C++ smart pointer

They provide greater functionality than a normal pointer. They help in:

**Construction and destruction**

Smart pointers release the memory they allocated by means of normal destructor activity. In this way, they prevent resource leaks.

**Copying and assignment**

They give sufficient freedom and flexibility for making shallow/deep copies.

**Dereferencing**

Smart pointers can be used to perform the lazy fetching strategy.

template<class T> // template for smart

class SmartPtr { // pointer objects

public:

SmartPtr(T\* realPtr = 0); // create a smart ptr to an

// obj given a dumb ptr to

// it; uninitialized ptrs

// default to 0 (null)

SmartPtr(const SmartPtr& rhs); // copy a smart ptr

~SmartPtr(); // destroy a smart ptr

// make an assignment to a smart ptr

SmartPtr& operator=(const SmartPtr& rhs);

T\* operator->() const; // dereference a smart ptr

// to get at a member of

// what it points to

T& operator\*() const; // dereference a smart ptr

private:

T \*pointee; // what the smart ptr

}; // points to

## Construction, Assignment, and Destruction of Smart Pointers

Consider the auto\_ptr template from the standard C++ library. An auto\_ptr object is a smart pointer that points to a heap-based object until it (the auto\_ptr) is destroyed. When that happens, the auto\_ptr's destructor deletes the pointed-to object.

template<class T>

class auto\_ptr {

public:

auto\_ptr(T \*ptr = 0): pointee(ptr) {}

~auto\_ptr() { delete pointee; }

...

private:

T \*pointee;

};

This works fine provided only one auto\_ptr owns an object. But what should happen when an auto\_ptr is copied or assigned?

auto\_ptr<TreeNode> ptn1(new TreeNode);

auto\_ptr<TreeNode> ptn2 = ptn1; // call to copy ctor;

// what should happen?

auto\_ptr<TreeNode> ptn3;

ptn3 = ptn2; // call to operator=;

// what should happen?