ECE 220 Computer Systems & Programming

Lecture 24 –Overloading, Inheritance & Polymorphism

ECE ILLINOIS

ILLINOIS

Review: Pass by Value / Address (Pointer) / Reference

Let’s take a look at our most familiar swap example…

Pass by value: void swap_val(int x, int y);

Pass by address: void swap_ptr(int *x, int *y);

Pass by reference: void swap_ref(int &x, int &y);

```
void swap_ref(int &x, int &y){
    int temp = x;
    x = y;
    y  = temp;
}
```

```
int main(){
    int x = 1;
    int y = 2;
    swap_val(x, y);         //pass by value
    swap_ptr(&x, &y);       //pass by address (pointer)
    swap_ref(x, y);         //pass by reference
}
```

***see the ref_vs_ptr.cpp (github)**

More on Reference

- An alias for a variable/object
- Similar to pointer but safer
- No need to dereference, use it just like a variable/object
- Should use "." instead of "->" to access members

Copy constructor and pass by constant reference

```
class Rectangle{
    //default access is private
    int width, height;
    public:
    //copy constructor
    Rectangle(const Rectangle &obj){
    width = obj.width;
    height = obj.height;}
    //other methods omitted here for simplicity
}; copy construct example: copy_constructor2.cpp (github)
```

Operator Overloading

Redefine built-in operators such as +, -, *, <, >, = in C++ to do what you want

```
Example:
class Vector {
   Protected:
   double angle, length;
   public:
   //constructors & other member functions
   …
   vector operator +(const Vector &b) {
       Vector c;
       double ax = length*cos(angle);
       double bx = b.length*cos(b.angle);
       double ay = length*sin(angle);
       double by = b.length*sin(b.angle);
       double cx = ax+bx;
       double cy = ay+by;
       c.length = sqrt(cx*cx+cy*cy);
       c.angle = acos( cx/c.length );
       return c;}
}; Example: test_overloading_L24.cpp (github)
```

```
Vector c(1.5,2);
Vector d(2.6,3);

//before operator overload
Vector e = c.add(d);

//after operator overload
Vector e = c + d;
```

Inheritance & Abstraction

C++ allows us to define a class based on an existing class, and the new class will inherit members of the existing class.

- the **existing** class –
- the **new** class –

A derived class inherits all base class member functions with the following exceptions:

- Constructors, destructors and copy constructors of the base class.
- Overloaded operators of the base class.
- The friend functions of the base class.

```
class orthovector : public vector{
    protected:
    int d; //direction can be 0,1,2,3, indicating r, l, u, d
    public:
    orthovector(int dir, double l){
        const double halfPI = 1.507963268;
        d = dir;
        angle = d*halfPI;
        length = l;
    }
    orthovector() {d = 0; angle = 0.0; length = 0.0;}
    double hypotenuse(orthovector b){
        if((d+b.d)%2 == 0) return length + b.length;
        return (sqrt(length*length + b.length*b.length));
    }
};
```

Access	public	protected	private
Same Class	Y	Y	Y
Derived Class	Y	Y	N
Outside Class	Y	N	N

Polymorphism

- The word **polymorphism** means having many forms. Typically, polymorphism occurs when there is a hierarchy of classes and they are related by inheritance.
- a call to a member function will cause a **different function to be executed** depending on the type of the object that invokes the function

Example:

```
//base class
class Shape{
    protected:
    double width, height;
    public:
    Shape() {width = 1; height = 1;}
    Shape(double a, double b) { width = a; height = b; }
    double area() { cout << “Base class area unknown” << endl;
                    return 0; }
}; Example: polymorphism_simple.cpp (on github)
```

```
int main(){
    Rectangle rec(3,5);
    Triangle tri(4,5);

    rect.area();
    tri.area();

    return 0;
}
```

```
//derived classes
class Rectangle : public Shape{
    public:
    Rectangle(double a, double b) : Shape(a,b){}
    double area() {


    }
};

class Triangle : public Shape{
    public:
    Triangle(double a, double b) : Shape(a,b){}
    double area() {


    }
};
```

Declared Type vs. Actual Type

```
int main(){
        Shape *ptr;
        Rectangle rec(10,7);
        Triangle tri(10,5);

        //use ptr to point to rec object
        ptr = &rec;
        ptr->area();

        //use ptr to point to tri object
        ptr = &tri;
        ptr->area();

        return 0;
}
```

What does this program print?

Virtual Function

- **virtual functions** are member functions in the base class you expect to redefine in the derived classes
- derived class declares instances of that member function

```
//base class
class Shape{
    protected:
    double width, height;
    public:
    Shape() {width = 1; height = 1;}
    Shape(double a, double b) { width = a; height = b; }


};
```