Introduction to Inheritance

CS 250 - Session 7
C++ Programming

Topics Covered

- Reading: 567-603
- What is inheritance? Why use it?
- Class relationships:
  - Is-a, has-a, uses-a, and implemented-as
- Syntax for basic public inheritance
- Pointers to base classes and polymorphism
- Virtual functions and dynamic binding
- Homework 11 & 12

Why Use Inheritance?

- 1) To create new, specialized classes
  - Reuse existing code. Don’t need source code
  - Vendors can distribute customizable libraries
  - Bugs fixed in base class replicated in derived
    » Bugs introduced in base replicated as well
- 2) To create generalized classes [Is-a]
- 3) To create customizable frameworks
  - Examples: MFC, MacApp
Methods of Code Reuse I

- Method 1: Direct Manipulation
  - You have a Box class that draws shapes
    - Box a(1, 1, 10, 10);
    - a.draw();
  - Now you want to add support for color
    - 1.) Get source code
    - 2.) Add bgColor and fgColor fields
    - 3.) Change constructors & methods

Methods of Code Reuse II

- Problems with direct manipulation
  - You must have access to the source code
  - You may (?) introduce new bugs
  - Programs that use Box must be rewritten
  - Can mitigate last problem by using default args
    - Box::Box(int, int, int, int, // org args
               int=BLACK,
               int=WHITE);

Methods of Code Reuse III

- Method 2: Clone & Change
  - Most common method of code reuse
  - Copy class, change name, add new features
  - Box copied to ColorBox
  - Changes won't affect existing programs
  - Only new users will have to change programs
  - Disadvantages?
    - Now maintaining two versions of Box class
    - Keeping in synch is a nightmare [Lava flows]
    - Still need access to existing source code
Methods of Code Reuse IV

- **Method 3: Inheritance**
  - Use existing box as *base class*
    - In other OOP languages, called *superclass*
    - Many prefer to think of it as a *parent class*
  - Add new fields and methods to *derived class*
    - In other OOP languages, called *subclass*
    - May prefer to think of it as a *child class*

Methods of Code Reuse V

- **Deriving a ColorBox**

  Data
  - x1, y1, x2, y2

  Methods
  - Constructors
  - SetX(), GetX()
  - Draw()

Inherited Fields

Additional Fields

New Constructors

Inherited Methods

Additional Methods

Overridden Methods

Inheritance and Generalization

- **Inheritance allows "class generalization"**
  - Suppose you have several functions that all contain the same code sequence
  - You "factor" the sequence into a function
  - This is "code generalization"

- **Inheritance does this with classes**
  - Common attributes and methods can be shared in a related "family" of classes
  - Called a "class hierarchy" [MFC as an example]
Inheritance is a technique for implementing particular kinds of class relationships.

Different kinds of class relationships:
- The uses-a relationship [no inheritance]
- The has-a relationship [no inheritance]
- The is-a relationship [public inheritance]
- The implemented-as relationship [private inheritance]

The uses-a relationship
- Occurs when your class asks an object of another class to carry out a task
  - Asking the cout and cin objects to perform I/O

The has-a relationship [composition]
- Occurs when your class contains data members which are objects of another class
  - A Label has a String, Color, Font, etc.

The is-a relationship
- Occurs when one class is a "kind of" another
  - Ostrich is a Bird is an Animal
  - A Label is a Component, so is a Button
- In C++, IS A is implemented by public inheritance like this:

```cpp
class Bird : public Animal
{
    // Code & data for Bird class goes here
};
```
Class Relationships III

The implemented-as relationship
- When one class uses the implementation of another, but not the interface
- Example: creating Stack from Array
- A stack is not an array
  » You should only access top of a stack
  » You can access any element in an array
- Implemented in C++ by private inheritance
  - class Stack : private Array {}
Basic Inheritance III

- **What is accessible?**
  - Members `x` and `y` can be used in any member function in the class (print(), setX(), etc.)
  - Cannot be used through a Point object

  ```cpp
  Point p1(100, 200);
  p1.x = 200;       // Error
  p1.setX(200);    // OK
  ```

