Homework 2: Shared Pointers (60 points; due Friday, October 22nd at 11:59 PM)

In this homework, you will make your own smart pointer type. You will write the following class to develop a referenced counted smart pointer. You will also implement a few other member functions to resemble the functionality of an ordinary raw pointer. Basically, this is a problem of designing a single, non-trivial class and overloading a few pointer related operators. You may not use any of the STLs except for stdexcept ("So what you're saying is 'exceptions is the only exception'?" – Former CS 30 Student) to do this, i.e., please don't try and use a `shared_ptr` to implement the class. You can start with this code, and you may add other member functions, if you want.

```
template <typename T>
class smart_ptr {
public:
    smart_ptr();
      // Create a smart_ptr that is initialized to nullptr. The
      // reference count should be initialized to nullptr.

    explicit smart_ptr(T* raw_ptr);
      // Create a smart_ptr that is initialized to raw_ptr. The
      // reference count should be one.  Make sure it points to
      // the same pointer as the raw_ptr.

    smart_ptr(const smart_ptr& rhs);
      // Copy construct a pointer from rhs. The reference count
      // should be incremented by one.

    smart_ptr(smart_ptr&& rhs);
      // Move construct a pointer from rhs.

    smart_ptr& operator=(const smart_ptr& rhs);
      // This assignment should make a shallow copy of the
      // right-hand side's pointer data. The reference count
      // should be incremented as appropriate.

    smart_ptr& operator=(smart_ptr&& rhs);
      // This move assignment should steal the right-hand side's
      // pointer data.

    bool clone();
      // If the smart_ptr is either nullptr or has a reference
      // count of one, this function will do nothing and return
      // false. Otherwise, the referred to object's reference
      // count will be decreased and a new deep copy of the
```

```
      // object will be created. This new copy will be the
      // object that this smart_ptr points and its reference
      // count will be one.

    int ref_count() const;
      // Returns the reference count of the pointed to data.

    T& operator*();
      // The dereference operator shall return a reference to
      // the referred object. Throws null_ptr_exception on
      // invalid access.

    T* operator->();
      // The arrow operator shall return the pointer ptr_.
      // Throws null_ptr_exception on invalid access.

    ~smart_ptr();          // deallocate all dynamic memory

private:
    T* ptr_;               // pointer to the referred object
    int* ref_;             // pointer to a reference count
};
```

Here is an example of how the above class might work.

```
int* p { new int { 42 } };
smart_ptr<int> sp1 { p };

// Ref Count is 1
cout << "Ref count is " << sp1.ref_count() << endl;
{
   smart_ptr<int> sp2 { sp1 };
   // Ref Count is 2
   cout << "Ref count is " << sp1.ref_count() << endl;
   // Ref Count is 2
   cout << "Ref count is " << sp2.ref_count() << endl;
}

// Ref Count is 1
cout << "Ref count is " << sp1.ref_count() << endl;

smart_ptr<int> sp3;

// Ref Count is 0
```

```
cout << "Ref count is " << sp3.ref_count() << endl;

sp3 = sp1;

// Ref Count is 2
cout << "Ref count is " << sp1.ref_count() << endl;
// Ref Count is 2
cout << "Ref count is " << sp3.ref_count() << endl;

smart_ptr<int> sp4 = std::move(sp1);

cout << *sp4 << " " << *sp3 << endl;        // prints 42 42
cout << *sp1 << endl;   // throws null_ptr_exception
```

The arrow operator will only compile and work if the referred object is a class type:

```
struct Point { int x = 2; int y = -5; };

int main ( )
{
   smart_ptr<Point> sp { new Point };
   cout << sp->x << " " << sp->y << endl;   // prints 2 -5
   return 0;
}
```

Here is an example of the clone member function:

```
smart_ptr<double> dsp1 { new double {3.14} };
smart_ptr<double> dsp2, dsp3;

dsp3 = dsp2 = dsp1;

cout << dsp1.ref_count() << " " << dsp2.ref_count() << " "
     << dsp3.ref_count() << endl; // prints 3 3 3
// prints 3.14 3.14 3.14
cout << *dsp1 << " " << *dsp2 << " " << *dsp3 << endl;

dsp1.clone();        // returns true

cout << dsp1.ref_count() << " " << dsp2.ref_count() << " "
     << dsp3.ref_count() << endl; // prints 1 2 2
// prints 3.14 3.14 3.14
cout << *dsp1 << " " << *dsp2 << " " << *dsp3 << endl;
```

Requirements/Hints

Here are some of the requirements for writing the class. Test the implemented member functions to verify their basic functionality, and also how much of the desired semantics that is achieved by this implementation, and also if there are any undesired effects.

1. The `null_ptr_exception` is an exception that you will define, it should be derived from an STL exception.
2. Label the above member functions as `noexcept` where appropriate.
3. You may add additional member functions, if you'd like.
4. Recognize that move constructors/assignments result in the reference count remaining the same, hence there's no need to change it.
5. Think about writing as exception safe code as possible. What type of guarantees should each member function have?

Extensions

If you have time, see if you can research how to get the following to work for a `smart_ptr<X> sp`. These require additional functions:

1. Direct null pointer test: `if (sp)`
2. Negated direct null pointer test: `if (!sp)`
3. Explicit equality test for null pointers, `sp == nullptr , nullptr == sp`
4. Explicit inequality test for null pointers, `sp != nullptr , nullptr != sp`
5. The above should be true without allowing `delete sp` to compile
6. Initialization to nullptr: `smart_ptr<X> sp = nullptr`

Turn It In

You will submit this homework via Canvas. Turn in one zip file that contains your solutions to the homework problem. The zip file must contain only the files `smart_ptr.h` and `main.cpp`. The header file `smart_ptr.h` will contain all the code from the top of this specification (`include`s, `typedef`, `class smart_ptr`) and proper guards, as well as have helpful comments that tell the purpose of the major program segments and explain any tricky code. If you don't finish everything you should return dummy values for your missing definitions. Remember that, for templates, your class definition and implementation must be in the same file. The main file `main.cpp` can have the main routine do whatever you want because we will rename it to something harmless, never call it, and append our own main routine to your file. Our main routine will thoroughly test your functions. You'll probably want your main routine to do the same. Your code must be such that if we insert it into a suitable test framework with a main routine and appropriate #include directives, it compiles. (In other words, it must have no missing semicolons, unbalanced parentheses, undeclared variables, etc.)