C++ Programming
Lesson 4

TINPRO02-5B

HOGESCHOOL ROTTERDAM

PROGRAMMA

Recap

Inheritance
Multiple inheritance
Abstract classes
Polymorphism
Virtual function overriding
Virtual destructors

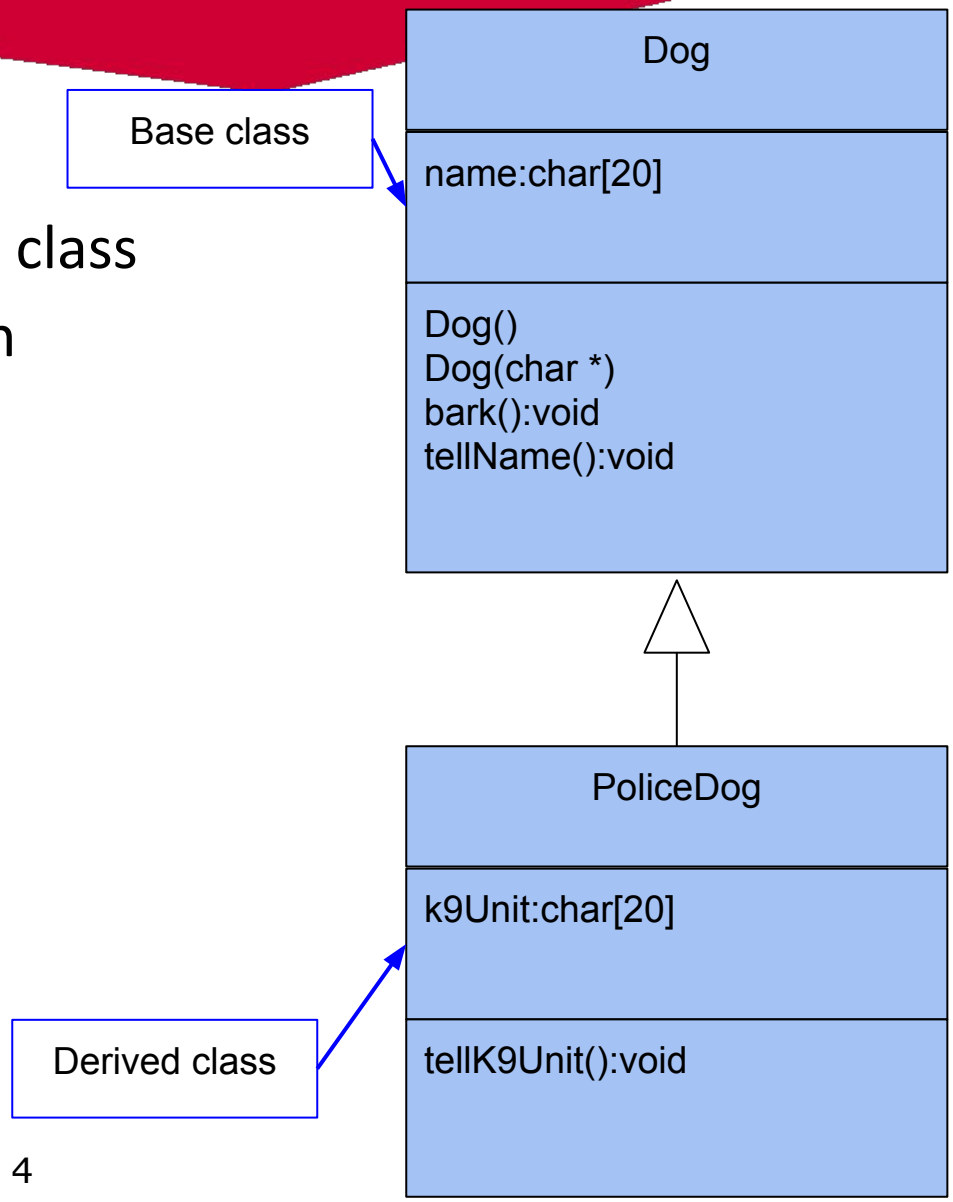
Recap Lesson 3

- Classes
- Constructors & destructors
- Function & constructor overloading
- Copy constructor

Inheritance

- Derived class inherits
  all the members of the base class
- Derived class can add its own members

```
class PoliceDog: public Dog{

};
```



Inheritance: what is inherited?

