

Polymorphism

OOP Principles, Virtual Members, Polymorphism



SoftUni Team

Technical Trainers



SoftUni



Software University

<https://softuni.bg>

1. Polymorphism
 - What is Polymorphism?
 - Types of Polymorphism
 - Override Methods
 - Overload Methods
2. Virtual Members and Overriding
3. Using Polymorphism
4. Specifics and Good Practices





sli.do

#cpp-oop



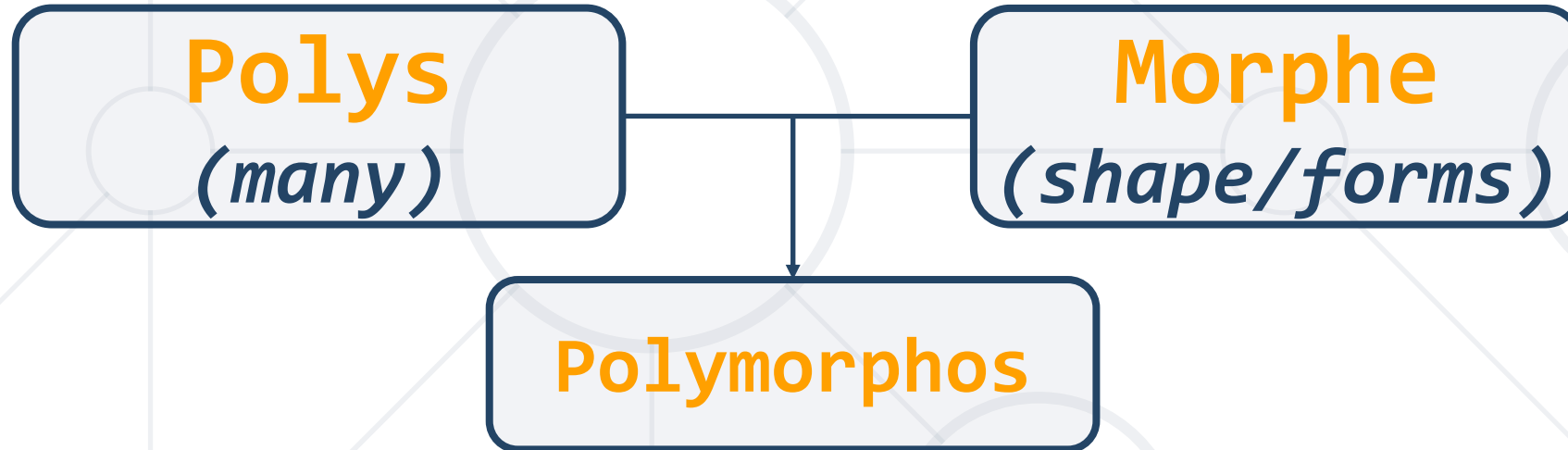
ANIMAL



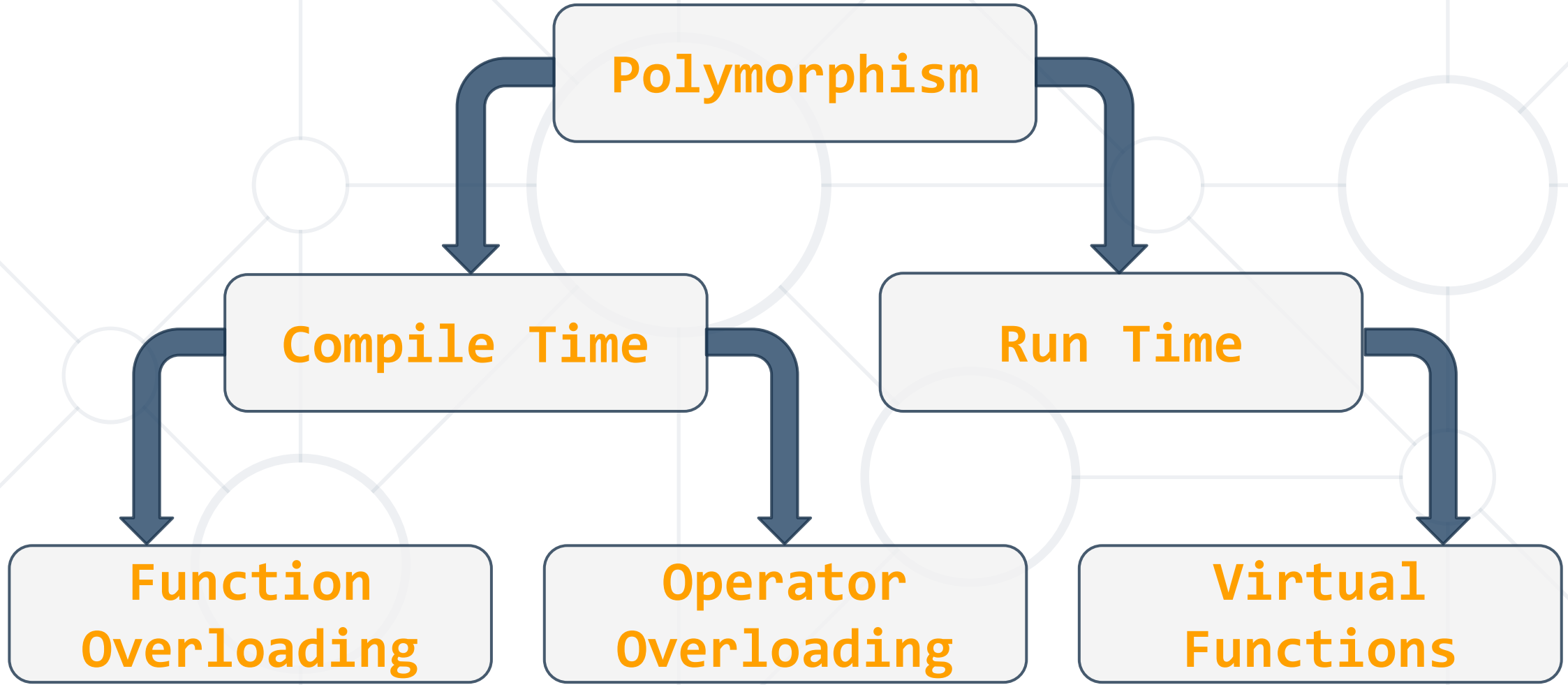
Polymorphism

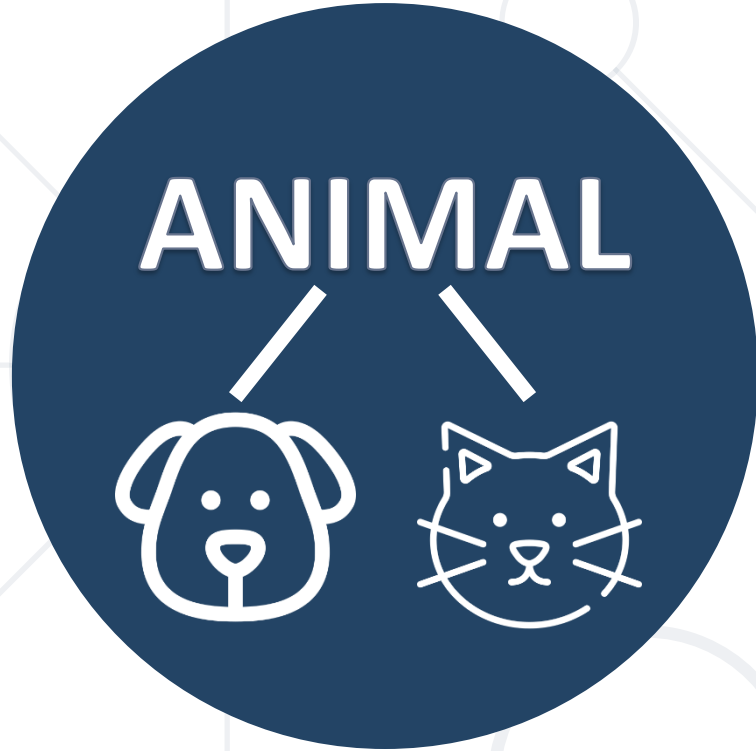
What is Polymorphism?

