**Raw Pointer *are nothing but* Classic pointers in C++**

A *pointer* is a type of variable. It stores the address of an object in memory, and is used to access that object. A *raw pointer* is a pointer whose lifetime is not controlled by an encapsulating object, such as a [smart pointer](https://docs.microsoft.com/en-us/cpp/cpp/smart-pointers-modern-cpp?view=msvc-160). A raw pointer can be assigned the address of another non-pointer variable, or it can be assigned a value of [**nullptr**](https://docs.microsoft.com/en-us/cpp/cpp/nullptr?view=msvc-160). A pointer that hasn't been assigned a value contains random data.

A pointer can also be *dereferenced* to retrieve the value of the object that it points at. The *member access operator* provides access to an object's members.

C++

int\* p = nullptr; // declare pointer and initialize it

// so that it doesn't store a random address

int i = 5;

p = &i; // assign pointer to address of object

int j = \*p; // dereference p to retrieve the value at its address

A pointer can point to a typed object or to **void**. When a program allocates an object on the [heap](https://wikipedia.org/wiki/Heap) in memory, it receives the address of that object in the form of a pointer. Such pointers are called *owning pointers*. An owning pointer (or a copy of it) must be used to explicitly free the heap-allocated object when it's no longer needed. Failure to free the memory results in a *memory leak*, and renders that memory location unavailable to any other program on the machine. Memory allocated using **new** must be freed by using **delete** (or **delete[]**). For more information, see [new and delete operators](https://docs.microsoft.com/en-us/cpp/cpp/new-and-delete-operators?view=msvc-160).

**Smart Pointers – This is modern C++ (Refer C++ 11, 14, 17 and further)**

*Smart Pointer* – In modern C++ programming the STL include *smart pointer,* which help to ensure that the program is free from memory leak and resources leaks and exception-safe.

*Smart Pointer* are defined in the std namespace in the [<memory>](https://docs.microsoft.com/en-us/cpp/standard-library/memory?view=msvc-160) header file. They are crucial to the [RAII](https://docs.microsoft.com/en-us/cpp/cpp/object-lifetime-and-resource-management-modern-cpp?view=msvc-160) or *Resource Acquisition Is Initialization* programming idiom. The main goal of this idiom is to ensure that resource acquisition occurs at the same time that the object is initialized, so that all resources for the object are created and made ready in one line of code. In practical terms, the main principle of RAII is to give ownership of any heap-allocated resource—for example, dynamically-allocated memory or system object handles—to a stack-allocated object whose destructor contains the code to delete or free the resource and also any associated clean-up code.

* **unique\_ptr**  
  Allows exactly one owner of the underlying pointer. Use as the default choice for POCO unless you know for certain that you require a shared\_ptr. Can be moved to a new owner, but not copied or shared. Replaces auto\_ptr, which is deprecated. Compare to boost::scoped\_ptr. unique\_ptr is small and efficient; the size is one pointer and it supports rvalue references for fast insertion and retrieval from C++ Standard Library collections. Header file: <memory>. For more information, see [How to: Create and Use unique\_ptr Instances](https://docs.microsoft.com/en-us/cpp/cpp/how-to-create-and-use-unique-ptr-instances?view=msvc-160) and [unique\_ptr Class](https://docs.microsoft.com/en-us/cpp/standard-library/unique-ptr-class?view=msvc-160).

A std::unique\_ptr owns of the object it points to and no other smart pointers can point to it. When the std::unique\_ptr goes out of scope, the object is deleted. This is useful when you are working with a temporary, dynamically-allocated resource that can get destroyed once out of scope.

