Queens College, CUNY, Department of Computer Science
Object-Oriented Programming in C++
CSCI 211/611
Summer 2018
Instructor: Dr. Sateesh Mane

due date Saturday, August 4, 2018, 11.59 pm (i.e. asap)

Homework: Polymorphism #2

- **Experience with other classes has demonstrated that in many cases the source of difficulty is not the mathematics or the programming.**
- **The source of difficulty is the English (understanding the text).**
- **If you do not understand the words in the lectures or homework, THEN ASK.**
- **If you do not understand the concepts in the lectures or homework, THEN ASK.**
- **Send me an email, explain what you do not understand.**
- **Do not just keep quiet and then produce nonsense in exams.**

- **Consult your lab instructor for assistance.**
- **You may also contact me directly, but I cannot promise a prompt response.**
- Please submit your inquiry via email, as a file attachment, to `Sateesh.Mane@qc.cuny.edu`.
- Please submit one zip archive with all your files in it.
  1. The zip archive should have either of the names (CS211 or CS611):

     `StudentId_first_last_CS211_hw_polymorphism2.zip`
     `StudentId_first_last_CS611_hw_polymorphism2.zip`
  2. The archive should contain one "text file" named "hw_polymorphism2.[txt/docx/pdf]" (if required) and cpp files named "Q1.cpp" and "Q2.cpp" etc.
  3. Note that a text file is not always required for every homework assignment.
  4. Note that not all questions may require a cpp file.

General information

- You should include the following header files, to run the programs below.

```
#include <iostream>
#include <fstream>
#include <sstream>
#include <iomanip>
#include <string>
#include <map>
#include <cmath>
```

- If you require additional header files to do your work, feel free to include them.
- **Include the list of all header files you use, in your solution for each question.**
- The questions below do not require complicated mathematical calculations.
- If for any reason you require help with mathematical calculations, **ask the lab instructor or the lecturer.**

Q1 Polymorphic “calc” classes

- This is an admittedly foolish artificial example, which could be implemented more sensibly in other ways, but the point here is to demonstrate polymorphism and an abstract base class with pure virtual functions.
- Our “polymorphic library” consists of the base class `CalcBase` and a function to use it.
- We shall write derived classes to sum the elements of an array (and store the results) in different ways.
- The inheritance tree has three levels, with various of virtual functions.

$$\begin{array}{ccc} & & \text{CalcProd} \\ \text{CalcSum} & & \text{CalcInv} \\ \nwarrow & & \nearrow \\ & \text{CalcBase} & \end{array}$$

Q2 Polymorphic "library"

- **Write a base class with a virtual destructor.**
- **The class has pure virtual functions. This makes it an abstract base class.**

```
class CalcBase {
public:
    virtual ~CalcBase() {                              // virtual destructor
        cout << "CalcBase destructor" << endl;
    }

    void calc(const vector<double> &v) {               // not virtual
        double d = value(v);                           // call virtual function
        set(d);                                        // call virtual function
    }

    virtual double get() const = 0;                    // pure virtual, const
    virtual void set(double d) = 0;                    // pure virtual, not const

protected:
virtual double value(const vector<double> &v) = 0;     // pure virtual, protected
};
```

- **Write the following function, which references only the base class.**

```
void PolyLib(CalcBase &ref_base)      // reference to base class
{
  vector<double> v;
  v.push_back(1.1);
  v.push_back(2.2);
  v.push_back(3.3);

  ref_base.calc(v);                   // non-virtual function
  cout << ref_base.get() << endl;     // virtual function
}
```

- **This is our polymorphic library.**
  1. *Someone else, the users of our library, will write the derived classes.*
  2. Whoever they are, this is the interface and they must override the pure virtual functions.

Q3 Derived class `CalcSum`

- **Write a derived class `CalcSum` which overrides the pure virtual functions.**
- Because we override all the pure virtual functions, objects of type `CalcSum` can be instantiated.
- The class has a protected data member `double x`;
- The default constructor initializes $x = 0$.
- The destructor is unnecessary, since there is no dynamic memory.
  However, we print a debugging statement in the destructor.
- The method `value` computes the average of the elements in the vector.
- *Do not waste time checking if the size of v is zero. Assume the data is valid.*
- **It is not necessary to write "virtual" in the derived classes.**
  1. A method that is tagged virtual in the base class is automatically virtual in the derived class.
  2. However, I prefer to write the virtual keyword in all the classes.
  3. It helps me to keep track of which methods are virtual.

```
class CalcSum : public CalcBase {
public:
  CalcSum() { x = 0; }
  ~CalcSum() {
    cout << "CalcSum destructor" << endl;
  }

  virtual double get() const;                      // accessor, return value of x
  virtual void set(double d);                      // mutator set x = d;

protected:
  virtual double value(const vector<double> &v) {
    // double sum = v[0] + v[1] + ...              // sum of elements of vector
    return sum / v.size();                         // return average
  }

  double x;                                        // data
};
```

