Chris Dang Lab 5

1.1 Create a Rock.h header file. Check when finished\_\_x\_\_

1.2 Add enum RockName {BASALT, DOLOMITE, GRANITE, GYPSUM, LIMESTONE, MARBLE,

OBSIDIAN, QUARTZITE, SANDSTONE, SHALE, ROCK\_OVERFLOW} ;

under the public section of the Rock class. Check when finished\_\_x\_\_

1.3 Put a private section in your class that contains a single data member myName type RockName. Check when finished\_\_x\_\_

1.4 Load rocktester.cpp and compile. Describe what happens.

**An error shows that GRANITE was undefined.**

What happens is that the enumerated data type RockName is inside the declaration of the class rock so the scope only extends to the curly brace at the close of the declaration.

Add to qualifier Rock to GRANITE so that it reads Rock::GRANITE in rocktester.cpp and recompile. What happens? **The program compiles normally with no errors.**

Now, move enumerated type in Rock.h ahead of the class definition, and remove the Rock:: qualifier in rocktester.cpp. Compile and check. Check when finished\_\_x\_\_

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.) This will produce a lot of error messages. Find one of the messages that indicates multiple rock definitions and record it below. **error x2011: ‘RockName’: ‘enum’ type redefinition**

2.1 Add the following to the header file Rock.h

#ifndef ROCK

#define ROCK

. . . // put header code here

#endif

Then remove the duplicate #include “Rock.h” in rocktester.cpp. Check when finished\_\_x\_\_

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 when finished\_\_x\_\_

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 when finished and executes correctly\_\_x\_\_

4.1 Write a function member display() with the 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, appen the keyword const to the function heading in both the prototype and its definition.

function\_heading const

Put in the definition() in the implementation file Rock.cpp. You should reuse the code you write in project 2.2 to output rock enumerators with minor modifications.

When you have complete your display() function memeber 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. Then compile Rock.cpp and rocktester.cpp, link and execute the resulting binary executable. Record below the values output for:

sample\_\_\_ROCK\_OVERFLOW rockVal\_\_\_GRANITE

4.2 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 o p e r a t o r « () function in R o c k . h after

the end o f 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 o p e r a to r < < () should have:

• Return type o s tr e a m &—a reference to an output stream so that the output can be chained

• Signature ( o s tr e a m &, c o n s t Rock &)

So the prototype has the form

o s tr e a m & o p e r a t o r « ( o s t r e a m & o u t, c o n s t Rock & r o c k V a l) ;

The actual function body should do two things:

1) It should call the d i s p l a y () function member of its Rock parameter, and

2) It should return the o s tr e a m for chaining.

Add this definition to R o c k .c p p and then test that it works. To test it, comment out or remove the

lines labeled "BEGIN PART 4" and "END PART 4" in r o c k t e s t e r . c p p . Compile your

modified R o c k .c p p and r o c k t e s t e r . c p p files, link, and execute.

You should have gotten the same output as in step 4. Did you?\_\_**Yes**

5.1 Overload the >> operator:

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.

Enter BASALT as first Rock and then marble for second Rock. What are their outputs? **BASALT and MARBLE**

Enter Granite as first Rock and then gypSUM for second Rock. What are their outputs? **GRANITE and GYPSUM**

Enter feldspar as first Rock and then 99 for second Rock. What are their outputs? **ROCK\_OVERFLOW and ROCK\_OVERFLOW**

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 the outputs for the given inputs:

Input: BASALT output: **Igneous**

Input: Marble output: **Metamorphic**

Input: Shale output: **Sedimentary**

Input: Feldspar output: **ROCK\_OVERFLOW is not a Rock**

7. 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.

Input: BASALT output: **Intermediate**

Input: Marble output: **Fine**

Input: shale output: **Fine**

Input: Feldspar output: **ROCK\_OVERFLOW is not a Rock**

Project 4.1

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?

**Yes, it should be a const function because myName is not being changed at all**

Record the outputs for each input:

Input: BASALT Output: BASALT

Input: Marble Output: MARBLE

Input: shale Output: SHALE

Input: feldspar Output: ROCK\_OVERFLOW Check when finished\_\_x\_\_

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

Your objective now is to add the relational operators < and ==, where x < y for R o ck objects means that the value in myName of \* precedes the value in myName of yin the enumeration declaration.

Unlike the output operator o p e r a to r <<( ) and the input operator o p e r a t o r » ( ), the relational

operators o p e r a to r < () and o p er ato r= = () 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

s a m p le == GRANITE

but we cannot make the reverse expression

GRANITE == sample

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

Overload o p e r a to r < () and o p e r a t o r = = () to compare two R o ck objects. Each should have two

constant reference parameters of type R o ck and return one of the b o o l values t r u e or f a l s e . Your

parameters will look, for example, like c o n s t Rock & a R o ck l.

When you have finished overloading the operators, you should test them by commenting out or removing

the lines " BEGIN PART 9 ' ' a n d "END PART 9" in r o c k te s t e r . cpp. Then recompile R o c k .c p p

and r o c k t e s t e r . cp p , link, and execute.

Check here when the relational operators execute correctly\_\_x\_\_

10: What’s Going On Here?

So far you’ve compared R o c k s —for example, r o c k l < r o c k 2 .

You might want to compare a Rock object with a RockName—for

example, r o c k < GRANITE. To see what happens when you try this, do the following:

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

r o c k t e s t e r . cpp.

2) Recompile and link R o c k . c p p and r o c k t e s t e r . cp p and execute the resulting binary executable.

3) Describe any errors you encounter in this part o f the program

**No errors to report**

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

d o u b le \_ v a lu e < in t\_ v a lu e

the i n t value gets converted {promoted)to a d o u b le value so that two d o u b le values are compared:

d o u b le \_ v a lu e < in t\_ v a lu e \_ c o n v e r te d \_ to \_ d o u b le

To see this process in operation do the following:

1) Insert an output statement like c o u t « "Rock C o n s tr u c to r \n " ; in your explicit-value

constructor.

2) Recompile and link R o c k . cp p and r o c k t e s t e r . cp p and execute the program.

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

**The compiler does some work automatically to match the data type being assigned to the container**

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,

R o c k f i l e .t x t 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 R o ck type. You will use the file R o c k f i l e . t x t for

input, which contains a random collection of rocks. The program you write should do the following:

1. Declare an integer array c o u n t [ ] whose indices are integers and all of whose elements are initialized

to 0.

2. Read names of rocks from the file R o c k F i le . t x t , and for each rock, increment the appropriate

element of c o u n t [ ] by 1; for example, if the rock is B a s a l t , then c o u n t [ 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 c o u n t 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 c o u n t.

R o c k N a m e 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:

c o u n t[ r o c k V a l. n a m e ( ) ]

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

values:

Rock r ;

f o r ( r = BASALT; r < ROCK\_OVERFLOW; in c r e m e n t r to n e x t r o c k )

{ ...

}

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 f or-loop would be to modify the n e x t () 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.nextO)

{ . . .

}

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 prefixoperator.

• operator++ (int) with one int parameter is the postfixoperator; 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 o p e r a to r + + () should be put in the Rock.c p p 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 n e x t () 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 r e tu r n it.

