Smart Pointers

- In Smart Pointer when the object is destroyed it frees the memory as well. So, we don't need to delete it as Smart Pointer does will handle it.
- A *Smart Pointer* is a wrapper class over a pointer with an operator like * and -> overloaded. The objects of the smart pointer class look like normal pointers. But, unlike *Normal Pointers* it can deallocate and free destroyed object memory.
- The idea is to take a class with a pointer, destructor and overloaded operators like * and ->. Since the destructor is automatically called when an object goes out of scope, the dynamically allocated memory would automatically be deleted.
- **Example:**

```
#include <iostream>
using namespace std;

// A generic smart pointer class
template <class T>
class SmartPointer
{
    private:
      T *ptr;   // Actual pointer

    public:
      SmartPointer(T *p = NULL)
      {
           ptr = p;
      }

      ~SmartPointer()
      {
          delete ptr;
      }

    // Overloading dereferencing operator
    T& operator*()
    {
        return *ptr;
    }


    // Overloading arrow operator so that members of T can be accessed like a pointer
    // useful if T represents a class or struct or union type
    T& operator->()
    {
        return *ptr;
    }
```

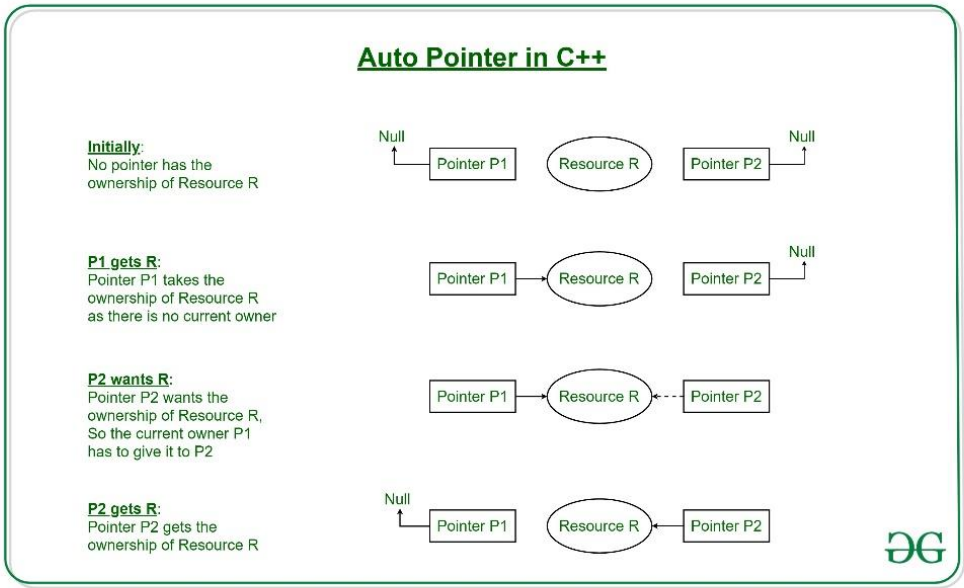
```
};

int main()
{
    SmartPointer<int> ptr(new int());
    *ptr = 20;
    cout << *ptr << endl;
}
```

- Types of Smart Pointers
  1.auto_ptr
  2.unique_ptr
  3.shared_ptr
  4.weak_ptr

1. auto_ptr:

- auto_ptr is a smart pointer that manages an object obtained via new expression and deletes that object when auto_ptr itself is destroyed.
- An object when described using auto_ptr class it stores a pointer to a single allocated object which ensures that when it goes out of scope, the object it points to must get automatically destroyed.
- It is based on **exclusive ownership model** i.e. two pointers of the same type can't point to the same resource at the same time.



