|  |
| --- |
| Lab04 Polymorphism |
|  |
| Exercise 1: Function Overriding and Upcasting |
| Guess the output of the following program. Then compile and run it to confirm your answer.  #include <iostream>  #include <string>  using namespace std;  class Human {  string name;  public:  Human (string name)  : name(name) {}  string getName() { return name; }  void shout() {  cout << "I'm a human. Name = " << name << "\n\n";  }  };  class Student : public Human {  int id;  public:  Student (string name, int id)  : Human(name), id(id) { }  };  int main() {  Human \*h1 = new Human("Michael"); // LINE1  h1->shout(); // LINE2  delete h1;  Student \*s1 = new Student("Kelly", 111); // LINE3  s1->shout(); // LINE4  h1 = new Student("Obama", 222); // LINE5  h1->shout(); // LINE6  delete h1;  Human& hr = \*s1; // LINE7  hr.shout(); // LINE8  delete s1;  };  Answer the following questions:   1. At the line labeled LINE1, what is the type of h1 and what type of object does h1 create? 2. At the line labeled LINE3, what is the type of s1 and what type of object does s1 create? 3. At the line labeled LINE5, what is the type of h1 and what type of object is h1 pointing to? 4. At the line labeled LINE7, what is the type of hr and what is the type of the object hr is referencing? 5. Which lines do upcasting occur? 6. What is the output? 7. Which method has been called? 8. Does function overriding occur? Why? 9. Add code to override the superclass’ shout() method at Student class. 10. At LINE6, h1 actually points to a Student object but it invokes Human::shout() instead of Student::shout(). Why? 11. At LINE8 hr actually references to a Student object, but it invokes Human::shout() instead of Student::shout(). Why? 12. At LINE4, Student::shout() is invoked correctly. Why? 13. LINE6 and LINE8 do not produce the best answer. Why? 14. What is the condition that makes the wrong method to be invoked in Step 13? |
| Exercise 2: Dynamic Polymorphism |
| Update your class in Exercise 1 to use dynamic polymorphism.  class Human {  string name;  public:  Human (string name)  : name(name) {}  void shout() {  cout << "I'm a human. Name = " << name << endl;  }  string getName() { return name; }  };  Then, answer the following question:   1. What is the new output? 2. Which method is invoked at LINE6 and LINE8? 3. Is the correct method being invoked now? 4. Under what circumstance must we use dynamic polymorphism? |

|  |
| --- |
| Exercise 3: Abstract Class and Pure Virtual Function |
| The Shape class in the following Class Diagram is an abstract class because it has 2 pure virtual functions: calcPerimeter() and dispaly(). (Italic denotes abstract or pure virtual.)   |  |  |  |  | | --- | --- | --- | --- | |  |  | *Shape* |  | |  |  | #perimeter:double |  | |  |  | +Shape()  *-calcPerimeter():void*  *+display():void* |  | |  |  |  |  | | Point |  | Triangle |  |  | | -x:int  -y:int |  | -points:Point[3] |  |  | | +Point(x:int,y:int)  +getX():int  +getY():int  +distance(p:Point):  double  +print():void  -calcPerimeter():void  +display():void | 3 1 | +Triangle(p1:Point,  p2:Point,p3:Point)  -calcPerimeter():void  +display():void |  |  | |  |  |  |  |  | |  | 1 |  |  | Circle | |  |  |  |  | -center:Point  -radius:double | |  |  |  | 1 | +Circle(center:Point,  radius:double)  -calcPerimeter():void  +display():void |   Answer the following questions:   1. Why it makes sense to declare the 2 methods as pure virtual. 2. The program below has compile errors. What cause the errors? Subclass Circle and subclass Triangle must implement the pure virtual function of the superclass Shape in order to become a concrete class (instantiate-able). 3. Fix the error so that it produces the expected output.   Expected output:  Point:  (1,1)  Perimeter = 0  Circle:  (1,1)  Radius = 10  Perimeter = 62.8  Triangle:  (0,0)  (3,0)  (3,4)  Perimeter = 12  #include <iostream>  #include <string>  #include <cmath>  using namespace std;  class Shape {  protected:  double perimeter;  virtual void calcPerimeter() = 0;  public:  Shape() : perimeter(0) {}  virtual void display() = 0;  };  class Point : public Shape {  int x;  int y;  public:  Point(int x = 0, int y = 0) : x(x), y(y) {}  int getX() { return x; }  int getY() { return y; }  double distance ( Point& p) {  int dx = x - p.x;  int dy = y - p.y;  return sqrt (dx\*dx + dy\*dy);  }  void calcPerimeter() { perimeter = 0; }  void print() { cout << "(" << x << "," << y << ")"; }  void display() {  cout << "Point:\n";  cout << "(" << x << "," << y << ")\n";  cout << "Perimeter = " << perimeter << endl;  }  };  class Circle : public Shape {  Point center;  double radius;  public:  Circle(Point center, double radius)  : center(center), radius(radius) {  calcPerimeter();  }  };  class Triangle : public Shape {  Point points[3];  public:  Triangle(Point p1, Point p2, Point p3) {  points[0] = p1;  points[1] = p2;  points[2] = p3;  calcPerimeter();  }  };  int main() {  Point p(1,1);  Circle c(Point(1,1), 10); // radius = 10.  Triangle t(Point(0,0), Point(3,0), Point(3,4));  p.display();  c.display();  t.display();  } |
| Exercise 4: Dynamic Polymorphism |
| The following main function and showShape function use the classes in the previous exercise but there are errors. Fix the errors in the 2 functions. Uses dynamic polymorphism.  void showShape(Shape s) {  cout << "Showing ";  s.display();  }  int main() {  Point p(1,1);  Circle c(Point(1,1), 10); // radius = 10.  Triangle t(Point(0,0), Point(3,0), Point(3,4));  showShape(p);  showShape(c);  showShape(t);  Shape s[] = { Point(1,1),  Circle(Point(1,1), 10),  Triangle(Point(0,0), Point(3,0), Point(3,4))};  for (int i = 0; i < 3; i++) {  s[i].display();  }  } |
|  |

