Polymorphism

Inheritance for Polymorphism

- Casting type to base class
- Virtual Methods
- Pure Virtual Methods

Type Casting

- Forced type conversion
- Used here to make an object of a derived class look like it belongs to base class
- Syntax: static_cast<BaseClass> (variablename)
- C-style syntax is deprecated

Storage for Classes

<table>
<thead>
<tr>
<th>Person</th>
<th>Employee</th>
<th>Staff</th>
</tr>
</thead>
<tbody>
<tr>
<td>_name</td>
<td>_name</td>
<td>_name</td>
</tr>
<tr>
<td>_address</td>
<td>_address</td>
<td>_address</td>
</tr>
<tr>
<td>_dept</td>
<td>_dept</td>
<td>_dept</td>
</tr>
<tr>
<td></td>
<td>_wage</td>
<td></td>
</tr>
</tbody>
</table>

Assigning Derived to Base

Employee ellen(...);
Person p = ellen;

Only fields of Person copied
- _name
- _address
- _dept

Rest is lost

Casting Type to Base Class

Use pointers (or references) to objects

Employee* e = new Employee(e_person, e_dept);
Person* p = static_cast<Person*>(e);

Pointer p is now an alias for pointer e

However,

e->getDept(); // OK, e is an Employee*
p->getDept(); // Error, p is a Person*
Polymorphism via Casting

bool Search(const Name& f_nme, Person** a, int dim) {
  int mid = (dim - 1)/2;
  int left = 0; int right = dim - 1;
  while (((a[mid] -> getName()) != f_nme) && (left < right)) {
    if (a[mid] -> getName() < f_nme) {
      right = mid; mid = (mid - left)/2;
    } else {
      left = mid; mid = (right - mid)/2;
    }
  }
  return (a[mid] -> getName()) != f_nme;
}

Polymorphism via Casting (2)

Person* a[4];
Professor* p = new Professor(e1, sal1);
Student* s = new Student(snm, sadd, mjr, ...);
Staff* st1 = new Staff(e2, wage2);
Staff* st2 = new Staff(e3, wage3);
a[0] = static_cast<Person*>(p);
a[1] = static_cast<Person*>(s);
a[2] = static_cast<Person*>(st1);
a[3] = static_cast<Person*>(st2);
search(name, a, 4);

Notes on Poly via Casting

- Array can hold any derived class of Person
- Search uses getName() method of Person
- Polymorphism limited to Person hierarchy
- Cast makes compiler view derived object as a Person object
  - If Staff had replaced getName() method, the Person method would still be used
  - Casting does not allow us to use methods particular to derived classes

Virtual methods

- Methods can be declared as virtual
- Sets up dynamic binding mechanism
  - Use pointer for base class to point to object of derived class (like before)
  - Method call for virtual method is dynamically bound to method of derived class
- Methods declared as virtual in base class are virtual for all derived classes

Example: Rectangles

- Classes to represent rectangles in a graphics program
  - Plain rectangles – display as lines
  - Filled rectangles – display with fill color
- Make (unfilled) Rectangle the base class

Rectangle without Virtual

class Rectangle {
  public:
    Rectangle(Location, Location);
    
    // Draws an empty rectangle
    void draw(Canvas&) const;
}; ...
Filled Rectangle without Virtual

class FilledRect : public Rectangle {
    public:
        FilledRect(Location, Location, Color);
        // Draws a filled rectangle
        void draw(Canvas&) const;
        ...
    };

Using Rectangle

Rectangle* a[2];
Rectangle* plain = new Rectangle(loc1, loc2);
FilledRect* red = new FilledRect(loc3, loc4, Color::red);
a[0] = plain;
a[1] = static_cast<Rectangle*> red;
a[0]->draw(windowcanv);
a[1]->draw(windowcanv);

Display of Rectangles

Problem

• Problem: Method Rectangle::draw doesn’t know about fill colors
• Solution: make draw a virtual method

Non-virtual Functions

Rectangle
    void draw()

    void Rectangle::draw() {
        
    }

FilledRect
    void draw()

    void FilledRect::draw() {
        
    }

Rectangle with Virtual

class Rectangle {
    public:
        Rectangle(Location, Location);
        // Default is to draws an empty rectangle
        virtual void draw(Canvas&) const;
        ...
    };

Note: Function definition doesn’t need virtual modifier.
Filled Rectangle with Virtual

class FilledRect : public Rectangle {
    public:
        FilledRect (Location, Location, Color);
        // Draws a filled rectangle
        void draw(Canvas&) const; ...
    }

Note: method is virtual in all derived classes; modifier not needed

Using Rectangle

Rectangle* a[2];
Rectangle* plain = new Rectangle(loc1, loc2);
FilledRect* red = new FilledRect (loc3, loc4, Color::red);
a[0] = plain;
a[1] = static_cast<Rectangle*> red;
a[0]->draw(windowcanv);
a[1]->draw(windowcanv);

Display of Rectangles (2)

Virtual Functions (1)

Rectangle
virtual void draw() { }

FilledRect
void draw() { }

Virtual Functions (2)

Virtual Destructors

- Destructors of base classes should be declared virtual
- Ensures that they will be called when derived objects are destructed
Implicit Dynamic Binding

```cpp
class Rectangle {
public:
    virtual void draw(Canvas&) const;
    void resize(int width, int height, Canvas& c);
};
```

```cpp
void Rectangle::resize(int width, int height, Canvas& c) {
    // code to change lower right location
    draw(c); // really this->draw(c);
}
```

Implicit Dynamic Binding (cont)

For derived classes only need to redefine `draw`

```cpp
class FilledRect : public Rectangle {
public:
    virtual void draw(Canvas&) const;
};
```

If applied to `FilledRect` object, `resize(...)` will use `FilledRect::draw`

Recognizing Dynamic Binding

- Nonvirtual methods always statically bound
- Virtual methods
  - Static when
    - Applied to object: `rect.draw(canv);`
    - Class explicitly named: `Rectangle::draw(canv)`
  - Dynamic when applied to
    - Pointer (see rectangle example)
    - Implicit object (the "this" pointer)

Virtual Methods and Changes

- Virtual methods help with extensions
- Ex. Adding a “labeled” rectangle that contains a text label
- Only need to define new derived class
- No existing code needs to be changed for new class to be usable

Type Casting (again)

- Widening - convert from derived to base class
  - Always safe
  - Use `static_cast<>`
- Narrowing
  - Converting type from base to derived class
  - Requires run-time type check
  - Use `dynamic_cast<>`
  - Class must have at least one virtual method

Dynamic Casting

- Dynamic cast incorporates a type check
- Can write code like

  ```cpp
  Rectangle* s;
  ...
  FilledRect* r = dynamic_cast<FilledRect*>(s);
  if (r != NULL) {
    // r equals s, points to FilledRect object
  } else {
    // s points to object that is not a FilledRect
  }
  ```