- As shown in the below program, copying or assigning of pointers changes the ownership i.e. source pointer has to give ownership to the destination pointer.
- **Example:**

  Input:

```
#include <iostream>
#include <memory>
```

```
using namespace std;

class A
{
   public:
      void show() { cout << "A::show()" << endl; }
};

int main()
{
   // p1 is an auto_ptr of type A
   auto_ptr<A> p1(new A);
   p1->show();

   // returns the memory address of p1
   cout << p1.get() << endl;

   // copy constructor called, this makes p1 empty.
   auto_ptr<A> p2(p1);
   p2->show();

   // p1 is empty now
   cout << p1.get() << endl;

   // p1 gets copied in p2
   cout << p2.get() << endl;

   return 0;
}
```

Output:

```
A::show()
0x1b42c20
A::show()
0
0x1b42c20
```

- The copy constructor and the assignment operator of auto_ptr do not actually copy the stored pointer instead they transfer it, leaving the first auto_ptr object empty. This was one way to implement strict ownership so that only one auto_ptr object can own the pointer at any given time i.e. auto_ptr should not be used where copy semantics are needed.

- **Why is auto_ptr deprecated?**
  It takes ownership of the pointer in a way that no two pointers should contain the same object. Assignment transfers ownership and resets the rvalue auto pointer to a null pointer. Thus, they can't be used within STL containers due to the aforementioned inability to be copied.

2. unique_ptr:

- unique_ptr was as a replacement for auto_ptr.
- unique_ptr is a new facility with similar functionality, but with improved security (no fake copy assignments), added features (deleters) and support for arrays. It is a container for raw pointers. It explicitly prevents copying of its contained pointer as would happen with normal assignment i.e. it allows exactly one owner of the underlying pointer.
- So, when using unique_ptr there can only be at most one unique_ptr at any one resource and when that unique_ptr is destroyed, the resource is automatically claimed. Also, since there can only be one unique_ptr to any resource, so any attempt to make a copy of unique_ptr will cause a compile-time error.

```
unique_ptr<A> ptr1 (new A);
 // Error: can't copy unique_ptr
 unique_ptr<A> ptr2 = ptr1:
```

- But, unique_ptr can be moved using the new move semantics i.e. using std::move() function to transfer ownership of the contained pointer to another unique_ptr.

```
// Works, resource now stored in ptr2
unique_ptr<A> ptr2 = move(ptr1):
```

- So, it's best to use unique_ptr when we want a single pointer to an object that will be reclaimed when that single pointer is destroyed.
- **Example:**

Input:

```
#include <iostream>
#include <memory>
using namespace std;

class A
{
   public:
      void show()
      {
         cout << "A::show()" << endl;
      }
};

int main()
{
   unique_ptr<A> p1(new A);
   p1->show();

   // returns the memory address of p1
   cout << p1.get() << endl;

   // transfers ownership to p2
   unique_ptr<A> p2 = move(p1);
   p2->show();
   cout << p1.get() << endl;
```

```
    cout << p2.get() << endl;

    // transfers ownership to p3
    unique_ptr<A> p3 = move(p2);
    p3->show();
    cout << p1.get() << endl;
    cout << p2.get() << endl;
    cout << p3.get() << endl;

    return 0;
}
```

Output:

```
A::show()
0x1c4ac20
A::show()
0
0x1c4ac20
A::show()
0
0
0x1c4ac20
```

- **When to use unique_ptr?**
  Use unique_ptr when you want to have single ownership(Exclusive) of the resource. Only one unique_ptr can point to one resource. Since there can be one unique_ptr for single resource its not possible to copy one unique_ptr to another.

