CS 202 - Computer Science II

Project X(tra)

This Project is OPTIONAL: If you choose to submit it, its grade will <u>override your worst graded Project</u> (if there is any improvement). In this context, you may use it to either compensate for a Project where you underperformed so far, or even skip a future one (so even if you have a good average on your Projects you should still consider completing this).

Due date (FIXED): Thursday, 4/9/2020, 11:59 pm

Objectives: The main objectives of this project are to review and strengthen your ability to create and use dynamic memory wrapped in classes.

Description:
For this project you will create your own **SmartPtr (Smart Pointer)** class. A Smart Pointer serves the purpose of wrapping a set of useful behaviors around a common Raw Pointer, such as:

i. Automatically handle allocation of Dynamic Memory if necessary, when a SmartPointer object is created.
ii. Automatically handle deallocation of Dynamic Memory if appropriate, when a SmartPointer object lifetime ends.
iii. Provide access to the Dynamic Memory it encapsulates (via the actual Raw Pointer) using the same notation (the same operators) as a Raw Pointer, so that it is exactly as easy to use.
iv. Automatically handle cases such as a) when a Smart Pointer is used to point to the data already allocated by another SmartPointer, and avoid re-allocation, or b) when a SmartPointer's lifetime ends but there also exists another SmartPointer pointing to the same data, and avoid deallocating early (understand when the last SmartPointer corresponding to that memory is destroyed, and only then delete the data).

The following header file excerpt gives the required specifications for the class:

```
//Necessary preprocessor #define(s)
...
//Necessary include(s)
...
//Class specification
class SmartPtr{

  public:
    SmartPtr( );                                                  //(1)
    SmartPtr( DataType * data );                                  //(2)
    SmartPtr( const SmartPtr & other );                           //(3)

    ~SmartPtr( );                                                 //(4)

    SmartPtr & operator=( const SmartPtr& rhs );                  //(5)

    DataType & operator*( );                                      //(6)
    DataType * operator->( );                                     //(7)

  private:
    size_t * m_refcount;                                          //(8)
    DataType * m_ptr;                                             //(9)

};
```

Specifications explained:

You will notice that the Smart Pointer encapsulates a raw pointer **m_ptr** of type **DataType (9)**.
(This means that the Smart Pointer works with dynamically allocated DataType objects, but since DataType is a class that you can define yourselves, it is still flexible and modular enough given what C++ practices we know so far).

DataType is a Class which is given to you (Header .h & Implementation .cpp files both), and it is simple enough to be considered self-explanatory.

The **SmartPtr** Class will contain the following **private** data members:

- **(9) m_ptr,** a DataType Pointer, pointing to the Dynamically Allocated data. These is the Dynamic Memory encapsulated by the SmartPtr class, i.e. the memory that needs to be:
  a) allocated by the class' own methods when appropriate,
  b) deallocated by the class' own methods when appropriate,
  c) addressable and accessible via the class' own methods.
- **(8) m_refcount**, a `size_t` Pointer, pointing to a dynamically allocated positive integer (`size_t`) variable. It is a reference-counting helper variable, keeping track of how many SmartPtr objects refer to the same Dynamic Memory behind m_ptr.
  The *value* (0,1,2,…) *pointed-to* by m_refcount denotes how many SmartPointer objects are currently pointing to the same Dynamically Allocated memory as the one pointed by m_ptr.
  But m_refcount is not a `size_t` but a Pointer to a `size_t`.
  This is because it needs to be a *shared value* between different SmartPtr objects, so that when one SmartPtr object updates it, all others can see the change. E.g. when one SmartPtr object gets destroyed, it should update the *value pointed-to* by m_refcount by decrementing it, to denote that there is one less SmartPtr object alive that points to the Dynamically Memory pointed to by m_ptr.

and will have the following **public** member functions:

