**smart pointers in c++:**

new -> Allocates memory in the heap

delete -> Deletes the memory from the heap

-> Smart pointers are used to automatically assign and delete heap memory

**"What is a smart pointer?"**

It's a type whose values can be used like pointers, but which provides the additional feature of automatic memory management: When a smart pointer is no longer in use,

the memory it points to is deallocated.

**"When should I use one?"**

In code which involves tracking the ownership of a piece of memory, allocating or de-allocating;

the smart pointer often saves you the need to do these things explicitly.

**"But which smart pointer should I use in which of those cases?"**

**Use std::unique\_ptr** when you want your object to live just as long as a single owning reference to it lives.

For example, use it for a pointer to memory which gets allocated on entering some scope and de-allocated on exiting the scope.

**Use std::shared\_ptr** when you do want to refer to your object from multiple places - and do not want your object to be de-allocated

until all these references are themselves gone.

**Use std::weak\_ptr** when you do want to refer to your object from multiple places - for those references for which it's ok to ignore

and deallocate (so they'll just note the object is gone when you try to dereference).

**Note:**

Don't use the **boost:: smart pointers** or **std::auto\_ptr** except in special cases which you can read up on if you must.

**"So when should I use regular pointers then?"**

Mostly in code that is oblivious to memory ownership. This would typically be in functions which get a pointer from someplace else and do not allocate nor de-allocate, and do not store a copy of the pointer which outlasts their execution.

**Smart pointers:**  
**NOTES:**

1. smart pointer is a class which wraps a raw pointer , to manage the lifetime of the pointer

In this video we will learn what is the actual use of smart pointers in c++.

It is basically a wrapper which wraps raw pointer and handles the life time of the object for you,

means if there is no reference to the pointer it will delete it otherwise it will hold.

2. It will make sure that the object is deleted if it is not referenced any more.

**Trivia:**

auto\_ptr is deprecated in c++11, it's not there in c++ 17

**There are basically three types of smart pointers.**

**TYPES:**

1. unique\_ptr

2. shared\_ptr

3. weak\_ptr

And the basic job of smart pointer is, it actually makes sure that we don't have memory leaks.

\*/

/\* **unique\_ptr:**

-> Allows only one owner of the underlying pointer

-> you cannot copy the object of the unique\_ptr

-> you can move it and take the reference/pointer of it

-> your pointer will be single all the time

\*/

/\* **shared\_ptr:**

-> Allows multiple owners of the same pointer (Reference count is maintained)

\*/

/\* **weak\_ptr:**

-> It is a special type of a shared\_ptr which doesn't count the reference

-> when you have two shared pointers and both have cyclic dependency, one can be made weak\_ptr

\*/  
\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**unique pointer in c++**

**NOTES:**

0. unique\_ptr is a class template.

1. unique\_ptr is one of the smart pointer provided by

c++11 to prevent memory leaks.

2. unique\_ptr wraps a raw pointer in it, and de-allocates the

raw pointer, when unique\_ptr object goes out of scope.

3.similar to actual pointers we can use -> and \* on the object of

unique\_ptr, because it is overloaded in unique\_ptr class

4. when exception comes then also it will de-allocate the memory

hence no memory leak.

5. Not only object we can create array of objects of unique\_ptr.

**Operations:**

release, reset, swap, get, get\_deleter

**Shared pointer in c++:**

There are few points about it:

0. shared\_ptr is a smart pointer which can share the ownership of object (managed object).

1. Several shared\_ptr can point to the same object (managed object).

2. It keep a reference count to maintain how many shared\_ptr are pointing to the same object.

and once last shared\_ptr goes out of scope then the managed object gets deleted.

**Trivia:**

1.

control block keeps track of reference count of your managed object. whenever you assign your smart pointer to

another smart pointer, this control block will kick in and increment the reference count.

once the last shared\_ptr goes out of scope, then the managed object gets deleted.

2.

< control block is thread safe >

The Reference count in control block is thread safe as per the c++ standards. No matter, how many times you declare

and re-assign shared\_ptr, the count increment/decrement is automatically taken care of in a thread-safe manner.

3. shared\_ptr is threads safe and not thread safe. [what is this??]

a. control block is thread safe

b. managed object is not

It is not thread safe. we can use mutex and other methods to make it thread safe manually.

4. There are three ways shared\_ptr will destroy the managed object.

a. If the last shared\_ptr goes out of scope.

b. If you initialize shared\_ptr with some other shared\_ptr.

c. If you reset shared\_ptr.

5. Reference count doesn't work when we use reference (Address of the shared\_ptr) or pointer of shared\_ptr.  
  
  
\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**weak\_ptr in c++**

0. If we say unique\_ptr is for unique ownership and shared\_ptr is for shared ownership then

weak\_ptr is for non-ownership smart pointer.

1. It actually reference to an object which is managed by shared\_ptr.

This means to have a weak\_ptr, there need to be an shared\_ptr

2. A weak\_ptr is created as a copy of shared\_ptr.

3. We have to convert weak\_ptr to shared\_ptr in order to use the managed object.

4. It is used to remove cyclic dependency between shared\_ptr.  
  
  
  
\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_