|  |
| --- |
| Take Home Exercises |

1. Create an inheritance hierarchy of Rodent: Mouse, Gerbil, Beaver, and Hamster. In the base class Rodent defines methods that are common to all Rodents for instance: place(), tail(), and eat(). In the derived classes Mouse, Gerbil, Beaver, and Hamster, overrides these methods to provide unique behavior specific to them. Create an array of pointers to Rodent, fill it with different specific types of Rodents, and call your base class methods to display the specific behavior of the different types of rodents. A sample output of the program is given below:

Sample Output:

Mice are found all over the world.

Mice have long and baitless tail.

Mice eat cardboards, papers, clothes.

Gebrils are found in some parts of Africa and Asia.

Gebrils have long tail.

Gebrils eat seeds, roots, insects, parts of plants.

Beavers are found in Northern Europe and North Americal.

Beavers have broad tail.

Beavers ear bark.

Hamsters are found in Western Europe.

Hamsters have short tail.

Hamsters eat cereals.

1. Package-delivery services, such as FedEx®, DHL® and UPS®, offer a number of different shipping options, each with specific costs associated.
   1. Create an inheritance hierarchy to represent various types of packages. Use Package as the base class of the hierarchy, then include classes TwoDayPackage and OvernightPackage that derive from Package. Base class Package should include data members representing the name, address, city, state and ZIP code for both the sender and the recipient of the package, in addition to data members that store the weight (in ounces) and cost per ounce to ship the package. Package’s constructor should initialize these data members. Ensure that the weight and cost per ounce contain positive values. Package should provide a public member function calculateCost that returns a double indicating the cost associated with shipping the package. Package’s calculateCost function should determine the cost by multiplying the weight by the cost per ounce. Derived class TwoDayPackage should inherit the functionality of base class Package, but also include a data member that represents a flat fee that the shipping company charges for two-day-delivery service. TwoDayPackage’s constructor should receive a value to initialize this data member. TwoDayPackage should redefine member function calculateCost so that it computes the shipping cost by adding the flat fee to the weight-based cost calculated by base class Package’s calculateCost function. Class OvernightPackage should inherit directly from class Package and contain an additional data member representing an additional fee per ounce charged for overnight-delivery service. OvernightPackage should redefine member function calculateCost so that it adds the additional fee per ounce to the standard cost per ounce before calculating the shipping cost. Write a driver program that creates objects of each type of Package and tests member function calculateCost.
   2. Use the Package inheritance hierarchy created in Exercise (a) to create a program that displays the address information and calculates the shipping costs for several Packages. The program should contain a vector of Package pointers to objects of classes TwoDayPackage and OvernightPackage. Loop through the vector to process the Packages polymorphically. For each Package, invoke get functions to obtain the address information of the sender and the recipient, then print the two addresses as they would appear on mailing labels. Also, call each Package’s calculateCost member function and print the result. Keep track of the total shipping cost for all Packages in the vector, and display this total when the loop terminates.