- From the Greek



- Such as a word having **several different meanings** based on the context
- Often referred to as **the third pillar of OOP**, after encapsulation and inheritance





Compile Time Polymorphism

Compile Time Polymorphism

- Also known as Static Polymorphism
- This type of polymorphism is achieved by
 - Function overloading
 - Operator overloading
 - Template metaprogramming
- Argument lists could **differ** in
 - Number of parameters
 - Data type of parameters
 - Sequence of Data type of parameters

Rules for Overloading Method

- **Overloading** can take place in the **same class** or in its **sub-class**
- **Constructors** in C++ can be **overloaded**
- Overloaded methods must have a **different argument list**
- They may have the **same** or **different return types**

Function Overloading

```
public:
    void func(int x) { // function with 1 int parameter
        cout << "value of x is " << x << endl; }
    void func(double x) { // function with same name but 1 double parameter
        cout << "value of x is " << x << endl; }
    void func(int x, int y) { // function with same name and 2 int parameters
        cout << "value of x and y is " << x << ", " << y << endl; }
};

int main() {
    func(13);
    func(13.2);
    func(33,43);
}
```

Output:
value of x is 13
value of x is 13.2
value of x and y is 33,43

Operator Overloading

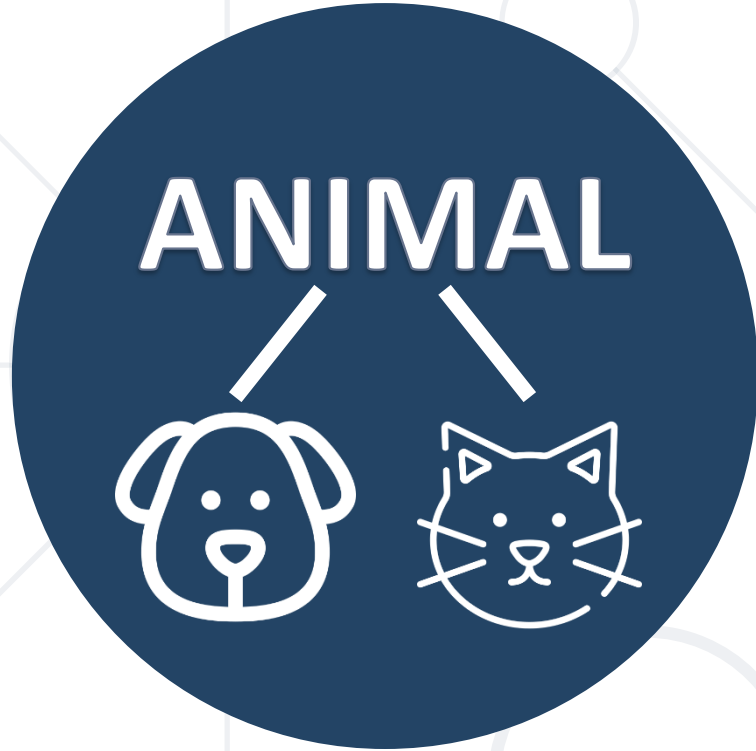
```
class Complex {
private:
    int real, imag;
public:
    Complex(int r = 0, int i=0) { real = r; imag = i; }

    Complex operator + (Complex const &obj) { // This is automatically called when '+' is used with
        Complex res; // between two Complex objects
        res.real = real + obj.real
        res.imag = imag + obj.imag;
        return res; }

    Void print() {count << real << " + i"<< imag << endl;
};

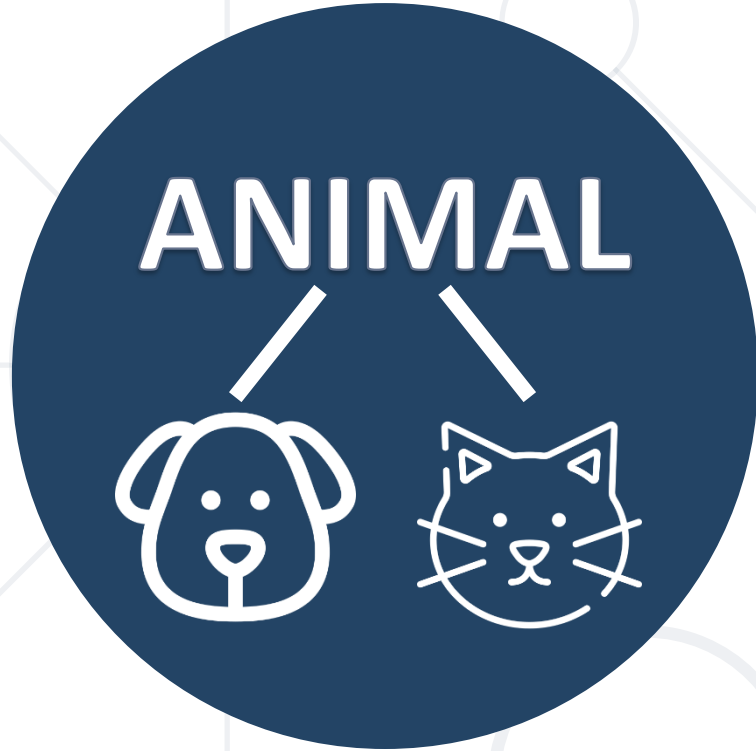
int main(){
    Complex c1(10, 5), c2(2,4);
    Complex c3 = c1 + c2; // An example call to "operator+"
    c3.print();
}
```

