CS-571 2023F HW02 20

Homework 2

Shailee Manojkumar Patel

spatel31@binghamton.edu

Harsimran Singh Avtar Singh Dhillon

hdhillon@binghamton.edu

1. The following questions apply to the attached Java code in Strng.java, Point.java, ColorPoint.java, and TestPoint.java.

```
public class TestPoint {
    public static void main (String[] args){
        ColorPoint pa = new ColorPoint( x: 1, y: 2, color: 3);      //Line 0
        Point pb = new Point( x: 4, y: 5);
        //pa = pb;                              // Line 1
        // pb = pa;                              // Line 2
        // pa = (ColorPoint)pb;       // Line 3
        // pb = (Point)pa;                 // Line 4
        System.out.println("Point pb is: " + pb);
        // System.out.println("Point pb is: " + pb.toStrng()); // Line 5
    }
}
```

a. After the line with the comment "Line 0" in TestPoint.java is executed, does the Java infrastructure create one continuous area on the heap with x, y and color fields; or does Java create two distinct areas, one with x and y, and the other with color?

Answer: The line 0 in TestPoint.java is

ColorPoint pa = new ColorPoint(1,2,3);

Here we are creating a new ColorPoint object **'pa'**. Where the ColorPoint class extends Point class. We have 2 variables **'x'** and **'y'** in the parent class **'Point'**. And we have another variable **'color'** in the child class **'ColorPoint'**. So when we create a new object of the child class **'ColorPoint'** a single memory will be allocated for both the child class **'ColorPoint'** and the parent class **'Point'.** The variables will be initialized by values passed to the object using the constructor defined in the child class.

So to conclude Java will only create one continuous area on the heap with x, y and color fields.

b. If the line with the comment "Line 1" in TestPoint.java were uncommented, but lines 2, 3, 4, and 5 are left commented, will the program still compile? If not, what error will occur? If so, what is the value of the x field in the object referenced by pa?

Answer: The line 1 in TestPoint.java is

pa = pb

If we were to uncomment this line and leave the lines 2, 3, 4 and 5 as commented. The program won't compile and give an error message as "**java: incompatible types: Point cannot be converted to ColorPoint**". We will get this message because we are trying to downcast the object which is not allowed implicitly in java.

c. If the line with the comment "Line 2" in TestPoint.java were uncommented, but lines 1, 3,4 and 5 are left commented, will the program still compile? If not, what error will occur? If so, what is the value of the x field in the object referenced by pb?

Answer: The Line 2 in TestPoint.java is:

pb = pa;

If we were to uncomment this line and leave the lines 1, 3, 4 and 5 as commented. The program would compile as we are performing upcasting which is allowed in java. Over here we are passing the reference of the child class to the parent class. So the parent class object **'pb'** will hold the values of the child class object **'pa'** i.e pb will have values of the variable **'x'** and **'y'** as **'1'** and **'2'** respectively. The value of the **x** field in the object referenced by pb is **1**.

```
Point pb is: (1,2)
```

d. If the line with the comment "Line 3" in TestPoint.java were uncommented, but lines 1, 2,4 and 5 are left commented, will the program still compile? If not, what error will occur? If so, what would happen when you ran the program?

Answer: The line 3 in TestPoint.java is:

pa = (ColorPoint)pb;

If we were to uncomment this line and leave the lines 1, 2, 4 and 5 as commented. The program won't compile and give an error message "**Exception in thread "main" java.lang.ClassCastException: class Point cannot be cast to class ColorPoint (Point and ColorPoint are in an unnamed module of loader 'app')**". Casting the object **'pb'** of class **'Point'** to **'pa'** of class **'ColorPoint'** will produce 'ClassCastException' for any non-null value.

e. If the line with the comment "Line 4" in TestPoint.java were uncommented, but lines 1, 2,3 and 5 are left commented, will the program still compile? If not, what error will occur? If so, what would happen when you ran the program?