A publicly derived class inherits access to every member of a base class except:

- its constructors and its destructor
- its assignment operator members (operator=)

Even though the constructors and destructor of the base class are not inherited as such, the parameterless constructor of the base class is automatically called by the constructor of the derived class.

Inheritance: Java vs C++

Java:

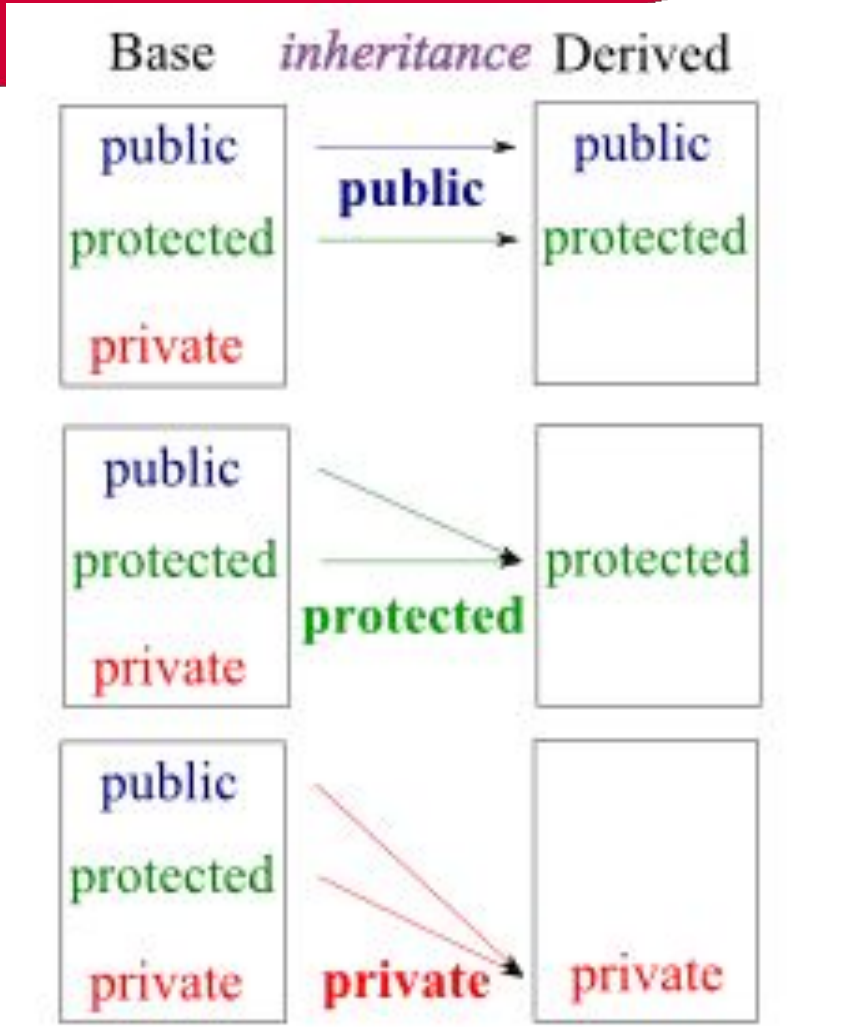
- The private members are not inherited
- If we want to call parameterized constructor then we must use super to call the parent constructor
- Does not support multiple inheritance
- Methods are virtual by default.

C++

- The private members are inherited
- If we want to call parameterized constructor of a parent class, we must use Initializer list.
- Supports multiple inheritance
- Explicitly use virtual keyword.

