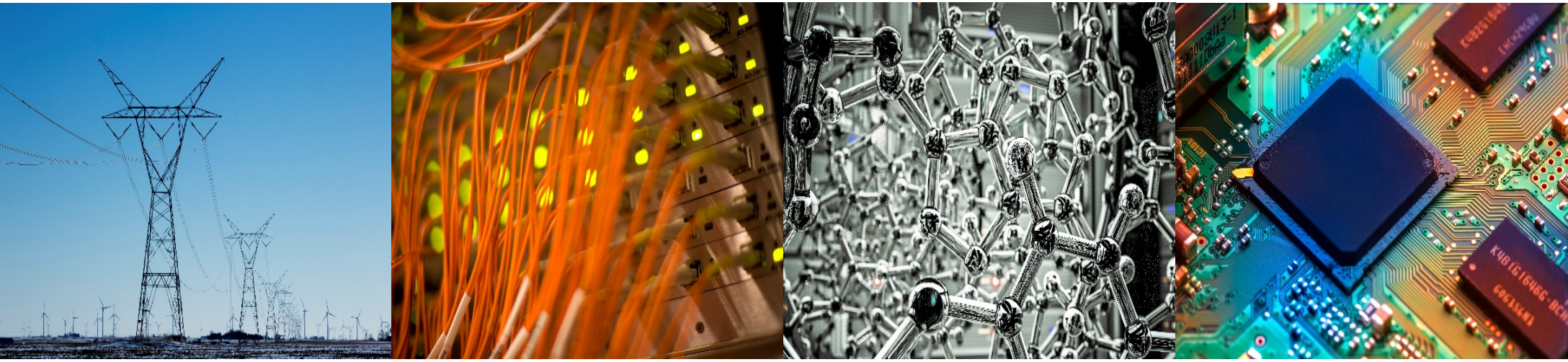
ECE 220 Computer Systems & Programming

Lecture 32 – C++ Inheritance & Polymorphism

July 31, 2020

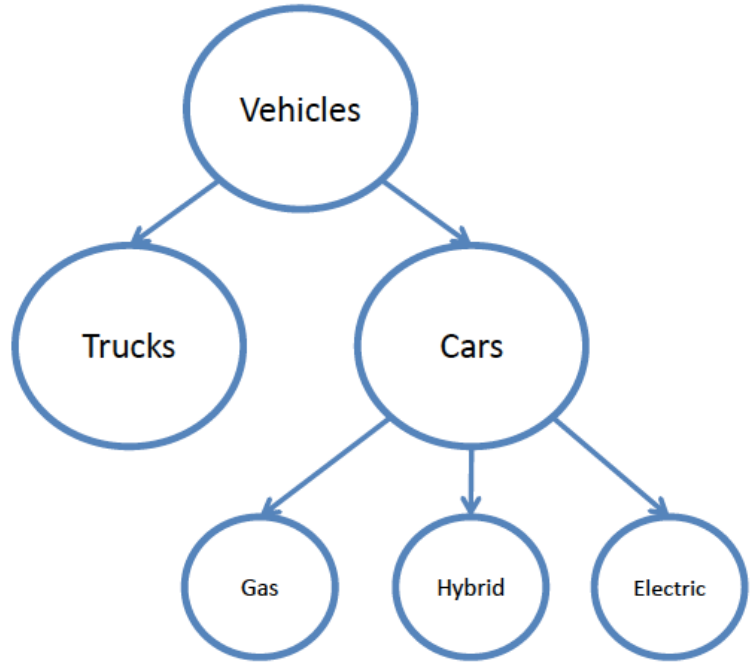


ILLINOIS
Electrical & Computer Engineering
GRAINGER COLLEGE OF ENGINEERING

- Final exam practice questions posted
- Final: 7pm on Friday, August 7th
- Conflict: 10:30am on Saturday, August 8th

Lecture 31 Recap

- C structures + functions + hierarchy = C++ classes



- Object-Oriented Programming features
- Constructor & Destructor
- Basic I/O

Dynamic Memory Allocation

new – operator to *allocate* memory (similar to *malloc* in C)

delete – operator to *deallocate* memory (similar to *free* in C)

Example:

```
int *ptr;
ptr = new int;
delete ptr;

int *ptr;
ptr = new int[10];
delete [] ptr;
```

Exercise – Pointer to an Object

```
int main(){
        Rectangle rect1(3,4);
        Rectangle *r_ptr1 = &rect1;
        //print rect1's area through r_ptr1

        Rectangle *r_ptr2, *r_ptr3;
        r_ptr2 = new Rectangle(5,6);
        //print area of rectangle pointed to by r_ptr2

        r_ptr3 = new Rectangle[2]{Rectangle(),Rectangle(2,4)};
        //print area of the 2 rectangles in the array


        //deallocate memory


        return 0;
}
```

Pass by Value / Address (Pointer) / Reference in C++

Let's take a look at our most familiar *swap* example.

