Maciej Sobieraj

Lecture 5

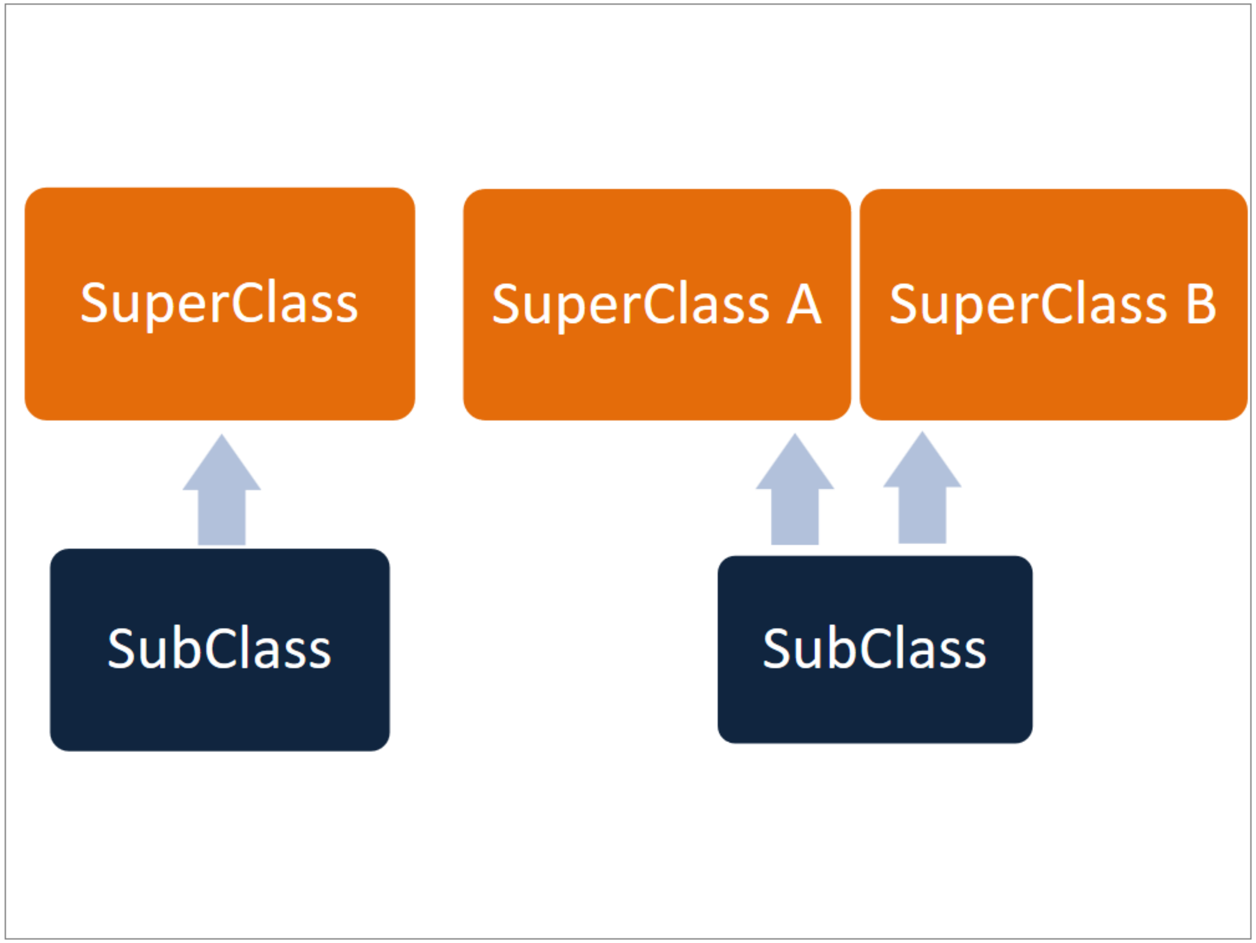
Outline

1. **Inheritance**
   1. **Defining class hierarchy**
   2. Classes, inheritance and type compatibility
   3. Polymorphism and virtual methods
   4. Objects as parameters and dynamic casting
   5. Various supplements

Defining a simple subclass

- We can use each class as a base (or a foundation) to define or build another class (**a subclass**).
- It’s also possible to use **more than one class to define a subclass**.
- We can also write about superclasses as **base** classes, and subclasses as **derived** classes.

Defining a simple subclass



Defining a simple subclass

- This class will serve as **a superclass**

```
#include <iostream>
using namespace std;
class Super {
private:
	int storage;
public:
	void put(int val) { storage = val; }
	int get(void) { return storage; }
};
int main(void) {
	Super object;

	object.put(100);
	object.put(object.get() + 1);
	cout << object.get() << endl;
	return 0;
}
```

Defining a simple subclass

- If we want to **define a class named Y as a subclass of a superclass named X**, we use the following syntax

```
class Y : {visibility specifier} X { … };
```

- If there’s more than one superclass, we have to enlist them all using commas as separators, like this:
  - **class** A : X, Y, Z { … };

Defining a simple subclass

- The *Sub* class introduces neither new variables nor new functions.
- When we **omit the visibility specifier**, the compiler assumes that we’re going to apply a “**private inheritance**”.

```
class Sub : Super {
};

int main(void) {
        Sub object;

        object.put(100);
        object.put(object.get() + 1);
        cout << object.get() << endl;
        return 0;
}
```

Defining a simple subclass

- “**private inheritance**” means that **all public superclass components turn into private access, and private superclass components won't be accessible** at all.
- We have to tell the compiler that **we want to preserve the previously used access policy.** We do this by using a “public” visibility specifier:

```
class Sub : public Super {
};
```

Defining a simple subclass

- Subclass **has lost access to the private components of the superclass**.
- We cannot write a member function of the *Sub* class which would be able to directly manipulate the *storage* variable.