Not only public inheritance

- Protected inheritance:
  - all public and protected members of the base class become <u>protected</u> in the derived class
- Private inheritance:
  - all public and protected members of the base class become <u>private</u> in the derived class



Private inheritance pitfalls

```
class Person {};
class Student:private Person {};   // private
void eat(const Person& p){}        // anyone can eat
void study(const Student& s){}     // only students study

int main()
{
    Person p;     // p is a Person
    Student s;    // s is a Student
    eat(p);       // fine, p is a Person
    eat(s);       // error! s isn't a Person
    return 0;
}
```

Constructors in inherited classes

- The constructor is not inherited in the derived class, but can be called from the constructor of the derived class
- A derived class object has all the member variables of the base class. When a derived class constructor is called, these member variables need to be allocated memory and should be instantiated.
- This memory allocation must be done by a constructor: the most convenient place for this is the base class constructor

Constructors in inherited classes: example

```
class SuperClass{
    public:
        SuperClass(char &foo){
            // do something with foo
        }
};
class SubClass : public SuperClass{
    private:
         int bar;
    public:
        SubClass(char &foo, int b): SuperClass(foo), bar(b)    // Call the
superclass constructor in the subclass' initialization list.
        {
            // deliberately empty         }
};      }
};
```

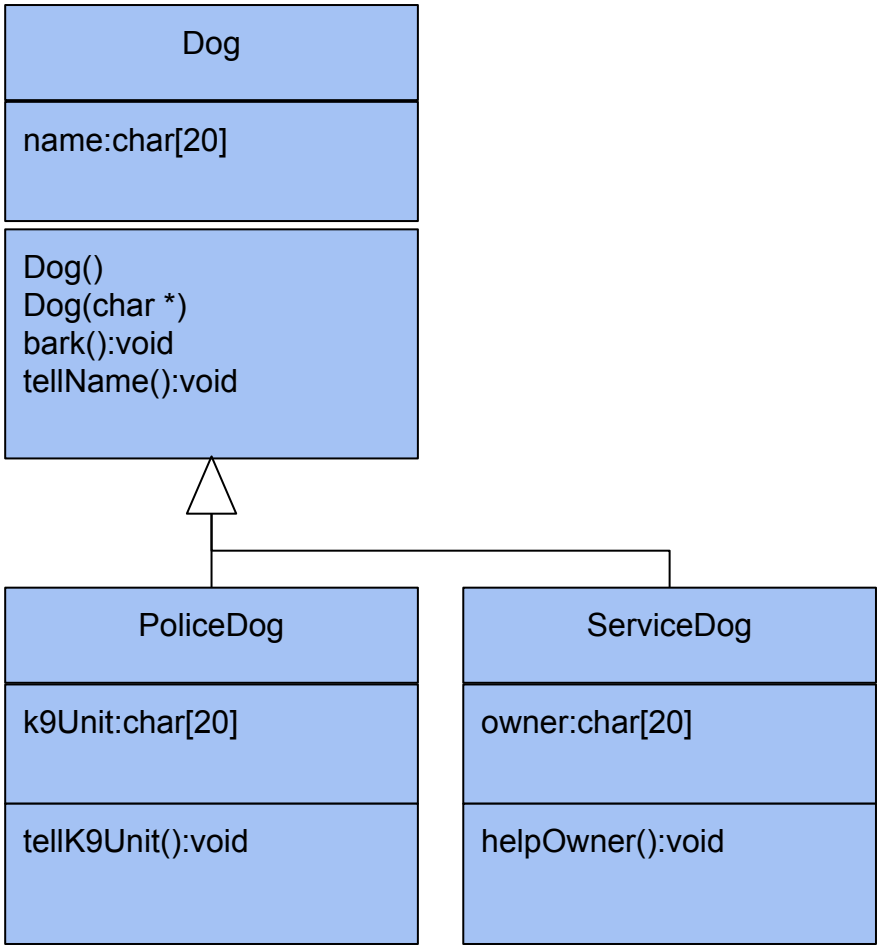
Pitfall: private members from base class

- Private members from the base class are inherited in derived classes
- However they are not directly accessible , not even in a member function definition for a derived class
- A private member can be accessed indirectly via an accessor or mutator member function
- A private member function is simply not available
- It is just as if the private members are not inherited

