Lecture 09a
OO Implementation & Dispatch (redux)
March 21st, 2016

Announcements
• Reading: Chapter 6
  – Quiz due by start of class today
  – No reading or quiz next Monday
• Recitations:
  – More Google Test
  – Mandatory
• PA6 is assigned today
  – Due a week from Wednesday (2 extra days)
• No Office Hours for me this week
• Wednesday & Friday
  – Guest lecturer (Fritz Sieker)
  – Important material

PA6
• Opportunity to use inheritance
  – Where?
  – Why not inclusion?
• Opportunity to use dynamic dispatch
  – What method?

Dispatch
• How methods are selected and invoked
• Methods/functions selected by signature
  – Name must match (e.g. WarmUp)
  – Arguments must match
    • Data types must match
    • Including hidden arguments
    • Parameter names are irrelevant
  – Return types not used for selection
• Polymorphism creates ambiguities
  – Animal::warmUp();
  – Mammal::warmUp();

Dispatch (II)
• Static Dispatch
  – Functions selected at compile time
  – Based on declared data types
  – This is the C++ default
• Dynamic Dispatch
  – Functions selected at run-time
  – Based on run-time data type of object
    • Run-time type is "sticky"
    • But does not apply to copies
    • Signaled by the ‘virtual’ keyword in C++

Dispatch Syntax
• A method uses virtual dispatch if it is declared virtual
  – E.g.: virtual bool warm_up();
• If a method overwrites an inherited method that
  it virtual, the new method is virtual, too.
  – Assume Equine inherits Mammal
  – Mammal’s warm_up is virtual
  – So Equine’s warm_up is virtual too
  – E.g.: bool warm_up(); // inside Equine, virtual
• Very bad style to rely on this
  – Declare overwrites to be virtual too
  – Otherwise readers have to loop up the parent class...
Returning to Our Example...

```cpp
class Animal {
public:
    bool WarmUp() {return false;}
    void Behave() {WarmUp();}
};
class Mammal : public Animal {
public:
    bool WarmUp() {shiver(); return true;}
};
```

What Happens?

```cpp
Mammal m;
m.Behave();
```

- Does Shiver() get called?
- Does Behave() return true or false?

- In Java: Shiver() gets called
- In C++: your choice
  - As written, Shiver() does NOT get called

Back to WarmUp()

```cpp
class Animal {
public:
    bool WarmUp() {return false;}
    void Behave() {WarmUp();}
};
class Mammal : public Animal {
public:
    bool WarmUp() {shiver(); return true;}
};
```

Static Dispatch: WarmUp()

- 'm' is created as a Mammal
- m.Behave() passes m to Animal::Behave()
  - m is the hidden argument
  - The hidden argument is of type Animal
  - Polymorphism
- In Animal::Behave(), 'this' is of type Animal
- Animal::Behave() calls WarmUp()
  - "this' is the hidden argument
  - "this' is of type Animal
  - So Animal::WarmUp() is called

WarmUp() version 2

```cpp
class Animal {
public:
    virtual bool WarmUp() {return false;}
    void Behave() {WarmUp();}
};
class Mammal : public Animal {
public:
    virtual bool WarmUp() {shiver(); return true;}
};
```

Dynamic Dispatch: WarmUp()

- 'm' is created as a Mammal
- m.Behave() passes m to Animal::Behave()
  - m is the hidden argument
  - The hidden argument is of type Animal
  - m's run-time type is still Mammal
- Animal::Behave() calls WarmUp()
  - "this' is the hidden argument
  - The run-time type of 'this' is still Mammal
  - So Mammal::WarmUp() is called
Organization of Inheritance

- Assume Quagga inherits Equine
- Equine inherits Mammal
- Mammal inherits Animal

This address can be:
- Animal*: Animal*
- Mammal*: Mammal*
- Equine*: Equine*
- Quagga*: Quagga*

Polymorphism doesn’t have to change the address, just the (compile-time) type

Back Again to Animals & Mammals...

class Animal {
public:
    bool warmup() {return false;}
    void behave() {warmup();}
    virtual int& lifespan() {return lifespan;}
    protected:
        int lifespan;
};
class Mammal : public Animal {
public:
    bool warmup() {shiver(); return true;}
    inline int& body_temp() {return body_temp;}
    protected:
        int body_temp;
};

Dynamic Dispatch

- If LifeSpan() is virtual, it function call can’t be hardwired...

What happens?
The compiler doesn’t know how ‘a’ was created

It might be an instance of a class that overwrote LifeSpan()...

So it doesn’t know what function to call...

Virtual Function Pointer Table (VFPT)

- Every class with virtual functions has a VFPT
- Every entry in this table is the address of a virtual method
- If LifeSpan() is the 1st entry for Animal, it is also the 1st entry for every derived class from Animal
- Note: one table per class, not per object

void Foo(Animal& a){
a.LifeSpan() = 75;
}

What happens?
The compiler jumps to the address in the VFPT* (1st field)

This will be the VFPT of the object as it was created

LifeSpan will be the 1st entry in this table (selected by name)

Jump to the code at that address

Implementing Dynamic Dispatch

- There is 1 exception to inheritance == union
- For objects with virtual functions
  - The first field is a pointer to the VFPT
  - Before any instances of parent classes
  - It points to the VFPT of the object as created

Implementing Dynamic Dispatch II

- Dynamic dispatch is implemented through the VFPT

What happens?
The compiler jumps to the address in the VFPT* (1st field)

This will be the VFPT of the object as it was created

LifeSpan will be the 1st entry in this table (selected by name)

Jump to the code at that address
Compile-time type vs Run-time type

- Compile-time type is how an object is declared
  - How it was created doesn’t matter
  - Used for static dispatch
  - Remember: this is the default

- Run-time type is how an object was created
  - As determined by its VFPT ptr
  - Used for dynamic dispatch
  - Exploited by virtual methods