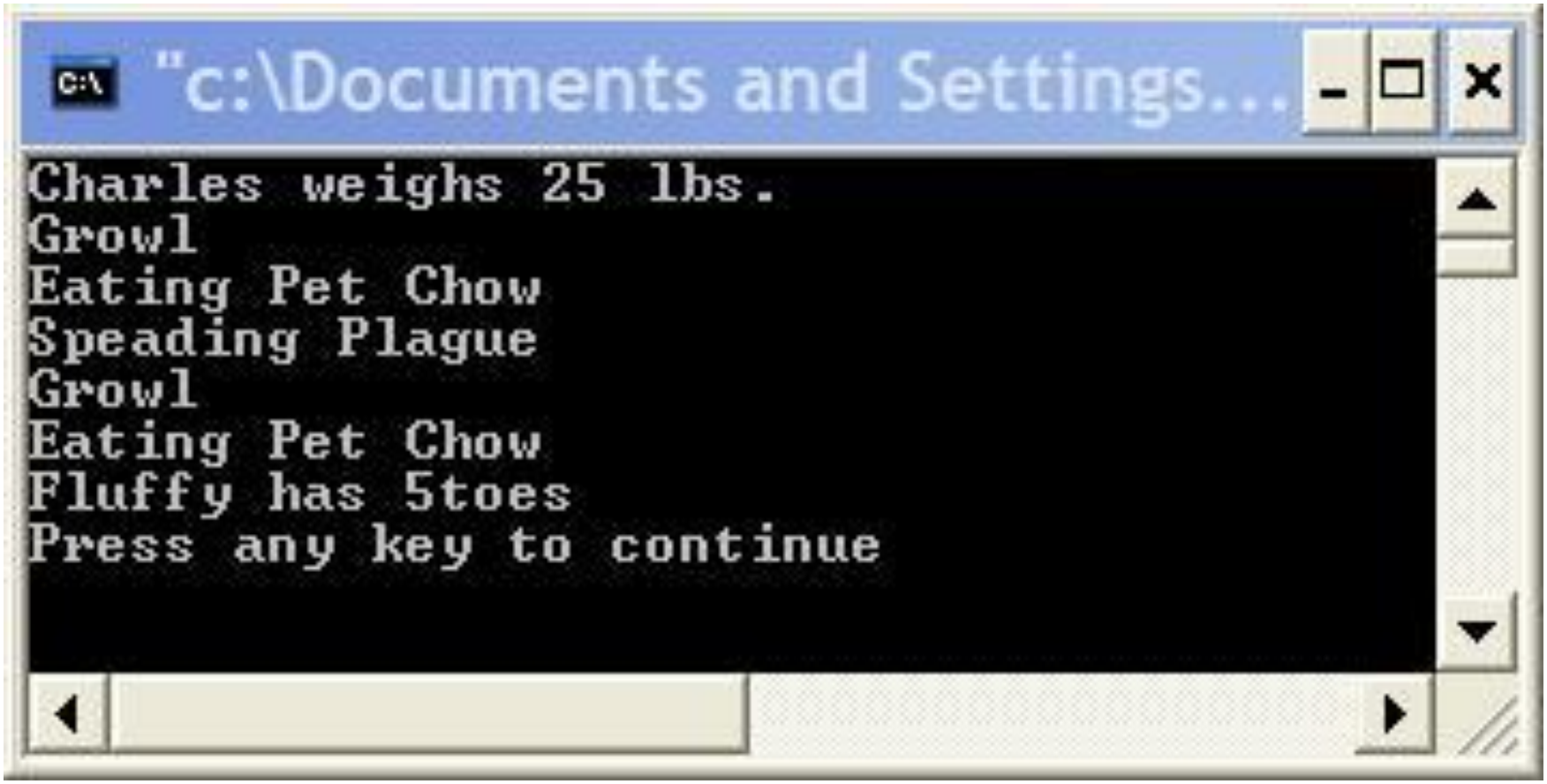
Inheritance

```
#include
    #include
    using namespace std;

    class Pet
    {
    public:
    // Constructors, Destructors
    Pet(): weight(1), food("Pet Chow") {}
    ~Pet() {}

    //Accessors
    void setWeight(int w) {weight = w;}
    int getWeight() {return weight;}

    void setfood(string f) {food = f;}
    string getFood() {return food;}

    //General methods
    void eat();
    void speak();

    protected:
    int weight;
    string food;

    };

    void Pet::eat()
    {
    cout << "Eating " << food << endl;
    }

    void Pet::speak()
    {
    cout << "Growl" << endl;
    }

    class Rat: public Pet
```

```
class Rat: public Pet
    {
    public:
    Rat() {}
    ~Rat() {}

    //Other methods
    void sicken() {cout << "Spreading Plague" << endl;}

    };

class Cat: public Pet
    {
    public:
    Cat() : numberToes(5) {}
    ~Cat() {}

    //Other accessors
    void setNumberToes(int toes) {numberToes = toes;}
    int getNumberToes() {return numberToes;}

    private:
    int numberToes;
    };
    int main()
    {
    Rat charles;
    Cat fluffy;

    charles.setWeight(25);
    cout << "Charles weighs " << charles.getWeight() << "
    lbs. " << endl;
    charles.speak();
    charles.eat();
    charles.sicken();

    fluffy.speak();
    fluffy.eat();
    cout << "Fluffy has " << fluffy.getNumberToes() << "
    toes " << endl;

    return 0;
    }
```