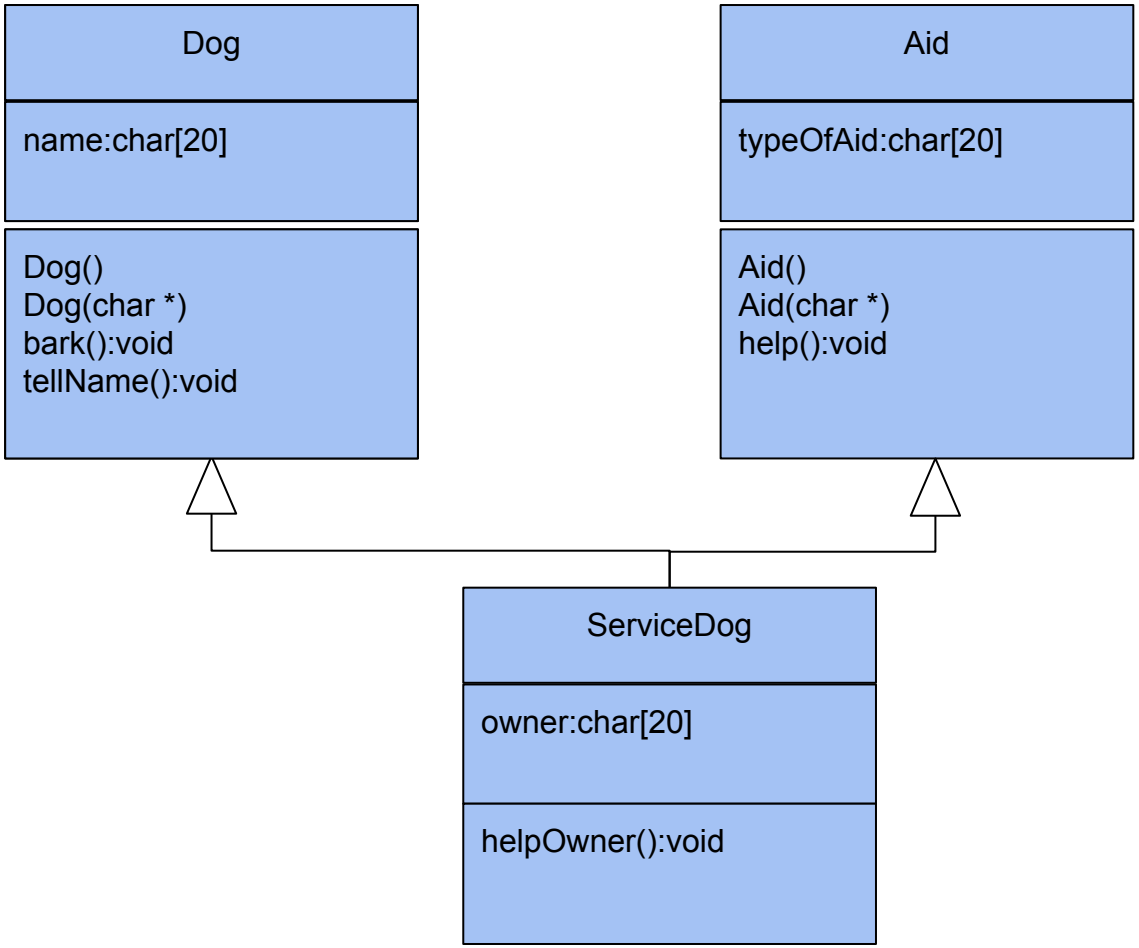
Encapsulation

- The fact that private member functions are not accessible by derived classes should not be a problem: their use should be limited to the class in which they are defined
- If you want this member function to be used as a helping function in a number of derived classes then it should be defined public
- Think of encapsulation->design decisions!

Multiple inheritance?



Multiple inheritance



Example of Multiple inheritance

```
class Dog{

};

class Aid{

};

class ServiceDog: Dog, Aid{

};
```

Polymorphism

- Derived from a Greek word meaning “many forms”
- In types (classes) related by inheritance, we can use “many forms” of these types while ignoring the differences among them
- When we have a pointer or reference to a base class and we call a function defined in this class, we don’t know the type of object on which that member is executed

Polymorphism

- Example 1: what happens?
- Example 2: polymorphism

Abstract base class

- Abstract base classes are classes that can only be used as base classes,
- They are allowed to have virtual member functions without definition (known as **pure virtual functions**).

```
// abstract class CPolygon
class Polygon {
  protected:
    int width, height;
  public:
    void set_values (int a, int b)
      { width=a; height=b; }
    virtual int area () =0;
};
```

