

cis15
advanced programming techniques, using c++
summer 2008
lecture # IV.1

topics:

- inheritance
- composition of classes

resources:

- Pohl, chapters 8 and 11

an example

- consider the program `patrol.cpp` (posted on the class web page)
- this program models a world in which there is a patrol car and some suspects
- the patrol car wanders around looking for suspects and arrests them
- this is similar to assignment from unit II, where you had a patrol car running around looking for suspects; the main difference is that now the suspects are moving too
- the class definition for the `patrol` class is as follows

```
class patrol {  
private:  
    point location;  
    int arrested;  
  
public:  
    patrol() { arrested = 0; }  
    int getX() const;  
    int getY() const;  
    void set( int x, int y );  
    void print() const;  
    void move();  
    void move( direction d );  
    void arrest();  
    bool busy();  
};
```

composition

- the `patrol` class includes a member of the `point` class, which we have used before (many times!)
- we say that the `patrol` class is related to the `point` class by *composition*
- *composition* means that one class contains a data member that is an *instance* of another class, i.e., a data member that is a variable whose data type is another class
- another example of composition in `patrol.cpp` is that the class `world` contains both `patrol` and `suspect` instances

privacy

- note that several of the function members (methods) of `patrol` look like those for `point`
 - `getX()`
 - `getY()`
 - `set(int x, int y)`
 - `print()`
- these function members provide a way to access the values of the data members of the instance of `point`, which is a data member of `patrol`
- since the data members (`x` and `y`) are `private`, we cannot access them directly in `patrol`
 - we have to refer to them indirectly by using the public function members of `point`

overloading

- the rest of the methods in the `patrol` class give us the functionality we want from `patrol`, allowing it to move, to look for suspects and to report whether it is busy (i.e., if there are still suspects at large in the world for it to arrest)
 - `move()`
 - `move(direction d)`
 - `arrest()`
 - `busy()`
- note that we have two versions of the `move()` function: one that takes no arguments and one that takes one argument
- creating two versions of the same function, distinguished by their different argument lists, is called *overloading*
- we used overloading when talking earlier in the term about different kinds of constructors

```
void patrol::move(){
    direction d;
    d = static_cast<direction>( rand() % 4 );
    move( d );
}

// overloaded function
void patrol::move( direction d ){
    int x = location.getX();
    int y = location.getY();
    switch( d ) {
    case north: y = (y + 1) % WORLD_SIZE;
                break;
    case south: y = (y - 1);
                if ( y < 0 ) y = WORLD_SIZE;
                break;
    case east:  x = (x + 1) % WORLD_SIZE;
                break;
    case west:  x = (x - 1) % WORLD_SIZE;
                if ( x < 0 ) x = WORLD_SIZE;
                break;
    }
    location.set( x, y );
}
```

inheritance

- we can create a class that is an *extension* of another class—instead of redefining similar functions in an existing class we can extend from existing class and then add functionality to it.
- the class being extended is called the *superclass* (or *base*) class, and the resulting classes that are derived from extending the base class are called *subclasses* or *derived* classes
- since both `patrol` and `suspect` class instances need to use same `move` functions, we can modify both classes to inherit the `move` functions from `point` class
- look at `patrol12.cpp`, which is a modified version of `patrol.cpp` in which the `suspect` and `patrol` classes are redefined as subclasses of the `point` class

```

...
class suspect : public point {
private:
    bool caught;
public:
    suspect() { caught = false; }
    void jailed();
};

void suspect::jailed(){
    cout << "caught!" << endl;
    caught = true;
}

class patrol : public point {
private:
    int arrested;
public:
    patrol() : arrested(0) { set(0,0) }
    void arrest();
    bool busy();
};

void patrol::arrest(){

```

cis15-summer2008-ozgelen-lectIV.1

9

```

    cout << "arrested suspect!";
    arrested++;
}

// the patrol is busy until all the suspects has been arrested
bool patrol::busy() {
    if ( arrested < NUM_SUSPECTS )
        return true;
    else
        return false;
}

...
void point::move(){
    direction d;
    d = static_cast<direction>( rand() % 4 );
    move( d );
}

// overloaded function
void point::move( direction d ){
    int x = getX();
    int y = getY();
    switch( d ) {

```

cis15-summer2008-ozgelen-lectIV.1

10

```

case north: y = (y + 1) % WORLD_SIZE;
            break;
case south: y = (y - 1);
            if ( y < 0 ) y = WORLD_SIZE;
            break;
case east:  x = (x + 1) % WORLD_SIZE;
            break;
case west:  x = (x - 1) % WORLD_SIZE;
            if ( x < 0 ) x = WORLD_SIZE;
            break;
}
set( x, y );
}

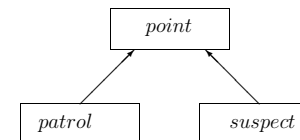
```

cis15-summer2008-ozgelen-lectIV.1

11

inheritance

- the relationship between the classes is illustrated by:



- that is the class patrol and the class suspect are both *subclasses* of the class point
- put another way, every instance of a patrol is an instance of point and every instance of suspect is an instance of point
- an instance of a subclass *inherits* all the members of its super class
- which means that suspect and patrol inherit x, y, getX(), getY(), set() and print() from point
- note that x and y are private, which means that they cannot even be accessed directly by the subclasses of point (later we'll explain how they could be)

cis15-summer2008-ozgelen-lectIV.1

12

- normally we want to do more than have a subclass just be a copy of the superclass—like we do with `suspect` and `patrol`
- what we often want to do is to have the subclass add things to the superclass
- (In Java this is explicit. When we define a subclass it is by saying it extends the superclass).
- in our example `patrol2.cpp`, the classes `suspect` and `patrol` are examples of this

- here `suspect` is extended with:
 - a private data member `caught`, which records whether the `suspect` instance has been arrested yet
 - a public function member `jailed` that sets the `caught` flag to true when a `suspect` instance has been arrested

- thus `suspect` has all of the data members of `point` as well as the additional ones listed here

- as a result we can do this:

```
suspect thief;
thief.set( 2, 3 );
```

which calls the `set` method on the `suspect` object named `thief`

- `suspect` *inherits* the `set` method from `point`

overriding and inheritance

- a subclass definition can re-define a function member defined in the superclass
- this is called *overriding*
- (don't confuse it with *overloading*!)
- we can, for example, override the definition of `move` in `patrol`
- the program `patrol3.cpp` has:

```
class people : public point {
public:
    void move();
    void move(direction d);
};
```

which has two subclasses:

```
class patrol : public people {
public:
    patrol() { arrested = 0; }
```

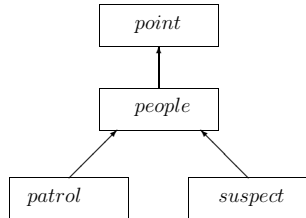
```
    void arrest();
    bool busy();
    move();
private:
    int arrested;
};

and

class suspect : public people {
public:
    suspect();
    void jailed();
private:
    bool caught;
};
```

- the `suspect` class uses the default `move()` functions, but the `patrol` class *overrides* the default definitions by providing its own (see the code, but the main difference is that the `patrol`'s `move()` function moves 2 spaces at a time instead of 1)

- a picture of this new hierarchy is shown below:



protected

- earlier we mentioned that we could not directly access a superclass' private data members in a subclass, but instead had to use the superclass' public function members to get access to the private data members
- however, if we really wanted access to the superclass' data members, we could declare them as `protected` instead of `private`
- `protected` data members sit somewhere between public members, which are accessible to any object, and private members, which are only accessible within that class
- roughly speaking, `protected` members are like private data members but are also accessible by members of derived classes
- we will talk more about `protected` later on.

comparing objects

- we can compare two instances of an object. for example, suppose we have a function that compares if one point is more to the west than another point:

```

bool point::moreWest( point a, point b ) {
    if ( a.getX() > b.getX() )
        return true;
    else
        return false;
}
  
```

- we can pass this function two instances of `point`, two instances of `people`, two instances of `patrol` or any combination such as a `point` and a `patrol`

virtual functions and abstract classes

- the program `patrol4.cpp` is another version of our world in which we define a new class `civilian` and something called a virtual function
- this is the function `defeated()` in the example
- notice that this function is first defined in the `people`
- but the function definition is preceded by the keyword `virtual` and has a funny prototype:

```

class people : public point {
public:
    void move();
    void move( direction d );
    virtual void defeated() = 0;
};
  
```

- because we have defined a virtual function, the class `people` is now called *abstract*
- any class that has at least one pure virtual function is an *abstract class*
- we do this when we know we will want to define a function (e.g., `defeated()`) but we don't want to give it any default behavior in the superclass

- the virtual function `defeated()` in `people` will never be called
- because an abstract class can never be instantiated
- it can only be extended, to create other classes
- the other classes, e.g., `suspect` and `civilian`, define their own versions of `defeated()` and then these are not abstract classes but instead can be instantiated

summary

- this lecture has looked at a number of issues related to object oriented programming in C++:
 - composition of classes
 - function overloading
 - inheritance
 - function overriding
 - comparing objects
 - virtual functions
 - abstract classes