Output


"c:\Documents and Settings...
Charles weighs 25 lbs.
Growl
Eating Pet Chow
Speading Plague
Growl
Eating Pet Chow
Fluffy has 5toes
Press any key to continue

comments

- notice that the data members of the Pet base class are declared as protected. Protected indicates that publicly derived subtypes of the base class will be able to directly access these variables.
- If they were declared to be private, only the Pet class could directly access them; subclasses could not.
- If they were declared public, any class or part of code could accessed them. This would defeat a key goal of object-oriented design: encapsulating data within a class and exposing the data only through the public interface.
-

comments

- The last thing to notice in this example is that constructorsThe last thing to notice in this example is that constructors, including copy constructorsThe last thing to notice in this example is that constructors, including copy constructors, and destructors are not inherited. Each subtype has its own constructor and destructor.

comments

- Rat and Cat objects inherit the methods of the base class, Pet. We can call the speak method and the eat method on objects of type Rat and Cat. These methods were defined only in Pet and not in the subclasses.
- Also, notice the each subclass extends the base class by adding methods and members. The Rat class has the "sicken" method. The Cat class has methods and members related to the number of toes an individual cat object has.
- This is the most common software reuse by **extending additional behaviors** in OO system development

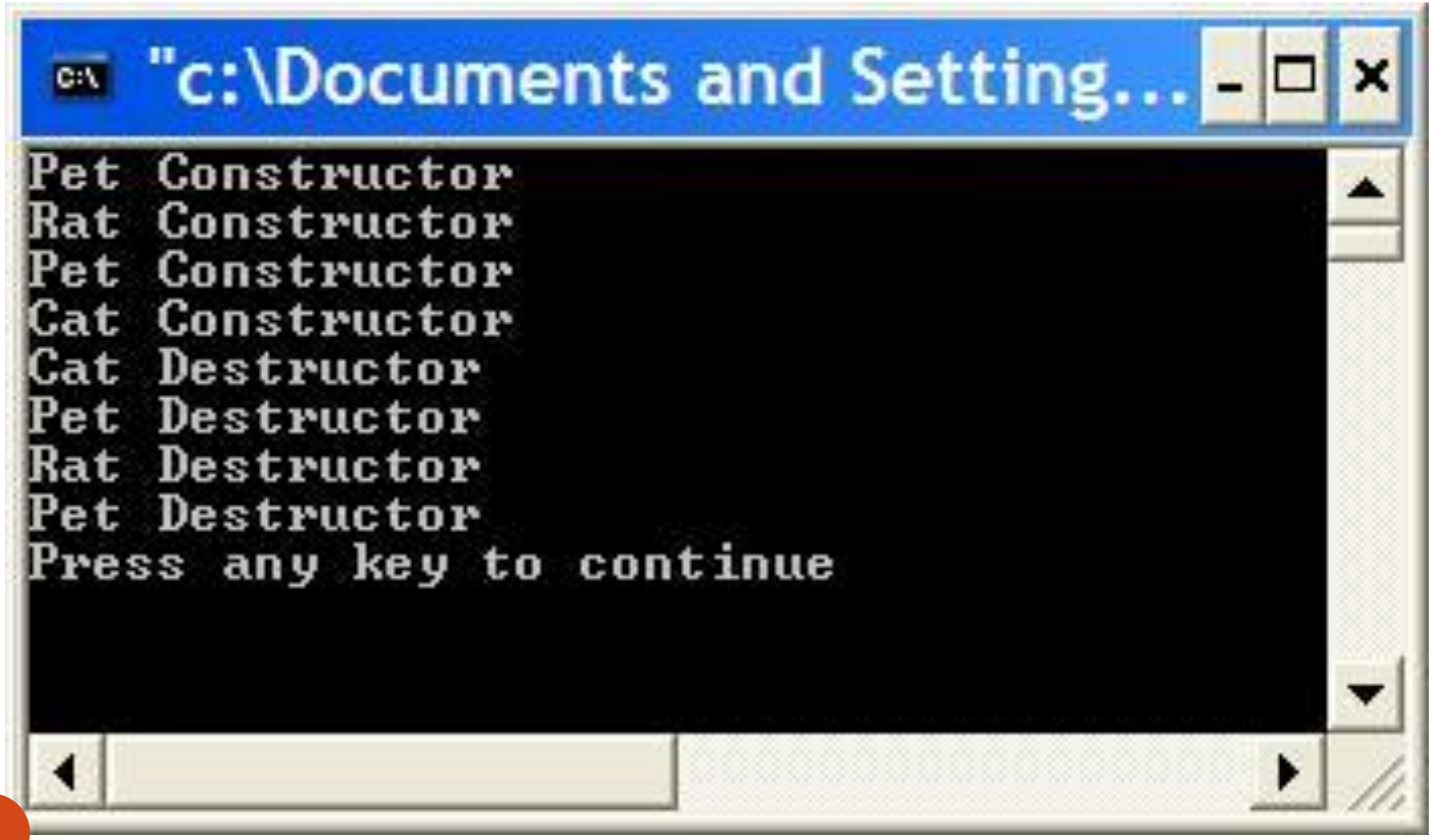
basic principle of inheritance

- The base class contains common members and methods used by the subclasses
- The subtypes are more specialized than the base class.

```
#include
    #include
    using namespace std;

    class Pet
    {
    public:
    // Constructors, Destructors
    Pet(): weight(1), food("Pet Chow")
    {
    cout << "Pet Constructor" << endl;
    }
    ~Pet()
    {
    cout << "Pet Destructor" << endl;
    }

    // Rest of code unmodified from first example
    ......

    };


    class Rat: public Pet
    {
    public:
    Rat()
    {
    cout << "Rat Constructor" << endl;
    }
    ~Rat()
    {
    cout << "Rat Destructor" << endl;
    }

    // Rest of code unmodified from first example
    ......

    };

    class Cat: public Pet
```

```
#include

    class Cat: public Pet
    {
    public:
    Cat() : numberToes(5)
    {
    cout << "Cat Constructor" << endl;
    }
    ~Cat()
    {
    cout << "Cat Destructor" << endl;
    }

    // Rest of code unmodified from first example
    ......

    };

    int main()
    {
    Rat charles;
    Cat fluffy;

    //charles.setWeight(25);
    //cout << "Charles weighs " <<
    charles.getWeight() << " lbs. " << endl;
    //charles.speak();
    //charles.eat();
    //charles.sicken();

    //fluffy.speak();
    //fluffy.eat();
    //cout << "Fluffy has " <<
    fluffy.getNumberToes() << "toes " << endl;

