# Problem with the raw pointers in C++

When we create raw pointers in C++, many times the program is unable to deallocate the memory for some reason which causes the memory leak. The memory leak is the most significant issue in C++. Consider the following example.

```
#include <iostream>
#include <string>

using namespace std;

class MyClass {
public:
    void method()
    {
        int* ptr = new int();
        *ptr = 100;
        int res = *ptr / 0;  // Division by zero error and crash
        delete ptr; // deallocate memory
    }
};


int main()
{
    MyClass m1;
    m1.method();
}
```

We can clearly see that before the deallocation of the memory, the program crashes due to division by zero error. This causes a memory leak. The memory allocated to the preceding raw pointer isn't released as the memory pointed by pointers is not freed up during the stack unwinding process. Whenever an exception is thrown by a function and the exception is not handled by the same function, stack unwinding is guaranteed. However only the automatic local variables will be cleaned up during the stack unwinding process, not the memory pointed by the pointers. This results in memory leaks.

# Smart Pointers

A smart pointer is a class that wraps a raw pointer and maintains the lifetime of the pointer. The main job of a smart pointer is to remove the chances of a memory leak. It also helps to avoid the double deletion problem.

Smart pointer overloads the -> and * operators to mimic the real pointer.

```
// Smart Pointer implementation in C++

template <class T>
class CSmartPointer
{
private:
    T* m_ptr;
public:
    // ctor
    explicit CSmartPointer(T* t = nullptr) :m_ptr(t) {}
    //dtor
    ~CSmartPointer() {
        // release memory
        delete m_ptr;
    }

    // overload dereference operator
    T& operator* ()
    {
        return *m_ptr;
    }

    // overload arrow operator
    T* operator->()
    {
        return m_ptr;
    }

};
```

# Smart Pointers provided by C++

1) auto_ptr *(deprecated)*
2) unique_ptr
3) shared_ptr
4) weak_ptr

# auto_ptr *(deprecated)*

- The auto_ptr smart pointer takes a raw pointer, wraps it, and ensures the memory pointed by the raw pointer is released back whenever the auto_ptr object goes out of scope.
- At any time, only one auto_ptr smart pointer can point to an object. Hence, whenever one auto_ptr pointer is assigned to another auto_ptr pointer, the ownership gets transferred to the auto_ptr instance that has received the assignment; the same happens when an auto_ptr smart pointer is copied.

```
int main()
{
   auto_ptr<Student> ptr1(new Student(1, "ptr1", 3));
   auto_ptr<Student> ptr2(new Student(2, "ptr2", 4));

   ptr2 = ptr1; // transfer ownership
   ptr2->print();

   return 0;

}
```

In the above example, when we assign a pointer from ptr1 to ptr2, the ownership transfers from ptr1 to ptr2, and ptr1 will no longer pointing to anything.

### Drawback of auto_ptr:

- An auto_ptr object can't be stored in an STL container

- The auto_ptr copy constructor will remove the ownership from the original source, that is, auto_ptr.
- The auto_ptr copy assignment operator will remove the ownership from the original source, which is auto_ptr.

The above points violate the copy ctor and assignment operator intention.

# unique_ptr

The unique_ptr is a smart pointer that wraps the raw pointer and helps to automatically deallocate the memory that is allocated dynamically. The unique_ptr allows only one smart pointer to exclusively own a heap-allocated object.

The unique_ptr is a smart pointer in C++ that provides exclusive ownership of a dynamically allocated object. It ensures that only one unique_ptr can point to the object, and when the unique_ptr goes out of scope or explicitly reset, it automatically deallocates the associated memory.

**Create unique_ptr pointer:**

To create a unique_ptr we can use make_unique or constructor.

```
std::unique_ptr<int> uniqueInt1 = std::make_unique<int>(42);

std::unique_ptr<int> uniqueInt2(new int(24));
```

**Transfer the ownership**

```
std::unique_ptr<int> transferredPtr = std::move(uniqueInt1);
```

**Checking for NULL**

```
if (uniqueInt1.get() == nullptr) {
    std::cout << "uniqueInt1 is null" << std::endl;
}
```

**Resetting a unique_ptr. This releases the ownership and deallocates the managed object.**

```
uniqueInt1.reset(); // Releases ownership and deallocates the integer
```

**Custom delete method**

```
// Using a lambda function as a custom deleter
auto customDeleter = [](int* ptr) {
    std::cout << "Custom deleter called for " << *ptr << std::endl;
    delete ptr;
};

std::unique_ptr<int, decltype(customDeleter)> customPtr(new int(42),
customDeleter);
```

# shared_ptr

The shared_ptr is a smart pointer that allows multiple smart pointers to share ownership of the same dynamically allocated object. It keeps, track of the number of shared pointers pointing to the object using a reference count mechanism. When the last shared_ptr pointing to the object goes out of scope or is explicitly reset, the object is automatically deallocated.