Defining a simple subclass

```
#include <iostream>

using namespace std;

class Super {
private:
	int storage;
public:
	void put(int val) { storage = val; }
	int get(void) { return storage; }
};

class Sub : public Super {
};

int main(void) {
	Sub object;

	object.put(100);
	object.put(object.get() + 1);
	cout << object.get() << endl;
	return 0;
}
```

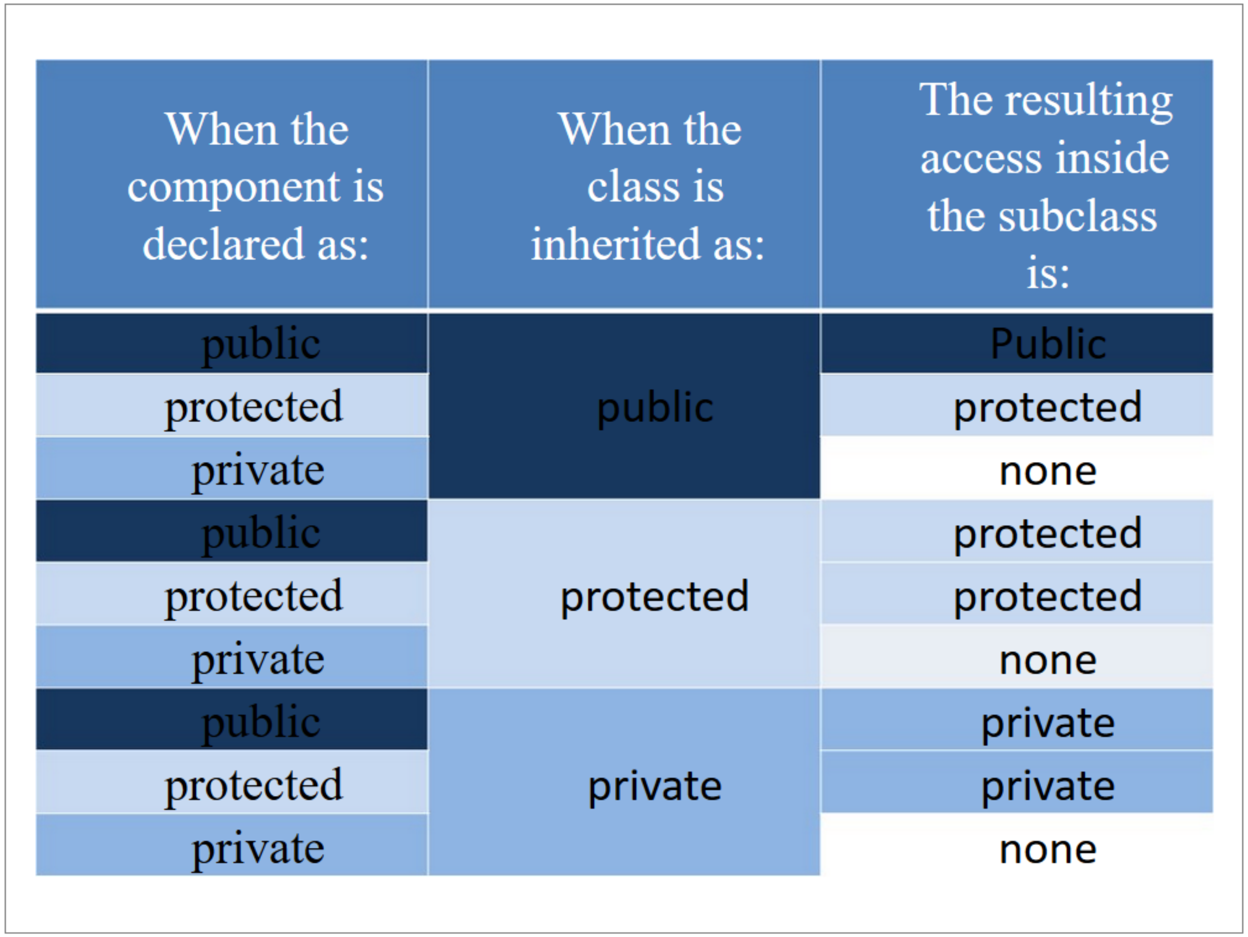
Defining a simple subclass

- The keyword *protected* means that any component marked with it **behaves like a public component when used by any of the subclasses and looks like a private component to the rest of the world**.

Defining a simple subclass

```
#include <iostream>

using namespace std;

class Super {
protected:
	int storage;
public:
	void put(int val) { storage = val; }
	int get(void) { return storage; }
};

class Sub : public Super {
public:
	void print(void) { cout << "storage = " << storage << endl; }
};

int main(void) {
	Sub object;

	object.put(100);
	object.put(object.get() + 1);
	object.print();
	return 0;
}
```

Defining a simple subclass



When the component is declared as:	When the class is inherited as:	The resulting access inside the subclass is:
public	public	Public
protected		protected
private		none
public	protected	protected
protected		protected
private		none
public	private	private
protected		private
private		none

