

COMPOSITION AND INHERITANCE

Today

- Today we will look at:
 - Composition; and
 - Inheritance
- These are the cornerstones of object oriented programming.
- This material is taken from Pohl, Chapter 8.

An example

- Consider the program `rabbit.cpp` which you can download from the class web site (Unit IV).
- This models a small ecosystem which holds:
 - A rabbit
 - Some carrots
- The rabbit runs around looking for carrots and eating them.
- The class definition for the `rabbit` class is as follows

```
class rabbit {  
  
private:  
    point location;  
    int    consumed;  
  
public:  
    rabbit(){consumed = 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 `rabbit` class includes a member of the `point` class, which we have played with before.
- We say that `rabbit` is related to `point` by *composition*.
- This just means what we see here — one class has an instance of another as a data member.
- Another example of composition in `rabbit.cpp` is that the class `world` contains instances of both `carrot` and `rabbit`.

- Several of the function members (methods) of `rabbit` look like those for `point`.
 - `getX()`
 - `getY()`
 - `set(int x, int y)`
- These data members provide a to alter the values of the attributes of the instance of `point` that is a member of `rabbit`.
- Since the data member `location` is `private` within `rabbit`, any program we write can't just use the function members of `point`.

- So this, for example:

```
rabbit peter;  
peter.location.getX();
```

would not compile.

- Instead we have to write *accessor* functions like `rabbit::getX()` which calls `point::getX()`
- This is the reason behind the functions that look like they only exist to call similar functions in `point`.

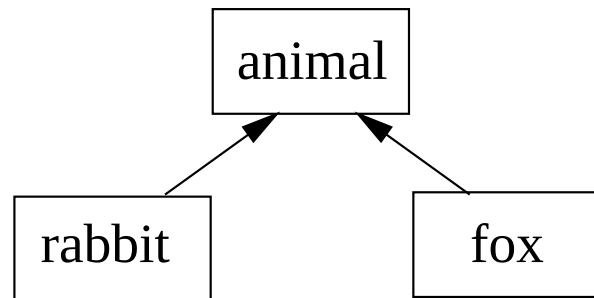
- Other methods are new:
 - `move()`
 - `eat()`
 - `hungry()`
- These give us the functionality we want from `rabbit`, allowing it to move, to report whether it is hungry, and to eat.
- If you haven't done so already, you should run the program `rabbit` and see how it works.
- Now imagine that we want to extend the program to include a fox, which runs around the world and eats rabbits.
- One way we could do this is to write a `fox` class that looks like the following.

```
class fox {  
  
    private:  
        point location;  
        int    consumed;  
  
    public:  
        fox(){consumed = 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();  
};
```

- This is exactly like the `rabbit` class since `fox` and `rabbit` are so similar.
- Both have a location in the world, move around, and eat things.
- Since they are so similar, writing both out seems a bit repetitive, and dull with it.
- It turns out that there is an alternative to doing this.
- The alternative is to use *inheritance* and this is considered better style than having lots of classes with (more or less) the same functionality.

Inheritance

- A program that handles the `fox` and `rabbit` example using *inheritance* is `rabbit3.cpp` on the class web page.
- The relationship between the classes is summarised by:



- That is the class `rabbit` and the class `fox` are both *subclasses* of the class `animal`.
- Alternatively, every instance of a `rabbit` is an instance of `animal` and every instance of `fox` is an instance of `animal`.

- We define `fox` as:

```
class fox : public animal {  
  
};
```

- This is the syntax for saying that `fox` has exactly the same members as `animal`.
- The keyword `public` indicates that all the `public` members of `animal` remain `public` in `fox`.
- If we replaced `public` with `private`, then all the `public` members of `animal` would become `private` in `fox`.
- We will say more about this next lecture.

- Normally we want to do more than have a subclass just be a copy of the superclass.
- 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).
- `rabbit` is an example of this.

```
class rabbit : public animal {  
  
private:  
  
    bool  eaten;  
  
public:  
  
    rabbit(){eaten = false;};  
    void beEaten();  
};  
  
void rabbit::beEaten(){  
    cout << "Drat that fox!" << endl;  
    eaten = true;  
}
```

- Here `rabbit` extends `animal` with:
 - A private data member `eaten`, which records whether the `rabbit` has been eaten by the fox; and
 - A public function member `beEaten` that takes appropriate action when the `rabbit` is eaten.
- Thus `rabbit` has all of the data members of `animal` as well as the additional ones listed here.
- As a result we can do this:

```
rabbit peter;  
peter.set(2, 3);
```

which calls the `set` method on the `rabbit peter`.

- `rabbit` *inherits* the `set` method from `animal`.

Overriding and inheritance

- A sub-class definition can re-define a function member defined in the super-class.
- This is called *overriding*.
- We can, for example, override the definition of move in fox.
- The program rabbit3.cpp has:

```
class fox : public animal {  
  
public:  
    void move();  
    void move(direction d);  
  
