Lab 4.1 Enumeration Classes—Classy Rocks

Project 4.1 More Classy Rocks—Continuation of Lab 4.1

NOTES TO THE INSTRUCTOR

This lab and project are intended to demonstrate the "correct" way to build the new data type `Rock` considered in Lab Exercise 2.2 and Project 2.2, namely, using a class. The lab exercise begins the design and building of the first part of this class, and the project continues the development. Both are intended to provide a review of (or perhaps a first look at) how to build a class in C++ and to illustrate how class objects are self-contained. Rather than pass enumeration values around to various functions for processing, enumeration-class objects operate on themselves with built-in operations. In particular, input and output operations are developed for the `Rock` class in the lab exercise; the relational operators < and == and the increment operator ++ are overloaded for the `Rock` class in the project.

Notes:

1. Parts 4 and 5 of the lab exercise deal with overloading the output operator << and the input operator >>. There are two ways to do this:

   - Define a function member `display()` and then have `operator<<()` call it; and define a function member `read()` and then have `operator>>()` call it.
   - Make `operator<<() and operator>>()` friend functions of class `Rock`.

   You should tell students which method you want them to use. The first is consistent with OOP and with the idea of inheritance and polymorphism considered later in the text and is the method used in the lab exercise. The second involves less typing and shows students how the *friend* mechanism is used (which isn't really consistent with the spirit of objective-oriented programming and can be easily overused).

2. Either or both of Parts 6 and 7 of the lab exercise may be omitted if you think the lab exercise is too long.

3. Because the first part of the project is a continuation of the lab exercise, you may want to have the students keep the lab handout to use in the project and then hand it in with the project handout, the final `Rock` files, and the driver program. The project's application could then be handed in at a later date.

4. The application in Project 4.1 requires an operation to increment a `Rock` value so that loops can be written that range over the different `Rock` values. There are two ways to do this:

   - Use a `next()` function as in Lab & Project 2.2. In this approach, one need only modify this function to make it a function member of the `Rock` class.
   - Overload the increment operator ++ for class `Rock`.

   You should tell students which method you want them to use—the first clearly is simpler, but the second is another illustration of operator overloading. In Project 4.1's description, the second procedure is used.

5. Lab Exercise 4.1 uses the driver program `rocktester.cpp` to test the `Rock` class as it is being developed. The application in Project 4.1 uses the data file `Rockfile.txt` (which is the same as the file used in Project 2.2). These files can be downloaded from the website whose URL is given in the preface.

Course Info: _______________ *Name:* ____

Lab 4.1 Classy Rocks

Background: In Lab 2.2 we explored using C++'s enumerated types to implement a `Rock` type. There were several limitations to this approach. A major one is that if we wanted to use a `Rock`, operations had to be implemented by passing the rock value to each application. Moreover, C-style enumerations do not support overloading arithmetic operators such as +, −, ++, −−, and so on.

Objective: This lab exercise will focus on implementing a rock type using a C++ class. In the course of developing this rock class, we will learn how to give various capabilities to our rocks:

- They will be able to construct themselves, so we can build new rocks. These can either have default values or be created with explicitly supplied values.
- We want to be able to display rocks using the standard output streams such as `cout`, which means that we will need to overload the output operator <<.
- We want to be able to read a rock from an input stream such as `cin`, which means that we will need to overload the input operator >>.
- Rocks should be able to tell us about their kind and texture.

Approach: C++ classes package data and operations together and thus provide an excellent way to implement ADTs. If we create a `Rock` class, then we are making *smart rocks* that *know* how to do things, i.e., the operations are part of a `Rock` object. As you create your `Rock` class, you will test it incrementally using a program called `rocktester.cpp`. If you are working in a Unix environment, a `Makefile` is helpful with compiling and linking. You should get a copy of these files using the procedure specified by your instructor. (They can be downloaded from the website whose URL is given in the preface.)

As you progressively develop your `Rock` class, you will also be modifying the `rocktester.cpp` program and learning about building test drivers for your classes. If you need more information about classes, see Chapter 4 of the text *ADTs, Data Structures, and Problem Solving with C++*, 2E.

When you have completed Lab 4.1, your next task will be to continue with Project 4.1, which will add relational operators to your `Rock` class and an increment operator. This will produce a class that is fairly rich in operations, almost as if it were part of the C++ language.

