Lecture 09b
Multiple Inheritance

Oct. 17th, 2017

Announcements

• Midterm #2: October 31st
  – Week from Tuesday
  – Halloween! (spooky)
• PA6 due Tuesday
  – Any questions?

OO Topics Still to Discuss

• Shadowing ✔ (1 slide review)
• Multiple Inheritance ✔ (2 slide review)
  – New to C++ (not in Java)
• Pure Virtual Classes
  – Replaces Java interfaces
  – Benefits from multiple inheritance
• Slicing
  – A problem new to C++ (not in Java)

Shadowing

• Shadowing is when:
  – two fields have the same name
  – two non-virtual methods have the same signatures
• When fields or methods are shadowed
  – The compiler will select the version defined in the current class
  – Otherwise, the one defined in the immediate parent
  – Otherwise, the parent of the parent
  – And so on up the hierarchy…
• Fully-qualified names over-ride this default

Multiple Inheritance Example

class Quagga : public Equine,
  public ZooProperty {
public:
  Quagga() {name = "Pete";}
protected:
  string name;
};

Quagga Visualized

• Quagga’s contain an Equine
• Quagga’s contain a ZooProperty
• Two lifespans!
What happened to interfaces?

- Java has classes and interfaces
- Interfaces provide a 'public face'
  - Shared among multiple classes
  - But no implementation:
    - No constructors
    - No data fields
    - No implemented methods
      - Well, static methods are allowed in interfaces in Java 8
      - And constant values
  - Nonetheless, interfaces are needed

Interfaces in C++

- In C++, interfaces are just classes
- Example: ZooProperty might be an interface
  - Animals might be zoo property
  - Toys in the gift shop might be zoo property
  - Garbage cans might be zoo property
- These are totally different concepts
  - But they share properties
    - Monetary value
    - Location in zoo

Java Interfaces

- In Java, interfaces are limited
  - They can define methods like
    - Value()
    - Location()
  - But they can only implement static methods
  - And they cannot store data like
    - Double value
    - (x,y) location
- Why does Java have these limitations?

C++ ZooProperty Interface

```cpp
// A C++ interface as a class
class ZooProperty {
public:
  ZooProperty(double value, int x, int y) :
    loc_x(x), loc_y(y) {} 
  inline double Value() const {return value;} 
  inline double X() const {return loc_x;} 
  inline double Y() const {return loc_y;} 
protected:
  double value;
  double loc_x;
  double loc_y;
};
```

C++ Interfaces

- In C++, interfaces are just classes
  - Often with no parent
  - At the least, orthogonal to the main is-a hierarchy
- Multiple inheritance allows them to be added to other classes
- C++ interface classes support
  - Data fields
  - Non-static (& static) methods
  - Constructors & Destructors

Pure Virtual Methods

- Imagine a ZooProperty method called InCaseOfEmergency()
  - For animals, rush them to shelter
  - For toys, close shop
  - For garbage cans, do nothing
- How do you implement this method at the level of ZooProperty?
  - Note: this is a lot like Animal::WarmUp()
- Answer: you don’t
  - You define it, but don’t implement it
Pure Virtual Example

```
// A C++ interface as a class
class ZooProperty {
public:
  ZooProperty(double value, int x, int y)
    : loc_x(x), loc_y(y) {}
  virtual bool InCaseOfEmergency() = 0;

  inline double Value() const {return value;}
  inline double X() const {return loc_x;}
  inline double Y() const {return loc_y;}
protected:
  double value;
  double loc_x;
  double loc_y;
};
```

Pure Virtual Methods (II)

- Syntax: `virtual signature = 0;`
  - Note that only virtual methods can do this
- Semantics:
  - The method is defined but not implemented
  - Think of it like
    - Allocating an entry in the Virtual Function Pointer Table
    - And then filling that entry with NULL
  - It is now an error to create an instance of the class
    - In our example, `ZooProperty zp;` is now an error.
  - It is an error to create an instance of any class the inherits this one, unless that class overrides the method

Pure Virtual Methods (III)

- Pure virtual methods enforce interfaces
- Any class that inherits a class with pure virtual methods must override those methods
- Otherwise, it cannot be instantiated
- Classes with pure virtual methods are therefore interfaces
  - But not all interfaces have pure virtual methods
  - Referred to as Pure Virtual Classes
  - Even though they may have other, implemented methods and data fields

Object Oriented Programming (Review!!!)

