Ceng 241 Advanced Programming
Final
Jan 12, 2004 15.40–17.30
Good Luck!

1 (25 Pts) Create a class **Rectangle** with attributes **length** and **width**, each of which defaults to 1. Provide member functions that calculate the **perimeter** and the **area** of the rectangle. Also, provide *set* and *get* functions for the **length** and **width** attributes. The set functions should verify that **length** and **width** are each floating-point numbers larger than 0.0 and less than 20.0. Write a complete program for the class **Rectangle** with the above capabilities.

```
#ifndef HEADER_H
#define HEADER_H
class Rectangle {
public:
   Rectangle( double = 1.0, double = 1.0 );  // default constructor
   double perimeter();  // perimeter
   double area();  // area
   void setWidth( double w ); // set width
   void setLength( double l ); // set length
   double getWidth();           // get width
   double getLength();          // get length
private:
   double length;           // 1.0 < length < 20.0
   double width;            // 1.0 < width < 20.0
}; // end class Rectangle
#endif

// member function definitions
#include "header.h"
Rectangle::Rectangle( double w, double l )
{
   setWidth(w);  // invokes function setWidth
   setLength(l); // invokes function setLength
}

double Rectangle::perimeter()
{
return 2 * ( width + length ); // returns perimeter
}

double Rectangle::area()
{
return width * length;          // returns area
}

void Rectangle::setWidth( double w )
```

```
{
width = w > 0 && w < 20.0 ? w : 1.0; // sets width
}

void Rectangle::setLength( double l )
{
length = l > 0 && l < 20.0 ? l : 1.0; // sets length
}

double Rectangle::getWidth()
{
return width;
}

double Rectangle::getLength()
{
return length;
}

// main
#include <iostream>
using std::cout; using std::endl; using std::fixed;
#include <iomanip>
using std::setprecision;
#include "header.h"

int main()
{
   Rectangle a, b( 4.0, 5.0 ), c( 67.0, 888.0 );
   cout << fixed;
   cout << setprecision( 1 );
   cout << "a: length = " << a.getLength()
    << "; width = " << a.getWidth()
        << "; perimeter = " << a.perimeter() << "; area = "
        << a.area() << ’\n’;
   cout << "b: length = " << b.getLength()
    << "; width = " << b.getWidth()
        << "; perimeter = " << b.perimeter() << "; area = "
        << b.area() << ’\n’;
   cout << "c: length = " << c.getLength()
    << "; width = " << c.getWidth()
        << "; perimeter = " << c.perimeter() << "; area = "
        << c.area() << endl;
   return 0;
}
```

2 (25 Pts) Create a class called **Complex** for performing arithmetic with complex numbers. Complex numbers have the form

$$realpart + imaginarypart * i$$

where **i** is

$$\sqrt{-1}$$

- The class must be enable input and output of complex numbers through the overloaded $\gg$ and $\ll$ operators, respectively.
- Overload the multiplication operator to enable multiplication of two complex numbers as in algebra.
- Overload the $==$ and $! =$ operators to allow comparisons of complex numbers.
- Write a complete program.

```
 ---------------------------------------------
//complex1.h
#ifndef COMPLEX1_H
#define COMPLEX1_H
class Complex {
public:
   Complex( double = 0.0, double = 0.0 );       // constructor
   Complex operator+( const Complex & ) const;  // addition
   Complex operator-( const Complex & ) const;  // subtraction
   const Complex &operator=( const Complex & ); // assignment
   void print() const;                          // output
private:
   double real;       // real part
   double imaginary;  // imaginary part
};
#endif
 ---------------------------------------------
//complex1.cpp
#include <iostream>
using std::cout;
#include "complex1.h"
// Constructor
Complex::Complex( double r, double i )
   : real( r ), imaginary( i ) { }
// Overloaded addition operator
Complex Complex::operator+( const Complex &operand2 ) const
{
   return Complex( real + operand2.real,
                   imaginary + operand2.imaginary );
}
// Overloaded subtraction operator
```

```
Complex Complex::operator-( const Complex &operand2 ) const
{
   return Complex( real - operand2.real,
                   imaginary - operand2.imaginary );
}
// Overloaded = operator
const Complex& Complex::operator=( const Complex &right )
{
   real = right.real;
   imaginary = right.imaginary;
   return *this;   // enables cascading
}
// Display a Complex object in the form: (a, b)
void Complex::print() const
   { cout << ’(’ << real << ", " << imaginary << ’)’; }
----------------------------------------------
#include <iostream>
using std::cout;
using std::endl;
#include "complex1.h"
int main()
{
   Complex x, y( 4.3, 8.2 ), z( 3.3, 1.1 );
   cout << "x: ";
   x.print();
   cout << "\ny: ";
   y.print();
   cout << "\nz: ";
   z.print();
   x = y + z;
   cout << "\n\nx = y + z:\n";
   x.print();
   cout << " = ";
   y.print();
   cout << " + ";
   z.print();
   x = y - z;
   cout << "\n\nx = y - z:\n";
   x.print();
   cout << " = ";
   y.print();
   cout << " - ";
   z.print();
   cout << endl;
   return 0;
}
```