    return 0;
    }
```

output


"c:\Documents and Setting...
Pet Constructor
Rat Constructor
Pet Constructor
Cat Constructor
Cat Destructor
Pet Destructor
Rat Destructor
Pet Destructor
Press any key to continue

comments

- The base classThe base class part of an object is always constructed first and destroyed last. The subclass part of an object is constructed last and destroyed first.
- The reason for this is that each object of a subtype consists of multiple parts, a base class part and a subclass part. The base class constructor forms the base class part. The subclass constructor forms the subclass part. Destructors clean up their respective parts.
- 每一個 subclass 要自行負責建構與清除自己的特異化的的部分

Passing arguments into constructors

- In the previous example, all the classes used default constructors. That is, the constructors took no arguments. Suppose that there were constructors that took arguments. How would this be handled? Here's a simple example.

```
#include <iostream>
#include <string>
using namespace std;

class Pet
{
public:
// Constructors, Destructors
Pet () : weight(1), food("Pet Chow")
{}
Pet(int w) : weight(w), food("Pet
Chow") {}
Pet(int w, string f) : weight(w),
food(f) {}
~Pet() {}

//Accessors
void setWeight(int w) {weight = w;}
int getWeight() {return weight;}

void setfood(string f) {food = f;}
string getFood() {return food;}

//General methods
void eat();
void speak();

protected:
int weight;
string food;

};
```

```
void Pet::eat()
{
cout << "Eating " << food << endl;
}

void Pet::speak()
{
cout << "Growl" << endl;
}

class Rat: public Pet
{
public:
Rat() {}
Rat(int w) : Pet(w) {}
Rat(int w, string f) : Pet(w,f) {}
~Rat() {}

//Other methods
void sicken() {cout << "Speading Plague"
<< endl;}

};

class Cat: public Pet
{
public:
Cat() : numberToes(5) {}
Cat(int w) : Pet(w), numberToes(5) {}
Cat(int w, string f) : Pet(w,f),
numberToes(5) {}
Cat(int w, string f, int toes) : Pet(w,f),
numberToes(toes) {}
~Cat() {}

//Other accessors
void setNumberToes(int toes)
{numberToes = toes;}
int getNumberToes() {return
numberToes;}

private:
int numberToes;
};
```

```
int main()
{
Rat charles(25,"Rat Chow");
Rat john;//Default Rat
constructor
Cat fluffy(10,"rats");
Cat buffy(10,"fish",6);

cout << "Charles weighs " <<
charles.getWeight() << " lbs. "
<< endl;
charles.speak();
charles.eat();
charles.sicken();

cout << "John weighs " <<
john.getWeight() << " lbs. " <<
endl;
john.speak();
john.eat();
john.sicken();

fluffy.speak();
fluffy.eat();
cout << "Fluffy has " <<
fluffy.getNumberToes() << "toes
" << endl;

buffy.speak();
buffy.eat();
cout << "Buffy has " <<
buffy.getNumberToes() << "toes
" << endl;

return 0;
}
```

comments

- Rat and Cat constructors that take arguments, which are in turn passed to the appropriate Pet constructor. The base class, Pet, constructor is added to the member initialization list of the derived class constructors.
- Also notice that for the derived class (Rat and Cat) default constructors, the Pet default constructor does not need to be explicitly called.

