
NJIT
THE EDGE IN KNOWLEDGE

CS 280
Programming Language Concepts

Fall 2022

Inheritance and Polymorphism

Introduction

- Inheritance
  - Software reusability
  - Create new class from existing class
  - Derived class inherits from base class
    - Derived class
      - More specialized group of objects
      - Behaviors inherited from base class and can be customized
      - Additional behaviors

Introduction

- Abstraction
  - Focus on commonalities among objects in system
- “is-a” vs. “has-a”
  - “is-a”
    - Inheritance
    - Derived class object treated as base class object
    - Example: Car *is a* vehicle
      - Vehicle properties/behaviors also car properties/behaviors
  - “has-a”
    - Composition
    - Object contains one or more objects of other classes as members
    - Example: Car *has a* steering wheel

Base Classes and Derived Classes

- Base classes and derived classes
  - Object of one class "is an" object of another class
    - Example: Rectangle is quadrilateral.
      - Class **`Rectangle`** inherits from class **`Quadrilateral`**
      - **`Quadrilateral`**: base class
      - **`Rectangle`**: derived class
  - Base class typically represents larger set of objects than derived classes
    - Example:
      - Base class: **`Vehicle`**
        - Cars, trucks, boats, bicycles, …
      - Derived class: **`Car`**
        - Smaller, more-specific subset of vehicles

Base Classes and Derived Classes

- Inheritance hierarchy
  - Inheritance relationships: tree-like hierarchy structure
  - Each class becomes
    - Base class
      - Supply data/behaviors to other classes

    OR
    - Derived class
      - Inherit data/behaviors from other classes

Base Classes and Derived Classes

- **`public`** inheritance
  - Specify with:

    `Class TwoDimensionalShape : public Shape`
    - Class **`TwoDimensionalShape`** inherits from class **`Shape`**
  - Base class **`private`** members
    - Not accessible directly
    - Still inherited
      - Manipulate through inherited member functions
  - Base class **`public`** and **`protected`** members
    - Inherited with original member access
  - **`friend`** functions
    - Not inherited

`protected` Members

- **`protected`** access
  - Intermediate level of protection between **`public`** and **`private`**
  - **`protected`** members accessible to
    - Base class members
    - Base class **`friend`**s
    - Derived class members
    - Derived class **`friend`**s
  - Derived-class members
    - Refer to **`public`** and **`protected`** members of base class
      - Simply use member names

Constructors and Destructors in Derived Classes

- Instantiating derived-class object
  - Chain of constructor calls
    - Derived-class constructor invokes base class constructor
      - Implicitly or explicitly
    - Base of inheritance hierarchy
      - Last constructor called in chain
      - First constructor body to finish executing
      - Example: **`Shape`**/**`Circle`**/**`Cylinder`** hierarchy
        - **`Shape`** constructor called last
        - **`Shape`** constructor body finishes execution first
    - Initializing data members
      - Each base-class constructor initializes data members
        - Inherited by derived class

Constructors and Destructors in Derived Classes

- Destroying derived-class object
  - Chain of destructor calls
    - Reverse order of constructor chain
    - Destructor of derived-class called first
    - Destructor of next base class up hierarchy next
      - Continue up hierarchy until final base reached
        - After final base-class destructor, object removed from memory

Inheritance Example

```
class Zoo {
  int x;
public:
  Zoo(int x) : x(x){}
  int getX() {return x; }
};
```

- Declare Zoo zz(3);
- Call zz.getX()
- Method will return 3 (zz's value for x) that was set by the constructor for zz.

Inheritance Example

- Any children of Zoo will inherit from Zoo.
- Therefore the getX() method is valid for all subclasses/children of Zoo.
- In this example, cityZoo cc(3,5) will return 5 from a call to cc.getX(), because x is set to 5 by the constructor call to Zoo

```
class cityZoo : public Zoo {
  int y;
Public:
  cityZoo(int y, int x) : Zoo(x), y(y) {}
};
```

Inheritance: Redefinition Example

```
class Zoo {
  int x;
public:
  Zoo(int x) : x(x){}
  int getX() { return x; }
};
class cityZoo : public Zoo {
  int y
Public:
  cityZoo(int y, int x): Zoo(x), y(y) {}
  int getX() { return y; }
};
```

```
Declaring
Zoo v1(5);
cityZoo v2(7,5);

Causes
v1.getX() to return 5
v2.getX() to return 7
```