Defining a simple subclass

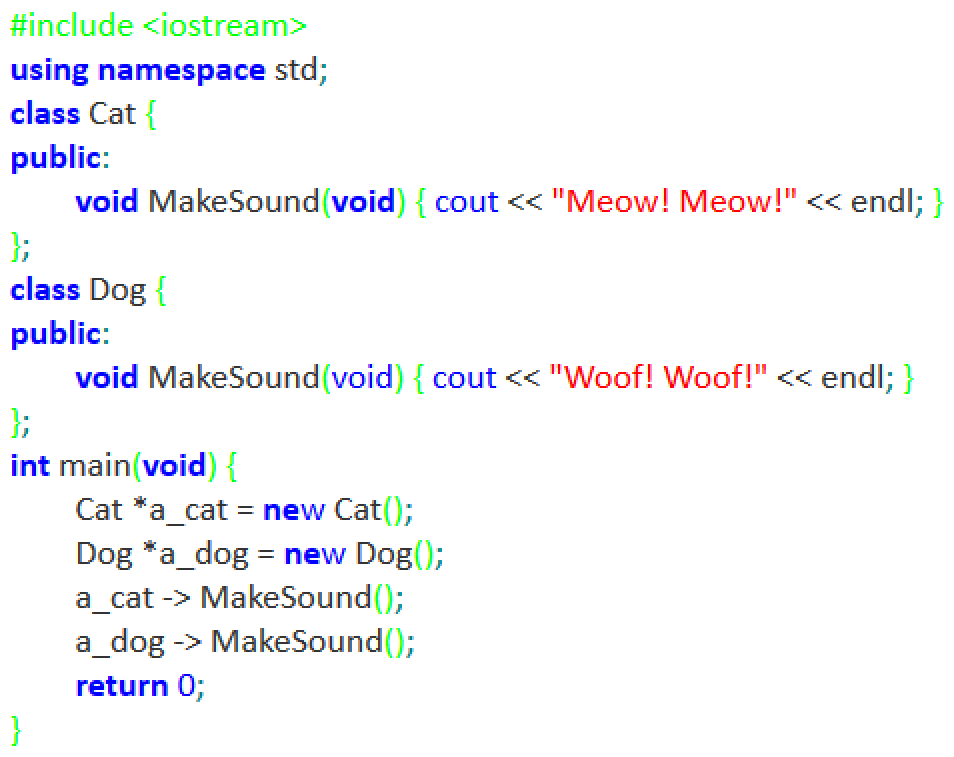
```
#include <iostream>
using namespace std;
class SuperA {
protected:
    int storage;
public:
    void put(int val) { storage = val; }
    int get(void) { return storage; }
};
class SuperB {
protected:
    int safe;
public:
    void insert(int val) { safe = val; }
    int takeout(void) { return safe; }
};
class Sub : public SuperA, public SuperB {
public:
    void print(void) {
        cout << "storage = " << storage << endl;
        cout << "safe    = " << safe << endl;
    }
};
int main(void) {
    Sub object;

    object.put(1);    object.insert(2);
    object.put(object.get() + object.takeout());
    object.insert(object.get() + object.takeout());
    object.print();
    return 0;
}
```

Outline

Type compatibility – the simplest case

- **Each new class constitutes a new type of data**. Each object constructed on the basis of such a class is like **a value of the new type**.
- This means that **any two objects may (or may not) be compatible** in the sense of their types.

Type compatibility – the simplest case



```
#include <iostream>
using namespace std;
class Cat {
public:
	void MakeSound(void) { cout << "Meow! Meow!" << endl; }
};
class Dog {
public:
	void MakeSound(void) { cout << "Woof! Woof!" << endl; }
};
int main(void) {
	Cat *a_cat = new Cat();
	Dog *a_dog = new Dog();
	a_cat -> MakeSound();
	a_dog -> MakeSound();
	return 0;
}
```

Type compatibility – the simplest case

- The objects of the *Cat* class are not compatible with the objects of the *Dog* class, although the structure of both classes is identical. Neither of the following assignments is valid and both of them will **cause a compiler error**:
  - a_dog = a_cat;
  - a_cat = a_dog;
- **Objects derived from classes which lie in different branches of the inheritance tree are always incompatible**.

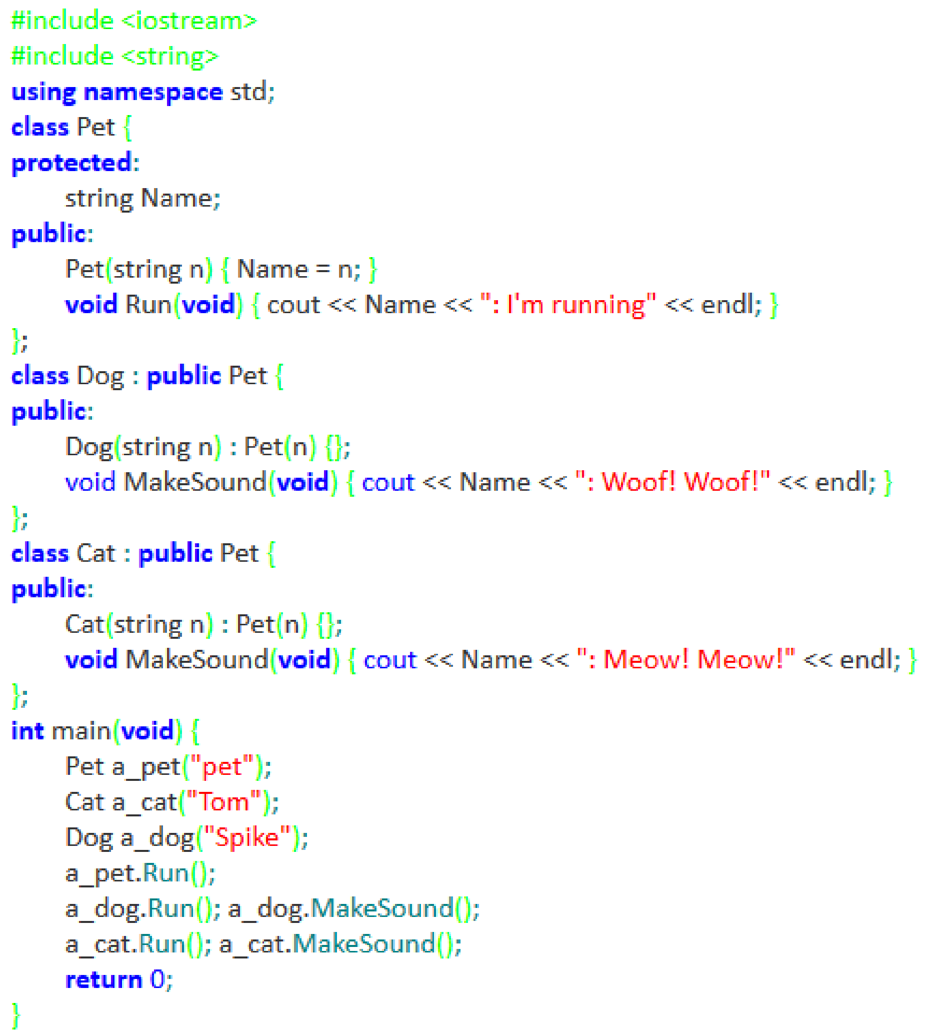
Type compatibility – more complex case

- The *Dog* and *Cat* classes are now **descendants** (to be precise, **children**) of the common base class *Pet*.
- We've also equipped all the classes with constructors.
- Our pets are also able to run.

