

HW 3 Solution

HW3a solution

```
int main(int argc, char **argv) {  
  
    Base *b = new Base( );  
    Derived *d = new Derived( );  
  
    b->f1( ); // prints "Base f1" L1  
    b->f2( ); // prints "Base f2" L2  
  
    d->f1( ); // prints "Base f1" L3  
    d->f2( ); // prints "Derived f2" L4  
  
    b = d;  
    b->f1( ); // prints "Base f1" L5  
    b->f2( ); // prints "Derived f2" L6  
}
```

L1 implies there must be an *f1*
in Base

L2 implies there must be an *f2*
in Base

So we know there is an *f1* and
f2 in Base

```
int main(int argc, char **argv) {
```

```
    Base *b = new Base( );  
    Derived *d = new Derived( );
```

```
    b->f1( ); // prints "Base f1" L1  
    b->f2( ); // prints "Base f2" L2
```

```
    d->f1( ); // prints "Base f1" L3  
    d->f2( ); // prints "Derived f2" L4
```

```
    b = d;  
    b->f1( ); // prints "Base f1" L5  
    b->f2( ); // prints "Derived f2" L6
```

```
}
```

HW3a solution

L3 implies there is no *f1* in *Derived*, otherwise *Derived's f1* would be called since *d* points to a *Derived* object

L4 implies there must be an *f2* in *Derived*, otherwise *Base's f2* would be called, since *Derived* would have inherited *Base's f2*.

So we know there is an *f1* and *f2* in *Base* (from the previous slide) and that there is no *f1* in *Derived*, but there is an *f2* in *Derived*, from this slide.

```
int main(int argc, char **argv) {
```

```
    Base *b = new Base( );  
    Derived *d = new Derived( );
```

```
    b->f1( ); // prints "Base f1" L1
```

```
    b->f2( ); // prints "Base f2" L2
```

```
    d->f1( ); // prints "Base f1" L3
```

```
    d->f2( ); // prints "Derived f2" L4
```

```
    b = d;
```

```
    b->f1( ); // prints "Base f1" L5
```

```
    b->f2( ); // prints "Derived f2" L6
```

```
}
```

HW3a solution

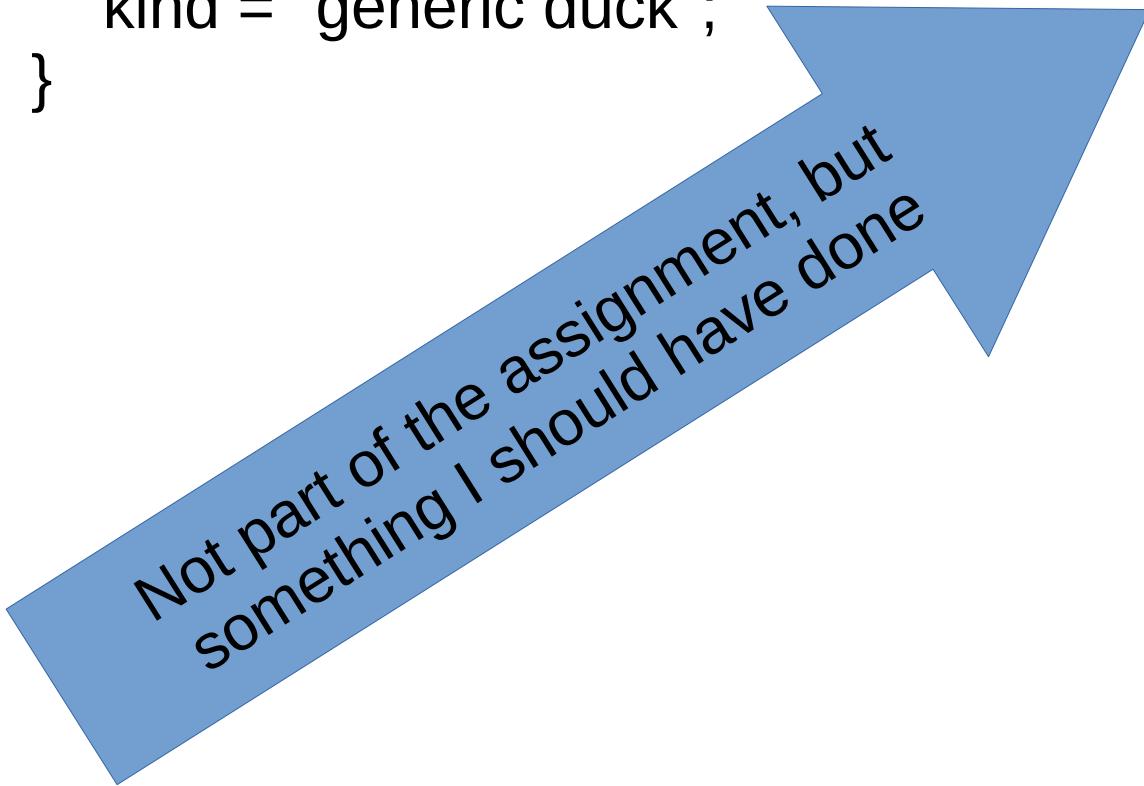
L5 implies that either *f1* in *Base* is not virtual, so there is no polymorphism, (inclusive) or it is virtual and there is no *f1* in *Derived*. We know from *L4* that there is no *f1* in derived.