Pass by value:

```
void swap_val(int x, int y);
```

Pass by address (pointer):

```
void swap_ptr(int *x, int *y);
```

Pass by reference:

```
void swap_ref(int &x, int &y);
```

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

```
void swap_ptr(int *x, int *y){
    int temp = *x;
    *x = *y;
    *y  = temp;
}
```

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

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
};
```

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;}
};
```

```
Vector a(1.5,2);
Vector b(2.6,3);

//before operator overload
Vector c = a.add(b);

//after operator overload
Vector c = a + b;
```

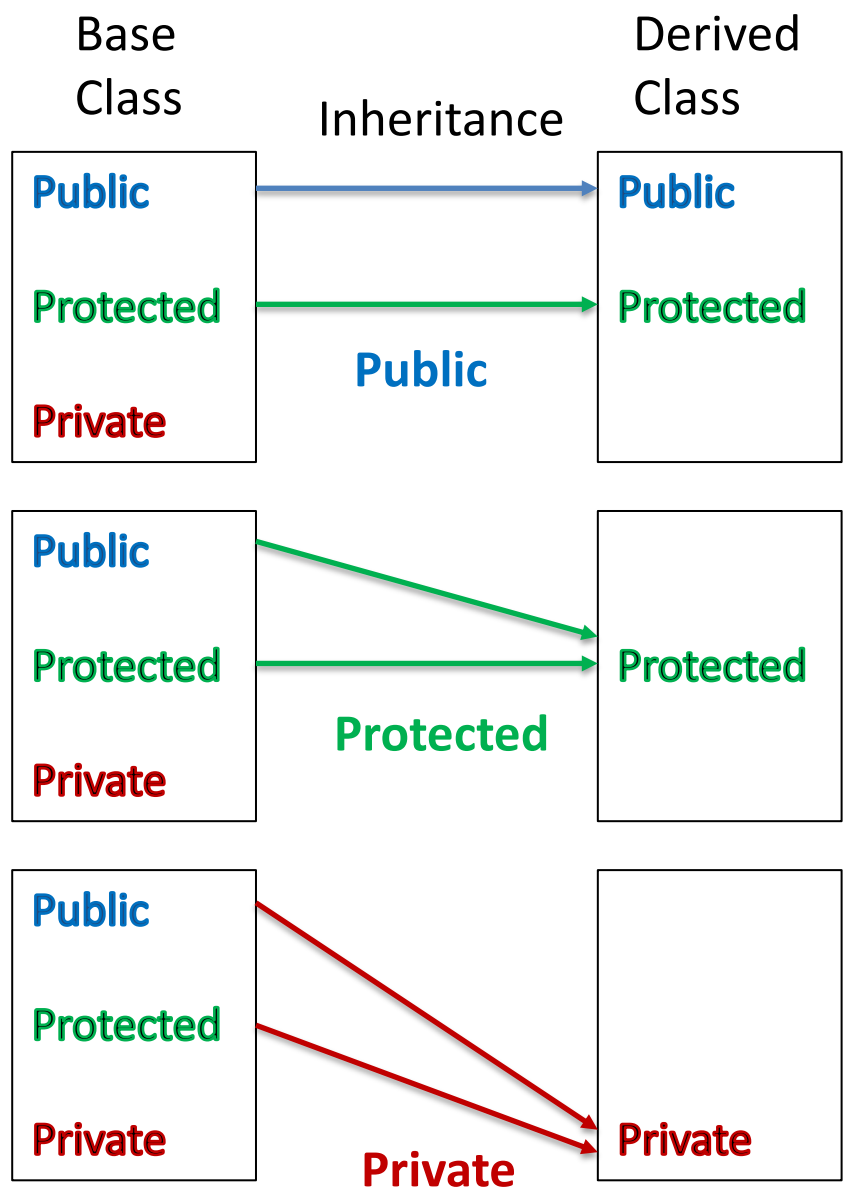
Inheritance & Abstraction

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 –

Exceptions in inheritance (things not inherited):

- 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

- 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; }
};
```

```
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() {


   }
};
```