- **(1) Default Constructor** – will:
  a) Dynamically Allocate a new DataType object and keep track of its address via m_ptr.
  b) Dynamically allocate a new `size_t` variable and set the *value it points-to* to 1 to denote that there exists one SmartPtr object pointing the newly allocated Dynamic Memory behind m_ptr.
  Remember: This will be assigned to m_refcount and will be *shared* among any other future SmartPtr objects that will be used to point to the same Dynamic Memory as the one behind m_ptr, so that they all know when it is time to deallocate that memory.
  c) Right before it returns, it should print out:
  "SmartPtr Default Constructor for new allocation, RefCount=<refcount>" where <refcount> the actual *value pointed-to* by m_refcount.
- **(2) Parametrized Constructor** – will take a Pointer to DataType as a parameter. This means that this Constructor takes in already pre-allocated data, and wraps itself around that raw pointer. It will:
  a) Not perform any Dynamic Allocation since the data Pointer should correspond to pre-allocated data, therefore it will use m_ptr to keep track of that data directly.
  b) Dynamically allocate a new `size_t` variable and keep track of it via m_refcount. Depending on whether the data Pointer passed is **`nullptr`** or not, the *value pointed-to* by m_refcount should be set to 0 or 1 to denote that the SmartPtr object does not correspond to valid memory, or that there exists one SmartPtr object pointing the Dynamic Memory behind m_ptr.
  c) Right before it returns, it should print out:
  "SmartPtr Parametrized Constructor from data pointer, RefCount=<refcount>" where <refcount> the actual *value pointed-to* by m_refcount.

- **(3) Copy Constructor** – will take another SmartPtr object as a parameter. This means that this Constructor has access to the pre-allocated data of the other object, and the pre-allocated reference-counting variable too (which will already have a *point-to value*).
  a) Not perform any Dynamic Allocation since the other object's m_ptr should correspond to pre-allocated data, therefore it will use m_ptr to keep track of that data directly.
  b) Bind its m_refcount to the same *shared* reference-counting variable as the other object's m_refcount when appropriate.
  Note: Depending on whether the other object's m_ptr is **`nullptr`** or not, the m_refcount of the newly instantiated SmartPtr should either be newly allocated (and the *value it points-to* should be initialized to 0), or it should be bound to the other object's m_refcount and the *value it points-to* should be incremented (++), to denote that there now exists one additional SmartPtr object pointing the Dynamic Memory behind m_ptr.
  Hint: Generally speaking in this implementation, SmartPtr objects corresponding to the same Dynamic Memory should also be bound to the same *m_refcount object. SmartPtr objects that correspond to no valid Dynamic Memory however (**`nullptr`**), should each have their own *m_refcount object.
  c) Right before it returns, it should print out:
  "SmartPtr Copy Constructor, RefCount=<refcount>" where <refcount> the actual *value pointed-to* by m_refcount.
- **(4) Destructor** – will:
  a) Check if the *value pointed-to* by m_refcount indicates that the calling SmartPtr object (the one whose lifetime is just now expiring, so its Destructor is getting called) is the last and only one referencing that m_refcount (this can happen for a **`nullptr`** initialized SmartPtr, or a SmartPtr copied or assigned from it, in this implementation). If so, it should reallocate the Dynamic Memory held by m_refcount and return.
  b) Otherwise, it should decrement the *value pointed-to* by m_refcount to denote that one less SmartPtr object is now pointing to the Dynamic Memory behind m_ptr.
  c) Examine whether the calling SmartPtr object (the one whose lifetime is just now expiring, so its Destructor is getting called) is the *last* one referencing the Dynamic Memory behind m_ptr. To do that, it will have to examine the *value pointed-to* by m_refcount (after it has been decremented).
  d) If now it is found to be the last one, it should dellocate the Dynamic Memory both behind m_ptr, and the shared variable m_refcount.
  e) Right <u>before any deallocation happens in all cases</u>, it should print out:
  "SmartPtr Destructor, RefCount=<refcount>" where <refcount> the actual *value pointed-to* by m_refcount (its value after decrement).
- **(5) operator=** will perform assignment from a SmartPtr object. This means that it will:
  a) First take care of releasing its handle on its own Dynamic Memory. *Note*: This does not necessarily mean to directly deallocate it, there might be other SmartPtr objects referencing the same Dynamic Memory! Review the description of the Destructor to understand how releasing with respect for other SmartPtr objcets will need to work.
  b) Then switch to referencing the same values as the other SmartPtr object. This means that both m_ptr and m_refcount will need to be repointed there, and any additional considerations (such as mutating the *value pointed-by* m_refcount, and how to handle a case where the other object holds a **`nullptr`** pointed for its Dynamic Memory, etc) should be handled as per the Copy Constructor.
  c) Right before it returns, it should print out:
  "SmartPtr Copy Assignment, RefCount=<refcount>" where <refcount> the actual *value pointed-to* by m_refcount.