Answer: The line 4 in TestPoint.java is:

pb = (Point)pa;

If we were to uncomment this line and leave the lines 1, 2, 3 and 5 as commented. The program would compile and produce an output as we are trying to perform upcasting over here. In this case scenario we are doing an explicit upcasting i.e specifying the type of the object explicitly while assigning it to another object. The object pb will be assigned with the values of the object pa where the values of pa(1,2,3) will override the values of pb(4,5) for the variable x and y. The code will print out the statement **"Point pb is: (1,2)"**

```
Point pb is: (1,2)
```

f. If the line with the comment "Line 5" in TestPoint.java were uncommented, but lines 1,2,3, and 4 are left commented, how many objects of the String class will be allocated when the program is run? Show the value of each String object created. (This was the way that Java Strings originally worked, but newer versions of Java optimize the "+" append method for Strings to use String Builders that do not require new objects for each append.)

Answer: The line 5 in TestPoint.java is:

System.out.println("Point pb is: " + pb.toStrng());

If we were to uncomment this line and leave the lines 1, 2, 3 and 4 as commented we will have the following output.

```
Point pb is: (4,5)
Made string from (
count : 1
Made string from 4
count : 2
Made string from (4
count : 3
Made string from ,
count : 4
Made string from (4,
count : 5
Made string from 5
count : 6
Made string from (4,5
count : 7
Made string from )
count : 8
Made string from (4,5)
count : 9
Point pb is: (4,5)
```

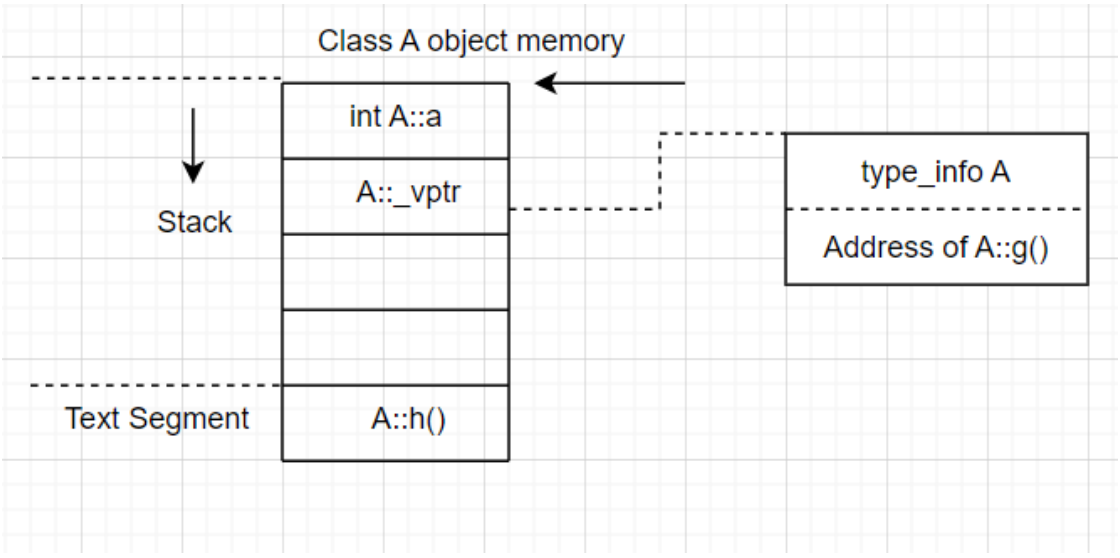
When the Line 5" in TestPoint.java is executed, there will be 9 String objects allocated to perform the toStrng() method. The toStrng() method is performing the append operation. The above screenshot shows the value of each String object created when the code is run. The final output printed would be **"Point pb is: (4,5)",** which is achieved by appending each string object to the existing one.