Step 1: Begin Developing Class `Rock`

Class Declaration:
The form of a class declaration for the `Rock` type is:

```
class Rock
{
  public:
    // Put declarations of things that should be available outside the
    // class here -- in particular, prototypes of public function members.

  private:
    // Put declarations of things that should not be available outside
    // the class here -- in particular, declarations of data members.
};
```

1.1 Create a `Rock.h` header/interface file. A *header file* informs users about *what* the class can do—and is thus often called an *interface file*—and the corresponding *implementation file* contains the actual code that carries out the class' operations—i.e., it shows *how* they work. Implementations can be delivered in either source code or object code so that, if desired, one can keep the implementation secret while publishing the interface. You will be making the source code for your implementation available.

Check here when finished ____

1.2 Now start with the enumerated type used for modeling rocks in Lab 2.2 and Project 2.2, but change its name to `RockName`:

```
enum RockName {BASALT, DOLOMITE, GRANITE, GYPSUM, LIMESTONE, MARBLE,
               OBSIDIAN, QUARTZITE, SANDSTONE, SHALE, ROCK_OVERFLOW};
```

First, try putting this `RockName` declaration in the `public` section of the `class Rock`.

Check here when finished ____

1.3 Now put a private section in your class that contains a single data member `myName` of type `RockName`.

Check here when finished ____

1.4 Now load `rocktester.cpp` and compile it. Describe what happens in the space below.

WARNING:
MAY NOT WORK
AS YOU EXPECT

Error(s) that result are caused by the compiler encountering the identifier `GRANITE` in the last output statement of Part 0, but it can't find a declaration of it. The problem is that the declaration of the enumerated data type `RockName` is *inside* the declaration of class `Rock`, which means that its scope extends only to the curly brace at the close of this declaration.

To refer to this type outside the class or to the enumerators listed in it, one must qualify them so that the compiler knows where to look for this declaration. Adding the qualifier `Rock` to `GRANITE` so that it reads `Rock::GRANITE` in `rocktester.cpp` should get rid of the problem. Do this and then recompile and execute. What happens?

What you have seen is that if we put an enumeration type definition inside the public part of a class, we have to qualify each enumerator (as well as the type name) with the class name each time it is used outside the class declaration. This is somewhat of a nuisance, right? So, to avoid this inconvenience, we will:

- Move the enumerated type in `Rock.h` ahead of the class definition
- Remove the `Rock::` qualifier in `rocktester.cpp`

Now recompile and execute the program to check that everything is okay.

Check here when finished ____

So the program currently uses two types:

1) `RockName`, the enumerated type defining the names of rocks, e.g. `BASALT`, `DOLOMITE`, `GRANITE`, `GYPSUM`, etc.; and

2) `Rock`, the name of a class that includes a variable of type `RockName`. You might think of `RockName` as a type of something that is inside a `Rock` object:

```
sample
+-------------------------+
| myName | GRANITE      | |
+-------------------------+
```

Step 2: Defending Against Redundant Definitions

Comment out (using `//` at their beginning) or remove the lines labeled `"BEGIN PART 1"` and `"END PART 1"` in `rocktester.cpp`. (Note that they come earlier in the source code than those for Part 0.) Then save and recompile the program.

You may get a plethora of error messages. But if you scan them, you should see some that indicate that there is more than one definition of `enum Rockname` and also multiple definitions of the `class Rock`. Find one of these and record it below.

Perhaps you didn't notice it, but we included `Rock.h` twice. C++'s preprocessor, being very literal, did just what we told it to do. The result is that the type `RockName` is defined twice, as is the class `Rock`. The compiler was confused and complained. While we would not do this intentionally, it can happen very easily when programs become complex. We might have a program that uses two classes, both of which include `Rock.h`, and it thus gets included twice in our program.

2.1 We need a mechanism to ensure that only a single copy of a header is included.

> You can do this for `Rock.h` by wrapping it in the following preprocessor code (called a *conditional compilation directive*):
>
> ```
> #ifndef ROCK
> #define ROCK
>    ... //put header code here
> #endif
> ```
>
> The first time that `Rock.h` is encountered, `ROCK` has not been defined, so it is defined (to be 1) and the code that follows `Rock.h` is included. The second time `Rock.h` is encountered, `ROCK` has already been defined and so the code is skipped.
>
> *Note*: It is common practice (but not mandatory) to use the name of the class in all caps in the directive.