**Create shared_ptr:**

```
std::shared_ptr<int> sharedInt1 = std::make_shared<int>(42);
std::shared_ptr<int> sharedInt2(new int(24));
```

**Sharing ownership:**

```
std::shared_ptr<int> sharedIntCopy = sharedInt1; // Now both sharedInt1 and
sharedIntCopy own the same integer
```

**Checking count:**

```
int count = sharedInt1.use_count(); // Returns the number of shared pointers
sharing ownership
```

# weak_ptr

weak_ptr is a smart pointer in C++ that is primarily used in conjunction with shared_ptr to break potential circular references and avoid memory leaks. It allows to observe or access the shared object without affecting its reference count.

**Breaking Circular References:**

Consider there are three classes: A, B and C. Class A and B have an instance of C, while C has an instance of A and B. A depends of C and C depends on A, similarly B depends on C and C depends on B.

```
#include <iostream>
#include <string>
#include <memory>
#include <sstream>

using namespace std;

class C;

class A
{
private:
    shared_ptr<C> ptr;
public:
    A() {
        cout << "ctor A" << endl;
    }

    ~A() {
        cout << "dtor A" << endl;
    }

    void setObject(shared_ptr<C> ptr) {
        this->ptr = ptr;
    }
};

class B
{
private:
    shared_ptr<C> ptr;

public:
    B() {
        cout << "ctor B" << endl;
```

```
    }

    ~B() {
        cout << "dtor B" << endl;
    }

    void setObject(shared_ptr<C> ptr) {
        this->ptr = ptr;
    }
};

class C {
private:
    shared_ptr<A> ptr1;
    shared_ptr<B> ptr2;

public:
    C(shared_ptr<A> ptr1, shared_ptr<B> ptr2) {
        cout << "ctor C" << endl;
        this->ptr1 = ptr1;
        this->ptr2 = ptr2;
    }

    ~C() {
        cout << "dtor" << endl;
    }

};


int main()
{
    shared_ptr<A> a(new A());
    shared_ptr<B> b(new B());
    shared_ptr<C> c( new C(a, b));

    a->setObject(c);
    b->setObject(c);

    //PROBLEM!
```

```
   /* This code never call the destructor.
   */

   return 0;

}
```

The above code never calls the destructor. The reason is that shared_ptr internally maintain the reference counting algorithm to decide whether the shared object has to be destructed. However, it fails here because object A can't be deleted unless object C is deleted. Object C can't be deleted unless object A is deleted. Similarly for the other cyclic dependence objects.

To overcome, we need the weak_ptr.

```
#include <iostream>
#include <string>
#include <memory>
#include <sstream>

using namespace std;

class C;

class A
{
private:
    //shared_ptr<C> ptr;
    weak_ptr<C> ptr;
public:
    A() {
        cout << "ctor A" << endl;
    }

    ~A() {
        cout << "dtor A" << endl;
    }

    void setObject(weak_ptr<C> ptr) {
        this->ptr = ptr;
    }
};

class B
{
private:
    //shared_ptr<C> ptr;
```

```
    weak_ptr<C> ptr;
public:
    B() {
        cout << "ctor B" << endl;
    }

    ~B() {
        cout << "dtor B" << endl;
    }

    void setObject(weak_ptr<C> ptr) {
        this->ptr = ptr;
    }
};

class C {
private:
    shared_ptr<A> ptr1;
    shared_ptr<B> ptr2;

public:
    C(shared_ptr<A> ptr1, shared_ptr<B> ptr2) {
        cout << "ctor C" << endl;
        this->ptr1 = ptr1;
        this->ptr2 = ptr2;
    }

    ~C() {
        cout << "dtor" << endl;
    }

};


int main()
{
    shared_ptr<A> a(new A());
    shared_ptr<B> b(new B());
    shared_ptr<C> c( new C(a, b));

    a->setObject(c);
    b->setObject(c);

    return 0;

}
```

**OUTPUT:**

```
ctor A
ctor B
ctor C
dtor
dtor B
dtor A
```