2. The following questions apply to the attached C++ code in prob2.cpp.

```cpp
#include <iostream>

using namespace std;

class A {
  public: int a;
  virtual void g();
  void h();
};
class B : public A {
  public: virtual void h();
  void f();
  int b;
};
class C  : public B {
  public: void h();
  void p();
};

int main() {
  return 0;
}
```

a. Describe the memory layout for an object in class A.

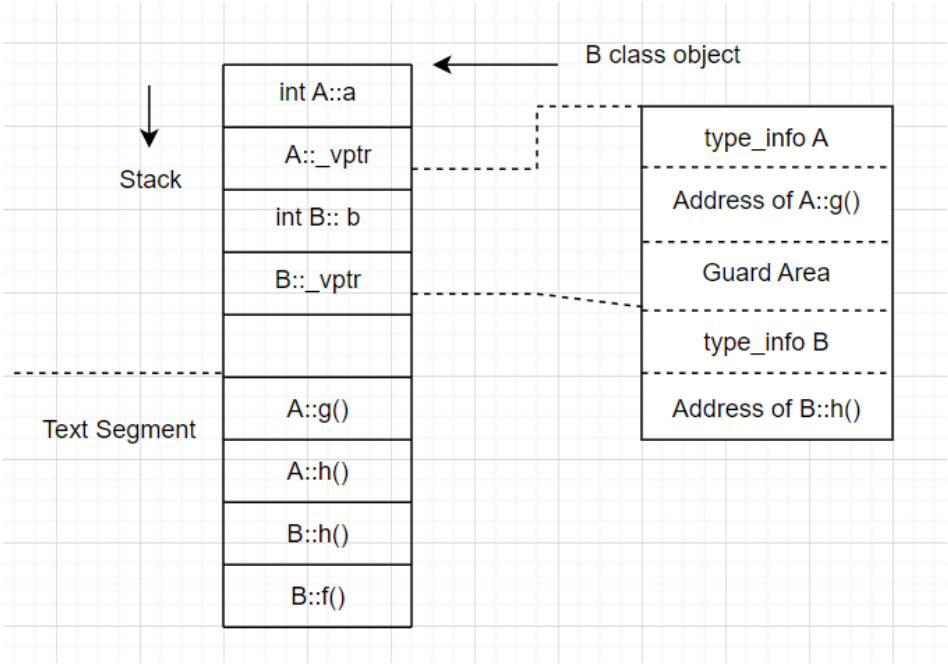
Memory layout for an object in class A:



Stack	Text Segment	Padding	Notes
A:: int a	void h()	NA	
Virtual Pointer g()		NA	This pointer points to the virtual method table of class A.

b. Describe the memory layout for an object in class B.

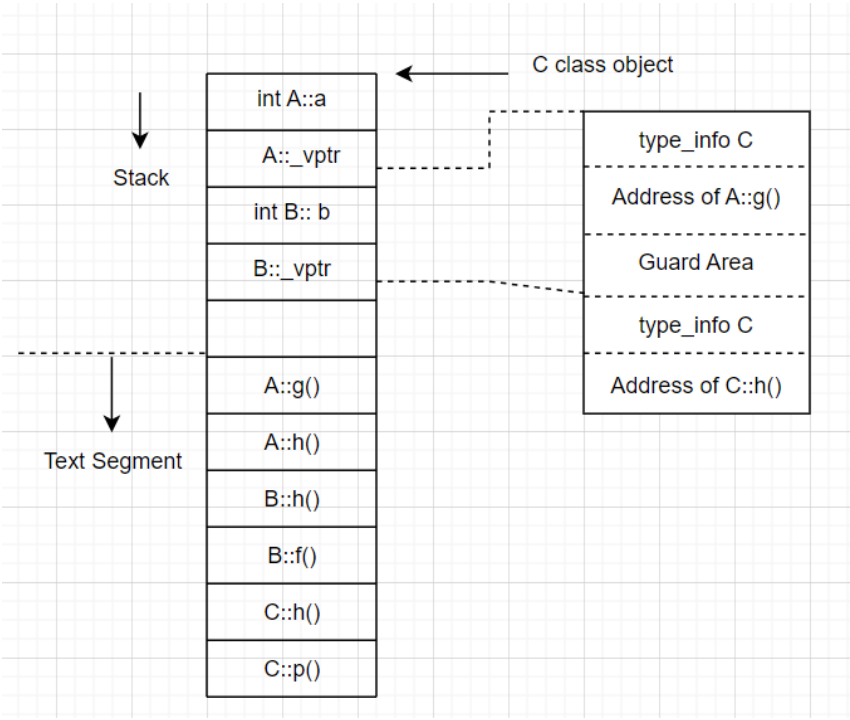
Memory layout for an object in class B.



