10 points.
Due: Friday, October 29, 2021 at 11:59 pm

Section 1 Program Summary

In this program, you will create a program to simulate "panning" for valuable stones. The primary focus is to get practice managing pointers in a couple of lists, and using inheritance and polymorphism to simplify the management. In order to allow you to focus on the list and pointer management, we will provide *most* of a `main.cpp`. You'll just need to fill in a couple cases inside `main.cpp`.
There are four kinds of Rock in the program - Ruby, Gold, Diamond, and regular old Sandstone. This program allows you to collect unidentified rocks, inspect them, and keep any sufficiently large Rock of a given type (except for Sandstone).

Section 2 Classes

In this section, we'll give details on how to implement the program outlined above.

(a)

Rock

This will be an abstract base class for the different kinds of Rocks. Each Rock should have a `string` classification, and a mass (`float`). Note that subclasses will need access to their classification, so these should be protected variables.
The public functions for the `Rock` class are listed below:

- `Rock(float _mass)` (parameterized constructor)
  Note that this parameterized constructor does not include an input for the classification. Hardcode the classification to be "Rock." Rocks of any kind won't be classified until they are inspected.
- `void Print()`
  This function will be the same for every kind of Rock. Output a message with the classification and mass of the Rock (which is given in grams).
  For example: "A *classification*, weighing *mass* g", where *classification* and *mass* are replaced by the appropriate values.
- `virtual bool Inspect()`
  This should be a pure virtual function, to be overridden by the subclasses.

(b)

Rock Subclasses

You should create four subclasses of Rock:
Gold, Ruby, Diamond, and Sandstone.
In each subclass, you should create a parameterized constructor that takes a `float _mass` as parameter, and passes this along to the parent constructor. You should also implement `bool Inspect()` for each subclass, as follows:

- Gold: set the `classification` to "Gold", and return true if mass is at least 1, else false.
- Ruby: set the `classification` to "Ruby", and return true if mass is at least 5, else false.
- Diamond: set the `classification` to "Diamond", and return true if mass is at least 3, else false.
- Sandstone: set the `classification` to "Sandstone", and return false (we never collect a sandstone).

Important Note: Because the definitions for Rock, Gold, Ruby, Diamond, and Sandstone are very simple, we will **not** ask you to create .cpp files for them. You should instead implement all the functions **directly in** the .h files.

(c)

RockPile Class

This class will essentially be a dynamically-sized "stack" data structure, storing pointers to Rocks. A RockPile needs to have a `string` name, as well as a "count" and "size" (as ints). Finally, RockPile should have a `Rock**` variable called `pile`. This is a pointer to a Rock pointer.
Recall, we create a dynamic array of integers with an integer pointer. In this case, we can use a Rock pointer pointer to create a dynamic array of Rock pointers. This is a common pattern when maintaining a list of objects utilizing polymorphism.
The public functions for `RockPile` are listed below:

- `RockPile(string _name)` (param'ed constructor)
  This should set the RockPile name to `_name`, set the count to 0, the size to a default size of your choice, and create a new array of `Rock*` with your chosen default size.
- `~RockPile()` (destructor)
  Your destructor should loop over any Rock* remaining in the pile, deleting each, and finally delete the `pile` variable.
- `void Add(Rock* _rock)`
  This function should add the given Rock* to the pile. You can assume the Rock* already points to a Rock, no need to create a new object.
  If adding the rock makes the pile full, Add should call the Grow function (details below).
- `Rock* Remove()`
  This function will return the last Rock* in the pile, and set the old pointer to NULL or nullptr. (note, you should *not* delete it, as this would turn the Rock* you return into a dangling pointer)
  If removing this Rock* brings the count to less than a quarter of the pile array's size, Remove should call the Shrink function (details below).
- `int GetSize()`
  Simply return the current size of your array (we'll just use this for testing).
- `int GetCount()`
  Simply return the current number of Rocks of your array (we'll just use this for testing).
- `void Print()`
  First, this function should output the name on its own line,
  then the size and count on a line,
  and finally call Print() on each Rock currently in the Pile.

There should be two private functions in RockPile as well:

- `void Grow()`
  This function will double the size of the pile array. First, you'll need to allocate a new dynamic array of Rock* (recommend you do this with a temporary local variable). Then, you must copy each pointer from the old pile to the new temp pile. Finally, you should delete the old pile (but not the pointers in it), and point the RockPile's `pile` variable to the new pile.
- `void Shrink()`
  This function will halve the size of the pile array. Note, since we only call this if the count is a quarter of size, the post-condition of Shrink is the same as that of Grow - namely, the new pile will be half-full. As with Grow, you'll need to allocate a new dynamic array of Rock* (again recommend you do this with a temporary local variable). Then, you must copy each pointer from the old pile to the new temp pile. Finally, delete the old pile and point the `pile` variable to the new pile.

Section 3 Main Program

For this program, we've given you the bulk of the code for main.cpp. We have also provided a file called RockMaker.h. Inside this header is a function called `PanForRock()`, which returns a pointer to a new Rock object.
You will need to fill in two cases in the switch statement in `main.cpp`.

1. `case Rock`: For this case, you simply need to call PanForRock(), and add the new rock pointer to the "inspection" `RockPile`.
2. `case INSPECT`: Here, you first need to remove a Rock* from the "inspection" pile. You should then call Inspect() on the given Rock*. If the result is true, then we'll keep the Rock, so you should output a message, call Print on the Rock*, and add it to the "keepers" `RockPile`. If the result is false, then the Rock wasn't worth keeping, and you should delete it.

Section 4 Submission

Submit a zip file, which contains two .cpp files called `main.cpp` and `RockPile.cpp`, as well as header files `Rock.h`, `Ruby.h`, `Gold.h`, `Diamond.h`, `Sandstone.h`, `RockPile.h`, and `RockMaker.h`.

Your zip file should be named *NetID*`_asg3.zip`, where *NetID* is your Net ID (the username you use on Canvas, *not* the 10-digit number on your wiscard).

In each cpp and header file, other than RockMaker.h, you should include a comment at the top of the file with your name and NetID.

Section 5 Hints

To build the program, use the following command: `g++ main.cpp RockPile.cpp -o main`
(recall that we aren't using .cpp files for Rock and its subclasses, each will be fully implemented in the .h file(s)).

Section 6 Rubric

Item	**Points**
Program builds and runs without compile errors	2
Rock has the specified functions, including pure virtual function Inspect	1
Gold, Ruby, Diamond, and Sandstone are all subclasses of Rock, and correctly implement their versions of Inspect()	2
RockPile can Add and Remove Rock* without crashing	2
RockPile Grows and Shrinks at appropriate times, without crashing	2
The ROCK and INSPECT cases of main.cpp have been correctly filled in to get a new rock and inspect a rock, respectively	1
Total	**10**