Pointers and Inheritance

- Because of the rules of inheritance, a `cityZoo` "`is-a`" `Zoo`
- Therefore a pointer to a `Zoo` can point to a `cityZoo`

```
Zoo* ptr1 = &v1;
Zoo* ptr2 = &v2;

ptr1->getX() will be 5
ptr2->getX() will be 5
```

```
Declaring
Zoo v1(5);
cityZoo v2(7,5);

Causes
v1.getX() to return 5
v2.getX() to return 7
```

Pointers and Inheritance

- What happened to the getX in cityZoo? Why isn't it being called?
  - Because ptr1 and ptr2 are pointers to Zoo, so they will use Zoo's methods.
  - Saying

```
cityZoo * cZ= &v2; cZ ->getX();
```

    - will call the getX() method… because `cZ` points to a `cityZoo`
- However, the required behavior is:
  - If the Zoo pointer really points at a Zoo object, then I want to call the Zoo method… but if the Zoo pointer actually points to a cityZoo object, I want to call the cityZoo method.
  - This is called "Dynamic Binding", where we figure out, at runtime, what our type is and which method to call.

Pointers and Inheritance

- pointer
  - Base-pointer can aim at derived-object
    - But can only call base-class functions
  - Calling derived-class functions is a compiler error
    - Functions not defined in base-class
- Common theme
  - Data type of pointer/reference determines the functions it can call.
- Typically, pointer-class determines functions
- **`virtual`** functions
  - The virtual keyword turns on dynamic binding.
  - Object (not pointer) class determines function called.

Virtual Methods

```
class Zoo {
  int x;
public:
  Zoo(int x) : x(x){}
  virtual int getX() {
  return x;
 }
};
class cityZoo : public Zoo {
  int y
Public:
 cityZoo (int y, int x): Zoo(x), y(y) {}
 int getX() { return y; }
};
```

```
Declaring
Zoo v1(5);
cityZoo v2(7,5);
Zoo *p1 = &v1;
Zoo *p2 = &v2;

Virtual causes dynamic
binding
p1->getX() will return 5
P2->getX() will return 7
```

Polymorphism

- Polymorphism
  - "Program in the general"
  - Treat objects in same class hierarchy as if all base class
    - Derived-class object can be treated as base-class object
  - Virtual functions activate dynamic binding
  - Makes programs extensible
    - New classes added easily, can still be processed
  - Dynamic binding
    - Choose proper function to call at run time
    - Only occurs off pointer handles
      - If function called from object, uses that object's definition

Polymorphism

- Aim pointers (base, derived) at objects (base, derived)
  - Base pointer aimed at base object
  - Derived pointer aimed at derived object
    - Both straightforward
  - Base pointer aimed at derived object
    - "is a" relationship
      - **`cityZoo`** "is a" **`Zoo`**
    - Will invoke base class functions
  - Function call depends on the class of the pointer
    - Does not depend on object to which it points
    - With **`virtual`** functions, this can be changed

Abstract Classes

- Abstract classes
  - Main purpose: to be a base class (called abstract base classes)
  - Incomplete
    - Derived classes fill in "missing pieces"
  - Cannot make objects from abstract class
    - However, can have pointers and references
- Concrete classes
  - Can instantiate objects
  - Implement all functions they define
  - Provide specifics

Abstract Classes

- To make a class abstract
  - Need one or more "pure" virtual functions
    - Declare function with initializer of 0

```
virtual void draw() const = 0;
```

  - **Regular virtual functions**
    - **Have implementations, overriding is optional**
  - **Pure virtual functions**
    - **No implementation, must be overridden**
  - **Abstract classes can have data and concrete functions**
    - **Required to have one or more pure virtual functions**

Abstract Classes

- Abstract base class pointers
  - Useful for polymorphism
- Application example
  - Abstract class **`Shape`**
    - Defines **`draw`** as pure virtual function
  - **`Circle`**, **`Triangle`**, **`Rectangle`** derived from **`Shape`**
    - Each must implement **`draw`**
  - Screen manager client knows that each object can draw itself

Example

```
class Shape {
    protected:
        const double PI = 3.14159;
    public:
        // virtual function that returns shape area
        virtual double getArea() const;
        // virtual function that returns shape perimeter
        virtual double getPerimeter() const;
        // pure virtual functions; overridden in derived classes
        virtual string getName() const = 0; // return shape name
        virtual void print() const = 0;     // output shape

    }; // end class Shape
```