Type compatibility – more complex case

- Let’s summarize what we created.
  - objects derived from the *Pet* class are **able to run**
  - objects derived from the *Dog* and *Cat* classes are **able to run** (they inherit this ability from their superclass); they can also **make sounds** (note that this skill is not available for objects of the *Pet* class)
- And so:
  - *Cat* and *Dog* objects can do all the things *Pets* are able to do
  - *Pets* cannot do all the thing that *Cat* and *Dog* can do

Type compatibility – more complex case



```
#include <iostream>
#include <string>
using namespace std;
class Pet {
protected:
	string Name;
public:
	Pet(string n) { Name = n; }
	void Run(void) { cout << Name << ": I'm running" << endl; }
};
class Dog : public Pet {
public:
	Dog(string n) : Pet(n) {};
	void MakeSound(void) { cout << Name << ": Woof! Woof!" << endl; }
};
class Cat : public Pet {
public:
	Cat(string n) : Pet(n) {};
	void MakeSound(void) { cout << Name << ": Meow! Meow!" << endl; }
};
int main(void) {
	Pet a_pet("pet");
	Cat a_cat("Tom");
	Dog a_dog("Spike");
	a_pet.Run();
	a_dog.Run(); a_dog.MakeSound();
	a_cat.Run(); a_cat.MakeSound();
	return 0;
}
```

Type compatibility – more complex case

- We can assume:
  - **objects of the subclass have at least the same capabilities as the superclass objects**
  - **objects of the superclass may not have the same capabilities as the subclass objects**
- This leads us to the following conclusion:
  - **objects of the superclass <u>are compatible</u> with objects of the subclass**
  - **objects of the subclass <u>are not compatible</u> with objects of the superclass**

Type compatibility – more complex case

- This means that:
  - you **can** do the following:
    - a_pet = **new** Dog("Huckleberry");
    - a_pet -> Run();
  - but you **cannot** do anything like this:
    - ~~a_pet -> MakeSound();~~
  - because *Pets* don’t know how to make sounds (in our world of classes, at least)
  - you are **not allowed** to do the following:
    - ~~a_dog = new Pet("Strange pet");~~

Type compatibility – more complex case

```
#include <iostream>
#include <string>
using namespace std;
class Pet {
protected:
	string Name;
public:
	Pet(string n) { Name = n; }
	void Run(void) { cout << Name << ": I'm running" << endl; }
};
class Dog : public Pet {
public:
	Dog(string n) : Pet(n) {};
	void MakeSound(void) { cout << Name << ": Woof! Woof!" << endl; }
};
class Cat : public Pet {
public:
	Cat(string n) : Pet(n) {};
	void MakeSound(void) { cout << Name << ": Meow! Meow!" << endl; }
};
int main(void) {
	Pet *a_pet1 = new Cat("Tom");
	Pet *a_pet2 = new Dog("Spike");

	a_pet1 -> Run();
	// 'a_pet1 -> MakeSound();' is not allowed here!
	a_pet2 -> Run();
	// 'a_pet2 -> MakeSound();' is not allowed here!

	return 0;
}
```

ype compatibility – how to recover the lost

- Why are we not allowed to command our pet to make a sound?
- The problem comes from **static checks made by the compiler during the compilation process**.
- The compiler is convinced that pets cannot make sounds and won’t allow us even to try to do that.

Type compatibility – how to recover the lost

- We can do this using the **cast operators**.

```
static_cast<target_type>(an_expression)
```

  - static_cast<Dog *>(a_pet)
- forces the compiler to assume that *a_pet* is (temporarily) converted into a pointer of type *Dog **.

Type compatibility – back to our pets

```
#include <iostream>
#include <string>
using namespace std;
class Pet {
protected:
        string Name;
public:
        Pet(string n) { Name = n; }
        void Run(void) { cout << Name << ": I'm running" << endl; }
};
class Dog : public Pet {
public:
        Dog(string n) : Pet(n) {};
        void MakeSound(void) { cout << Name << ": Woof! Woof!" << endl; }
};
class Cat : public Pet {
public:
        Cat(string n) : Pet(n) {};
        void MakeSound(void) { cout << Name << ": Meow! Meow!" << endl; }
};
int main(void) {
        Pet *a_pet1 = new Cat("Tom");
        Pet *a_pet2 = new Dog("Spike");
        a_pet1 -> Run();
        static_cast<Cat *>(a_pet1) -> MakeSound();
        a_pet2 -> Run();
        static_cast<Dog *>(a_pet2) -> MakeSound();
        return 0;
}
```

Type compatibility – abusing owner’s power

```
#include <iostream>
#include <string>
using namespace std;
class Pet {
protected:
	string Name;
public:
	Pet(string n) { Name = n; }
	void Run(void) { cout << Name << ": I'm running" << endl; }
};
class Dog : public Pet {
public:
	Dog(string n) : Pet(n) {};
	void MakeSound(void) { cout << Name << ": Woof! Woof!" << endl; }
};
class Cat : public Pet {
public:
	Cat(string n) : Pet(n) {};
	void MakeSound(void) { cout << Name << ": Meow! Meow!" << endl; }
};
int main(void) {
	Pet *a_pet1 = new Cat("Tom");
	Pet *a_pet2 = new Dog("Spike");
	a_pet2 -> Run();
	static_cast<Cat *>(a_pet2) -> MakeSound();
	a_pet1 -> Run();
	static_cast<Dog *>(a_pet1) -> MakeSound();
	return 0;
}
```

Type compatibility – abusing owner’s power

- The compiler **isn’t able to check if the pointer being converted is compatible with the object it points to**.
- **Full pointer validity verification is possible when and only when the program is being executed**
- The “C++” language has a second conversion operator designed especially for this case.
- Its name is somewhat suggestive: ***dynamic_cast***.

Type compatibility – final case

- The rule stating that **objects lying at higher levels are compatible with objects at lower levels** of the class hierarchy works even when the **inheritance chain is arbitrarily long**.

