MULTIPLE INHERITANCE

Today

- Today we will look in more detail at how C++ achieves inheritance.
- This (hopefully) explains some of the strange things that you have seen so far.
- We will end by looking at multiple inheritance, where a class inherits from more than one class.
- Again the textbook covers this in Chapter 8.
- And again it doesn't really cover it in enough detail.
- Let's start with a recap.

Composition and inheritance

- We use *composition* when one class contains a data member that is an object of another class.
- Thus in `rabbit4.cpp`, the class `living` contains a data member `location` which is an object of the class `point`.
- Thus `living` and `point` are related by composition.
- Any object of type `living` thus includes an object, called `location`, of type `point`.
- To access the `private` data members of `location` from within an object that contains it, we have to use the `public` function members of `point`.

- We use *inheritance* when one class extends another class, as in:

  ```
  class animal : public living
  ```

  from `rabbit5.cpp`.
- Here `living` is called the *base class* or *super-class* and `animal` is called the *sub-class*.
- We can think of this as meaning that an object of class `animal` contains all the data and function members of class `living`.
- If we had an object `a` of class `animal`, we would refer to its member `location` by:

  ```
  a.location
  ```

- And the data member `x` of `location` as:

```
a.location.x
```

- However, it is not quite as simple as that.
- The way that C++ implements inheritance is such that an object of class `animal` contains an object of class `living` (rather than the members of that object).
- Access to the members of this sub-object follow the usual access rules.
- Thus the `private` data members of `living` are not accessible from within `animal`.
- This is typically not what we want.

“public”, “private” and “protected”

- One way to handle the fact that a sub-class can’t access the `private` members of a base class is to write `public` methods that access them.
- Methods like `set`, `getX` and `getY` for `point`.
- Another approach is to redefine the `private` members as `protected`.
- Thus:

```
class living {

protected:

  point location;
  bool  eaten;

};
```

- Using `protected` here means that the members are treated as `public` in classes derived from `living` (like `animal`).
- However, for classes that are not derived from `living`, the `protected` data members are treated like they are `private`.
- This is exactly what we want in `rabbit4.cpp`.
- The general question of how sub-classes can access members of base classes is more complex than this, however.

Access to base class members

- Each member of a base class can be:
  - `public`
  - `protected`
  - `private`
- Classes can also be derived as:
  - `class A : public B`
  - `class A : protected B`
  - `class A : private B`
- These access levels interact.

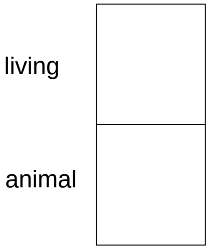
- If we have `class A : public B`
  - `public` and `protected` members of `B` remain `public` and `protected` in `A`.
- If we have `class A : protected B`
  - `public` and `protected` members of `B` are `protected` in `A`.
- If we have `class A : private B`
  - `public` and `protected` members of `B` become `private` in `A`.
- Of course, even if base class members are `private` they can be accessed by `friend` classes.
- (Now would be a good time to go back and recap `friend` classes).

Inheritance again

- Let's go back to what we said above, that:

  The way that C++ implements inheritance is such that an object of class `animal` contains an object of class `living` (rather than the members of that object).
- This is *literally* true.
- An object of class `animal` has two parts, an object of class `living` and an object with all the things that are in `animal` but not in `living`
- The fact that these are separate objects explains the problem with `private` data.
- It also explains some other stuff.

- So an `animal` object looks like:



- When we reference an `animal` object by name or by an `animal` pointer, the system will look first in the `animal` part.
- Only if it can't find the referenced member will it look in the `living` part.

- Thus if we have:

  ```
  animal a;
  a.beEaten();
  ```

  and

  ```
  animal *aptr = &a
  aptr->beEaten();
  ```

  then the copy of `beEaten()` that will be called will be the one in `animal`.
- If we want to call the one in `living` we can use:

  ```
  aptr->living::beEaten();
  ```

  explicitly calling the version in the `living` bit of `a`.

- If instead we have:

```
living *lptr = &a
lptr->beEaten();
```

then by default the version of `beEaten` that will be executed is the one in the `living` bit of `a` because the pointer is one that points to `living`.

- As we already saw, we can force it to call the version in `animal` by making it virtual.