This mechanism is rather clumsy, but it works well enough. Add these defensive preprocessor statements to your `Rock.h` file and then recompile and execute `rocktester.cpp`.

- If you get errors, make sure that you have entered the directives exactly as shown above (e.g., # is the first character on each line, and they do not end with semicolons).

- Then comment out or remove the duplicate `#include "Rock.h"` directive in `rocktester.cpp`. It is no longer needed.

Check here when finished ____

Step 3: Creating `Rock`'s Constructors

Normally, when we declare a new variable, we have to initialize it. Usually this is done by providing initializers in the declarations of the variable; for example,

```
int i = 10;    // or  int x(10);
double x[3] = {1.0,2.0,3.0};
```

Without initializers, memory locations will be allocated for the variables, but the values in them may not be meaningful.

Variables whose type is a class—called *objects*—should always be initialized so they form legitimate objects of that type. C++ accomplishes this by calling a special function called a *constructor*. There are several kinds of constructors, but two main types are of interest to us right now:

- *default constructor* : `ClassName();`
- *explicit-value constructor*: `ClassName(`*`parameter-list`*`);`

These are the first function members to be added to your `Rock` class:

- A **default constructor** so that users can make declarations like the following:

  ```
  Rock sample;    //default constructor
  ```

  The default constructor should initialize the `myName` data member to some rock value, e.g., `BASALT`.

- An **explicit-value constructor** so that users of the class can make declarations like the following:

  ```
  Rock rockVal(GRANITE);  //explicit-value constructor
  ```

  This constructor should initialize the `myName` data member to the specified rock name (`GRANITE`).

Note: An alternative is to use only an explicit-value constructor with a *default argument* for the parameter. Then if the user doesn't supply a value, the default argument will be used. (See Section 4.5 in the text for details.)

3.1 Add prototypes and documentation for these two constructors to your `Rock` class. Then test them by commenting out or removing the lines labeled `"BEGIN PART 2"` and `"END PART 2"` in `rocktester.cpp` and then compiling the program (but not linking—i.e., use the `-c` switch if you are using `g++`).

Check here when it compiles without errors ____

3.2 Now create an implementation file `Rock.cpp` with suitable opening documentation. Put the definitions of the constructors in it. *Remember that the names of function members must be qualified in their definitions:*

`ClassName::functionName( . . . )`

3.3 Now compile and link `rocktester.cpp` and `Rock.cpp` and execute the resulting binary executable.

Check here when it executes correctly ____

*Note: Later, for efficiency, we will **inline** simple functions like these constructors and put their definitions in the header file below the class declaration. Inlining a function has the effect of replacing each call to that function with the actual code of the function definition.*

Step 4: Adding an Output Operation

One operation that we usually add soon after the constructors is an output operation so that we can use it to check the implementations of the other operations. We will do this in two steps:

- First add a `display()` function member in the usual way.
- Then overload the output operator `<<`.

4.1 *Adding a* **display( )** *Function Member*

Write a function member `display()` with signature `(ostream &)` and return type `void` to output a `Rock` object. Put its prototype in the class. Make it a *const function*, since it does not change any data member. To do this, append the keyword `const` to the function heading in both the prototype and its definition.

```
function_heading const
```

Put the definition of `display()` in the implementation file `Rock.cpp`. *You should be able to reuse the code that you wrote in Project 2.2 to output rock enumerators with minor modifications.*

When you have completed your `display()` function member, you should test it. Comment out or remove the lines labeled `"BEGIN PART 3"` and `"END PART 3"` in `rocktester.cpp`. Then compile `Rock.cpp` and `rocktester.cpp`, link them, and execute the resulting binary executable. Record below the values output for:

`sample` ____________________ `rockVal` ____________________

4.2 *Overloading the Output Operator* **<<**

Using `display()` to output a `Rock` value works fine. But we do have to call it using the *dot* operator as in

```
sample.display(cout);
```

which breaks up the usual chain of `<<` operations. What we'd really prefer is that output for the `Rock` type be no different than output for any other type. To accomplish this we need to overload the output operator. This is a little tricky, so we will proceed slowly.

First add a prototype for a nonmember function for the `operator<<()` function in `Rock.h` *after the end of the class declaration*. Because it is not a member function (as noted on the next page), do not add a prototype for it inside the class `Rock` and do not qualify its name with the class name.