Type compatibility – final case

```
#include <iostream>
#include <string>
using namespace std;
class Pet {
protected:
	string Name;
public:
	Pet(string n) { Name = n; }
	void Run(void) { cout << Name << ": I'm running" << endl; }
};
class Cat : public Pet {
public:
	Cat(string n) : Pet(n) {};
	void MakeSound(void) { cout << Name << ": Meow! Meow!" << endl; }
};
class Persian : public Cat {
public:
	Persian(string n) : Cat(n) {};
};
int main(void) {
	Pet  *a_pet;
	Persian *a_persian;
	a_pet = a_persian = new Persian("Mr. Bigglesworth");
	a_persian -> MakeSound();
	static_cast<Persian *>(a_pet) -> MakeSound();
	return 0;
}
```

Outline

1. **Inheritance**
    1. Defining class hierarchy
    2. Classes, inheritance and type compatibility
    3. **Polymorphism and virtual methods**
    4. Objects as parameters and dynamic casting
    5. Various supplements

Overriding a method in the subclass

- When a subclass **declares a method of the name previously known in its superclass, the original method is overridden**.
- **The effects of the overriding may be reversed** (or voided) if you use the *static_cast* operator in reverse.

Overriding a method in the subclass

```
#include <iostream>
using namespace std;
class Pet {
protected:
    string Name;
public:
    Pet(string n) { Name = n; }
    void MakeSound(void) { cout << Name << " the Pet says: Shh! Shh!" << endl; }
};
class Cat : public Pet {
public:
    Cat(string n) : Pet(n) { }
    void MakeSound(void) { cout << Name << " the Cat says: Meow! Meow!" << endl; }
};
class Dog : public Pet {
public:
    Dog(string n) : Pet(n) { }
    void MakeSound(void) { cout << Name << " the Dog says: Woof! Woof!" << endl; }
};
int main(void) {
    Cat *a_cat;
    Dog *a_dog;

    a_cat = new Cat("Kitty");
    a_dog = new Dog("Doggie");
    a_cat -> MakeSound();
    static_cast<Pet *>(a_cat) -> MakeSound();
    a_dog -> MakeSound();
    static_cast<Pet *>(a_dog) -> MakeSound();
    return 0;
}
```

Overriding a method in the subclass

```
#include <iostream>
using namespace std;
class Pet {
protected:
	string Name;
public:
	Pet(string n) { Name = n; }
	void MakeSound(void) { cout << Name << " the Pet says: Shh! Shh!" << endl; }
};
class Cat : public Pet {
public:
	Cat(string n) : Pet(n) { }
	void MakeSound(void) { cout << Name << " the Cat says: Meow! Meow!" << endl; }
};
class Dog : public Pet {
public:
	Dog(string n) : Pet(n) { }
	void MakeSound(void) { cout << Name << " the Dog says: Woof! Woof!" << endl; }
};
int main(void) {
	Pet *a_pet1, *a_pet2;
	Cat *a_cat;
	Dog *a_dog;

	a_pet1 = a_cat = new Cat("Kitty");
	a_pet2 = a_dog = new Dog("Doggie");
	a_pet1 -> MakeSound();
	a_cat  -> MakeSound();
	a_pet2 -> MakeSound();
	a_dog  -> MakeSound();
	return 0;
}
```

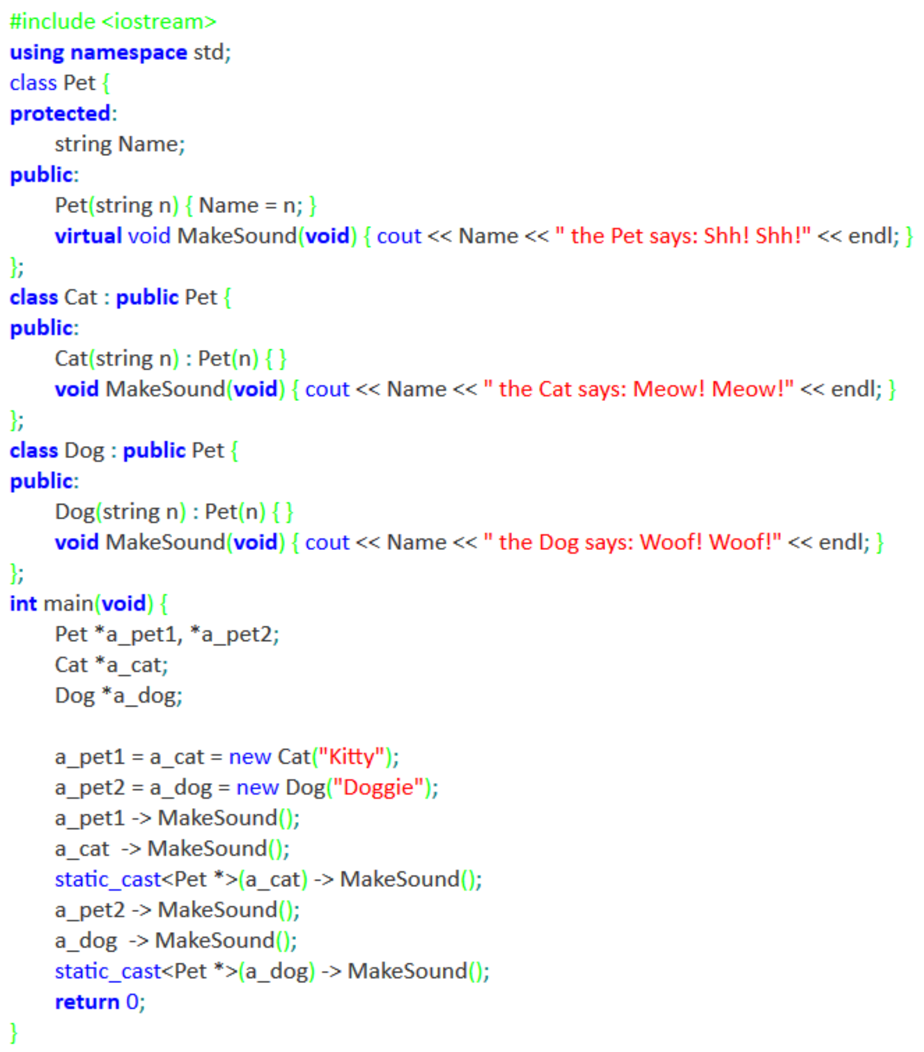
Overriding a method in the subclass

- **Polymorphism** is **a method to redefine the behaviour of a superclass** (but only the one that explicitly agrees to be treated in this way!) **without touching its implementation**.
- The word "**polymorphism**" means that the one and same class may show many ("**poly**" – like in "*polygamy*") forms ("*morphs*") not defined by the class itself, but **by its subclasses**.

