Smart Pointers

Matt Warner

1 Brief

Smart Pointers are objects that are designed to look, act, and feel like build-in pointers, but to offer greater functionality. They have a variety of applications, including resource management and the automation of repetitive coding tasks.

When you use smart pointers in place of C++ built-in pointers, you gain control over the following aspects of pointer behaivior.

- **Construction and destruction**. You determine when a smart pointer is created and destroyed. It is common to give smart pointers a default value of 0 to avoid the headaches associated with uninitalized pointers. Some smart pointers are made responsible for deleting the object they point to when the last smart pointer pointing to the object is destroyed. This can go a long way toward eliminating resource leaks.
- **Copying and assignment**. You control what happens when a smart pointer is copied or is involved in an assignment. For some smart pointer types, the desired behavior is to automatically copy or make an assignment to what is pointed to, i.e., to perform a deep copy. For others, only the pointer itself should be copied or assigned. For still others, these operations should not be allowed at all. Regardless of what behavior you consider “right,” the use of smart pointers lets you call the shots.
- **Dereferencing**. What should happen when a client refers to the object pointed to by a smart pointer? You get to decide. You could, for example, use smart pointers to help implemenet the lazy fetching strategy.

The implementation of a smart pointer would look something like this:

```cpp
// # smartpointer.h
template <class T> class SmartPtr {
public:

    // Ctor
    SmartPtr(T* Realptr = 0);
    // Copy ctor
    SmartPtr(const SmartPtr &rhs);

    // dtor
    ~SmartPtr();

    // Copy assignment ctor
    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

public:
    T* pointee;

};
```

Note:-

The copy constructor and assignment operator are both shown public here. For smart pointer classes where copying and assignment are not allowed, they would typically be declared private.

The two dereferencing operators are declared const, because dereferencing a pointer doesn't modify it.

Finally, each smart pointer-to-T object is implemented by containing a dumb pointer-to-T within it. It is this dumb pointer that does the actuall pointing.

Although You certainly can create your own implementation of a smart pointer, its important to know that the C++ standary library already provides you with various smart pointer utilities, which can be utilized in your code by including the ***memory*** library. In this library, there are three types of smart pointers, all of which have different use cases.

- **unique_ptr**
- **shared_ptr**
- **weak_ptr**

2 unique_ptr

unique pointers are smart pointers that are often used when we want single or exclusive ownership of a resource. Unique pointers are automatically deleted and have their memory freed when they go out of scope, So sole ownership of a resource is a must due to the issue of double deletion.

For example, lets say we had the following code:

```
int main( ) {
  smart_ptr<string> ptr(new string("Hello"))

  {
    smart_ptr<string> ptr2(ptr);
  }
  return EXIT_SUCCESS:
}
```

In this example, we have are using a smart pointer class that automatically deletes the smart pointer object when it goes out of scope. We create a smart pointer that holds a raw pointer to an string object, which we named **ptr**. We are then creating another smart pointer that is a copy of our first pointer, **ptr2**. The example code is problematic because when ptr2 goes and of scope and gets destroyed, it frees the memory that it owned, which is also owned by **ptr**. So if you were to then try and access the the memory owned by **ptr** Your program might crash. Furthermore, when **ptr** gets destroyed, the program might crash because the memory has already been freed when **ptr2** was destroyed.

When implementing your own smart pointer class. If you are not going to use reference counting, which will be talked about later, you should make copy and assignment constructors unavalible to prevent bugs in your code. What you should instead do is implement a move copy constructor and move assigment operator that tranfers ownership of an object. Heres what that implementation might look like.

```
#ifndef SMART_PTR_H
#define SMART_PTR_H

template <class T> smart_ptr {
public:

// Constructor.
smart_ptr(T* raw_ptr = nullptr) noexcept : pointee(raw_ptr) {}

// Move Copy Constructor.
smart_ptr(smart_ptr<T>&& rhs) noexcept {

  // Transfer owndership of *pointee to *this
  pointee = rhs.pointee;

  // rhs no longer owns anything
  rhs.pointee = nullptr;
}

// Move assignment operator.
smart_ptr<T>& operator=(smart_ptr<T> &&rhs) {

  // Prevent self assignement
  if (*this == rhs) { return *this; }
  // Free memory previously owned
  delete pointee;
  // Tranfer ownership
  pointee = rhs.pointee;
  // rhs no longer owns anything
  rhs.pointee = nullptr;

  return *this;
}

// Destructor.
~smart_ptr() noexcept { delete pointee; }

// Overloading operator[] for array types
// T& operator[](int i) { return pointee[i]; }

private:
  T* pointee;
};

T& operator*() { return *pointee; }
T* operator->( return ) { return pointee; }

// Removing access to these.
smart_ptr(const smart_ptr<T> &rhs) = delete;
smart_ptr<T>& operator=(const smart_ptr<T> &rhs) = delete;
#endif
```

Note:-

When after using the move constuctors, its important to note that the objects that has it memory stolen is not destroyed, it will just be pointing to nothing i.e., **nullptr**. So dont try and access it without making it point to something new.