  - Must use interface to access private members

Basic Inheritance IV

- **To create the Point3D class** we: [Point02.cpp]
  - Use the keyword `class`
  - The name of the derived class
  - A single colon
  - An optional derivation type (public, etc.)
  - A Point3D is a Point so we use `public`
  - The name of the base (parent) class
  - The remainder of the class definition

```cpp
class Point3D : public Point
{
    int z;
    public:
    int getZ() const { return z; }
    void setZ(int z_) { z = z_; }
    void print() const
    { cout << x << ", " << y << ", " << z; }
};
```
Using Inherited Members

- Why? Members x and y are private
  - Child class has no direct access to private members
    - Private members are inherited, however
  - Child class has direct access to public members
- Solution 1? Write print() like this:
  ```cpp
  void print() const
  {
    cout << getX() << ", " << getY() << " ", " << z;
  }
  ```
  - getX() and getY() are inherited by Point3D

Using Hidden Members I

- Solution 2: Let the parent do the work [Point3D.cpp]
  - Point::print() already prints x,y.
  - Can we call the parent function directly?
  ```cpp
  void print() const
  {
    print(); cout << " ", " << z;
  }
  ```
  - Unfortunately, this blows up when run
    - print() in Point3D hides print() in Point
    - print(); refers to Point3D::print()

Using Hidden Members II

- Solution? Use scope resolution operator
  ```cpp
  void print() const
  {
    Point::print();
    cout << ", " << z;
  }
  ```
- In C++, names overload only in same scope
  - Overloaded functions in same class is OK
  - Same name in derived class hides all inherited funcs
Overloading and Hiding

- The Bird classes

```cpp
class Bird {
public:
    Speak() { }
    Speak(String) { }
};
class Crow:public Bird {};
class Duck:public Bird {
public:
    Speak() { };
};
```

- What happens here?
  - Bird b,
    Crow c,
    Duck d;
  - b.speak();
  - b.speak("Hi");
  - c.speak();
  - c.speak("Hi");
  - d.speak();
  - d.speak("Hi");

Protected Access

- Protected access is between public and private
  - Child [derived] classes are allowed direct access
  - Other classes are not
  - We could rewrite Point class like this:

```cpp
public class Point {
    protected:
        int x, y; // Point3D can directly access
};
```

- In general, better to avoid protected access

Using Point3D

- The Point3D class has no constructor
  - What happens when we do this?
    - Point3D p3;
      p3.print();
  - Members x and y have value 0, z undefined
  - Calls the default Point [no arg] constructor
  - Can we do this? Point3D p3(10, 20);
  - No. Constructors are not inherited
Point3D Constructors I

- We can write a default constructor like this:
  ```
  Point3D() : z(0) { }
  ```
- Default Point constructor called automatically
- Better solution is to write a working constructor
  - Provide default values like Point constructor
  - Need to provide values to pass to parent

Point3D Constructors II

- A Point3D working constructor: [Point04.cpp]
  ```
  // Defined inside the Point3D class
  Point3D (int x_, int y_, int z_)
  : Point(x_, y_), z(z_)
  {} 
  ```
- Notes
  - Call Point constructor in initialization list
  - If you do not, default Point constructor called
  - Cannot directly initialize x and y
  
Construction Order I

- Suppose you create a Widget like this:
  ```
  Widget w(12734, "Blue", 3.45);
  ```
- Constructors will be called in following order
  - Look for Widget(int, char *, double) in Widget
    - Will not look in base class if not found
  - Initialization list searched for base constructor
    - If not found, then default base constructor called
  - Initialization list searched for member constructors
  - Body of constructor is executed
Construction Order II

- Here's the Widget class
  ```cpp
  class Widget : public Thing {
    string name;
    public:
    Widget(int, char double);
  };
  ```

- What happens here?
  ```cpp
  Widget::Widget(int n, char * s, double a)
  { name = s; }
  ```

Construction Order III

- What happens here?
  ```cpp
  Widget::Widget(int n, char * s, double a)
  : Thing(n, a)
  { name = s; }
  ```

- What happens here?
  ```cpp
  Widget::Widget(int n, char * s, double a)
  : Thing(n, a), name(s)
  {}
  ```

Order of Destruction

- A hierarchy may be many levels deep
  - Highest constructor called first
  - Lowest destructor called first