Multiple inheritance

- In statements of class derivation like

```
    class A : public B
```

we are not limited to deriving from a single base class.

- We can have, for example:

```
    class A : public B, public C
```

- This is called *multiple inheritance*.
- In the latter case `A` has all of the members of `B` and `C`.

- This offers scope for ambiguity.
- If `B` and `C` both have a function `print`, and `A` does not, then in

```
A adele;
adele.print()
```

is ambiguous.

- We have to say which `print` we want, for example:

```
adele.B::print();
```

- When we make a statement like:

```
    class A : public B, public C
```

there is no limit to the number of classes `A` can inherit from.

- However, the same class cannot appear twice.
- This does not stop a class inheriting from the same class twice though.

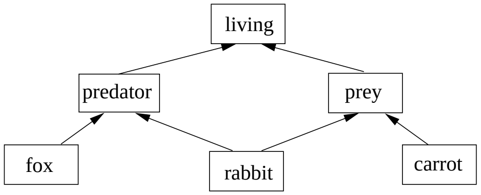
- As an example, consider a variation on the classes in `rabbit4.cpp`.
- We could have:

```
class predator: public living{

public:
void eat();
};

class prey: public living{

public:
void beEaten();
};
```

- `carrot` is then a sub-class of `prey`, and `fox` is a sub-class of `predator`.
- `rabbit` is both predator and prey (it eats carrots but is eaten by foxes), so we would define:

```
class rabbit: public predator, public prey
```

- Now we have the class hierarchy:



and `rabbit` now inherits from `living` twice, once through `predator` and once through `prey`.

- This means it has two copies of all the members that it inherits from `living`.
- If we have:

```
rabbit peter;
peter.location.set(1, 2);
```

it is ambiguous which `location` this refers to.

- As before, we can get around this by using class scope.

```
peter.prey::location.set(1, 2);
```

says to use the version of `location` inherited through `prey`.

- A more elegant solution is to use a `virtual` base class.

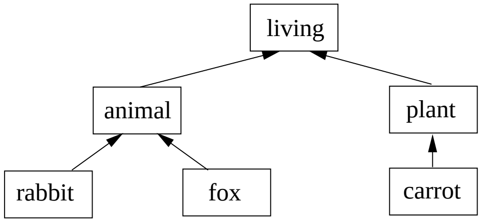
- If we define:

```
class predator: virtual public living{

public:
void eat();
};

class prey: virtual public living{

public:
void beEaten();
};

class rabbit: public predator, public prey{
};
```

then `rabbit` will only contain one copy of `living`.

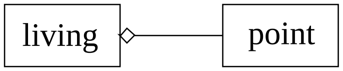
Unified Modelling Language

- We'll finish by recapping what we mentioned before about UML.
- UML is a method of designing and documenting object-oriented designs.
- We are already familiar with the idea of drawing the relationship between classes:



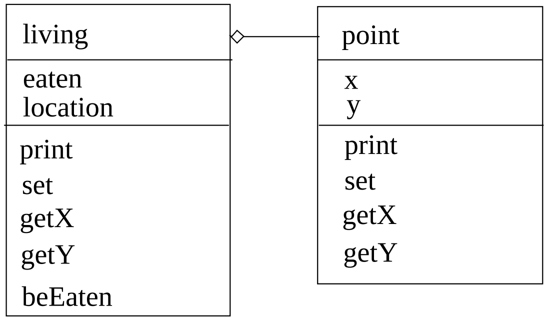
UML expands on this.

- UML uses the same notation as we have been using already to show inheritance between classes.
- UML adds a graphical representation of composition:



indicates that `living` includes an object of type `point`

- UML also shows the data and function members that a class contains.
- The full UML representation of `living` and `point` from `rabbit4.cpp` is shown on the next slide.



- Clearly we could expand the rest of the class hierarchy with this additional information.
- The idea behind UML is to use this graphical notation to develop the class design before coding.
- The diagrams also serve as a form of documentation.
- Tools for drawing UML diagrams, tutorials and much more can be found at `http://www.uml.org/`.

Summary

- This lecture looked in detail at inheritance.
- We started with a recap of the differences between composition and derivation.
- Then we looked at access to members from the base class.
- And we looked at the possibilities and problems of multiple inheritance.
- Finally we recapped some UML.