A complete list of invoked constructors

- Rat charles(25,"Rat Chow");
  Pet(int w, string f)
  Rat(int w, string f)

  Rat john;
  Pet()
  Rat()

  Cat fluffy(10,"rats");
  Pet(int w, string f)
  Cat(int w, string f)

  Cat buffy(10,"fish",6);
  Pet(int w, string f)
  Cat(int w, string f, int toes)

Output

```
"c:\Documents and Set...
Charles weighs 25 lbs.
Growl
Eating Rat Chow
Speading Plague
John weighs 1 lbs.
Growl
Eating Pet Chow
Speading Plague
Growl
Eating rats
Fluffy has 5 toes
Growl
Eating fish
Buffy has 6 toes
Press any key to continue
```

Overriding methods

- A derived classA derived class can use the methods of its base classA derived class can use the methods of its base class(es), or it can override them.
- The method in the derived class must have the same **signature** and return type as the base class method to override.
  - The signature is number and type of arguments and the constantness (const, non- const) of the method. When an object of the base class is used, the base class method is called.
- Note that overriding is different from **overloading.** With overloading, many methods of the same name with different signatures (different number and/or types of arguments) are created.
- With overriding, a subclass implements its own version of a base class method. The subclass can selectively use some base class methods as they are, and override others.

```
#include <iostream>
    #include <string>
    using namespace std;

    class Pet
    {
    public:
    // Constructors, Destructors
    Pet () : weight(1), food("Pet Chow") {}
    Pet(int w) : weight(w), food("Pet Chow") {}
    Pet(int w, string f) : weight(w), food(f) {}
    ~Pet() {}
    //Accessors
    void setWeight(int w) {weight = w;}
    int getWeight() {return weight;}

    void setfood(string f) {food = f;}
    string getFood() {return food;}

    //General methods
    void eat();
    void speak();

    protected:
    int weight;
    string food;

    };
    void Pet::eat() {
    cout << "Eating " << food << endl;
    }
    oid Pet::speak() {
    cout << "Growl" << endl;
    }
```

```
class Rat: public Pet
{
public:
Rat() {}
Rat(int w) : Pet(w) {}
Rat(int w, string f) : Pet(w,f) {}
~Rat() {}
//Other methods
void sicken() {cout << "Speading Plague" << endl;}
void speak();
};
void Rat::speak(){ cout << "Rat noise" << endl;}
class Cat: public Pet
{
public:
Cat() : numberToes(5) {}
Cat(int w) : Pet(w), numberToes(5) {}
Cat(int w, string f) : Pet(w,f), numberToes(5) {}
Cat(int w, string f, int toes) : Pet(w,f),
numberToes(toes) {}
~Cat() {}

//Other accessors
void setNumberToes(int toes) {numberToes = toes;}
int getNumberToes() {return numberToes;}

//Other methods
void speak();

private:
int numberToes;
};
void Cat::speak()  { cout << "Meow" << endl;   }
int main()
{
Pet peter;
Rat ralph;
Cat chris;

peter.speak();
ralph.speak();
chris.speak();

return 0;
l }
```

output


"c:\Documents and Set...
Growl
Rat noise
Meow
Press any key to continue

Notice

- remember that the return type and signature of the subclass method must match the base class method exactly to override.
- Another important point is that if the base class had overloaded a particular method, overriding a single one of the overloads will hide the rest.
  - For instance, suppose the Pet class had defined several speak methods.

Be very careful!

These are overloaded methods

- void speak();
  void speak(string s);
  void speak(string s, int loudness);
  If the subclass, Cat, defined only
  void speak();
  Then speak() would be overridden. speak(string s) and speak(string s, int loudness) would be hidden. This means that if we had a cat object, fluffy, we could call:
  fluffy.speak();
  But the following would cause compilation errors.
  fluffy.speak("Hello");
  fluffy.speak("Hello", 10);

- Generally, if you override an **overloaded base class** method you should either override every one of the overloads, or carefully consider why you are not. It is a safety protocol enforced by compiler to prevent you from doing such error

The Principle of Inheritance

level of abstraction is not uniform.


```
class EmployeeCensus: public ListContainer {
public:
    ...
    // public routines
    void AddEmployee( Employee employee );
    void RemoveEmployee( Employee employee );

    Employee NextItemInList();
    Employee FirstItem();
    Employee LastItem();
    ...
private:
    ...
};
```

- This class is representing two ADTs: an employee and a ListContainer.