3. shared_ptr:

- A shared_ptr is a container for raw pointers. It is a *reference counting ownership model* i.e. it maintains the reference count of its contained pointer in cooperation with all copies of the shared_ptr. So, the counter is incremented each time a new pointer points to the resource and decremented when the destructor of the object is called.
- **Reference Counting:** It is a technique of storing the number of references, pointers or handles to a resource such as an object, block of memory, disk space or other resources.
- An object referenced by the contained raw pointer will not be destroyed until reference count is greater than zero i.e. until all copies of shared_ptr have been deleted.
- So, we should use shared_ptr when we want to assign one raw pointer to multiple owners.
- **Example:**
  Input:

```
#include <iostream>
#include <memory>
using namespace std;

class A
```

```
{
   public:
      void show()
      {
         cout << "A::show()" << endl;
      }
};

int main()
{
   shared_ptr<A> p1(new A);
   cout << p1.get() << endl;
   p1->show();
   shared_ptr<A> p2(p1);
   p2->show();
   cout << p1.get() << endl;
   cout << p2.get() << endl;

   // Returns the number of shared_ptr objects referring to the same managed object.
   cout << p1.use_count() << endl;
   cout << p2.use_count() << endl;

   // Relinquishes ownership of p1 on the object and pointer becomes NULL
   p1.reset();
   cout << p1.get() << endl;
   cout << p2.use_count() << endl;
   cout << p2.get() << endl;

   return 0;
}
```

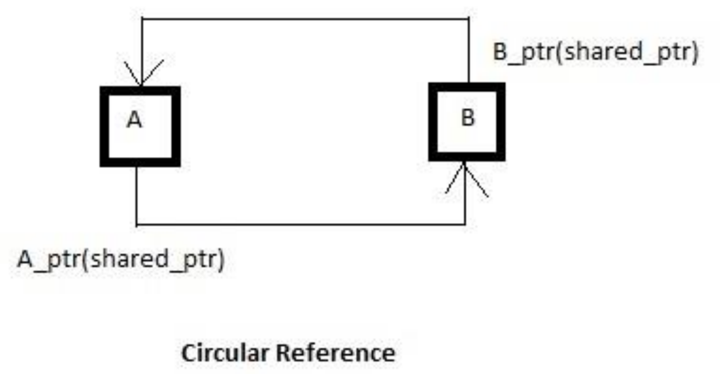
Output:

```
0x1c41c20
A::show()
A::show()
0x1c41c20
0x1c41c20
2
2
0
1
0x1c41c20
```

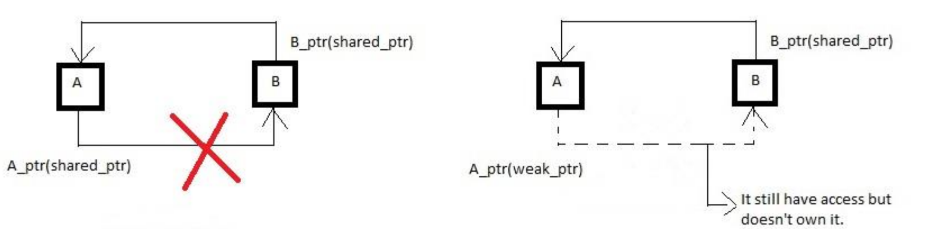
- **When to use shared_ptr?**
  Use shared_ptr if we want to share ownership of a resource. Many shared_ptr can point to a single resource. shared_ptr maintains reference count for this propose. when all shared_ptr's pointing to resource goes out of scope the resource is destroyed.

4. weak_ptr:

- 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.
- **Cyclic Dependency (Problems with shared_ptr):** Let's consider a scenario where we have two classes A and B, both have pointers to other classes. So, it's always like A is pointing to B and B is pointing to A. Hence, use_count will never reach zero and they never get deleted. The reason is if suppose pointers are holding the object and requesting for other objects then they may form a **Deadlock.**



- This is the reason we use weak pointers(weak_ptr) as they are not reference counted. So, the class in which weak_ptr is declared doesn't have a stronghold of it i.e. the ownership isn't shared, but they can have access to these objects.



- So, in case of shared_ptr because of cyclic dependency use_count never reaches zero which is prevented using weak_ptr, which removes this problem by declaring A_ptr as weak_ptr, thus class A does not own it, only have access to it and we also need to check the validity of object as it may go out of scope. In general, it is a design issue.
- **When to use 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).