Overriding a method in the subclass

- The word **virtual** means that the method will be **redefined** (replaced) at the level of the original class.

Overriding a method in the subclass



```
#include <iostream>
using namespace std;
class Pet {
protected:
    string Name;
public:
    Pet(string n) { Name = n; }
    virtual void MakeSound(void) { cout << Name << " the Pet says: Shh! Shh!" << endl; }
};
class Cat : public Pet {
public:
    Cat(string n) : Pet(n) { }
    void MakeSound(void) { cout << Name << " the Cat says: Meow! Meow!" << endl; }
};
class Dog : public Pet {
public:
    Dog(string n) : Pet(n) { }
    void MakeSound(void) { cout << Name << " the Dog says: Woof! Woof!" << endl; }
};
int main(void) {
    Pet *a_pet1, *a_pet2;
    Cat *a_cat;
    Dog *a_dog;

    a_pet1 = a_cat = new Cat("Kitty");
    a_pet2 = a_dog = new Dog("Doggie");
    a_pet1 -> MakeSound();
    a_cat  -> MakeSound();
    static_cast<Pet *>(a_cat) -> MakeSound();
    a_pet2 -> MakeSound();
    a_dog  -> MakeSound();
    static_cast<Pet *>(a_dog) -> MakeSound();
    return 0;
}
```

Overriding a method in the subclass

```
#include <iostream>
using namespace std;
class Pet {
protected:
        string Name;
public:
        Pet(string n) { Name = n; MakeSound(); }
        virtual void MakeSound(void) { cout << Name << " the Pet says: Shh! Shh!" << endl; }
};
class Cat : public Pet {
public:
        Cat(string n) : Pet(n) { }
        void MakeSound(void) { cout << Name << " the Cat says: Meow! Meow!" << endl; }
};
class Dog : public Pet {
public:
        Dog(string n) : Pet(n) { }
        void MakeSound(void) { cout << Name << " the Dog says: Woof! Woof!" << endl; }
};
int main(void) {
        Cat *a_cat;
        Dog *a_dog;

        a_cat = new Cat("Kitty");
        a_dog = new Dog("Doggie");

        return 0;
}
```

Overriding a method in the subclass

- We invoke the *MakeSound* method as part of the *Pet* constructor.
- The program will output the following lines:
  - Kitty the Pet says: Shh! Shh!
  - Doggie the Pet says: Shh! Shh!
- This means that the binding between the original functions and their polymorphic implementations is established when the subclass object is created, not sooner.

Overriding a method in the subclass

```
#include <iostream>
using namespace std;
class Pet {
protected:
	string Name;
public:
	Pet(string n) { Name = n; }
	virtual void MakeSound(void) { cout << Name << " the Pet says: Shh! Shh!" << endl; }
	void WakeUp(void) { MakeSound(); }
};
class Cat : public Pet {
public:
	Cat(string n) : Pet(n) { }
	void MakeSound(void) { cout << Name << " the Cat says: Meow! Meow!" << endl; }
};
class Dog : public Pet {
public:
	Dog(string n) : Pet(n) { }
	void MakeSound(void) { cout << Name << " the Dog says: Woof! Woof!" << endl; }
};
int main(void) {
	Cat *a_cat;
	Dog *a_dog;

	a_cat = new Cat("Kitty");
	a_cat -> WakeUp();
	a_dog = new Dog("Doggie");
	a_dog -> WakeUp();

	return 0;
}
```

Overriding a method in the subclass

- The virtual method may be invoked not only from outside the class but also from within.
- The code produces the following output:
  - Kitty the Cat says: Meow! Meow!
  - Doggie the Dog says: Woof! Woof!

Outline

1. **Inheritance**
   1. Defining class hierarchy
   2. Classes, inheritance and type compatibility
   3. Polymorphism and virtual methods
   4. **Objects as parameters and dynamic casting**
   5. Various supplements

Passing an object as a function parameter

- **Any object may be used as a function parameter** and, vice versa, **any function may have a parameter as an object of any class**.
- We can pass an object into a function: **by pointer** and **by reference**.

Passing an object as a function parameter

```
#include <iostream>
#include <string>
using namespace std;
class Pet {
protected:
	string name;
public:
	void NameMe(string name) { this -> name = name; }
	void MakeSound(void) { cout << name << " says: no comments" << endl; }
};
void PlayWithPetByPointer(string name, Pet *pet) {
	pet -> NameMe(name);
	pet -> MakeSound();
}
void PlayWithPetByReference(string name, Pet &pet) {
	pet.NameMe(name);
	pet.MakeSound();
}
int main(void) {
	Pet *p1  = new Pet;
	Pet p2;
	PlayWithPetByPointer("anonymous", p1);
	PlayWithPetByReference("no_name", p2);
	PlayWithPetByPointer("no_name", &p2);
	PlayWithPetByReference("anonymous", *p1);
	return 0;
}
```

Passing an object by value

```cpp
#include<iostream>
#include <string>
using namespace std;
class Pet {
protected:
    string name;
public:
    void NameMe(string name) { this -> name = name; }
    void MakeSound(void) { cout << name << " says: no comments" << endl; }
};
void NamePetByValue(string name, Pet pet) {
    pet.NameMe(name);
}
void NamePetByPointer(string name, Pet *pet) {
    pet -> NameMe(name);
}
void NamePetByReference(string name, Pet &pet) {
    pet.NameMe(name);
}
int main(void) {
    Pet pet;
    pet.NameMe("no_name");
    NamePetByValue("Alpha", pet);
    pet.MakeSound();
    NamePetByPointer("Beta", &pet);
    pet.MakeSound();
    NamePetByReference("Gamma", pet);
    pet.MakeSound();
    return 0;
}
```

