

cis15
advanced programming techniques, using c++
fall 2007
lecture # IV.1

topics:

- inheritance
- composition of classes

resources:

- Pohl, chapters 8 and 11

an example

- consider the program `robot.cpp` (posted on the class web page)
- this program models a world in which there is a robot and some spots of dirt
- the robot wanders around looking for spots of dirt and vacuuming them up
- this is kind of like the assignment from unit II, where you had a robot running around looking for coins; but as we go through this example, you'll see where the code becomes more sophisticated and more *object-oriented*
- the class definition for the `robot` class is as follows

```
class robot {  
private:  
    point location;  
    int num_vacuumed;  
public:  
    robot() { num_vacuumed = 0; }  
    int getX() const;  
    int getY() const;  
    void set( int x, int y );  
    void print() const;  
    void move();  
    void move( direction d );  
    void eat();  
    bool hungry();  
};
```

composition

- the `robot` class includes a member of the `point` class, which we have used before (many times!)
- we say that the `robot` 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 `robot.cpp` is that the class `world` contains both `robot` and `dirt` instances

privacy

- note that several of the function members (methods) of robot 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 robot
- since the data members (x and y) are private, we cannot access them directly in robot
 - we have to refer to them indirectly by using the public function members of point

overloading

- the rest of the methods in the robot class give us the functionality we want from robot, allowing it to move, to vacuum up spots of dirt and to report whether it is busy (i.e., if there are still spots of dirt in the world for it to vacuum)
 - move()
 - move(direction d)
 - vacuum()
 - 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 robot::move(){
    direction d;
    d = static_cast<direction>( rand() % 4 );
    move( d );
}

// overloaded function
void robot::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 );
}
```

extending classes

- since we are inherently lazy, we can create a class that is an *extension* of another class—instead of creating a whole new class (like dirt or robot) that contains all the functionality of one class (like point) and then adds functionality to it
- the class being extended is called the *super* (or *parent*) class, and the resulting classes that are derived from extending the parent are called *subclasses* or *children* classes
- look at robot2.cpp, which is a modified version of robot.cpp in which the dirt and robot classes are redefined as subclasses of the point class

```

class dirt : public point {
private:
    bool gone;
public:
    dirt() { gone = false; }
    void disappear();
};

void dirt::disappear(){
    cout << "poof!" << endl;
    gone = true;
}

class robot : public point {
private:
    int num_vacuumed;
public:
    robot() { num_vacuumed = 0; }
    void move();
    void move( direction d );
    void vacuum();
    bool busy();
};

```

cis15-fall2008-azhar-lectIV.1

9

```

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

// overloaded function
void robot::move( direction d ){
    int x = getX();
    int y = 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;
    }
    set( x, y );
}

```

cis15-fall2008-azhar-lectIV.1

10

```

void robot::vacuum(){
    cout << "shroooooop...";
    num_vacuumed++;
}

// the robot is busy until all the dirt has been found and vacuumed
bool robot::busy() {
    if ( num_vacuumed < NUM_SPOTS ){
        return true;
    }
    else {
        return false;
    }
}

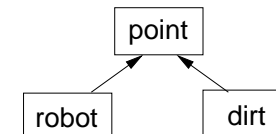
```

cis15-fall2008-azhar-lectIV.1

11

inheritance

- the relationship between the classes is illustrated by:



- that is the class robot and the class dirt are both *subclasses* of the class point
- put another way, every instance of a robot is an instance of point and every instance of dirt is an instance of point
- an instance of a subclass *inherits* all the members of its parent class
- which means that dirt and robot 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-fall2008-azhar-lectIV.1

12

- normally we want to do more than have a subclass just be a copy of the superclass—like we do with `dirt` and `robot`
- 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 `robot2`, the classes `dirt` and `robot` are examples of this

- here `dirt` is extended with:
 - a private data member `gone`, which records whether the `dirt` instance has been vacuumed up yet
 - a public function member disappear that sets the `gone` flag to true when a `dirt` instance has been vacuumed up
- thus `dirt` has all of the data members of `point` as well as the additional ones listed here
- as a result we can do this:


```
dirt spot;
spot.set( 2, 3 );
```

 which calls the `set` method on the `dirt` object named `spot`
- `dirt` *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 `robot`
- the program `robot3.cpp` has:

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

which has two subclasses:

```
class robot : public creature {
private:
    int num_vacuumed;
```

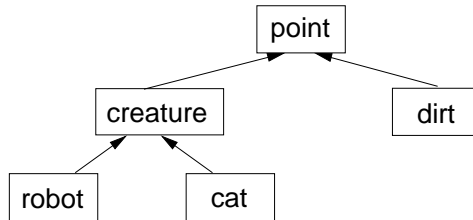
```
public:
    robot() { num_vacuumed = 0; }
    void vacuum();
    bool busy( int num_spots );
};
```

and

```
class cat : public creature {
public:
    void move();
    void move( direction d );
};
```

- the `robot` class uses the default `move()` functions, but the `cat` class *overrides* the default definitions by providing its own (see the code, but the main difference is that the `cat`'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.

more on inheritance

- since `robot` is a subclass of `creature`, we can carry out any operation on a `robot` that we can on a `creature` and on a `point`
- we already know that this is the case where the operations are function members of `creature` with simple parameters
- thus we can do:

```

robot rosie;
rosie.set( 2, 3 );
rosie.move();
  
```

calling methods from `point` and `creature` on `robot`

comparing objects

- we can compare two instances of an object:

```

bool robot::busier( robot r ) {
    if ( this->num_vacuumed > r.num_vacuumed )
        return true;
    else
        return false;
}
  
```

- we can pass this function an instance of a `robot` and get the result that compares the current instance (referenced by the pointer `this->`) to its argument instance

comparing objects using inheritance

- we can also do this with a superclass but pass an instance of a subclass
- for example, suppose we have a function that compares if one point is more to the west than another point:

```
bool point::moreWest( point p ) {  
    if ( this->getX() < p.getX() )  
        return true;  
    else  
        return false;  
}
```

- this could be called on two point objects:

```
point p1, p2;  
p1.set( 1, 2 );  
p2.set( 3, 4 );  
if ( p1.moreWest( p2 ) ) ...
```

- and it could also be called on two subclasses of point:

```
robot r1, r2;  
r1.set( 1, 2 );  
r2.set( 3, 4 );  
if ( r1.moreWest( r2 ) ) ...
```

virtual functions and abstract classes

- the program robot4.cpp is another version of our world in which we define something called a virtual function
- this is the function speak() in the example
- notice that this function is first defined in the creature
- but the function definition is preceded by the keyword virtual and has a funny prototype:

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

- because we have defined a virtual function, the class creature is now called *abstract*
- we do this when we know we will want to define a function (e.g., speak()) but we don't want to give it any default behavior in the superclass

- the virtual function speak() in creature 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., robot and cat, define their own versions of speak() and then these are not abstract classes but instead can be instantiated
- any class that has at least one pure virtual function is an *abstract class*
- you cannot create instances of abstract classes! (i.e., you cannot declare variables whose data type is an abstract class)

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