C++ Example of a Class Interface with Consistent Levels of Abstraction

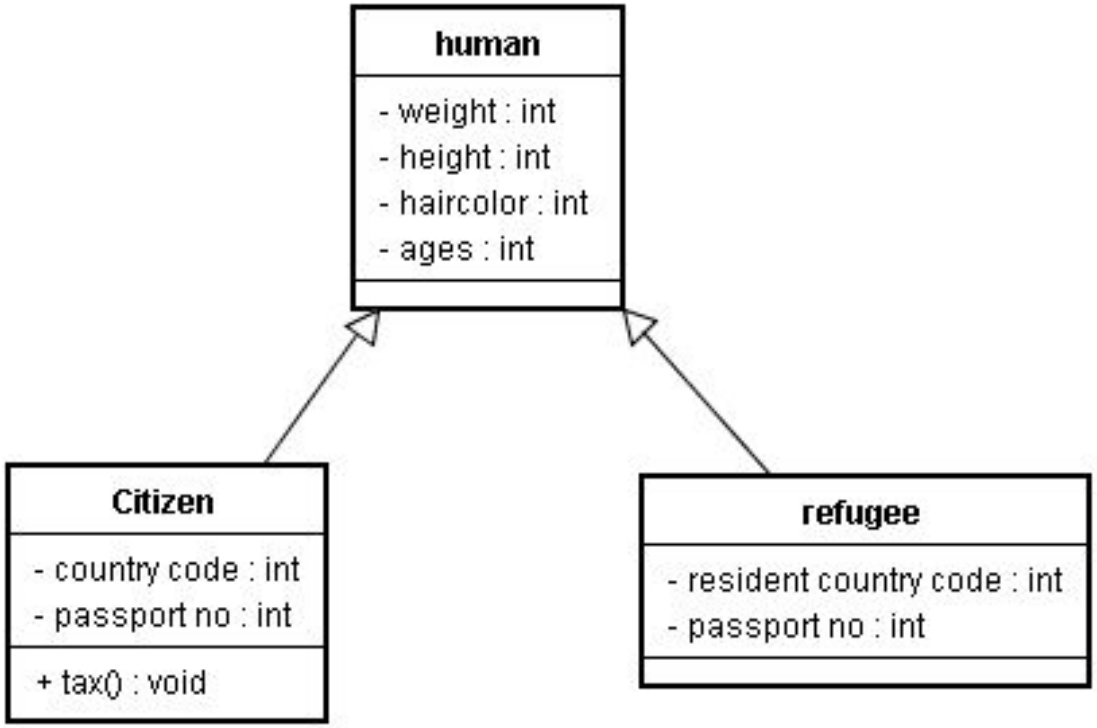
```
class EmployeeCensus {
public:
    ...
    // public routines
    void AddEmployee( Employee employee );
    void RemoveEmployee( Employee employee );
    Employee NextEmployee();
    Employee FirstEmployee();
    Employee LastEmployee();
    ...
private:
    ListContainer m_EmployeeList;
    ...
};
```

The abstraction of all these routines is now at the "employee" level.

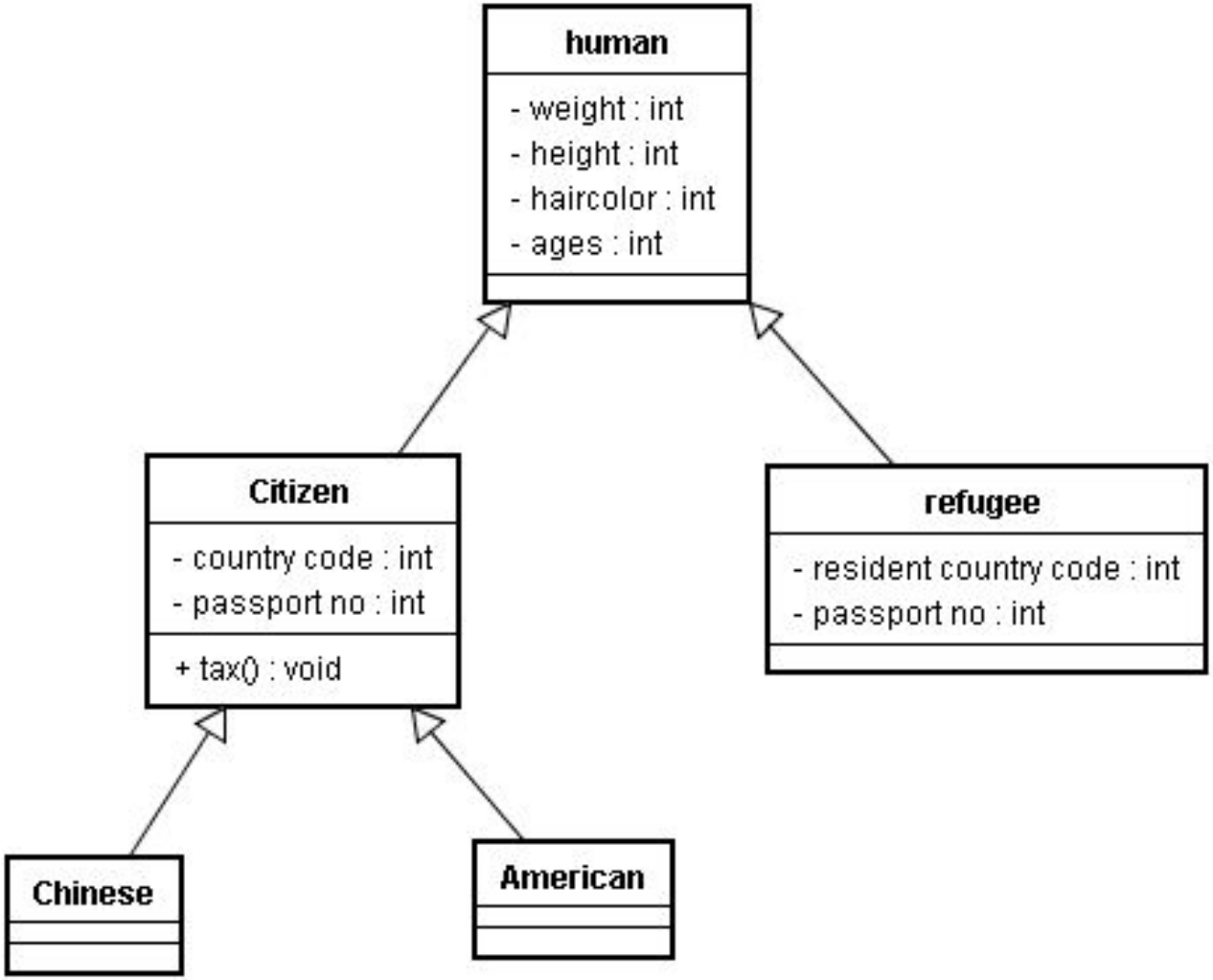
That the class uses the *ListContainer* library is now hidden.