Output:
12 + i9



Compile Time Polymorphism

LIVE DEMO



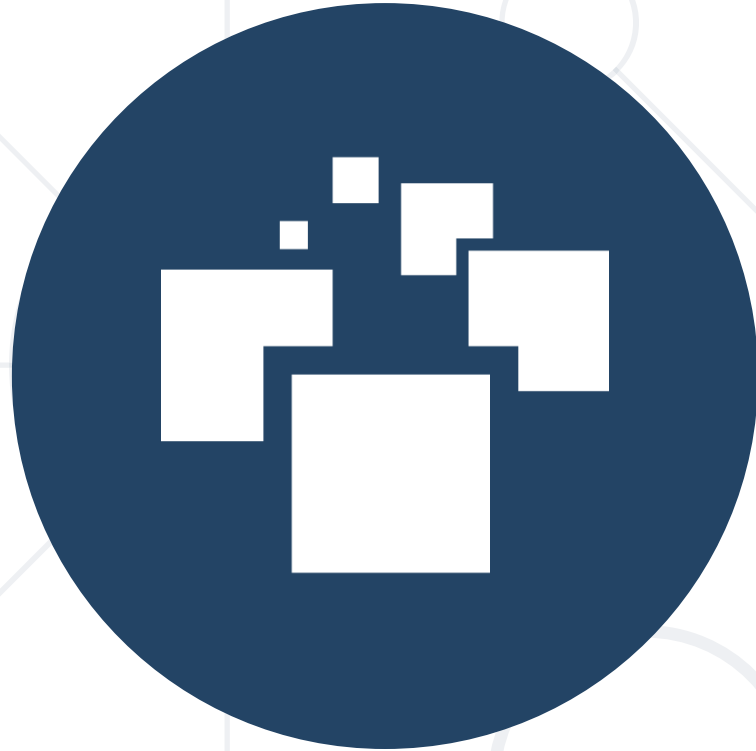
Runtime Polymorphism

- Also known as **Dynamic Polymorphism**
- This type of polymorphism is achieved by **virtual methods**
- In order for virtual methods to be used - the function definition in the derived class must have the same definition as the one base class. That base function is said to be **overridden**

- **Base** pointers/references can point to **derived** objects
 - **upcast**, NO slicing – not fitting larger into smaller object
 - **Derived d; Base* p = &d;**
 - **Base* p = new Derived(); ...**
- Accesses base members, regardless of hiding

```
Airplane plane(510, 2400, 90);  
Vehicle* v = &plane;  
cout << v->toString() << endl; // calls Vehicle::toString()
```