From *L2* and *L3* we know there is an *f2* in *Base* and *Derived*, thus *f2* must be virtual in *Base*.

So we know there is an *f1* and *f2* in *Base* (from the previous slide) and that there is no *f1* in *Derived*, but there is an *f2* in *Derived*, from this slide. Moreover, from *L6* we know *f2* is virtual, but *L5* doesn't give enough information as to whether *f1* is virtual or derived, so it can be either.

The Homework 3b solution

```
Duck::Duck( ) : quackBehavior(NULL), flyBehavior(NULL) {  
    kind = "generic duck";  
}
```



Not part of the assignment, but
something I should have done

Just because code
runs it doesn't mean
it is correct.

Base.h and Base.cpp

```
class Base {  
public:  
    Base( );  
    virtual ~Base( );  
  
    virtual void f1( );  
    virtual void f2( );  
};
```

```
#include "Base.h"  
#include <iostream>  
  
Base::Base( ) { }  
Base::~~Base( ) { }  
  
void Base::f1( ) {  
    std::cout << "Base f1" << std::endl;  
}  
  
void Base::f2( ) {  
    std::cout << "Base f2" << std::endl;  
}
```

Base.h and Base.cpp

```
class Derived :  
public Base {  
public:  
    Derived( );  
    virtual  
    ~Derived( );  
  
    void f2( );  
};
```

```
Derived::Derived( ) { }  
Derived::~~Derived( ) { }  
  
void Derived::f2( ) {  
    std::cout << "Derived f2" << std::endl;
```

The code for DogToy.cpp

```
#include <iostream>
#include "DogToy.h"

DogToy::DogToy( ) : Duck("DogToy") {
    setFlyBehavior(new NoFly( ));
    setQuackBehavior(new Squeak( ));
}
DogToy::~DogToy( ) { }
```

Code for the the other places I said to PUT code is similar, with different objects passed to the setter functions.

The strategy pattern

Design Principle: Identify the aspects of your application that change, and separate them from what doesn't change.

We do this by putting the invariant behaviors into the base class methods (e.g., `display`, which always prints the value of *kind*), and representing the changing behaviors as *HASA* relations (e.g., `HASA flyBehavior`, `HASA a quackBehavior`), or as functionality in the derived class, if appropriate.

This also makes it easy to change the behavior at runtime, by changing the object that `quackBehavior` points to, for example.

Design Principle: Program to the interface, not an implementation.

We do this by calling functions on derived objects, but call their functions through base class pointers. Thus, we call a QuackBehavior quack method, not a Squeak quack method.

// WE DO THIS

```
void Duck::quack( ) { // in Duck
    if (quackBehavior != NULL) {
        quackBehavior->quack( );
    }
}
```

// WE DON'T DO THIS

```
void Duck::quack( ) { // in DogToy
    if (Squeak != NULL) {
        Squeak->quack( );
    }
}
```

Design Principle: Favor composition over inheritance

When we used a HASA relation for quackBehavior and flyBehavior, we were *composing*, or combining, the functionality of the Duck class with the functionality provided by the QuackBehavior and FlyBehavior classes.

Again, composition allows us to factor out properties that are not invariants of the base class, and allows us to dynamically change the behavior of certain properties, something that is harder to do if these behaviors are hard-coded into the base class.

Our solution is not ideal

For QuackBehavior and FlyBehavior, the base QuackBehavior and FlyBehavior classes had to define *quack()* and *fly()* methods, respectively. Yet they were always overridden.

We would like a way to force the derived class programmer to define methods without having to define those methods ourselves.

This allows the behavior to only be defined in the derived class, where it is known, but for people using our classes to program to the interface provided by the base class. When we discuss *abstract methods* next week we'll have a way to do this. And we'll make use of it in another pattern.