To subclass or not to subclass?

- Consider you are a UN (United Nation) staff



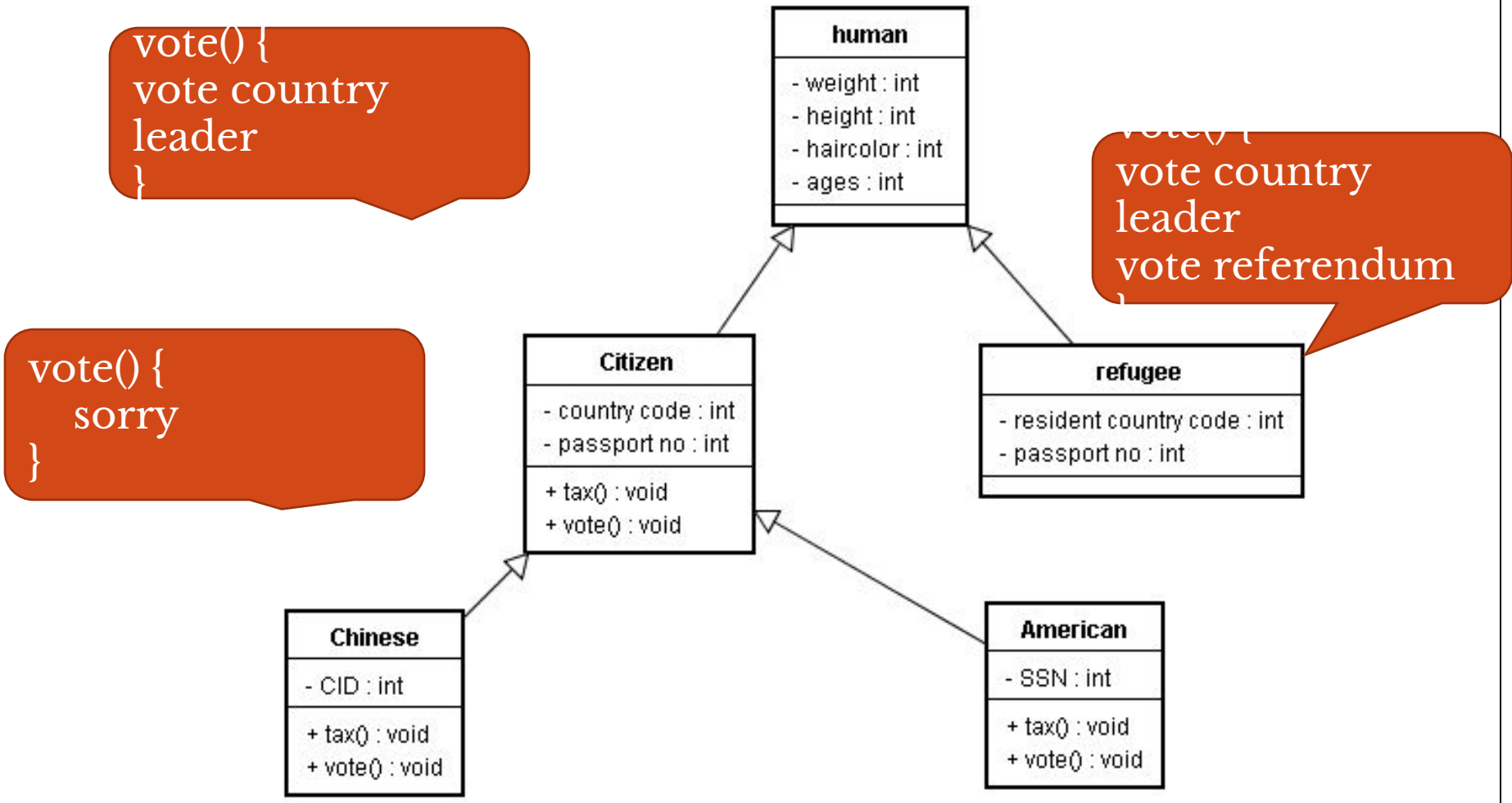
Do you want to keep going?