Additionally, we are following a few Core Guidelines here.
Core Guideline C.66: Make move operations noexcept.
Core Guideline C.64: A move operation should move and leave its source in a valid state.

Ideally, that moved-from should be the default value of the type.

As you can see, The implementations of the copy constructor and the assignment operator strictly **tranfer ownership**. farther down the class, we make sure to delete both the regular copy and assignment constructor. This is an added layer of protection that forces users of the smart pointer to use **std::move** when copying their pointers. This is what that looks like:

```
smart_ptr<int> ptr(new int(4));
// Using the move copy constructor
smart_ptr<int> ptr2(std::move(ptr));
```

```
// Using the move assignment operator
ptr2 = std::move(ptr);
```

We can use **std::move** this way because our constructor parameters are **r-value references**. That is was the double ampersand (&&) is. Without making the parameters an r-value reference, the compiler would yell at us if you tried using std::move. Because std::move casts a given argument into a r-value reference, so the constructors need to take r-value references as a parameter.

2.1 Custom deleters

The above implementation works fine for the most part, but completely falls apart when you try and create array pointers. For starters, We are not overloading **operator**[], so trying to index the array would simply not work. You would instead need to do some pointer arithmatic via some auxillary pointer to access the elements. The larger problem, however, is that the destructor will only ever call **delete** on the raw pointer which is a huge problem as **T***[] types should always be matched with **delete**[].

In order to implement a smart pointer class that safely allows you to creat pointer arrays, we need to introduce a second template parameter, a **deleter**. These are just **functors** that specify which verison of delete to call when the smart pointer is being destroyed. In our implementation of a unique pointer, we would write two custom deleters, one for non array types and one for array types.

```
// Custom deleter functor for non array types
template <typename Tp>
struct default_delete {

  default_delete() { }

  template <typename Up>
  default_delete(const default_delete<Up>& { }

  void operator()(Tp* ptr) const {
    static_assert(sizeof(Tp) > 0,
    "cannot delete pointer to incomplete type");

    delete ptr;
  }
};

// Custom deleter functor for array types
template <typename Tp>
struct default_delete<Tp[]> {

  void operator()(Tp* ptr) const {
    static_assert(sizeof(Tp) > 0,
    "cannot delete pointer to incomplete type");

    delete [] ptr;
  }
};
```

For our template parameters, we would then have:

```
template <class Tp, class Tp_Deleter = std::default_delete<Tp>>
```

These two template paramters form a pair where **Tp*** will point to the object managed by smart_ptr and **Tp_Deleter** will specify how to deallocate or destroy the object. With that being said, our private data member will be changed to a tuple, which will cotain this pair. So far, here are the changes to consider:

```
template<typename Tp, typename Tp_Deleter = default_delete<Tp>>
class smart_ptr {

  typedef std::tuple<Tp*,Tp_Deleter> tuple_type;

public:
  typedef Tp* pointer;
  typedef Tp element_type;
  typedef Tp_Deleter deleter_type;

  ...

private:
tuple_type M_t;
};
```

Now that we are working with a tuple instead of a raw pointer as our private data member, we need to make some changes to our constructors:

```
public:

// Constructors.
smart_ptr() : M_t(pointer(), deleter_type()) { }       // Sets tuple with type defaults.

smart_ptr(pointer p) : M_t(p, deleter_type()) { }     // passed a pointer but no deleter

smart_ptr(pointer p, deleter_type deleter) : M_t(p, deleter) {} // passed a pointer and
↪  deleter

// Move constructors.
smart_ptr(smart_ptr&& rhs) noexcept {
  std::get<0>(M_t) = rhs.release();

  std::get<1>(M_t) = std::forward<deleter_type>(rhs.get_deleter());
};

// class helpers
pointer get() const {
  return std::get<0>(M_t);
}

typename std::add_lvalue_reference_t<deleter_type>::type
get_deleter() const {
  return std::get<1>(M_t);
}

pointer release() {

  // gets the pointer from rhs
  pointer p = std::get<0>(M_t);

  // rhs pointer points to null
  std::get<0>(M_t) = 0;

  return p;
}
```

In the move constructor, since we are working with a tuple, we can use **std::get** to get its elements. For our raw pointer we make a call to our helper function **release()** to move the pointer from **rhs** to **this**. The next step in our move constructor uses **std::forward** to move the deleter into our newely constructed object. This technique is called **Perfect forwarding**. This ensures that the deleter is forwarded with the appropriate value category (lvalue or rvalue). Here is a conceptual example of **std::forward**

```
template<typename T>
T&& std::forward<T>(T&& param) {

  if (is_lvalue_reference<T>::value) { // If T indicates lvalue do nothing
    return param;
  } else {
    return std::move(param) // cast to rvalue
  }

}
```