3 (30 Pts) Write down all the shapes you can think of both two-dimensional and three-dimensional and form those shapes into a shape hierarchy. Your hierarchy should have an abstract base class **Shape** from which class **TwoDimensionalShape** and class **ThreeDimensionalShape** are derived (these classes should also be abstract). Once you have developed the hierarchy, define each of the classes in the hierarchy. Use a **virtual print** function to output polymorphically the type and dimensions of each class. Also include **virtual area** and **volume** functions so these calculations can be performed for objects of each concrete class in the hierarchy. Write a complete program that tests the **Shape** class hierarchy.

Hints:

```
   shapes[ 0 ] = new Circle( 3.5, 6, 9 );
   shapes[ 1 ] = new Square( 12, 2, 2 );
   shapes[ 2 ] = new Sphere( 5, 1.5, 4.5 );
   shapes[ 3 ] = new Cube( 2.2 );
```

```
// Definition of base-class Shape
#ifndef SHAPE_H
#define SHAPE_H
#include <iostream>
using std::ostream;
class Shape {
   friend ostream & operator<<( ostream &, Shape & );
public:
   Shape( double = 0, double = 0 );
   double getCenterX() const;
   double getCenterY() const;
   virtual void print() const = 0;
protected:
   double xCenter;
   double yCenter;
};
#endif

// Member and friend definitions for Shape
#include "shape.h"
Shape::Shape( double x, double y )
{   xCenter = x;
   yCenter = y;}
double Shape::getCenterX() const { return xCenter; }
double Shape::getCenterY() const { return yCenter; }
ostream & operator<<( ostream &out, Shape &s )
{  s.print();
   return out;}

// Definition of class TwoDimensionalShape
#ifndef TWODIM_H
#define TWODIM_H
#include "shape.h"
```

```
class TwoDimensionalShape : public Shape {
public:
   TwoDimensionalShape( double x, double y ) : Shape( x, y ) { }
   virtual double area() const = 0;
};
#endif

// Defnition of class ThreeDimensionalShape
#ifndef THREEDIM_H
#define THREEDIM_H
#include "shape.h"
class ThreeDimensionalShape : public Shape {
public:
   ThreeDimensionalShape( double x, double y ) : Shape( x, y ) { }
   virtual double area() const = 0;
   virtual double volume() const = 0;
};
#endif

// Definition of class Circle
#ifndef CIRCLE_H
#define CIRCLE_H
#include "twodim.h"
class Circle : public TwoDimensionalShape {
public:
   Circle( double = 0, double = 0, double = 0 );
   double getRadius() const;
   double area() const;
   void print() const;
private:
   double radius;
};
#endif

// Member function definitions for Circle
#include "circle.h"
#include <iostream>
using std::cout;
Circle::Circle( double r, double x, double y )
   : TwoDimensionalShape( x, y ) { radius = r > 0 ? r : 0; }
double Circle::getRadius() const { return radius; }
double Circle::area() const { return 3.14159 * radius * radius; }
void Circle::print() const
{
cout << "Circle with radius " << radius << "; center at ("
    << xCenter << ", " << yCenter << ");\narea of " << area() <<'\n';}
```

```
// Definition of class Square
#ifndef SQUARE_H
#define SQUARE_H
#include "twodim.h"
class Square : public TwoDimensionalShape {
public:
   Square( double = 0, double = 0, double = 0 );
   double getSideLength() const;
   double area() const;
   void print() const;
private:
   double sideLength;
};
#endif

// Member function definitions for Square
#include "square.h"
#include <iostream>
using std::cout;
Square::Square( double s, double x, double y )
   : TwoDimensionalShape( x, y ) { sideLength = s > 0 ? s : 0; }
double Square::getSideLength() const { return sideLength; }