Passing an object of a subclass

```
#include <iostream>
#include <string>
using namespace std;
class Pet {
protected: string name;
public:    Pet(string name) { this -> name = name; }
           void MakeSound(void) { cout << name << " is silent :(" << endl; }
};
class Dog : public Pet {
public: Dog(string name) : Pet(name) {}
        void MakeSound(void) { cout << name << " says: Woof!" << endl; }
};
class GermanShepherd : public Dog {
public: GermanShepherd(string name) : Dog(name) {}
        void MakeSound(void) { cout << name << " says: Wuff!" << endl; }
};
class MastinEspanol : public Dog {
public: MastinEspanol(string name) : Dog(name) {}
        void MakeSound(void) { cout << name << " says: Guau!" << endl; }
};
void PlayWithPet(Pet &pet) {
      pet.MakeSound();
}
```

Passing an object of a subclass

```
int main(void) {
	Pet pet("creature");
	Dog dog("Dog");
	GermanShepherd gs("Hund");
	MastinEspanol mes("Perro");
	PlayWithPet(pet);
	PlayWithPet(dog);
	PlayWithPet(gs);
	PlayWithPet(mes);
	return 0;
}
```

Passing an object of a subclass

- The expected output is:

  - creature is silent :(
  - Dog is silent :(
  - Hund is silent :(
  - Perro is silent :(

Passing an object of a subclass

```
#include <iostream>
#include <string>
using namespace std;
class Pet {
protected:string name;
public:   Pet(string name) { this -> name = name; }
          void MakeSound(void) { cout << name << " is silent :(" << endl; }
};
class Dog : public Pet {
public:   Dog(string name) : Pet(name) {}
      void MakeSound(void) { cout << name << " says: Woof!" << endl; }
};
class GermanShepherd : public Dog {
public:   GermanShepherd(string name) : Dog(name) {}
      void MakeSound(void) { cout << name << " says: Wuff!" << endl; }
};
class MastinEspanol : public Dog {
public:   MastinEspanol(string name) : Dog(name) {}
      void MakeSound(void) { cout << name << " says: Guau!" << endl; }
};
void PlayWithPet(Pet *pet) {
      pet -> MakeSound();
}
```

```
int main(void) {
      Pet *pet = new Pet("creature");
      Dog *dog = new Dog("Dog");
      GermanShepherd *gs = new GermanShepherd("Hund");
      MastinEspanol *mes = new MastinEspanol("Perro");
      PlayWithPet(pet);
      PlayWithPet(dog);
      PlayWithPet(gs);
      PlayWithPet(mes);
      return 0;
}
```

The dynamic_cast operator

- Firstly, we’ve modified the *MakeSound* method inside the top-level class – it’s **virtual** now.
- Secondly, we’ve made the class tree. We’ve added two additional levels to the tree.

The dynamic_cast operator

- **If** the *dynamic_cast* operator is used in the following way:
  - dynamic_cast<pointer_type>(pointer_to_object)
- **and** the conversion of *pointer_to_object* to the type of *pointer_type* is possible, **then** the result of the conversion is **a new pointer which is fully usable**.
- Otherwise, the result of the conversion is equal to NULL.

The dynamic_cast operator

```
#include <iostream>
#include <string>
using namespace std;
class Pet {
protected: string name;
public:     Pet(string name) : name(name) {}
        virtual void MakeSound(void) { cout << name << " is silent :(" << endl; }
};
class Dog : public Pet {
public:    Dog(string name) : Pet(name) {}
      void MakeSound(void) { cout << name << " says: Woof!" << endl; }
};
class GermanShepherd : public Dog {
public:    GermanShepherd(string name) : Dog(name) {}
      void MakeSound(void) { cout << name << " says: Wuff!" << endl; }
      void Laufen(void) { cout << name << " runs (gs)!" << endl; }
};
class MastinEspanol : public Dog {
public:    MastinEspanol(string name) : Dog(name) {}
      void MakeSound(void) { cout << name << " says: Guau!" << endl; }
      void Ejecutar(void) { cout << name << " runs (mes)!" << endl; }
};
void PlayWithPet(Pet *pet) {
      GermanShepherd *gs;
      MastinEspanol *mes;
      pet -> MakeSound();
      if(gs = dynamic_cast<GermanShepherd *>(pet))
            gs -> Laufen();
      if(mes = dynamic_cast<MastinEspanol *>(pet))
            mes -> Ejecutar();
}
```

The dynamic_cast operator

```
int main(void) {
        Pet *pet = new Pet("creature");
        Dog *dog = new Dog("Dog");
        GermanShepherd *gs = new GermanShepherd("Hund");
        MastinEspanol *mes = new MastinEspanol("Perro");
        PlayWithPet(pet);
        PlayWithPet(dog);
        PlayWithPet(gs);
        PlayWithPet(mes);
        return 0;
}
```

The dynamic_cast operator

- This is what you should see on the screen:

    - creature is silent :(
    - Dog says: Woof!
    - Hund says: Wuff!
    - Hund runs (gs)!
    - Perro says: Guau!
    - Perro runs (mes)!

The dynamic_cast operator

- The *PlayWithPet* function doesn’t have a pointer but a **reference**. In consequence, the following two parts of the programs have been changed too:
  - the main function invokes the *PlayWithPet* in a slightly different way (have a look)
  - the form of ***dynamic_cast*** utilization is quite different here; the operator takes the following form:
    - dynamic_cast<reference_type>(reference_to_object)
- and returns a newly transformed (converted) reference