- **(6) operator*** will act in the same way that it is used on Raw Pointers, i.e. it will Dereference the Dynamic Memory Object that is encapsulated within the SmartPtr:
  When you have a Raw Pointer, dereferencing returns a Reference-to the underlying object:
  ```
  DataType * data_pt = new DataType;  //data_pt is a raw pointer
  DataType & data_ref = *( data_pt ); //operator* on a raw pointer
  ```
  In the same principle, operator* will act as a Smart Pointer Dereference:
  ```
  SmartPtr data_pt;                   //data_pt is now a smart pointer
  DataType & data_ref = *( data_pt ); //operator* on a smart pointer
  ```
  and again have the same result (the goal is to maintain the same notation semantics so that the user of a SmartPtr class has near-zero things to learn in order to use it).
  Hint: Given the SmartPtr class declaration that you already know, where you are also given the return type, it should be straightforward how to implement this method.
- **(7) operator->** will act in the same way that it is used on Raw Pointers, i.e. it will Allow Object Member Access via the Dynamic Memory Pointer that is encapsulated within the SmartPtr:
  When you have a Raw Pointer, member-access is performed like:
  ```
  DataType * data_pt = new DataType;  //data_pt is a raw pointer
  int intVar = data_pt->getIntVar();  //operator-> on a raw pointer
  ```
  In the same principle, operator-> will act as a Member Access via Smart Pointer:
  ```
  SmartPtr data_pt;                   //data_pt is now a smart pointer
  int intVar = data_pt->getIntVar();  //operator-> on a smart pointer
  ```
  and again have the same result (the goal is to maintain the same notation semantics so that the user of a SmartPtr class has near-zero things to learn in order to use it).
  Hint: Given the SmartPtr class declaration that you already know, where you are also given the return type, it should be straightforward how to implement this method.

The SmartPtr.h header file should be as per the specifications. The SmartPtr.cpp source file you create will hold the required implementations. You should also create a source file projX.cpp which will be a test driver for your class.