Abstract base class

- Cannot be instantiated
- Must have derived classes
- In the derived classes the pure virtual methods of the base (abstract class) are implemented

```
class Rectangle: public Polygon {
    public:
      int area (void)
        { return (width * height); }
};

class Triangle: public Polygon {
    public:
      int area (void)
        { return (width * height / 2); }
};
```

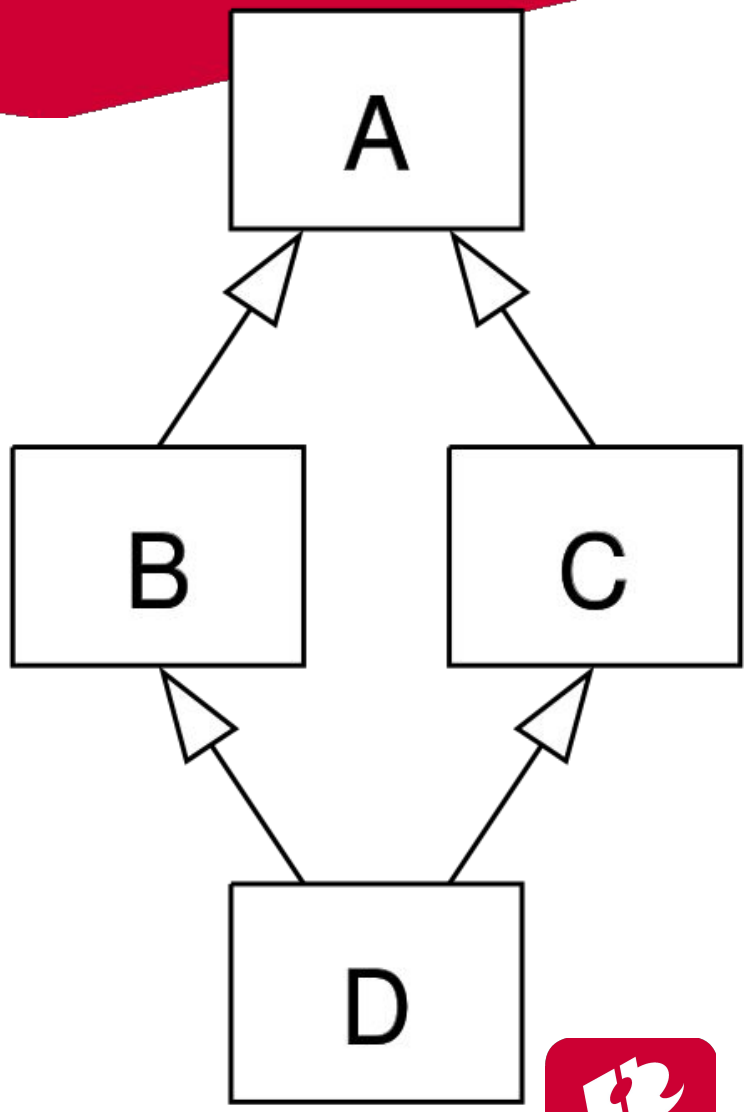
Polymorphism: list of pointers to ancestors

```
Polygon *polygonList[10];

...//create polygons with new

for (int i = 0; i<10; i++){
    cout<< polygonList[i]->area();
}
```

The diamond problem

The "**diamond problem**" is an ambiguity that arises when two classes B and C inherit from A, and class D inherits from both B and C.

If there is a method in A that B and C have overridden, and D does not override it, then which version of the method does D inherit: that of B, or that of C?