The dynamic_cast operator

```
#include <iostream>
#include <string>
using namespace std;
class Pet {
protected: string name;
public:     Pet(string name) : name(name) {}
        virtual void MakeSound(void) { cout << name << " is silent :(" << endl; }
};
class Dog : public Pet {
public:   Dog(string name) : Pet(name) {}
      void MakeSound(void) { cout << name << " says: Woof!" << endl; }
};
class GermanShepherd : public Dog {
public:   GermanShepherd(string name) : Dog(name) {}
      void MakeSound(void) { cout << name << " says: Wuff!" << endl; }
      void Laufen(void) { cout << name << " runs (gs)!" << endl; }
};
class MastinEspanol : public Dog {
public:   MastinEspanol(string name) : Dog(name) {}
      void MakeSound(void) { cout << name << " says: Guau!" << endl; }
      void Ejecutar(void) { cout << name << " runs (mes)!" << endl; }
};
void PlayWithPet(Pet &pet) {
      pet.MakeSound();
      dynamic_cast<GermanShepherd &>(pet).Laufen();
      dynamic_cast<MastinEspanol &>(pet).Ejecutar();
}
int main(void) {
      Pet pet("creature");
      Dog dog("Dog");
      GermanShepherd gs("Hund");
      MastinEspanol mes("Perro");
      PlayWithPet(pet);
      PlayWithPet(dog);
      PlayWithPet(gs);
      PlayWithPet(mes);
      return 0;
}
```

The dynamic_cast operator

- The program, compiled and run, produces the following, disappointing output:
  - creature is silent :(
  - terminate called after throwing an instance of 'std::bad_cast'
  - what(): std::bad_cast
  - This application has requested the Runtime to terminate it in an unusual way.
  - Please contact the application's support team for more information.

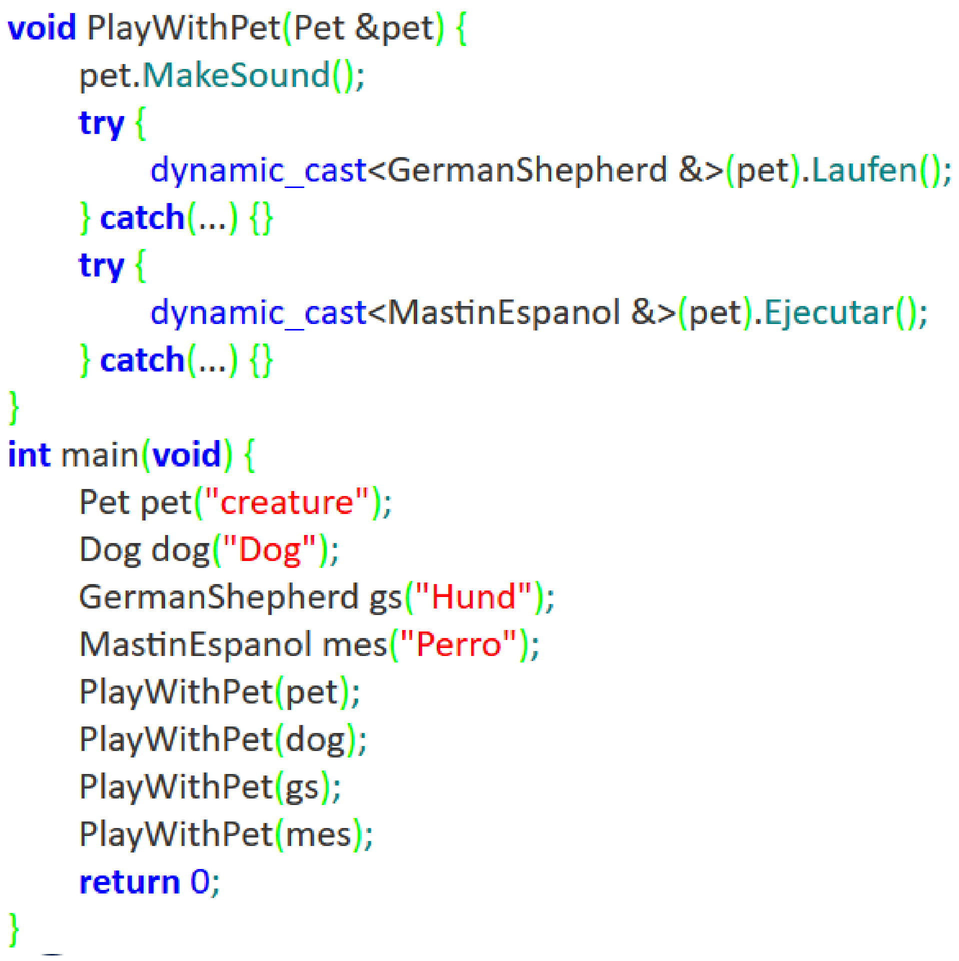
The dynamic_cast operator

- There's something new here: the **try-catch statement**. It looks like this:
  - try {
  - thing_we_want_to_try_although_we_are_not_quite_sure_if_it_is_reasonable;
  - } catch(…) {}

The dynamic_cast operator

```
#include <iostream>
#include <string>
using namespace std;
class Pet {
protected: string name;
public:     Pet(string name) : name(name) {}
        virtual void MakeSound(void) { cout << name << " is silent :(" << endl; }
};
class Dog : public Pet {
public:   Dog(string name) : Pet(name) {}
      void MakeSound(void) { cout << name << " says: Woof!" << endl; }
};
class GermanShepherd : public Dog {
public:   GermanShepherd(string name) : Dog(name) {}
      void MakeSound(void) { cout << name << " says: Wuff!" << endl; }
      void Laufen(void) { cout << name << " runs (gs)!" << endl; }
};
class MastinEspanol : public Dog {
public:   MastinEspanol(string name) : Dog(name) {}
      void MakeSound(void) { cout << name << " says: Guau!" << endl; }
      void Ejecutar(void) { cout << name << " runs (mes)!" << endl; }
};
```

The dynamic_cast operator



```
void PlayWithPet(Pet &pet) {
	pet.MakeSound();
	try {
		dynamic_cast<GermanShepherd &>(pet).Laufen();
	} catch(...) {}
	try {
		dynamic_cast<MastinEspanol &>(pet).Ejecutar();
	} catch(...) {}
}
int main(void) {
	Pet pet("creature");
	Dog dog("Dog");
	GermanShepherd gs("Hund");
	MastinEspanol mes("Perro");
	PlayWithPet(pet);
	PlayWithPet(dog);
	PlayWithPet(gs);
	PlayWithPet(mes);
	return 0;
}
```

The dynamic_cast operator

- The program produces this output:

  - creature is silent :(
  - Dog says: Woof!
  - Hund says: Wuff!
  - Hund runs (gs)!
  - Perro says: Guau!
  - Perro runs (mes)!