Stack	**Text Segment**	**Padding**	**Notes**
A:: int a	void h()	NA	
B:: int b	void f()	NA	
Virtual Pointer g()		NA	This pointer points to the virtual method table of class A.
Virtual Pointer h()		NA	This pointer points to the virtual method table of class B.

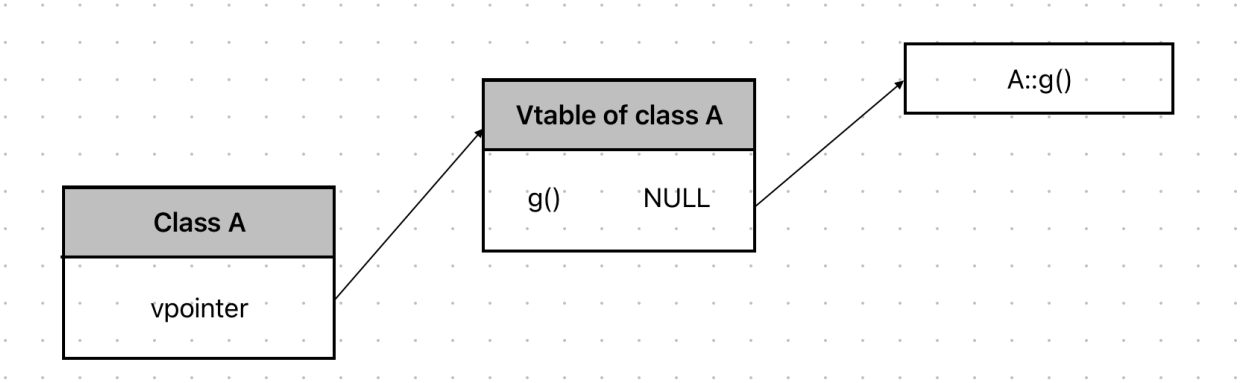
c. Describe the memory layout for an object in class C.

Memory layout for an object in class C.



Stack	Text Segment	Padding	Notes
A:: int a	void p()	NA	
B:: int b		NA	
Virtual Pointer g()		NA	This pointer points to the virtual method table of class A.
Virtual Pointer h()		NA	This pointer points to the virtual method table of class C.

d. Describe the VMT for class A.
Virtual Method table for class A:

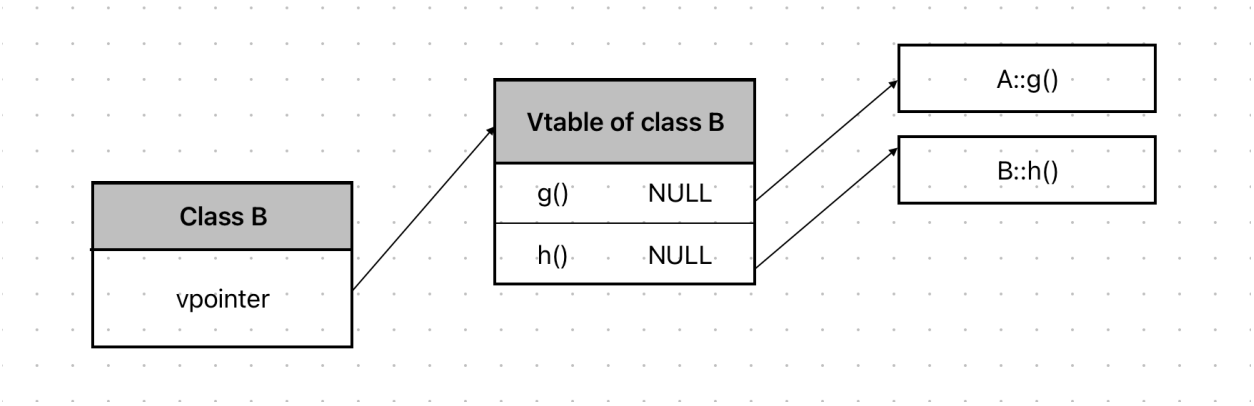


In this VMT:

- The vtable for class A contains an entry for the virtual function A::g().
- This entry points to the actual implementation of the g() function for class A.

When an object of class A or any derived class is created and a virtual function is called through a base class pointer or reference, the program looks up the appropriate function pointer in the vtable associated with the object's dynamic type to determine which implementation of the function to call.

e. Describe the VMT for class B.
Virtual Method table for class B:

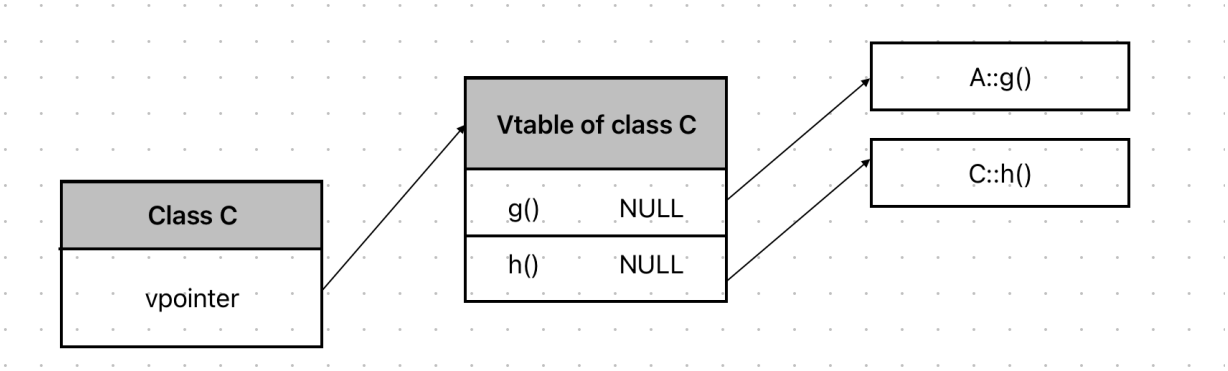


In this VMT:

- The first entry points to the implementation of the virtual function A::g() inherited from class A.
- The second entry points to the implementation of the virtual function B::h() as overridden in class B.

In 'class B', it overrides the 'h()' function declared in class A. When an object of class B is created and a virtual function is called through a pointer or reference of type A, the program uses the vtable associated with the object's dynamic type (in this case, class B) to determine which implementation of the virtual function to invoke.

f. Describe the VMT for class C.
Virtual Method Table for class C:



In this VMT:

- The first entry points to the implementation of the virtual function A::g() inherited from class A.
- The second entry points to the implementation of the virtual function C::h(), which is overridden in class C.

This VMT allows an object of class C to correctly dispatch calls to the virtual functions in the class hierarchy.

3. The following questions apply to the attached Java code in Pet.java, Cat.java, Dog.java, and Perform.java; which are almost exactly the same as the polymorphism example presented in class; but Perform.java has been changed.
   a. What does the code print out when the main function in Perform.java is invoked?
      When the main function in Perform.java is invoked, it prints:
      hmmph???
      hmmph???