The diamond problem: code error

```
class Animal  // base class{
```

```
funct.cpp:31:20: error: non-static member 'getWeight' found in multiple base-class subobjects of type 'Animal':
    class Liger -> class Tiger -> class Animal
    class Liger -> class Lion -> class Animal
   int weight = lg.getWeight();
                   ^
funct.cpp:15:8: note: member found by ambiguous name lookup
   int getWeight() { return weight;};
       ^
1 error generated.
```

```
        /*COMPILE ERROR, the code below will not get past any C++ compiler */

        int weight = lg.getWeight();
```

The diamond problem

And here is the problem: **because Liger derives from both the Tiger and Lion classes – which each have their own copy of the data members and methods of the Animal class- the Liger object "lg" will contain *two* subobjects of the *Animal* base class.**

Virtual inheritance

- Virtual inheritance solves the classic “Diamond Problem”.
- It ensures that the child class gets only a single instance of the common base class.
- The members of the base class will not appear twice in the derived class

The diamond problem solved

```
class Animal  // base class{
   int weight;
   public:
   int getWeight() { return weight;};
};

class Tiger : virtual public Animal { /* ... */ };
class Lion : virtual public Animal { /* ... */ };

class Liger : public Tiger, public Lion { /* ... */ };

int main(){
   Liger lg ;
   int weight = lg.getWeight();
}
```

Virtual function overriding

- Virtual functions ensure that the correct function is called for an object, regardless of the type of reference (or pointer) used for function call.
- They are mainly used to achieve Runtime polymorphism
- Functions are declared with a **virtual** keyword in base class.
- The resolving of function call is done at Run-time.

Rules for virtual function overriding

1. Virtual functions cannot be static and also cannot be a friend function of another class.
2. Virtual functions should be accessed using pointer or reference of base class type to achieve run time polymorphism.
3. The prototype of virtual functions should be same in base as well as derived class.
4. They are always defined in base class and overridden in derived class. It is not mandatory for derived class to override (or re-define the virtual function), in that case base class version of function is used.
5. A class may have virtual destructor but it cannot have a virtual constructor.

Virtual destructor

- Deleting a derived class object using a pointer to a base class that has a non-virtual destructor results in undefined behavior.
- To correct this situation, the base class should be defined with a virtual destructor.

Example

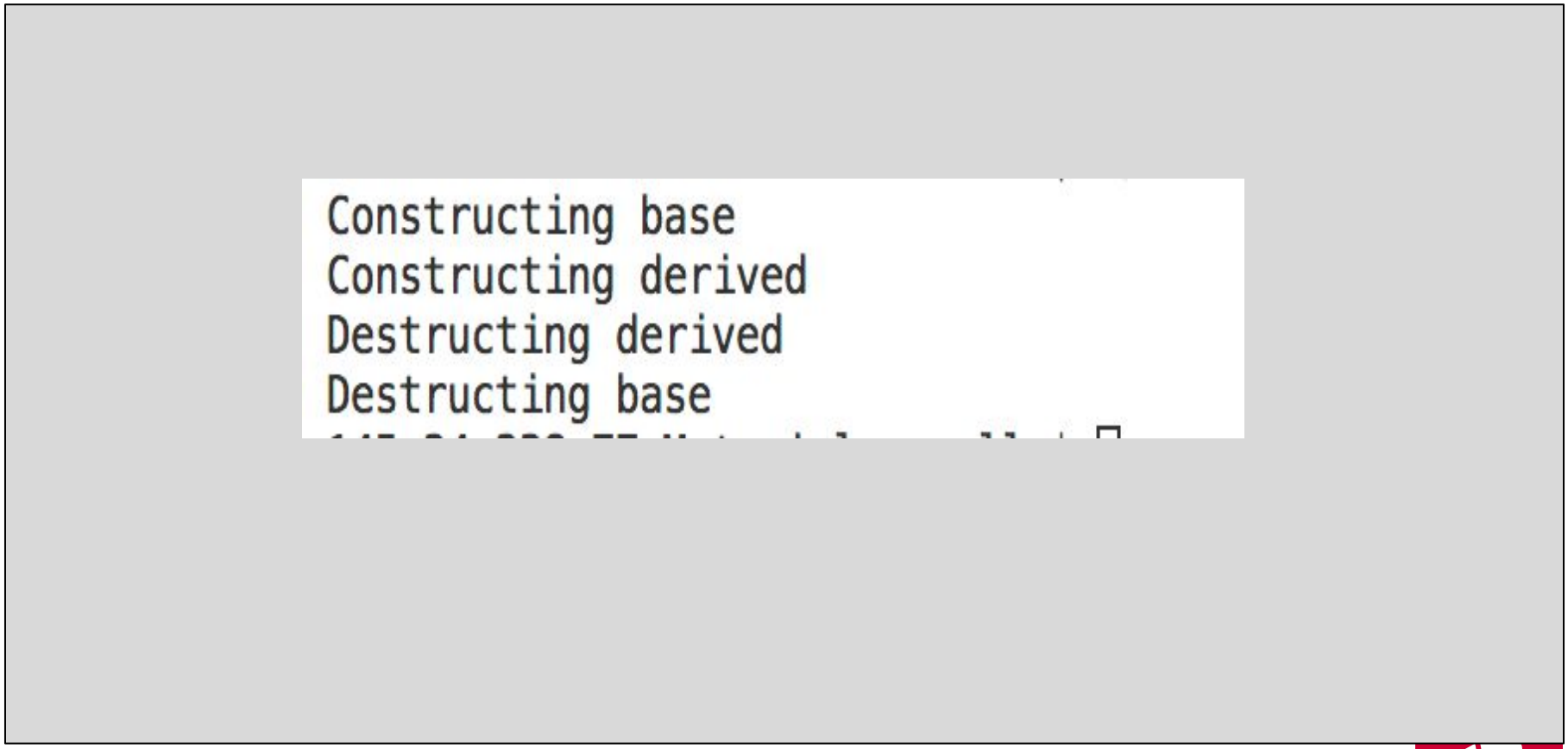
```
class Base {
```

```
Constructing base
Constructing derived
Destructing base
```

```
    delete b;
    return 0;
}
```

