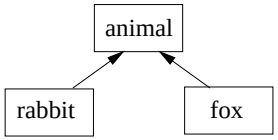
VIRTUAL FUNCTIONS

Today

- Today we will look at the topic of *virtual functions*.
- These turn out to be of great importance in object oriented programming.
- In particular they allow us to exploit inheritance to achieve polymorphism.
- This material is taken from Pohl, Chapter 8, but Pohl doesn't say much, so you might want to look at `www.learncpp.com` which has some explanations of virtual functions.
- The section on polymorphism on `www.cplusplus.com` also covers this material nicely.

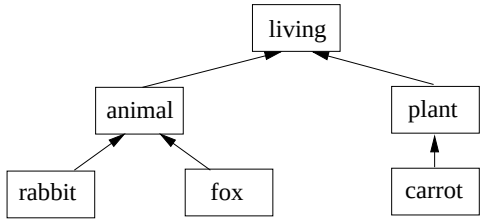
More Inheritance

- Last time we ended up with an inheritance hierarchy that looked like:



- That is the class `rabbit` and the class `fox` are both *subclasses* of the class `animal`.
- This lecture we will look at expanding it.

- 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`,.
- This gives us a heirarchy that looks like this:



- So `animal` is a subclass of `living` and so is `plant`.
- `carrot` is then a subclass of `plant`.

- With this re-arrangement of the class hierarchy, we have to reorganise the design of the classes.
- A revamped version of the example may be found in `rabbit4.cpp`.
- 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`.
- However, function `beEaten`, does apply to all living things and so we will define it in `living`.
- However, in our example, every class implements `beEaten` in its own way, so, we will *over-ride* the definition in all the sub-classes.

- Now, recall the function that we defined in the last lecture:
- We have:

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

- Since we can pass this two `rabbit`s, two `fox`es, or a `rabbit` and a `fox`, this gives us a simple form of polymorphism.
- With a function like `beEaten` that exists in every class, we can go further.

- Let's define:

```
void howDoYouDie(living *ptr){
  ptr->beEaten();
}
```

- This takes as its argument a pointer to a `living`, so a natural way to call this is:

```
living l;
living *lptr = &l;
howDoYouDie(lptr);
```

- Given the way `beEaten` is implemented, this will print:

```
I live, therefore I can be eaten
```


- Since `plant` is a subclass of `living`, we can also do this:

```
plant p;
lptr = &p;
howDoYouDie(lptr);
```

- However, the final line generates:

```
I live, therefore I can be eaten
```

because `howDoYouDie` calls the `beEaten` for `living` rather than the `beEaten` for `plant`

- The system picks the version of `beEaten` that matches the type of the pointer.
- How can we call the right `beEaten`, beairng in mind we still want to keep the pointer the same so that `howDoYouDie` remains polymorphic?

Virtual functions

- The answer is that we make `beEaten` a *virtual function*.
- We do this by adding the keyword `virtual`:

```
virtual void beEaten(){
  cout << "I live, therefore I can be eaten";
  cout << endl;
}
```

at the highest point up the inheritance hierarchy — here that is in `living`

- Now when we do this:

```
plant p;
lptr = &p;
howDoYouDie(lptr);
```

the C++ system does not just pick the version of `beEaten` to match the pointer.

- Instead, the `virtual` makes it go down the hierarchy *while the program is running* to find the *most specific* `beEaten`.
- Most specific means "lowest down the hierarchy"

- So in `rabbit4.cpp`, each of:
  - `plant`
  - `animal`
  - `carrot`
  - `rabbit`

  will call their own `beEaten`.
- However `fox`, which doesn't over-ride the version it inherits, will use the one from `animal`.

- C++ style suggests that we should define functions like `beEaten` that we know will be overridden as virtual functions.
- This allows us to define functions so they 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`.
- Making them virtual rather than non-virtual functions allows us to get the fucntionality we need
  - Each living thing will `beEaten` in a different way and we let the system pick the appropriate way in polymorphic functions.

Virtual destructors

- `www.learncpp.com` points out that you should *always* make destructors virtual in a super-class.
- Otherwise, just as with `beEaten`, there will be times when the wrong function will be called:
  - The one in the base class not the one in the derived class.
- If we are writing a destructor it suggests that we need to be sure to delete some memory.
- So make sure that the right destructor gets called, and it all gets deallocated.

Abstract classes

- Just to be confusing, C++ makes another use of the keyword `virtual`.
- Rather than:

```
virtual void beEaten(){
  cout << "I live, therefore I can be eaten";
  cout << endl;
}
```

`rabbit5.cpp` has `living` define:

```
virtual void beEaten() = 0;
```

- This is a *pure virtual function*.

- Clearly a function like this can't be called (what would it do?), so if we create one, it has two important consequences.
- The first is that we can't make any instance of a class that contains a pure virtual function.
- In `rabbit5.cpp` we can't make an `living` objects.
- The class `living` becomes what we call an *abstract* class

- The second consequence of the pure virtual function is that every sub-class of `living` has to define `beEaten`.
- If it doesn't, then that class becomes abstract, and we can't make any instances of it either.
- `rabbit5.cpp` is a version of the rabbit example with an abstract version of `living`.

- There are a couple of reasons to make `beEaten` a pure virtual function.
- First, because we want to *prevent* anyone making an instance of class `living`.
  - You can't create an instance of an abstract class.
- Second, because you want to force all sub-classes of `living` to define their own `beEaten`.
  - Otherwise you won't be able to make instances of them.
- In both cases we can use abstract classes to make the class hierarchy work the way that we want it to.

Summary

- This lecture has looked at virtual functions.
- When we create a virtual function we are enabling polymorphism by forcing the C++ system to find the right over-riding function at run-time.
- We also looked at pure virtual functions.
- Defining just one pure virtual function in a class makes it an abstract class and prevents us from making instances of it.
- Abstract classes are used to structure the inheritance hierarchy.