A std::unique\_ptr is created like this:

* + std::unique\_ptr<int> p1(new int);
  + std::unique\_ptr<int[]> p2(new int[50]);
  + std::unique\_ptr<Object> p3(new Object("Lamp"));

It is also possible to construct std::unique\_ptr with the help of the special function std::make\_unique, like this:

* + std::unique\_ptr<int> p1 = std::make\_unique<int>();
  + std::unique\_ptr<int[]> p2 = std::make\_unique<int[]>(50);
  + std::unique\_ptr<Object> p3 = std::make\_unique<Object>("Lamp");

**Characteristic of unique\_ptr**

1. main feature is **unique\_ptr** is, it is destroyed when gone out of scope
2. unique pointer is unique, it **DOES NOT ALLOW** to make copy of itself (prevention of pointer)
3. It will allow to move unique pointer to another unique pointer (differences the old one)
4. Basically copy\_contructor does not work
5. <https://en.cppreference.com/w/cpp/memory/unique_ptr>

* **shared\_ptr**Reference-counted smart pointer. Use when you want to assign one raw pointer to multiple owners, for example, when you return a copy of a pointer from a container but want to keep the original. The raw pointer is not deleted until all shared\_ptr owners have gone out of scope or have otherwise given up ownership. The size is two pointers; one for the object and one for the shared control block that contains the reference count. Header file: <memory>.
* A std::shared\_ptr owns the object it points to but, unlike std::unique\_ptr, it allows for multiple references. A special internal counter is decreased each time a std::shared\_ptr pointing to the same resource goes out of scope. This technique is called **reference counting**. When the very last one is destroyed the counter goes to zero and the data will be deallocated.
* This type of smart pointer is useful when you want to share your dynamically-allocated data around, the same way you would do with raw pointers or references.

A std::shared\_ptr is constructed like this:

* + std::shared\_ptr<int> p1(new int);
  + std::shared\_ptr<Object> p2(new Object("Lamp"));

There is an alternate way to build a std::shared\_ptr, powered by the special function std::make\_shared:

* + std::shared\_ptr<int> p1 = std::make\_shared<int>();
  + std::shared\_ptr<Object> p2 = std::make\_shared<Object>("Lamp");

**Characteristic of shared\_ptr**

* + 1. One of the main features of std::shared\_ptr is the ability to track how many pointers refer to the same resource. You can get information on the number or references with the method use\_count().

<https://www.internalpointers.com/post/beginner-s-look-smart-pointers-modern-c>

* + 1. Ownership of an object can be shared to another obj, (copy\_constructor works here)
    2. <https://en.cppreference.com/w/cpp/memory/shared_ptr>
* **weak\_ptr**Special-case smart pointer for use in conjunction with shared\_ptr. A weak\_ptr provides access to an object that is owned by one or more shared\_ptr instances, but does not participate in reference counting. Use when you want to observe an object, but do not require it to remain alive. Required in some cases to break circular references between shared\_ptr instances. Header file: <memory>.

A std::weak\_ptr is basically a std::shared\_ptr that doesn't increase the reference count. It is defined as a smart pointer that holds a **non-owning reference**, or a **weak reference**, to an object that is managed by another std::shared\_ptr.

* This smart pointer is useful to solve some annoying problems that you can't fix with raw pointers. We will see how shortly.

Create a std::weak\_ptr out of a std::shared\_ptr or another std::weak\_ptr.

* + std::shared\_ptr<int> p\_shared = std::make\_shared<int>(100);
  + std::weak\_ptr<int> p\_weak1(p\_shared);
  + std::weak\_ptr<int> p\_weak2(p\_weak1);

In the example above p\_weak1 and p\_weak2 point to the same dynamic data owned by p\_shared, but the reference counter doesn't grow.

**Characteristic of shared\_ptr**

* + 1. weak\_ptr is a smart pointer that holds a non-owning (“weak) reference of the object, meaning it does not own that object.
    2. Usefulness – to track the objects, holds temporary ownership of shared pointer
    3. weak\_ptr count will always remain 1 (when referenced)
    4. life of weak\_ptr is alive until shared\_ptr is alive.

*[Reference links]*

<https://www.internalpointers.com/post/beginner-s-look-smart-pointers-modern-c>

<https://en.cppreference.com/w/cpp/memory/shared_ptr>

<https://rufflewind.com/2016-03-05/unique-ptr>