The answer

- No
  - The country code is already capable of distinguishing citizenships of country
- This subclassing is meaningless until

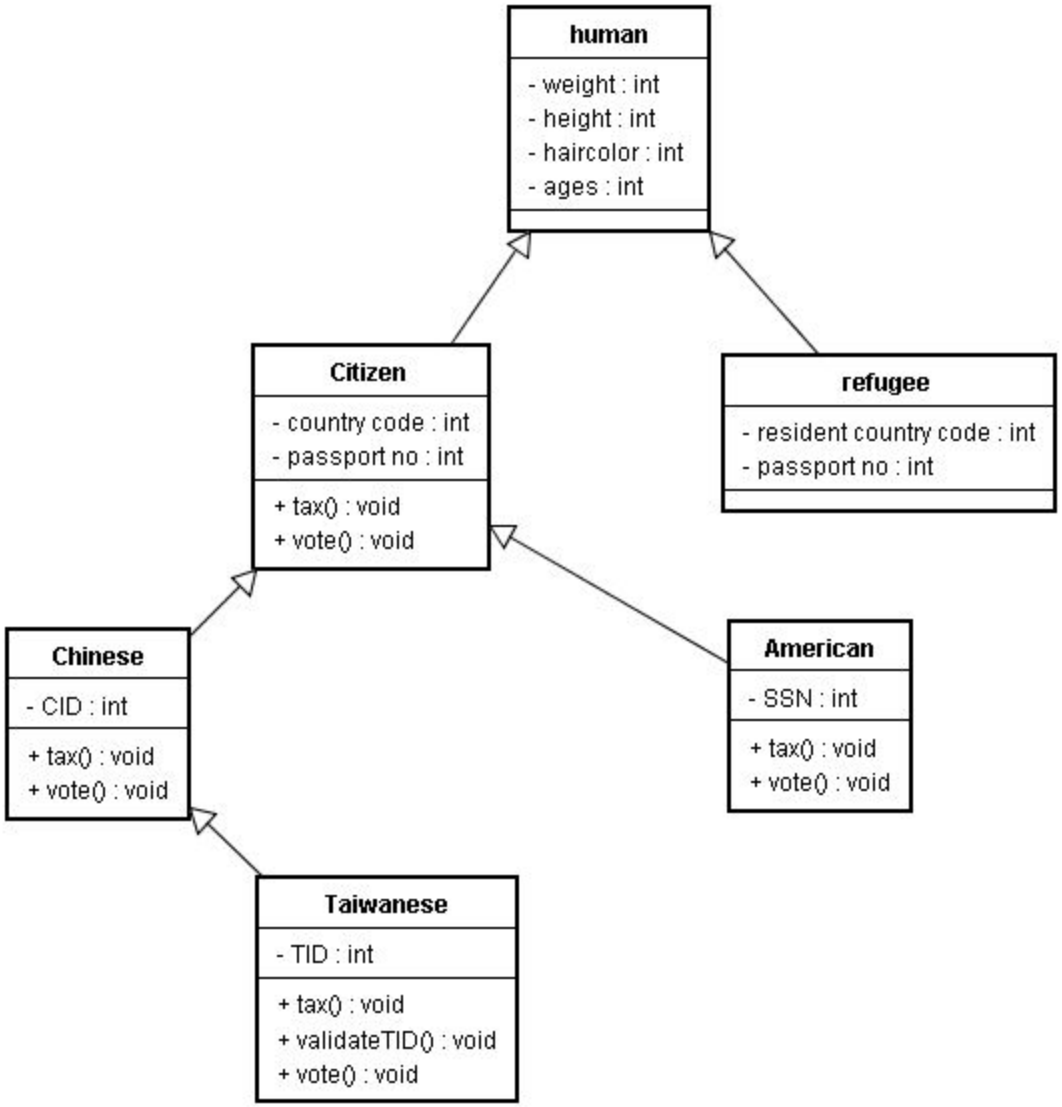
Until object's behaviors must be specialized and distinguished



Thumb Rule

- A citizen is a human?
- An American is a citizen?
  - => An american is a human?
  - Inheritance relation is transitive
- A Chinese is a citizen?
- A citizen is a human?
- A Chinese is a human?