Q4 Derived class `CalcInv`

- **Write a derived class `CalcInv` which overrides the pure virtual functions.**
- The class has a protected data member pointer `double *px`;
- The default constructor initializes `px` using dynamic memory.
- The destructor releases the dynamic memory (and we print a debugging statement).
- *To avoid wasting time, we make the copy constructor and assignment operator private.*
  1. Then we do not have to write a deep copy for them.
  2. We simply disable making copies of objects.
- **Note the following:**
  1. The method "value" sums the **inverses** of the elements in the vector.
  2. Do not worry about division by zero. Assume the data is valid.
  3. **Also note that the mutator sets `*pd = 1.0/d`.**
  4. Mathematically, the **harmonic mean** h of a set of numbers is defined as follows:

$$\frac{1}{h} = \frac{1}{n}\left(\frac{1}{v_0} + \cdots + \frac{1}{v_{n-1}}\right).$$

```
class CalcInv : public CalcBase {
public:
  CalcInv() : px(new double) { *px = 0; }     // constructor, allocate memory
  ~CalcInv() {                                // destructor, release memory
    delete px;
    cout << "CalcInv destructor" << endl;
  }

  virtual double get() const;             // accessor, return *px;
  virtual void set(double d);             // mutator, set *px = 1.0/d <-- ** note **

protected:
  virtual double value(const vector<double> &v) {
    // double sum = 1.0/v[0] + 1.0/v[1] + ...      // sum **inverses** of elements of v
    return sum / v.size();                         // return average
  }

  double *px;                                      // data

private:
  CalcInv(const CalcInv& orig) {}                  // private, disable copies
  CalcInv& operator= (const CalcInv& rhs) { return *this; }
};
```

Q5 Derived class `CalcProd`

- **Write a derived class `CalcProd` which overrides the pure virtual functions.**
- The class `CalcProd` inherits from `CalcInv`, not from the base class.
- The class has no data members of its own.
- We do not need a default constructor because there is nothing to initialize in the class.
  1. The compiler will generate a default constructor.
  2. It will invoke the `CalcInv` default constructor, which will allocate dynamic memory.
- The destructor is also unnecessary, because `CalcProd` does not allocate dynamic memory. However, we print a debugging statement in the destructor.
- **We override only the mutator.** The accessor uses `CalcInv`.
- **Note the following:**
  1. The method "value" computes the **product** of the elements in the vector.
  2. It calls the `std::pow` function to return output.
  3. **You must include the header `<cmath>`.**
  4. *Assume all the numbers are positive. Do not waste time on validation tests.*
  5. Mathematically, the **geometric mean** g of a set of numbers is defined as follows:

$$g = \left(v_0\, v_1\, \ldots\, v_{n-1}\right)^{1/n}.$$

```
class CalcProd : public CalcInv {          // ** derives from CalcInv **
public:                                    // not write default constructor
  ~CalcProd() {                            // destructor prints debugging statement
    cout << "CalcProd destructor" << endl;
  }
                                           // not override accessor
  virtual void set(double d);              // mutator set *px = d

protected:
  virtual double value(const vector<double> &v) {
    // double prod = v[0] * v[1] * ...           // product of elements of v
    return std::pow(prod, 1.0/v.size());         // ** include <cmath> header **
  }
};                                               // no data in class
```

Q6 Main program

- **Write a main program to test your code.**

  1. An example main program is given below.
  2. Use a base class pointer to point to a derived class object.
  3. Use a `CalcInv` pointer to point to a `CalcProd` object.
  4. Instantiate **objects** of derived classes and use them.

```
// relevant headers
using namespace std;

// class declarations

void PolyLib(CalcBase &ref_base)
{
  // etc
}

int main()
{
  CalcBase *pcs = new CalcSum;   // base class pointer to derived class
  CalcInv  ci;                   // object of derived class
  CalcInv  *pcp = new CalcProd;  // CalcInv pointer to CalcProd derived class

  PolyLib(*pcs);      // poly library uses correct derived class
  PolyLib(ci);
  PolyLib(*pcp);

  delete pcs;    // virtual destructor in base class, memory released correctly
  delete pcp;
  return 0;
}
```

Q7 Math: arithmetic, harmonic and geometric mean

- Suppose we have a set of n numbers $v_0, \ldots, v_{n-1}$.
- The **arithmetic, harmonic** and **geometric** means of the numbers are defined as follows.

$$
\begin{aligned}
a &= \frac{v_0 + \cdots + v_{n-1}}{n}\,, && \text{arithmetic} \\
\frac{1}{h} &= \frac{1}{n}\Bigl(\frac{1}{v_0} + \cdots + \frac{1}{v_{n-1}}\Bigr)\,, && \text{harmonic} \\
g &= \bigl(v_0\, v_1\, \ldots\, v_{n-1}\bigr)^{1/n}\,. && \text{geometric}
\end{aligned}
$$

- The arithmetic mean is what we usually call the average.
- The arithmetic mean always exists, if $n > 0$.
- The harmonic has problems of division by zero.
- The geometric has problems if some of the numbers are negative.
- Suppose all the numbers are positive.
  1. If the numbers are all equal, the means are all equal.
  2. Else they are all unequal and arithmetic mean is the largest and the harmonic mean is the smallest.
  $$h < g < a\,.$$