The function `operator<<()` should have:

- Return type `ostream &`—a reference to an output stream so that the output can be chained
- Signature `(ostream &, const Rock &)`

So the prototype has the form

```
ostream & operator<<(ostream & out, const Rock & rockVal);
```

The actual function body should do two things:

1) It should call the `display()` function member of its `Rock` parameter, and

2) It should return the `ostream` for chaining.

Add this definition to `Rock.cpp` and then test that it works. To test it, comment out or remove the lines labeled `"BEGIN PART 4"` and `"END PART 4"` in `rocktester.cpp`. Compile your modified `Rock.cpp` and `rocktester.cpp` files, link, and execute.

You should have gotten the same output as in step 4. Did you? ____________

Step 5: Adding an Input Operation

Overloading the input operator >> is not significantly different from overloading the output operator <<.

5.1 First, add a function member `read()` with a parameter of type `istream &` to your `Rock` class. Note that it cannot be a const function because it must modify the value of the `myName` data member of class `Rock`. Again, *you should be able to reuse the code that you wrote in Project 2.2 to input rock enumerators with minor modifications.*

5.2 Add a prototype and definition of `operator>>()` in much the same manner as you did for the output operator. Note that it should have

- Return type `istream &`—a reference to an input stream so that the input can be chained
- Signature `(istream &, Rock &)`

Then test that it works by commenting out or removing the lines labeled `"BEGIN PART 5"` and `"END PART 5"` in `rocktester.cpp`. Compile your modified `Rock.cpp` and `rocktester.cpp` files, link, and execute.

Record in the following table, the output produced for the given inputs:

Input for first `Rock`	Input for second `Rock`	Output for first `Rock`	Output for second `Rock`
`BASALT`	`marble`		
`Granite`	`gypSUM`		
`feldspar`	`99`		

A Quick Review of Binary Operators in C++

Let's begin by quickly reviewing how C++ binary operators work. Suppose we have a binary operator Δ. If `operator`Δ`(a,b)` is a member function of some class `C`, then the compiler will interpret an expression of the form `a` Δ `b` (where `a` is of type `C`) as

`a.operator`Δ`(b)`

Thus if we make `operator<<()` a function member of class `Rock`, an expression of the form `cout << rockVal` will be interpreted as

`cout.operator<<(rockVal)`

which confuses the compiler because the statement implies that `operator<<()` is a function member of the class `ostream` and would have only one parameter—a `Rock` parameter. But, of course, `ostream` does not have such a function, because we are trying to define such a function!

So this is the reason that neither `operator<<()` *nor* `operator>>()` *can be a member function of a user-defined class.*

Step 6: Adding a `kind()` Operation

In Lab 2.2 we added a `kind()`function that returns one of the strings "igneous," "metamorphic," or "sedimentary," depending on the kind of rock determined as follows:

- Basalt, granite, and obsidian are igneous.
- Marble, and quartzite are metamorphic.
- Dolomite, limestone, gypsum, sandstone, and shale are sedimentary.

6 Add a `kind()` function member to the `Rock` class. You can use the code developed in Lab 2.2 as a starting point. To test if it works correctly, comment out or remove the lines labeled `"BEGIN PART 6"` and `"END PART 6"` in `rocktester.cpp`, and then recompile, link, and execute.

Record in the following table the output produced for the given inputs:

Rock Input	Kind of Rock
BASALT	
Marble	
Shale	
Feldspar	

Step 7: Adding a `texture()` Operation

6 Now add a function member called `texture()` that returns one of the strings "coarse," "intermediate," or "fine" that indicates the texture of the rock, determined as follows:

- Dolomite, granite, gypsum, limestone, and sandstone are coarse in texture
- Basalt and quartzite are intermediate in texture, and
- Obsidian, marble, and shale are fine in texture

To test if it works correctly, comment out or remove the lines `"BEGIN PART 7"` and `"END PART 7"` from `rocktester.cpp`, and then recompile, link, and execute.

Record in the following table the output produced for the given inputs:

Rock Input	Texture of Rock
BASALT	
Marble	
shale	
feldspar	

You have finished! Hand in:

1) the lab exercise with answers filled in
2) a listing of your final program and library files
3) a demonstration that everything compiled, linked, and ran correctly
4) a sample run of your program

Course Info: ________________ ***Name:*** ____________________________________

Project 4.1 More Classy Rocks—Continuation of Lab 4.1

Objective: Your `Rock` class now has quite a few capabilities. In this project you will first continue the development of the class by adding additional operations to it. These operations are:

- An accessor `name()` that returns the name of the rock stored in `myName`
- Relational operators < and ==

Once these are added and tested, you will then use your class in a modified version of the bar-chart program in Project 2.2.