To subclass or not to subclass?
The UN staff wants to follow one-China policy



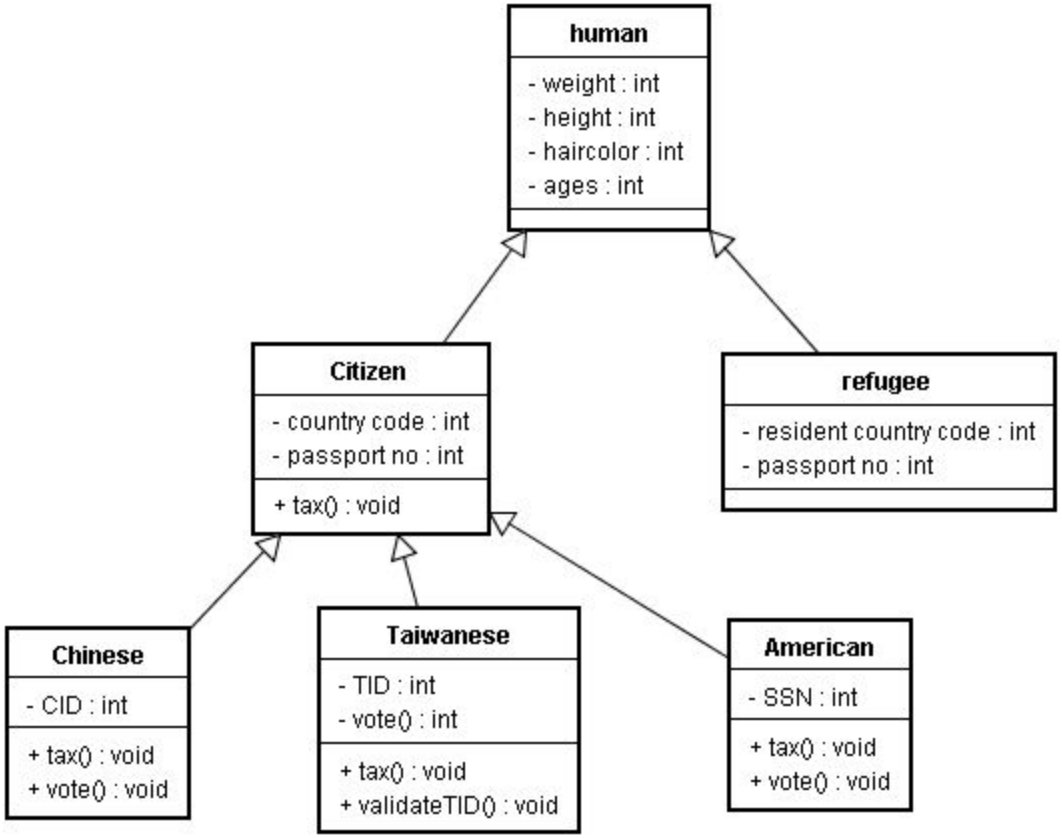
- Taiwanese all inherit CID
- Taiwanese overrides the tax() and vote() method
- Taiwanese has a specialized attributes called TID
- Taiwanese has a new method to validate if a TID is valid.

By so inheriting, the UN staff means

- Taiwanese is a Chinese, is a citizen, and is a human
- Chinese is not necessary a Taiwanese
- Taiwanese has an attribute called CID, but is never used (this is something wrong)

- OK..This is very heavy

How about this inheritance?



By this inheritance, you mean

- Taiwanese is not a Chinese
- But Taiwanese and Chinese are both citizens and human.
- Taiwanese does not inherit CID attributes

More exercise of overriding

```
#include <iostream>
using namespace std ;
class pet {

public:
 pet() { cout << "pet constructor" << endl ; }
 ~pet() { cout << "pet destructor" << endl ; }
 void speak() { cout << "Growl " << endl ; }
};

class cat: public pet {

public:
 cat() { cout << "cat constructor" << endl ; }
 ~cat() { cout << "cat destructor" << endl ; }
 void speak() { cout << "meow" << endl ; }

};
```

```
main() {
   pet insect ;
   cat pussy ;
   pet * nose = (pet *) new
    cat();

   insect.speak();
   pussy.speak();
   ((pet) pussy).speak();
   nose-> speak();
}
```

Output

```
ScreenTaker v3.10 - UNREGISTERED~
[ypc@romulans ~/cplus/t5]$ ./a
pet constructor
pet constructor
cat constructor
pet constructor
cat constructor
Growl
meow
Growl
pet destructor
Growl
cat destructor
pet destructor
pet destructor
```

Confused?

- When you use a subclass to override a base class's method, C++ will use the current type to determine the method
- This is not the polymorphism you expect.
- 當你用一個 subclass override 掉 base class 的 method 時, C++ 會根據目前物件的type來幫你呼叫methods
- 在運用多型的時候, 你會不想要讓這樣的事情發生, 要怎麼辦？