Smart Pointers

09 June 2021 07:07

Smart pointer is that it manages a dynamically allocated resource, and ensures the dynamically allocated object is properly cleaned up at the appropriate time
Always allocate smart pointers on the

- stack (as local variables or composition members of a class)
  we're guaranteed that the smart pointer will properly go out of scope when the function or object it is contained within ends, ensuring the object the smart pointer owns is properly deallocated.

std::unique_ptr - manage any dynamically allocated object that is not shared by multiple objects
https://www.cplusplus.com/reference/memory/unique_ptr/
https://en.cppreference.com/w/cpp/memory/unique_ptr/make_unique

std::shared_ptr -multiple std::shared_ptr pointing to the same dynamically allocated resource. Internally, std::shared_ptr keeps track of how many std::shared_ptr are sharing the resource. As long as at least one std::shared_ptr is pointing to the resource, the resource will not be deallocated, even if individual std::shared_ptr are destroyed. As soon as the last std::shared_ptr managing the resource goes out of scope (or is reassigned to point at something else), the resource will be deallocate
https://www.cplusplus.com/reference/memory/shared_ptr/
https://www.nextptr.com/tutorial/ta1358374985/shared_ptr-basics-and-internals-with-examples

std::weak_ptr
https://www.cplusplus.com/reference/memory/weak_ptr/

1. ## Unique_ptr copy assignment is disabled.. So
   ptr2 = std::move(ptr1);
2. ## Dereferencing unique_ptr
   std::unique_ptr has an overloaded operator* and operator->
   - Operator* returns a reference to the managed resource
   - Operator-> returns a pointer
3. ## unique_ptr is implicitly cast to bool to check if its not nullptr
   If(unique_ptr)
4. ## Favor std::array, std::vector, or std::string over a smart pointer managing a fixed array, dynamic array, or C-style string
   I
5. ## Use std::make_unique<type>(constructor arguments) .. Exception safe
   because -
   `std::unique_ptr<int> a{new int{}}; // new is one step, unique_ptr creation is another. Exception may cause memory leak`
   `Std::unique_ptr<int> b = std::make_unique<int>({}); //`creation of the object T and the creation of the std::unique_ptr happen inside the std::make_unique() function as atomic operation, where there's no ambiguity about order of execution.
6. ## Pass unique_ptr into functions
   using std::move if ownership has to be transferred to a new unique_ptr
   using lvalue reference if u wanna use the same unique_ptr in func
7. ## Returning unique_ptr from func is an example of C++ doing a move-return on l-value - automatically moved
   C++17 or newer, the return will be elided??
8. ## If additional shared_ptr needs to be created from existing shared_ptr - copy assign/initialization/move initialized
   I
9. ## Shared_ptr internal details
   - Pointer to control block which consists of
     - MO pointer / owned pointer
       - shared_ptr<type>(new **type**) --> type of MO pointer
     - ref count
     - weak count
     - deleter etc)
   - Raw Pointer / stored-pointer to managed object
     - shared_ptr<**type**> --> type of raw pointer

   When MO pointer/owned pointer is null -> EMPTY Shared pointer
   When Raw pointer/stored pointer is null -> NULL shared pointer
10. ## Use make_shared<type>(constructor argument).. Exception safe + Efficient: creates control block, MO in same allocation
    I
11. ## aliasing shared_ptr
    X = shared_ptr<T1>(new T)
    Shared_ptr<T2>(X, T2* y) --> raw pointer = T2* y.. MO pointer is pointing to same MO as X.. Manages this MO and not y
12. ## weak_ptr
    - std::weak_ptr was designed to solve the "cyclical ownership" problem
    - A std::weak_ptr is an observer -- it can observe and access the same object as a std::shared_ptr (or other std::weak_ptrs) but it is not considered an owner.
    - weak_ptr.lock()

    - ONLY way to access MO
    - lock the owned/managed pointer and add additional shared_ptr to manage MO

```
33                                          // sharing group:
                                           // --------------
sp1 = std::make_shared<int> (20);          // sp1
wp = sp1;                                  // sp1, wp
sp2 = wp.lock();                           // sp1, wp, sp2
sp1.reset();                               //      wp, sp2
sp1 = wp.lock();                           // sp1, wp, sp2
```

- weak_ptr.expired() -> check if MO pointer is null i.e. no shared pointer
- weak_ptr.use_count() -> check number of shared_ptr

13. F
    I
14. F
    I
15. F
    I
16. F
    I
17. F
    I
18. D
    I
19. F
    I
20. F
    I
21. F
    I
22. F
23. F
    I4334
24. F
    I
25. F
    I
26. F
    I
27. F
    I
28. F
    I
29. D
    I
30. F
    I
31. F
    I
32. F
    I
33. F
34. F
    I
35. F
    I
36. F
    I
37. F
    I
38. F
    I
39. F
    I
40. D
    I
41. F
    I

42. F
    I
43. F
    I
44. F
45. F
    I
46. F
    I
47. F
    I
48. F
    I
49. F
    I