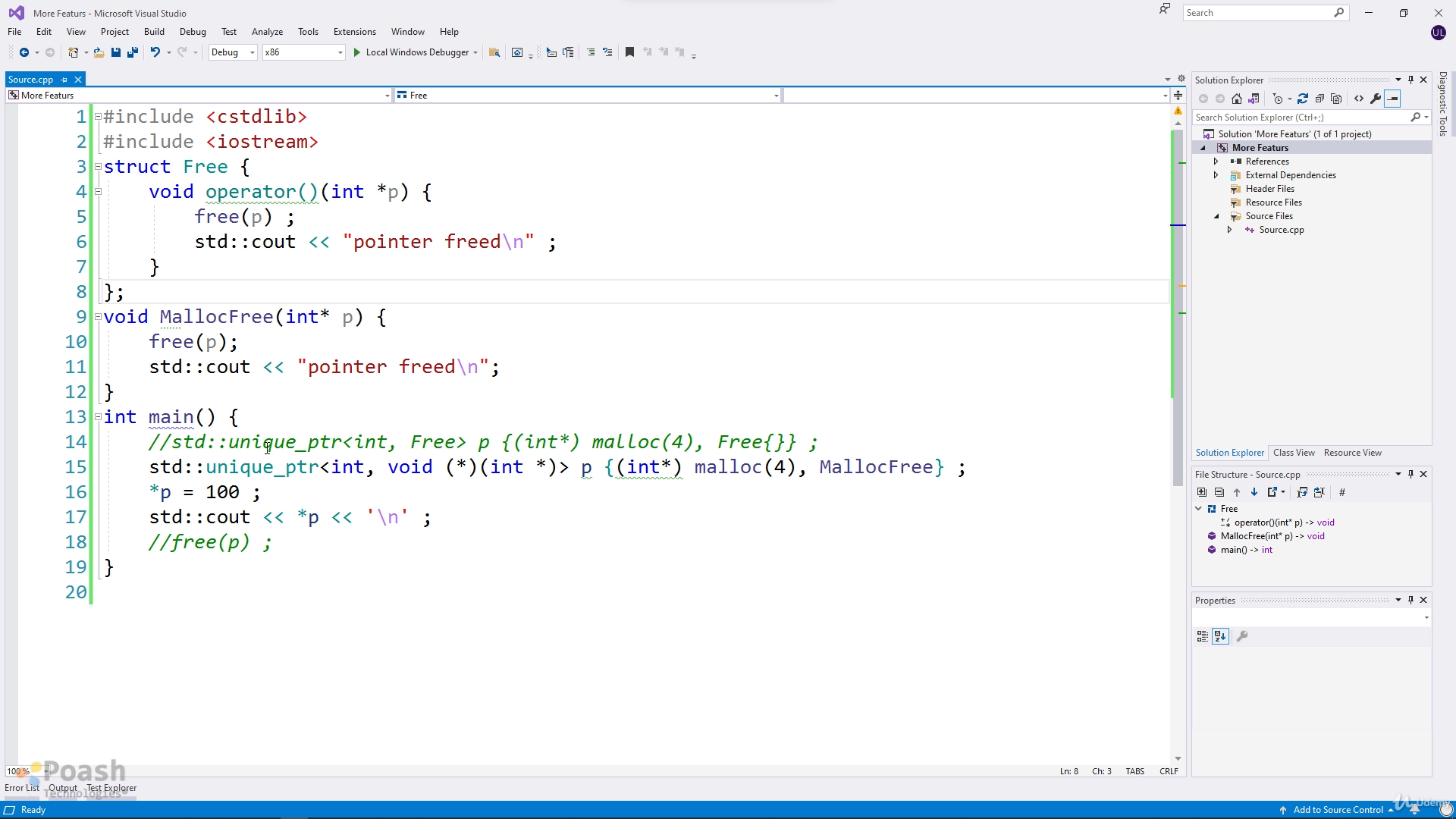
Description automatically generated

* Using a smart pointer here will cause undefined behavior.
* There is no guarantee it will always work because by default, the deleter of the smart pointer which is a unique pointer will call delete and we know that for **malloc** we have to invoke **free** for new **we** must invoke **delete**.
* The resources will not be released properly because the smart point by default will call to delete
* Smart pointers don't invoke delete directly, Instead, they invoke deleter is deleted is a callback that releases the resource and both smart pointers, if we want to release a different kind of a resource, then we can specify our deleter.
* I can either create the deleter as a global function a function object a lambda expression or any other object.