      In the 'main' method, an array of Pet objects is created which contains one 'Cat' object and one 'Dog' object. In the for loop, letsHear(pets[i]) method is called for each element in the pets array. The 'letsHear' method is overloaded with letsHear(Cat cat), letsHear(Dog dog), and letsHear(Pet pet). But the letsHear method is determined at compile-time based on the reference type and not during the runtime of the object. In this case, the reference type is pet for both elements in the pets array. Therefore, letsHear(Pet pet) of reference type Pet is called and "humph???" Is printed for both Cat and Dog objects as they are considered as Pet objects.

   b. Does the code in Perform.java demonstrate polymorphism? Why or why not?
      The code in Perform.java does not demonstrate polymorphism because polymorphism involves different objects to respond differently to the same method call in different ways based on their types at runtime. In Perform.java code, the method calls within the for loop. Following reasons depicts the code does not show polymorphism:
      - Method overloading: The letsHear method is overloaded by letsHear(Cat cat), letsHear(Dog dog), and letsHear(Pet pet).
      - Static Binding: The code uses static binding for overloaded method (letsHear), where decision of overloaded method is made at compile-time and not at runtime.

   c. How could you change Perform.java so that it prints out "Meow" for cats, and "Woof" for dogs?

```
class Perform {
  static void letsHear(Pet pet) {
      String sound = pet.speak();
      System.out.println(sound);
    }
    public static void main(String[] args) {
      Pet[] pets = { new Cat(), new Dog() };
      for (int i = 0; i < pets.length; i++)
        letsHear(pets[i]);
    }
}
```

      To modify Perform.java, we will use method overriding and polymorphism.
      In this code:
      - Modified letsHear method to accept a Pet object as parameter and now it calls the speak method of the Pet object to retrieve sound.
      - The speak method in Cat and Dog classes is overridden to return "Meow" and "Woof", respectively when called.
      - In the main method, we create an array of Pet objects containing Cat and Dog instances.
      - When the function letsHear(pets[i]) is called, polymorphism makes sure that the speak method of the relevant object type (either Cat or Dog) is called and the corresponding sound ("Meow" or "Woof") is printed.

4. Many people think the Java "interface" mechanism is an encapsulation mechanism. Their arguments are as follows: Suppose we have an interface, A that is implemented in class B. If programmers consistently use type A every time an instance of B is referred to, then any method, say m, defined in B but not specified in A is hidden and cannot be accessed by any code holding a reference of type A. There is, however, a fatal flaw in this argument. Explain how a piece of code referring to an instance of B with type A can, in fact, access m.

The code can access m by defining in class B but not specified in the interface A. This can be done by casting the reference of B to type B as a reference from type A to type B is always allowed, since B is a subclass of A.
For example, the following code will illustrate to access m:

```
interface A {
  void methodA();
}
class B implements A {
    void m(){
        System.out.println("Method m() in class B");
    }
    public void methodA() {
        System.out.println("MethodA() in class B");
    }
}
public class Main {
   public static void main(String[] args) {
       A obj = new B(); // B implements A.
       obj.methodA(); // This calls the method specified in interface A

       // Attempting to call m through interface reference results in a compilation error:
       // obj.m();  // Compilation error
       // Cast the reference back to type B to access the method m().
       if (obj instanceof B) {
           B bObj = (B) obj;
           bObj.m(); // This calls the method m() in class B
       }
   }
}
```

In this example, initially we use an interface reference A obj to refer to an instance of B. This restricts to only call the methods declared in the interface A. However, casting the reference back to B, we can access and call method 'm' in class B. This demonstrates that while interfaces do not provide complete encapsulation as they restrict access to interface specific methods, non-interface methods are accessible by explicitly casting to its original class type.

In practice, using interfaces helps to enforce a level of abstraction and makes interaction with objects simpler, but it does not entirely hide or protect all aspects of the underlying implementation.