The test driver has to demonstrate that your SmartPtr class works as specified:
- You should use all the meaningful test cases you can identify to demonstrate the use of all the class methods. The following example is considered as a starting point:

```
cout << endl << "Testing SmartPtr Default ctor" << endl;
SmartPtr sp1;  // Default-ctor
sp1->setIntVal(1);
sp1->setDoubleVal(0.25);
cout << "Dereference Smart Pointer 1: " << *sp1 << endl;
cout << endl << "Testing SmartPtr Copy ctor" << endl;
SmartPtr sp2 = sp1;  // Copy-initalization (Copy-ctor)
sp2->setIntVal(2);
sp2->setDoubleVal(0.5);
cout << "Dereference Smart Pointer 1: " << *sp1 << endl;
cout << "Dereference Smart Pointer 2: " << *sp2 << endl;
cout << endl << "Testing SmartPtr Assignment operator" << endl;
SmartPtr sp3;
sp3 = sp1;  // Assignment operator
sp3->setIntVal(4);
sp3->setDoubleVal(0.0);
cout << "Dereference Smart Pointer 1: " << *sp1 << endl;
cout << "Dereference Smart Pointer 2: " << *sp2 << endl;
cout << "Dereference Smart Pointer 3: " << *sp3 << endl;
```

```
cout << endl << "Testing SmartPtr Parametrized ctor with nullptr" << endl;
SmartPtr spNull( nullptr );  // nullptr initialization

cout << endl << "Testing SmartPtr Copy ctor with nullptr SmartPtr" << endl;
SmartPtr spNull_cpy( spNull );  // nullptr copy constructor

cout << endl << "Testing SmartPtr Assignment with nullptr SmartPtr" << endl;
SmartPtr spNull_assign;
spNull_assign = spNull;  // nullptr data assign

cout << endl <<
"End-of-Scope, Destructors called in reverse order of SmartPtr creation\n
(spNull_assign, spNull_cpy, spNull, sp3, sp2, sp1): " << endl;
```

The completed project should have the following properties:

- Additionally to your class code, you are required to examine and explain the output of the test driver you provide with projX.cpp.
  Your grade will be based on the **explanations** you provide in your **documentation file** (e.g. copy-paste the output from your terminal and explain what is happening line-by-line).
- Your code is required to follow the **file organization structure** demonstrated in your Labs, with subfolders for headers, source files, and a final build products location (generated during build).
- Your project's build should be based on a **CMakeLists.txt** script, which will be included with your deliverables.

The completed project should have the following properties:

- Written, compiled and tested using Linux.
- It must compile successfully using the g++ compiler on department machines or the provided Xubuntu VM image.
- The code must be commented and indented properly.
  Header comments are required on all files and recommended for the rest of the program.
  Descriptions of functions commented properly.
- A one page (minimum) typed sheet documenting your code. This should include the overall purpose of the program, your design, problems (if any), and any changes you would make given more time.

Turn in: Compressed file structure (with .cpp and .h files, and your CMakeLists.txt). Also, your project documentation file.

IMPORTANT: Creating a build configuration for Debugging:

- When requiring a build with debug symbols, usually you would specify this using the g++ command with the appropriate flag:

  `g++ -g ...`

  But **CMake** provides a convenient functionality for configuring a "standardized" debug build of your project with the required compiler flags and settings automatically handled by CMake. It does so via a configuration option called: `CMAKE_BUILD_TYPE`.
  If you want to enforce configuring your project build to have debug symbols enabled (because you intend to debug it using `gdb` for instance), you may run the cmake configuration command and specify this option as follows:

  `cmake -D CMAKE_BUILD_TYPE=Debug ..`

  Extra: Other possible values are:
  `-D CMAKE_BUILD_TYPE=Release` which enables recommended compiler optimizations to refactor the compiled code to make it more efficient,
  `-D CMAKE_BUILD_TYPE=RelWithDebInfo` which generates a "Release" (optimized) build but retains debug symbols for debugging (be careful with "Release" builds your code is optimized by the compiler and thus refactored).
- It is recommended to use such a configuration together with `gdb` (as demonstrated in your Labs) to trace any dynamic memory management bugs of your code.

The following are a list of restrictions:

- Your code may use the C++11 standard (or any standard higher or lower).

  Note: Usually, you would specify using the g++ command with some flags:
  `g++ -std=c++11 ...`

  But **CMake** provides the functionality of *autodetecting* your system's C++ compiler and generating the Makefiles to invoke the appropriate commands to be used when you eventually `make` your project.
  If you want to enforce configuring your project build to use a particular standard, you either do so everytime you run the cmake configuration command by:

  `cmake -D CMAKE_CXX_STARDARD=11 ..`

  Or you could put a line like the following inside your **CMakeLists.txt** script:

  `set(CMAKE_CXX_STANDARD 11)`

  You do not need to worry about either of these however, and for now just running the usual `cmake ..` for configuration should do the trick.