Next, we will discuss how to create this deleter using a function object

# A function object

* Is simply a function that has a state, and in C++ we can create function objects by overloading the function call operator.
* By creating a structure, class, or even a function.
* The argument should be the type of pointer that you want to release.



Should specify an object for this class of custom deleter

can either create an object and then pass it here or I can create a temporary object.



Function custom deleter

here I must specify the type of the function pointer

Pass here the address of the function.



When managing resources with smart pointers, situations may arise where the default behavior of smart pointers, which involves calling **delete** in their destructors, is inadequate. This is especially true when dealing with resources that were not acquired using **new**, such as legacy code or file streams. In such cases, using smart pointers without customization can lead to undefined behavior.

To address this, C++ provides a mechanism to specify custom deleters for smart pointers. Custom deleters are functions or function objects responsible for releasing the resource associated with the pointer when the smart pointer goes out of scope.

Here's a breakdown of using custom deleters with smart pointers:

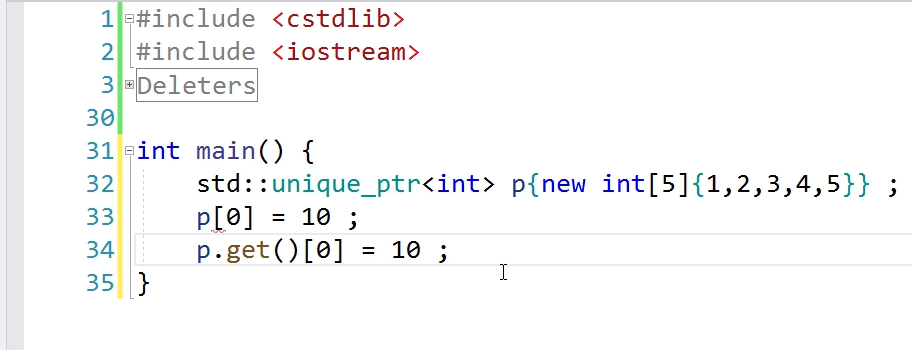
1. **Unique Pointers**:
   * Define a custom deleter, which can be a function object, lambda expression, or a global function.
   * Pass the custom deleter as the second template argument when creating the **std::unique\_ptr**.
   * Alternatively, specify the type of the deleter as a template argument and provide it as a constructor argument.
2. **Shared Pointers**:
   * Similar to unique pointers, define a custom deleter.
   * Pass the custom deleter as the second argument to the constructor of **std::shared\_ptr**.

By specifying custom deleters, you ensure that the appropriate cleanup logic is executed when the smart pointer goes out of scope. This allows you to manage resources safely and efficiently, even when they cannot be released using a simple **delete** call.

Using custom deleters with smart pointers provides flexibility and control over resource management, enabling you to handle a wide range of resource types and scenarios effectively.

# Dynamic Arrays

* It is possible to use smart pointers with dynamic arrays. The default deleter of the unique pointer will call delete and not delete subscript (delete []). So this might lead to undefined behavior.
* The second problem is using this smart pointer, we cannot directly access the elements of the dynamic array using the Syntax.



* To solve this problem use a partial specialization of unique pointer for array types.
* This will cause the compiler to choose another Class of unique PTR that provides the subscript operator through which you can access the elements.
* So no need to use this complex syntax. And the other advantage is that the leader will use, the correct delete, that is it will use the delete subscript.
* Ideally, we should avoid creating a dynamic like this. If you want a dynamic array that can grow at runtime, then it would recommend using a container

A screen shot of a computer program

Description automatically generated

This excerpt discusses the challenges and solutions related to using smart pointers with dynamic arrays in C++:

1. **Default Deleter Behavior:**
   * Smart pointers, like unique pointers, use a default deleter that calls **delete**, which may lead to undefined behavior when managing dynamic arrays.
2. **Inability to Access Elements:**
   * Due to the lack of a subscript operator, directly accessing elements of a dynamic array with smart pointers is inconvenient and error-prone.
3. **Issues with Shared Pointers:**
   * Shared pointers face similar problems, where the default deleter also calls **delete**, making it unsuitable for managing dynamic arrays.
4. **Custom Deleters as a Solution:**
   * While custom deleters can be used to address this issue, they require additional code and do not provide the convenience of the subscript operator.
5. **Partial Specialization of Unique Pointer:**
   * Smart pointers offer a solution through partial specialization, specifically for array types.
   * By using an empty subscript, the compiler selects a specialized version of **unique\_ptr** that supports the subscript operator, simplifying element access.
6. **Advantages of Specialization:**
   * Specialized **unique\_ptr** instances use the correct delete operation for arrays, avoiding issues associated with the default deleter.
7. **Compatibility and Best Practices:**
   * It's noted that specialization and the subscript operator for shared pointers were introduced in C++17, potentially causing compatibility issues with earlier standards.
   * While smart pointers offer solutions, it's advised to avoid managing dynamic arrays manually.
   * For dynamic arrays requiring runtime growth, containers like **vector** are recommended for automatic memory management.
   * However, if fixed-size dynamic arrays are necessary, smart pointers like **unique\_ptr** or **shared\_ptr** can be used, following established best practices.

Overall, the excerpt emphasizes the importance of proper resource management and highlights smart pointers' capabilities and limitations when dealing with dynamic arrays in C++.

# Make functions

1. **Discouragement of manual memory management**: Modern C++ discourages the direct use of **new** and **delete** operators for memory management due to the risks of memory leaks and undefined behavior.
2. **Utilization of smart pointers**: Instead of manually allocating memory, smart pointers are recommended. They automate memory management by automatically deallocating memory when the smart pointer goes out of scope.
3. **Global functions for smart pointer creation**: Smart pointers provide global functions, such as **make\_unique** and **make\_shared**, which simplify the process of creating smart pointers without the need for manual memory allocation.
4. **Usage of make\_unique**: **make\_unique** behaves like a factory function, creating an instance of a class on the heap and returning a **unique\_ptr** that owns the object. It automatically handles memory allocation and initialization.
5. **Implementation differences between unique and shared pointers**: Shared pointers have to store additional information related to reference counting, which is stored in a control block.
6. **Custom deleters**: One disadvantage of using **make** functions is the inability to specify a custom deleter. If a custom deleter is needed, manual construction of the smart pointer is required.
7. **Advantages of using make functions**: If a custom deleter is not needed, using **make** functions is recommended over manual memory allocation using **new**. It simplifies memory management and reduces the risk of memory leaks.

Overall, the use of smart pointers and **make** functions promotes safer and more efficient memory management practices in modern C++ programming

A screenshot of a computer program

Description automatically generatedA screenshot of a computer program

Description automatically generated

modern C++ practices regarding memory management and the usage of smart pointers, focusing on **make\_unique** and **make\_shared** functions:

1. **Discouragement of Manual Memory Management:**
   * Modern C++ discourages direct usage of **new** and **delete** operators for memory management due to the risk of memory leaks and other issues.
2. **Introduction of Smart Pointers:**
   * Smart pointers provide a safer alternative for managing dynamically allocated memory by automatically handling memory deallocation when the pointer goes out of scope.
3. **Usage of make\_unique and make\_shared:**
   * Instead of manually allocating memory using **new**, modern C++ encourages the use of **make\_unique** and **make\_shared** functions provided by smart pointers.
   * These functions allow the creation of smart pointers without the need to explicitly allocate memory using **new**.
4. **make\_unique Function:**
   * **make\_unique** behaves like a factory function, creating an instance of a class on the heap.
   * It accepts the type of object to construct and any arguments needed for its constructor.
   * Example usage demonstrates creating a **unique\_ptr** for an **int** object initialized with a value of 5.
5. **Dynamic Array Allocation:**
   * **make\_unique** can also be used to allocate memory for dynamic arrays by providing the array size as an argument.
   * However, initialization of dynamic arrays using **make\_unique** is not possible; initialization should be done separately.
6. **make\_shared Function:**
   * Similar to **make\_unique**, **make\_shared** is used to create shared pointers.
   * It is advantageous over manually allocating memory because it optimizes memory usage by storing additional information related to the underlying pointer in a control block.
7. **Custom Deleters:**
   * A limitation of using **make** functions is the inability to specify custom deleters.
   * If a custom deleter is needed, manual allocation and construction of the smart pointer are required.
8. **Recommendation:**
   * If custom deleters are not needed, it's recommended to use **make** functions for creating smart pointers instead of manually allocating memory.

Overall, the advantages of using **make\_unique** and **make\_shared** functions for memory allocation with smart pointers, promoting safer and more efficient memory management practices in modern C++ programming.