Adding More Operations to the `Rock` *Class*

Step 8: Adding an Accessor to `Rock`

You want to add an accessor function called `name()` to class `Rock` that returns the value (of type `RockName`) stored in the `myName` member. Should it be a `const` function? Why?

____

WARNING:
MAY NOT WORK
AS YOU EXPECT

Once you have it written, test it by adding statements between the lines labeled `"BEGIN PART 8"` and `"END PART 8"` in `rocktester.cpp` like those in Part 7 to enter some `Rock` values, access and output the value stored in their `myName` member (via `name()`). Then recompile, link, and execute `Rock.cpp` and `rocktester.cpp`. Record in the following table the output produced for the given inputs:

`Rock` Input	Name of `Rock`
`BASALT`	
`Marble`	
`shale`	
`feldspar`	

When you added the `name()` function, it returned a `RockName` value, but outputting this value produced a number and not a name. We have overloaded the output operator to work with a `Rock` object, but we also need to overload it for `RockName` values. The obvious solution is to modify the overloading created in Project 2.2 and add its prototype to `Rock.h` (below the class declaration) and its definition to `Rock.cpp`. Do this now and test it. You shouldn't have to make any changes in `rocktester.cpp`.

Check here when it executes correctly ____

Step 9: Overloading the relational operators < and == for `Rock`

Your objective now is to add the relational operators < and ==, where $x < y$ for `Rock` objects means that the value in `myName` of x precedes the value in `myName` of y in the enumeration declaration.

Unlike the output operator `operator<<()` and the input operator `operator>>()`, the relational operators `operator<()` and `operator==()` can be function members of `Rock`. However, making them member functions introduces an asymmetry in a relation which is intrinsically symmetric. We can make a comparison like

```
sample == GRANITE
```

but we cannot make the reverse expression

```
GRANITE == sample
```

Thus, it seems best to make these operators nonmember functions.

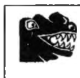
Overload `operator<()` and `operator==()` to compare two `Rock` objects. Each should have two constant reference parameters of type `Rock` and return one of the `bool` values `true` or `false`. Your parameters will look, for example, like `const Rock & aRock1`.

When you have finished overloading the operators, you should test them by commenting out or removing the lines `"BEGIN PART 9"` and `"END PART 9"` in `rocktester.cpp`. Then recompile `Rock.cpp` and `rocktester.cpp`, link, and execute.

Check here when the relational operators execute correctly ______

Step 10: What's Going On Here?

So far you've compared `Rocks`—for example, `rock1 < rock2`. You might want to compare a `Rock` object with a `RockName`—for example, `rock < GRANITE`. To see what happens when you try this, do the following:



WARNING:
MAY NOT WORK
AS YOU EXPECT

1) Comment out or remove the lines labeled `"BEGIN PART 10"` and `"END PART 10"` in `rocktester.cpp`.

2) Recompile and link `Rock.cpp` and `rocktester.cpp` and execute the resulting binary executable.

3) Describe any errors you encounter in this part of the program.

____

What? No problems! Why is that?

The compiler appears to have done some work for you. What has happened is similar to what happens when numbers of different types are compared. For example, in the mixed-type expression

`double_value < int_value`

the `int` value gets converted (***promoted***) to a `double` value so that two `double` values are compared:

`double_value < int_value_converted_to_double`

To see this process in operation do the following:

1) Insert an output statement like `cout << "Rock Constructor\n";` in your explicit-value constructor.

2) Recompile and link `Rock.cpp` and `rocktester.cpp` and execute the program.

3) Describe below your conclusion about how a `RockName` value gets converted to a `Rock` object.

____

Application of your **`Rock`** *Class—Bar-Graph Generator*

Now that you have a tested `Rock` class with quite a few capabilities, it's time to use it in an application: a bar-graph generator program like that in Project 2.2. You will run the program with the same data file, `Rockfile.txt` that was used there. You should get a copy of this file using the procedure specified by your instructor. (It can be downloaded from the website whose URL is given in the preface.)