- If you create an Elephant
  Animal(), Mammal(), Elephant()

- When Elephant is destroyed
  - Elephant(), -Mammal(), -Animal()
A Class Hierarchy

Suppose you have the following classes
- Pet, Cat, Dog
- Every Cat, Dog is-a Pet
- Cat inherits 2 methods, overrides 1
- Dog inherits 1, overrides 2

Pet
- ID: const int
- void speak();
- void license();
- int getId();

Cat
- void speak();

Dog
- void speak();
- void license();

Static Polymorphism

What happens here? [pet01.cpp]
- Pet p; Dog d; Cat c;
- p.speak(); d.speak(); c.speak();
- Each object responds appropriately
  - Compiler looks at type of p, d, c
  - Calls method based upon that type
  - Type must be known at compile-time
  - Called static or early binding

References and Pointers

Static binding can apply to pointers and references as well [pet01.cpp]
- Create a Dog * and a Cat &
  - // A Dog and a Dog pointer
  - Dog d, *dp = &d;
  - d.speak(); dp->speak();
  - // A Cat and a Cat reference
  - Cat c, crf = c;
  - c.speak(); crf.speak();
- Type of pointer used to decide what to call
We'd like to write general purpose functions
– Instead of functions like this: [pets03.cpp]

```
void showCat(Cat c) { c.speak(); c.license(); }
void showDog(Dog d) { d.speak(); d.license(); }
```

– Why can't we do this?

```
void show(Pet p) { p.speak(); p.license(); }
void show(Pet *p) { p->speak(); p->license(); }
```

– After all, every Dog is-a Pet, right?

Objects and Functions II

Here's what happens when you pass a Dog object to the function show(Pet p)

```
Pet c;    // Calls Cat::show()
Pet * p = &c;  // Calls Pet::show()
```

Pointers and Derived Objects

When you pass a pointer or reference, however, **no slicing** occurs
– Function calls are still statically bound, though
– Compiler needs to know what function to call
– Example: [pets04.cpp]

```
Pet * p = &c;
c.show(); // Calls Cat::show()
p->show(); // Calls Pet::show()
```
Virtual Functions I

- What we need is a way to tell the function that the Pet * p actually points to a Cat object
  - Means we must wait until runtime to decide which function should be called
  - This is called late binding
- C++ allows you to use both binding styles
  - Other languages [Java, etc.] only allow one
  - Static binding is a little more efficient
  - Late binding uses indirection to call function

Virtual Functions II

- To use late binding, put the keyword virtual in front of the function in the base class
  - In class Pet: [pets02.h]
    - virtual void speak() ...
    - virtual void license() ...
  - Optionally may be repeated in derived class
  - Cannot make a non-virtual function virtual
  - Cannot make a virtual function "unvirtual"

Virtual Functions III

- Use a virtual function through a base pointer
  - Example: [pets05.cpp]
    - Dog d; Cat c;
    - Pet * p = &d;
p->speak();
p = &c;
c->speak();
  - Compiler uses pointer’s dynamic type to decide exactly which function to call
Virtual Functions IV

- How virtual functions work

![Diagram of virtual function mechanisms for Dog and Cat objects]

Virtual Functions and Pointers

- Virtual functions are only virtual when called through a pointer or reference
  - Some examples: [pets06.cpp]
  
  ```cpp
  Cat c; c.speak(); // Static binding
  Pet p = c; p.speak(); // Pet::speak()
  Pet *p = &c; p->speak(); // Cat::speak()
  ```

  - Functions must take a reference or pointer

  ```cpp
  void speak(Pet p) { p.speak(); } // No
  void speak(Pet *pp) { pp->speak(); } // OK
  ```

Homework

- Due Sunday, March 21 [Grade early Monday AM]
  - Homework 11: BigInt Arithmetic
    - Add addition & subtraction to BigInt class
    - Use hmwk11.cpp & BigInt02.h, change BigInt02.cpp
  - Homework 13: Simple Inheritance
    - Create one base class Person
      * Data members: char *name, unsigned long ssn
      * Member functions: constructor, destructor, print()
    - HourlyEmployee derived class
    - SalariedEmployee derived class