**SmartPointers**

**What:**

Using smart pointers, we can make pointers to work in way that we don’t need to explicitly call delete. Smart pointer is a wrapper class over a pointer with operator like \* and -> overloaded.

There are three types of smart pointer:

1. Shared pointer

2. Unique pointer

3. Weak pointer

**When:**

Its used on object so the memory associated with a object is deallocated automatically when the pointer goes out of the scope.

**Why:**

For automatic deallocation of the memory associated with the pointer/ object when they goes out of scope.

**Advantages:**

It gurantess that memory leak is completely avoided.

Unique pointer:

A unique\_ptr does not share its pointer. It cannot be copied to another unique\_ptr, passed by value to a function, or used in any C++ Standard Library algorithm that requires copies to be made. A unique\_ptr can only be moved. This means that the ownership of the memory resource is transferred to another unique\_ptr and the original unique\_ptr no longer owns it. We recommend that you restrict an object to one owner, because multiple ownership adds complexity to the program logic. Therefore, when you need a smart pointer for a plain C++ object, use unique\_ptr, and when you construct a unique\_ptr, use the make\_unique helper function.

**Syntax:**

std::unique\_ptr<type> variable;

**snippet:**

std::unique\_ptr<int> p1(new int(5));

std::unique\_ptr<int> p2 = p1; //Compile error.

std::unique\_ptr<int> p3 = std::move(p1); // Transfers ownership. p3 now owns the // memory and p1 is set to nullptr.

p3.reset(); //Deletes the memory.

p1.reset(); //Does nothing.

Shared pointer:

Objects of shared\_ptr types have the ability of *taking ownership* of a pointer and *share* that ownership: once they take ownership, the group of owners of a pointer become responsible for its deletion when the last one of them releases that ownership.  
  
shared\_ptr objects release ownership on the object they *co-own* as soon as they themselves are destroyed, or as soon as their value changes either by an assignment operation or by an explicit call to shared\_ptr::reset. Once all shared\_ptr objects that share ownership over a pointer have released this ownership, the managed object is deleted.

**Syntax:**

std::shared\_ptr<type> variable;

**snippet:**

std::shared\_ptr<int> p0(new int(5)); // valid, allocates 1 integer and initialize it // with value 5

std::shared\_ptr<int[]> p1(new int[5]); // valid, allocates 5 integers

std::shared\_ptr<int[]> p2 = p1; //Both now own the memory.

p1.reset(); //Memory still exists, due to p2.

p2.reset(); //Deletes the memory, since no one else owns the memory.

Weak pointer:

A weak\_ptr is created as a copy of shared\_ptr. It provides access to an object that is owned by one or more shared\_ptr instances, but does not participate in reference counting. The existence or destruction of weak\_ptr has no effect on the shared\_ptr or its other copies. It is required in some cases to break circular references between shared\_ptr instances.

**Syntax:**

std::shared\_ptr<type> var1;

std::weak\_ptr<type> var2 {var1};

**snippet:**

std::shared\_ptr<int> p1 = std::make\_shared<int>(5);

std::weak\_ptr<int> wp1 {p1}; //p1 owns the memory.

{

std::shared\_ptr<int> p2 = wp1.lock(); //Now p1 and p2 own the memory.

// p2 is initialized from a weak pointer, so

// you have to check if the memory still exists!

if (p2) {

do\_something\_with(p2);

}

}

//p2 is destroyed. Memory is owned by p1.

p1.reset(); // Delete the memory.

std::shared\_ptr<int> p3 = wp1.lock();

//Memory is gone, so we get an empty shared\_ptr.

if (p3) { // code will not execute

action\_that\_needs\_a\_live\_pointer(p3);

}