double Square::area() const { return sideLength * sideLength; }
void Square::print() const
{
cout << "Square with side length " << sideLength << "; center at ("
     << xCenter << ", " << yCenter << ");\narea of " << area() << '\n';}

// Definition of class Shere
#ifndef SPHERE_H
#define SPHERE_H
#include "threedim.h"
class Sphere : public ThreeDimensionalShape {
public:
   Sphere( double = 0, double = 0, double = 0 );
   double area() const;
   double volume() const;
   double getRadius() const;
   void print() const;
private:
   double radius;
};
#endif

// Member function definitions for Sphere
#include "sphere.h"
#include <iostream>
using std::cout;
```

```
Sphere::Sphere( double r, double x, double y )
   : ThreeDimensionalShape( x, y ) { radius = r > 0 ? r : 0; }
double Sphere::area() const
   { return 4.0 * 3.14159 * radius * radius; }
double Sphere::volume() const
   { return 4.0/3.0 * 3.14159 * radius * radius * radius; }
double Sphere::getRadius() const { return radius; }
void Sphere::print() const
{
cout << "Sphere with radius " << radius << "; center at ("
     << xCenter << ", " << yCenter << ");\narea of "
     << area() << "; volume of " << volume() << '\n';}

// Definition of class Cube
#ifndef CUBE_H
#define CUBE_H
#include "threedim.h"
class Cube : public ThreeDimensionalShape {
public:
   Cube( double = 0, double = 0, double = 0 );
   double area() const;
   double volume() const;
   double getSideLength() const;
   void print() const;
private:
   double sideLength;
};
#endif

// Member function definitions for Cube
#include "cube.h"
#include <iostream>
using std::cout;
Cube::Cube( double s, double x, double y )
   : ThreeDimensionalShape( x, y ) { sideLength = s > 0 ? s : 0; }
double Cube::area() const { return 6 * sideLength * sideLength; }
double Cube::volume() const
   { return sideLength * sideLength * sideLength; }
double Cube::getSideLength() const { return sideLength; }
void Cube::print() const
{
cout << "Cube with side length " << sideLength << "; center at ("
     << xCenter << ", " << yCenter << ");\narea of "
     << area() << "; volume of " << volume() << '\n';}

// Driver to test Shape hierarchy
#include <iostream>
using std::cout;
```

```
#include <vector>
using std::vector;
#include "shape.h"
#include "circle.h"
#include "square.h"
#include "sphere.h"
#include "cube.h"
int main()
{
   vector < Shape * > shapes( 4 );
   shapes[ 0 ] = new Circle( 3.5, 6, 9 );
   shapes[ 1 ] = new Square( 12, 2, 2 );
   shapes[ 2 ] = new Sphere( 5, 1.5, 4.5 );
   shapes[ 3 ] = new Cube( 2.2 );
   for ( int x = 0; x < 4; ++x )
      cout << *( shapes[ x ] ) << '\n';
   return 0;
}
```

4 (25 Pts) Write a program designed to generate and handle a memory exhaustion error. Your program should loop on a request to create dynamic storage through operator **new**.

Hint:

The output of the program should be as the following:

```
Allocated 50000000 long doubles in ptr[ 0 ]
Memory Allocation Failed.
```

```
#include <iostream>
using std::cout;using std::cerr;
#include <new>
using std::bad_alloc;
#include <cstdlib>
int main()
{
   long double *ptr[ 10 ];
   try {
      // loop will cause memory exhaustion
      for ( int i = 0; i < 10; ++i ) {
         ptr[ i ] = new long double[ 50000000 ];
         cout << "Allocated 50000000 long doubles in ptr[ "
              << i << " ]\n";
      }
   }
   // catch bad_alloc exception
   catch ( bad_alloc ex ) {
      cerr << "Memory Allocation Failed.\n";
      exit( EXIT_FAILURE );
   }
   return 0;
}
```