Example (cont'd)

```
class Circle : public Shape {
  public:
    Circle( double rad = 0.0 ); // default constructor

    void setRadius( double );   // set radius
    double getRadius() const;   // return radius
    double getDiameter() const;        // return diameter
    double getCircumference() const;  // return circumference
    virtual double getArea() const;   // return area
    string getName() const;
    void print() const;  // output Circle object

  private:
       double radius;  // Circle's radius
}; // end class Circle
```

Example (cont'd)

```
class Rectangle : public Shape
{
    private:
                double length, width;  // length and width in feet

    public:
     Rectangle(double l, double w);

    double getArea();
    double getPerimeter();
    void print() const;  // output Rectangle object
}
```

```
int main(){
  vector< Shape * > shapeVector( 4 );
  Circle circle1( 3.5 ), circle2(7.5); // create a Circle
  Rectangle rectangle1( 3.3, 6.4 ), rectangle2(4.2, 8.3);
  shapeVector[ 0 ] = &circle1;
  shapeVector[ 1 ] = &rectangle1;
  shapeVector[ 2 ] = &circle2;
  shapeVector[ 3 ] = &rectangle2;
  for ( int i = 0; i < shapeVector.size(); i++ ) {
    string shapeName = shapeVector[i]->getName() ;
    shapeVector[i]->print();
    cout << "Area is " << shapeVector[i]->getArea() << endl;
    if (shapeName == "Circle")
       cout << "Circumference is"<<shapeVector[i]->getPerimeter()
    << endl;
    else if (shapeName == "Rectangle")
       cout << "Perimeter is " << shapeVector[i]->getPerimeter()
    << endl;
    cout << endl;
  }
  return 0;}
```

Recitation Assignment 9: Story Books

- A Publishing House publishes stories in three categories and has strict requirements for page counts in each one. Create an **abstract class** called *Story*, which has the following data members: story title (title), author name (author), number of pages (pages), type of story book (type), and a string message (msg). Include get and set member functions for each field. Follow the naming convention for the setter and getter functions for each data member, such as `setTilte(), getTitle()`, etc. The Story abstract class has a constructor that accepts the type of story book as a parameter to initialize its type field. The type of story books considered are Novel, Novella and Short Story. The function that sets the number of pages (setPages()) is abstract. The `setPages()` function accepts one parameter for the number of words per page.

Recitation Assignment 9 : Story Books

- Create three *Story* subclasses called *Novel*, *Novella*, and *ShortStory*. The three child classes represent three types of stories, each with a specific page requirement. Each subclass has extra data members as constants for the page lower and upper limits in that category, and an extra data member for the number of words in the book. For the *Novel* subclass, the lower limit of pages is 101. For the *Novella* subclass, the lower limit is 51 pages and the upper limit is 100. For the *ShortStory*, the lower limit is 5 and the upper limit is 50 pages. Include a constructor for each subclass that may be passed two parameters for initializing the number of words in the book and the type of story book; and include get and set member functions for the subclass data member. The `setPages()` member function should compute the possible number of pages in the book based on the number of words and the number of words per page passed as a parameter. The function should check whether the candidate book needs to be cut by a certain number of pages, or added certain number of pages to qualify for the type of story requirement. Otherwise, the book would be accepted. Accordingly, the function sets the msg data member with the required message to be used for display. See the example below for the format and contents of the messages.

Recitation Assignment 9 : Story Books

- Implement each class in a separate header file that carries the class's name, such as: "Story.h", "Novel.h", etc.

- The driver program "RA9prog.cpp" will be used to test your implementations. The program reads book information from a file and displays the list of candidate books along with possible messages to adjust the book to the book category requirement. An example of the expected output is shown below.

Test Case

```
List of candidate story books to be published:
1. Title: Goodbye Columbus
Author:  Philip Roth
Pages: 47
Story Category: Novella
Pages must be added to the book to satisfy the requirements of a Novella: 4

2. Title: The Pit and the Pendulum
Author:  Edgar Allan Poe
Pages: 31
Story Category: Short Story
Book is accepted.
.        .        .        .
4. Title: Of Mice and Men
Author:  John Steinbeck
Pages: 107
Story Category: Novella
Pages must be cut from the book to satisfy the requirements of a Novella: 7

5. Title: Breakfast at Tiffany's
Author:  Truman Capote
Pages: 120
Story Category: Novel
Book is accepted.
```


NJIT
THE EDGE IN KNOWLEDGE