Example: correct

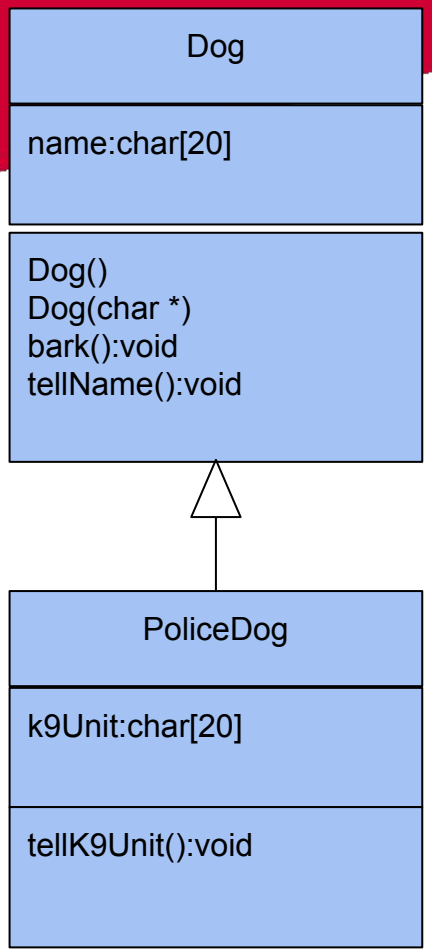
```
class Base {
```




```
    return 0;
}
```

Try it out!

- Create a class dog. Add a destructor to the class that prints “object is going to be destroyed”
- Add a class PoliceDog according to the class diagram
- *Create an array of pointers to Dog, and fill it in with many instances (created in the heap) of Dog and PoliceDog. Delete all the objects in the correct way*



Solution

Example of solution

DerivedDogs

Try it out!

- Write a class named `Circle`. `Circle` should have two private member variables with default values: `color ("Black")` and `radius (10.0)`. Provide constructors to set only color, set only radius, set both, or set neither value, and provide a destructor. Also write functions to print out the color and radius of the circle.
- Make a derived class named `Oval`. Create first a class diagram on paper before you start coding: which attributes do you need to add to `Oval`?

Solution

Example of solution

CircleOval

Try it out

- Use the Circle class and add a new class called CircleList which contains a list of pointers to Circle instances, use an array for this.
- CircleList has 3 constructors, an empty one, one where you give an array and one where you give an amount of Circles and a color (defaults to black) and it will generate `n` Circles with random sizes between 10 and 20.
- Also add methods to add a new Circle, remove a Circle, clear the whole pit and to print information about all the Circles in the list
- Make sure the circles are removed from the memory using `delete`

overtref jezelf