1. Encapsulation
   - Collect data & code that operates on that data in one object
   - Provide a single, public interface
     - Changes to the implementation are local
2. Polymorphism (inheritance)
   - Abstraction via “is-a” relation
   - Write code at different levels of abstract
   - Avoid redundant code
3. Inheritance as Union
   - A child class is the union of parent and new field/methods
     - Slicing: an unfortunate combination of these

Initialization

- When an object is made, its data fields are initialized
  - Primitive values are undefined
  - Object fields are initialized
    - Using the arguments provided in the constructor list
    - That thing that begins with :
  - Using default arguments if not initialized in the constructor list
  - More specifically, they are initialized in the order they are declared.
  - Initialization precedes entering the body of the constructor method

Inheritance as Union

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

This address can be Animal*, Mammal*, Equine*, Quagga*
Polymorphism doesn’t have to change the address, just the (compile-time) type

Animal:
  lifespan (int)
Mammal:
  body_temp (int)
Equine:
  mane_length (int)
  tail_length (double)
Quagga:
  Stripe_count (int)
Inheritance & Initialization

• Inheritance is union

• Therefore, parents are initialized before entering the child object's constructor
  – Initializing the parent is recursive
  – The parents' fields are initialized
  – Then its constructor is called
  – Of course, if the parent has a parent...

Inheritance & Constructors

• Assume Quagga inherits Equine, which inherits Mammal, which inherits Animal
  – Animal's fields are initialized
  – Animal's constructor is called
  – Mammal's fields are initialized
  – Mammal's constructor is called
  – ...

• Constructors are called in order, parent to child

Inheritance & Destructors

• Destructors are called in the opposite order
  – Quagga's destructor
  – Equine's destructor
  – Mammal's destructor
  – Animal's destructor

• Destructors are called in reverse order, child to parent

Collections in C++ (Review)

• Arrays, vectors, etc. collect like data items
  – We call them containers.
  – Most containers are objects
  – But C-style arrays are not

• In C++, containers can hold anything
  – vector<Mammal> is a vector of Mammals
  – vector<Mammal&> is a vector of references to Mammals
  – vector<Mammal*> is a vector of pointers to Mammals

• Java only supports containers of references

Implications of Storage Types

• Vector<Mammal>
  – Memory is contiguous
    • Fewer cache misses, more efficient access
  – Memory used: size() * sizeof(Mammal)
  – Push_back copies data
    • Constructor costs
    • No side effects

• Vector<Mammal&>
  – Mammals may be spread across memory
    • More cache misses
  – Memory: size() * (sizeof(Mammal) + sizeof(Mammal&))
  – Push_back does not copy data
    • No constructor costs
    • Potential side effects
    • Errors if the source falls out of scope...

• Vector<Mammal*> -- like vector<Mammal&> for these purposes
Polymorphism + Storage

- Polymorphism adds a wrinkle

```cpp
class Animal {
public:
  virtual string Type() const {return "Animal";} 
};

class Mammal : public Animal {
public:
  Mammal(int span) : lifespan(span) {}
  virtual string Type() const {return "Mammal";} 
  int Lifespan() const {return lifespan; 
  int lifespan;
};
```

What happens when...

```cpp
vector<Animal> avec;
vector<Animal&> aref_vec;
mammal m(30);
avec.push_back(m);
aref_vec.push_back(m);
cout << avec[0].Type() << " " 
  << aref_vec[0].Type() << endl;
```

... on the previous slide

- What is printed out?
  - Animal Mammal
- What copy constructor was called?
  - Animal(const Animal& a);
- How many times was it called?
  - Once
- Why did avec[0] print 'Animal'?
  - Because avec[0] was created by push_back as an Animal
- Why did aref_vec[0] print 'Mammal'?
  - You are printing from the original, not the copy
    - The original was made as a Mammal

Slicing

- What is the size (in memory) of avec[0]?
  - The size of a pointer (for the VFPT)
- What happened to the lifespan of 30?
  - It was “sliced” away
  - Really, the Animal copy constructor just never copied it