Announcements

- Quiz #6 due Tuesday (before class)
- Midterm #1 was returned last Thursday
  - Today is the last day I will bring them to class
  - You can still get them by meeting me in my office
  - I also have regraded tests
- PA4 is due Tuesday
  - Counts twice, so give it your best effort
  - Any questions?

PA4 Hints (Condensed)

1. No method longer than 15 lines
2. Lots of short, well-named methods
3. Multiple classes
   - Encapsulation
   - Abstraction

Object Oriented Programming

- Why do we use objects?
  - **Encapsulation**
    - Collect related data & code together
  - **Polymorphism**
    - Goal: abstraction
      - Quaggas have properties
      - Some properties shared by all Equines
      - Some shared by all Mammals
      - Or all Animals
    - Write methods at most abstract level possible
      - Avoids repeating code

What is inheritance?

- If class B inherits class A, what does that mean?
  - Inheritance is union
    - Every field in A is included in B
    - Every method in A is included in B
    - B then adds its own fields and methods
  - Note: you cannot inherit part of a parent
- Inheritance is mandatory in Java
  - Every class has a parent
    - By default, the class object
  - Every object therefore has extra data & methods
- Inheritance is optional in C++
  - Most classes have no parent
  - Makes inheritance “lighter”
  - Makes run-time typing harder
  - Java run-time type information is stored in the object class

A Classic Inheritance Hierarchy

- For future examples...
Method Overloading

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

What Happens?
Mammal m;
m.Behave();

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

• In Java: Shiver() gets called
• In C++: Shiver() does not get called
    – As written, at least
    – You shouldn’t be surprised to find out you have a choice...

Dispatch

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

Dispatch (II)

• Static Dispatch
    – Functions selected at compile time
    – Based on declared data types
    – This is how C dispatches functions
    – 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
    – This is the Java default
    – Signaled by the ‘virtual’ keyword in C++

Back to WarmUp()

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

Mammal m;
m.Behave();

• Note that virtual keyword never appears
• Therefore dispatch will be static

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 : Mammal → Animal
• 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

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

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
  - But m's sticky run-time type is still Mammal
- Animal::Behave() calls WarmUp()
  - 'this' is the hidden argument
  - WarmUp() is virtual
  - The run-time type of 'this' is still Mammal
  - So Mammal::WarmUp() is called

Dispatch (II) Redux

- 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++

Under the Hood

- How is inheritance implemented?
- How is dynamic dispatch implemented?
- How is static dispatch implemented?

Implementation: Animal

- How are Animal objects implemented?
  - What is sizeof(Animal)?
    - Caveat: implementations may differ...
    - Animal has 1 data field of type int
    - sizeof(Animal) == sizeof(int) \textit{potentially}
  - What about Animal's methods?
    - Implemented once per class
    - Not once per object
    - Executable code stored in memory

Back to Animals & Mammals...

// Note: nothing is virtual
class Animal {
public:
   bool WarmUp() {return false;}
   void Behave() {WarmUp();}
   inline int& LifeSpan() {return lifespan;}
protected:
   int lifespan;
};
class Mammal : public Animal {
public:
   bool WarmUp() {shiver(); return true;}
   inline int& BodyTemp() {return body_temp;}
protected:
   int body_temp;
};
Static Dispatch

- None of Animal's methods are virtual
- Therefore all dispatch is static

```
static void main() {
  Animal a;
  a.LifeSpan();
  // ignoring inline...
  Animal a;
  a.LifeSpan() = 75;
}
```

What happens?
- 'a' is a location on the stack
- Sizeof(a) == sizeof(int)
- 'LifeSpan' is a block of machine instructions in memory

At compile time, assembly code is generated to jump to the address of 'LifeSpan'

Implementation: Mammal

- How are Mammal objects implemented?
  - What is sizeof(Mammal)?
    - Same caveat as before...
    - Every Mammal contains an Animal object
    - Mammals also have their own int field (1: body_temp)
    - Sizeof(Mammal) == (sizeof(Animal) + sizeof(int))
    - In general: sizeof(child) == sizeof(parent) + sizeof(child's fields)
  - How is Mammal organized?
    - The first thing in a Mammal is an Animal
    - As a result, any Mammal* is also an Animal*

Organization of Inheritance

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

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& BodyTemp() {return body_temp;}
    protected
        int body_temp;
};
```

Dynamic Dispatch

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

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

What happens?
- The compiler doesn’t know how ‘a’ was created
- It might be an instance of a child 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
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 an entry in this table (selected by name)
  Jump to the code at that address

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

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