- No libraries except **`<iostream>`** and **`<fstream>`** and **`<string>`** or **`<cstring>`** / **`<string.h>`** allowed.
- No global variables except `const` ones.
- You are expected to employ code abstraction and reuse by implementing and using functions. The already provided code structure in the project description will be considered sufficient.
- You are expected to implement **`const`** correctness in your program design. This refers to class method qualifications, function parameter qualifications, etc.

Submission Instructions:

- You will submit your work via WebCampus
- Compress your:
  1. Code file structure (containing Source code files, Header files, CMakeLists.txt)
  2. Documentation

  Do not include executable or library files, nor any build, devel, or other non-required folders.
- Name the compressed folder:
  PA#_Lastname_Firstname.zip
  ([PA] stands for [ProjectAssignment], [#] is the Project number)
  Ex: PA7_Smith_John.zip

Verify: After you upload your .zip file, re-download it from WebCampus. Extract it, compile it and verify that it compiles and runs on the ECC systems.

- Code that does not compile will be heavily penalized –may even cost you your *entire* grade–. Executables that do not work 100% will receive partial grade points.
- It is better to hand in code that compiles and performs partial functionality, rather than broken code. You may use your Documentation file to mention what you could not get to work exactly as you wanted in the given timeframe of the Project.

Late Submission:

A project submission is "late" if any of the submitted files are time-stamped after the due date and time. Projects will be accepted up to 24 hours late, with 20% penalty.

Instructions to remotely test your project configuration and build on the ECC systems:

a) Download your Webcampus submission on your local computer. Let's say you submitted a file named **PAx_Smith_John.zip**, and you now downloaded it into your **Downloads** folder.

b) Navigate to that directory using your terminal, and check the file you downloaded is there.

   a. On **Ubuntu** you can open a terminal with Ctrl+Alt+T and then do:
      ```
      cd Downloads
      ls -al
      ```
   b. On a **Mac** you can open Spotlight and type "Terminal" and hit Enter, then do:
      ```
      cd Downloads
      ls -al
      ```
   c. On **Windows** in your Start Menu type "cmd" and click on Command Prompt, then:
      ```
      cd Downloads
      dir
      ```

c) Remotely copy your submission file from you local Downloads folder to your CSE Ubuntu user account inside its home/$USER folder.
   ```
   scp FILENAME NETID@ubuntu.cse.unr.edu:/nfs/home/NETID/FILENAME
   ```
   For example if the user NetID is **jsmith** and the submission file **PA7_Smith_John.zip**:
```
scp PAx_Smith_John.zip jsmith@ubuntu.cse.unr.edu:/nfs/home/jsmith/PAx_Smith_John.zip
```

d) Login to your CSE Ubuntu user account.
   ```
   ssh NETID@ubuntu.cse.unr.edu
   ```
   For example if the user NetID is **jsmith**:
   ```
   ssh jsmith@ubuntu.cse.unr.edu
   ```

e) Once you are in, check the contents to verify that you have successfully transferred the file:
   ```
   ls -al
   ```

f) Unzip the file into a folder with the same name:

a. If it is a **.zip** file then:
   ```
   unzip -o FILENAME.zip -d FILENAME
   ```
   Example:
   ```
   unzip -o PAx_Smith_John.zip -d PAx_Smith_John
   ```
b. If it is a **tar.gz** file then:
   ```
   mkdir FILENAME
   tar -xzvf FILENAME.tar.gz -C FILENAME
   ```
   Example:
   ```
   mkdir PAx_Smith_John
   tar -xzvf PAx_Smith_John.tar.gz -C PAx_Smith_John
   ```

g) The above will create a folder with the same name as your submission file, which will contain the unzipped content. Enter the directory and execute the known configuration and build sequence:
```
cd PAx_Smith_John
mkdir build
cmake ..
make
```