};
```

giving new definitions for how the fox moves.

Aside: “protected”

- It turns out (as you can see in `rabbit3.cpp`) that we need to make the `animal` vclass a little different from the `rabbit` class that we started with.
- The problem is that to allow `move` to be overridden in `fox`, we have to change the value of `location` in `fox`.
- Now, `location` is `private` to `animal`, so `fox` cannot alter it.
- One answer is to make `location` not `private` but `protected`.
- `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.
- So a protected data member in `animal` is accessible by `rabbit` and `fox`, but not by `carrot`.
- In contrast, if we made `location` public, it would also be accessible by objects of class `carrot`.
- We will talk more about protected later on.

More inheritance

- Since `rabbit` is a subclass of `animal`, we can carry out any operation on a `rabbit` that we can on a `animal`.
- We already know that this is the case where the operations are function members of `animal` with simple parameters.
- Thus we can do:

```
rabbit peter;  
peter.set(2, 3);  
peter.move();
```

calling methods from `animal` on `rabbit`.

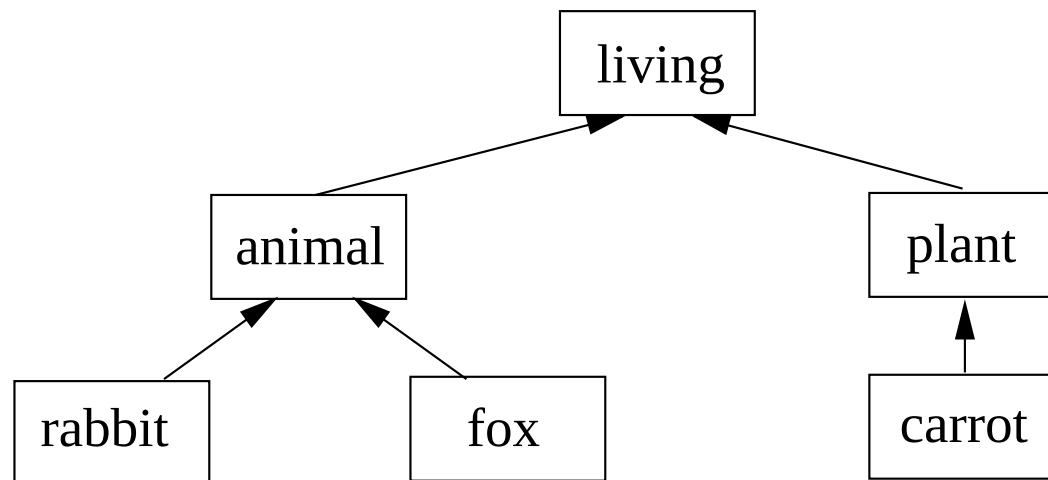
- It turns out we can go a bit further than this also.
- If we have:

```
bool animal::hungrier(animal a1, animal a2){  
    if(a1.consumed < a2.consumed){  
        return true;  
    }  
    else {  
        return false;  
    }  
}
```

we can pass this two rabbits, two foxes, or a rabbit and a fox.

Virtual functions

- The program `rabbit5.cpp` is a cleaner version of our little ecosystem.
- By defining a class `living`, we can exploit the fact that `carrot` has some aspects (to do with location) that are just like `rabbit` and `fox`.
- We have the class hierarchy:



- Not all of the functions that exist in the sub-classes make sense in the super class.
 - For example, since plants do not move, it makes little sense to have a `move` class in `living`.
- The function `beEaten`, does apply to all living things and so could be defined in `living`.
- However, in our example, every class implements `beEaten` in its own way.
- C++ style suggests that we should define functions like `beEaten` that we know will be overridden as *virtual functions*.
- We do this by adding the keyword `virtual` before the function prototype:

```
virtual void beEaten();
```

- The virtual function `beEaten` in `living` will never be called.
- All objects are carrots, rabbits, or foxes, and these all define their own way to `beEaten`.
- In such cases we should define `beEaten` in `living` as a *pure virtual function*.
- We do this by:

```
virtual void beEaten() = 0;
```
- Any class that has at least one pure virtual function is an *abstract class*.
- You cannot create instances of abstract classes.

Why have virtual functions?

- There are (at least) three reasons.
- First because it makes sense in the context of the class hierarchy.
 - The ability to be eaten is a property of living things so it should be defined at the level of `living`.
 - However, each living thing will `beEaten` in a different way so we don't want to say exactly how it will happen.
- Second, because we want to *prevent* anyone making an instance of class `living`.
- Third, because you want to force all sub-classes of `living` to define their own `beEaten`.
 - If you have a sub-class that doesn't create a new `beEaten` and override the abstract version, that sub-class will be virtual also, and you won't be able to make instances of it.

Summary

- This lecture has looked at a number of issues related to object oriented programming in C++.
 - Composition of classes
 - Inheritance
 - Overriding;
 - Virtual functions; and
 - Abstract classes.