Program Requirements

Write a program that uses your newly developed `Rock` type. You will use the file `Rockfile.txt` for input, which contains a random collection of rocks. The program you write should do the following:

1. Declare an integer array `count[]` whose indices are integers and all of whose elements are initialized to 0.

2. Read names of rocks from the file `RockFile.txt`, and for each rock, increment the appropriate element of `count[]` by 1; for example, if the rock is `Basalt`, then `count[BASALT]` should be incremented by 1. [Note that `RockName` values can be used as indices because each is associated with a nonnegative integer.]

3. Display the elements of `count` as a histogram (bar graph) something like the following:

```
BASALT      :XXXXXXXXXXXXXXX (15)

DOLOMITE    :XXXXX (5)
GRANITE     :XXXXXXXXXXXX (12)
GYPSUM      :XXX (3)
LIMESTONE   :XXXXXXXXXXXXXXXXXXXXXXXXXX (26)
MARBLE      :XXXXXXXXXXXXXX (14)
OBSIDIAN    :XXXXXXX (7)
QUARTZITE   : (0)
SANDSTONE   :XX (2)
SHALE       :X (1)
```

   where the length of each bar (the number of X's) and the number in parentheses indicate the number of times a rock with that name was found in the file.

But there are a couple of things to note:

(1) Objects of type `Rock` cannot be used as indices of an array because they are not integers. They are `Rock` objects. In particular they cannot be used as indices for the array `count`.

`RockName` enumerators can be used as indices because they are simply synonyms for integers. So you can use the values returned by the accessor function `name()` as an index in expressions of the form:

```
count[rockVal.name()]
```

(2) You need an operation to increment `Rock` values so you can run loops over the range of `Rock` values:

```
Rock r;
for (r = BASALT; r < ROCK_OVERFLOW; increment r to next rock)
{ ...
}
```

Note: The assignment of a `RockName` value to a `Rock` variable works for the same reason as the comparison of a `RockName` value with a `Rock` value.

One way to increment in the `for`-loop would be to modify the `next()` function from Lab 2.2 and Project 2.2 to make it a member function of the `Rock` class.

```
Rock r;

for (r = BASALT; r < ROCK_OVERFLOW; r = r.next())

{ ...

}
```

Another way, and the one you are to use, is to overload the ++ increment operator for class `Rock` so you can write loops like:

```
Rock r;

for (r = BASALT; r < ROCK_OVERFLOW; ++r)

{ ...

}
```

To do this you need to overload `operator++()` for class `Rock`. *You need overload only the prefix operator unless your instructor tells you to do both.*

To overload the ++ operator, you distinguish between prefix ++ and postfix ++ as follows:

- `operator++()` with no parameters is the *prefix* operator.
- `operator++(int)` with one `int` parameter is the *postfix* operator; no name need be given to the `int` parameter because it is not actually used in the definition.

Now you're ready to go—do the following:

- Add a prototype for `operator++()` to the `Rock` class that has no parameters and that will return a `Rock` value. This function should be a member function because it works "internally."
- The definition of `operator++()` should be put in the `Rock.cpp` file. This is accomplished in two steps:
  1. In the function heading, qualify its name as we do with all member functions.
  2. Use statements like those in the definition of the `next()` function from Lab 2.2 to find the successor of the data member `myName`, but change the value of `myName` to the successor value before you `return` it.

Hand in the items listed on the grade sheet.

Course Info: ________________ ***Name:*** ____

Project 4.1 Grade Sheet

Hand in:

1. This project handout with the answers filled in
2. Printouts showing:

   a. Listings of `Rock.h` and `Rock.cpp`

   Note: Be sure your header file has complete opening documentation and complete documentation of functions.

   b. A demonstration that the class library and driver program compile and link okay

   c. A sample run of the driver program
3. Printouts showing:

   a. A listing of the bar-graph program

   b A demonstration that the class library and this program compile and link okay

   c. A sample run of the bar-graph program from the application with the data file `RockFile.txt`

Attach this grade sheet to your printouts.

Category	Points Possible	Points Received
Class `Rock`	(85)	
Correctness of new operations	45	______
Structure/efficiency of functions	10	______
Opening documentation of `Rock.h`	5	______
Specifications of functions	15	______
Style/readability	10	______
Driver program: Sample run—adequate testing	20	______
Subtotal	**105**	______
Application program:	(75)	
Correctness	40	
Structure/efficiency	10	______
Documentation	10	______
Style/readability	5	______
Sample run	10	______
Subtotal	**75**	______
Total	**180**	______