Homework 12: Dynamic Dispatch and Polymorphism

Instructor: Mehmet Emre

CS 32 Fall '21

Due: 11/2 2pm
Name & Perm #: Bharat Kathi (5938444)

Homework buddy (leave blank if you worked alone):
Reading: PS 15.3

1 (10 pts)

In the lectures, I mentioned *dynamic dispatch* is a form of polymorphism, and that some object-oriented programming resources mistakenly conflate the two.

1. (4 pts) What is dynamic dispatch? Why is it "dynamic"? Explain briefly.

   Answer: Dynamic dispatch is one form of polymorphism in which objects behave appropriately given their constructed type, even if assigned to a base class type. It is called "dynamic" since the imple- mentation for a virtual function is not fixed or apparent during compilation until it is used in the program, during which the appropriate implementation is dynamically chosen.

2. (4 pts) What is the purpose of polymorphism? How does it differ from dynamic dispatch? Polymorphism is more general than dynamic dispatch, explain how it is more general.

   Answer: In polymorphism, an operation has more than one implementation. As such, dynamic dispatch is a form of polymorphism since functions can have the same name, but different implementations that are chosen during run-time. However, polymorphic functions and classes can also build upon each other, with different implementations being bound during compile-time.

2

On page 868 in PS, in Display 15.12, line 17, there is a use of the overloaded operator < on two objects, one of type `Sale` and another of type `DiscountSale`. The definition of that operator appears on lines 25 - 28 of Display 15.10 on p. 866. On line 27, there is an invocation of `first.bill()` and an invocation of `second.bill()`.

1. (2 pts) For `first.bill()` in the case of the invocation in Display 15.12 line 17, where is the definition of the member function bill() that is invoked? Give the name of the class whose `bill()` function is invoked.

Answer: .

```
Sale
```

2. (2 pts) For `second.bill()` in the case of the invocation in Display 15.12 line 17, where is the definition of the member function bill() that is invoked? Give the name of the class whose `bill()` function is invoked.

   Answer: .

   ```
   DiscountSale
   ```

3. (2 pts) The `bill()` member function is special in that the exact definition of the function used depends on what type of object it is invoked on–whether it is an instance of `Sale` or `DiscountSale`, which may not be known until run-time. What is the C++ keyword that is used in the definition of `bill()` that signals this so called *dynamic dispatch* of the member function?

   Answer: .

   ```
   virtual
   ```

3 (8 pts)

Assume we have a base class (e.g. Person) and derived class (e.g. Student), and there is some function such as `toString()` that is defined in both the base class and the dervied class.

For example, suppose that:

- for Person, `toString` returns the person's name, e.g. Chris Gaucho
- for Student, `toString` returns the person's name and their perm number in parentheses. e.g. Chris Gaucho (1234567).

We say that `toString()` is *overridding* in the derived class. However, in PS (15.3), Savitch makes a distinction between the two cases, one that is properly called *overriding* and another that should really be called *redefinition*. Most of the cases we've seen so far are really just *redefinition*. What is different, according to Savitch, in the case where this should be called *overriding*? Hint: the `override` keyword covered in class is relevant here.

Answer: The override keyword is used when the parent class also has a virtual method with the same signature so we are overriding its behavior rather than defining another method with the same name. In other words, if the function in the parent class is virtual, a child class would be overriding that function since a new implementation overrides the virtual one. If the function in the parent class isn't virtual, then a child class would simply be redefining that function. Although, from a programmers perspective, this distinction seems unnecessary, it is important to the compiler since it is treated differently by the compiler.

4

Given the following class definitions (you may assume all necessary libraries have already been included):

```
class A {
public:
    ~A() { cout << "A::~A()" << endl; }
    void f1() { cout << "A::f1()" << endl; }
    virtual void f2() { cout << "A::f2()" << endl; }
};
class B : public A {
public:
    virtual ~B() { cout << "B::~B()" << endl; }
    virtual void f1() { cout << "B::f1()" << endl; }
    void f2() { cout << "B::f2()" << endl; }
    virtual void f3() = 0;
};
class C : public B {
public:
    ~C() { cout << "C::~C()" << endl; }
    void f1() { cout << "C::f1()" << endl; }
    virtual void f3() { cout << "C::f3()" << endl; }
};
```

1. (6 pts) What will the output be if we ran the following code (be sure to include destructors' output):

   ```
   void f1() { C c1; A a1 = c1; a1.f1(); a1.f2(); }

   int main() { f1(); }
   ```

   Answer: .

   ```
   A::f1()
   A::f2()
   A::~A()
   C::~C()
   B::~B()
   A::~A()
   ```

2. (6 pts) What will the output be if we ran the following code (be sure to include destructors' output):

   ```
   void f2() { B* b1 = new C(); b1->f1(); b1->f2(); b1->f3(); delete b1; }

   int main() { f2(); }
   ```

   Answer: .

```
C::f1()
B::f2()
C::f3()
C::~C()
B::~B()
A::~A()
```