- Unless members are **virtual overrides**



Virtual Members and Overriding

- **virtual** methods – allow **derived** to change implementation
- **override** – placed after same-signature **virtual** in **derived**
 - **Base** has **virtual void f()**
 - **Derived** has **virtual void f() override**

```
class Vehicle {  
  
    ...  
    virtual void stop() {  
        this->speed = 0;  
    }  
};
```

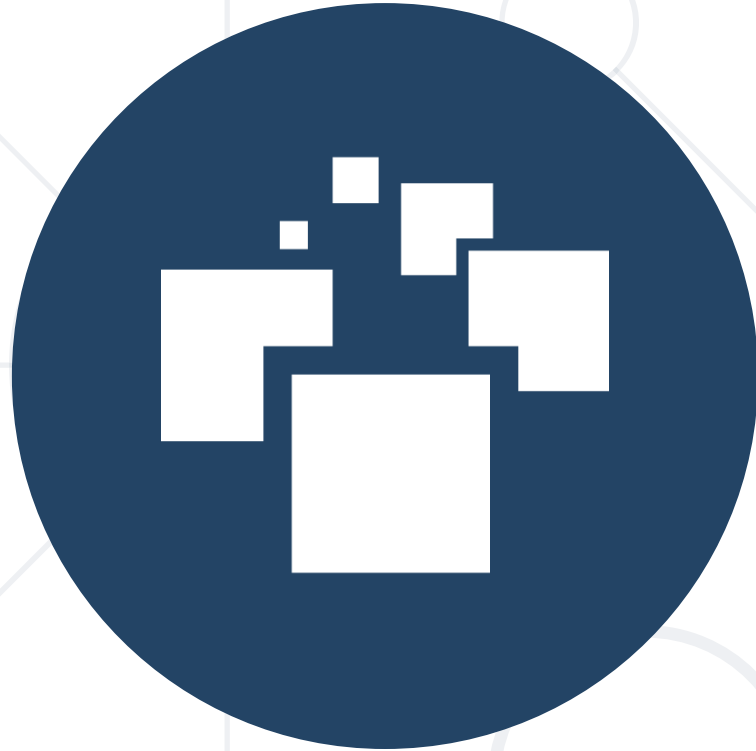
```
class Car : public Vehicle {  
    ...  
    virtual void stop() override {  
        Vehicle::stop();  
        this->parkingBrakeOn = true;  
    }  
};
```

- Call **virtual** method from **base** pointer to **derived** object calls:
 - **Derived** method if there's a matching member
 - **Base** method otherwise

```
virtual void stop() { ... }           // class Vehicle
virtual string toString() const { ... }
```

```
virtual string toString() const override { ... } // class Airplane
virtual void stop() override { ... }
```

```
Vehicle* v = new Airplane plane(510, 2400, 90);
cout << v->toString() << endl; // calls Airplane::toString()
v->stop(); // calls Airplane::stop()
```



virtual, override, and Base Pointers

LIVE DEMO

Rules for Overriding Method

- **Overriding** can take place in **sub-class**
- **Argument list** must be the **same** as that of the **parent method**
- The overriding method must have **same return type**
- **Access modifier** cannot be more **restrictive**
- **Private**, **static** and **final** methods can **NOT** be overridden
- The overriding method **must not** throw new or broader **checked exceptions**

■ Will this leak memory?

```
for (;;) {  
    IndexedContainer* c =  
        new IntArray(10);  
    delete c;  
}
```

a) Yes

b) No

c) Maybe

d) I don't know, can you repeat the question... 🎵🎵🎵

```
class IndexedContainer { public:  
    virtual int& operator[](int i) { ... };  
};  
class IntArray : public IndexedContainer {  
    int size; int* data;  
public:  
    IntArray(int size)  
        : size(size), data(new int[size]) {}  
    ~IntArray() { delete[] this->data; }  
  
    virtual int& operator[](int i) override  
        { ... }  
};
```

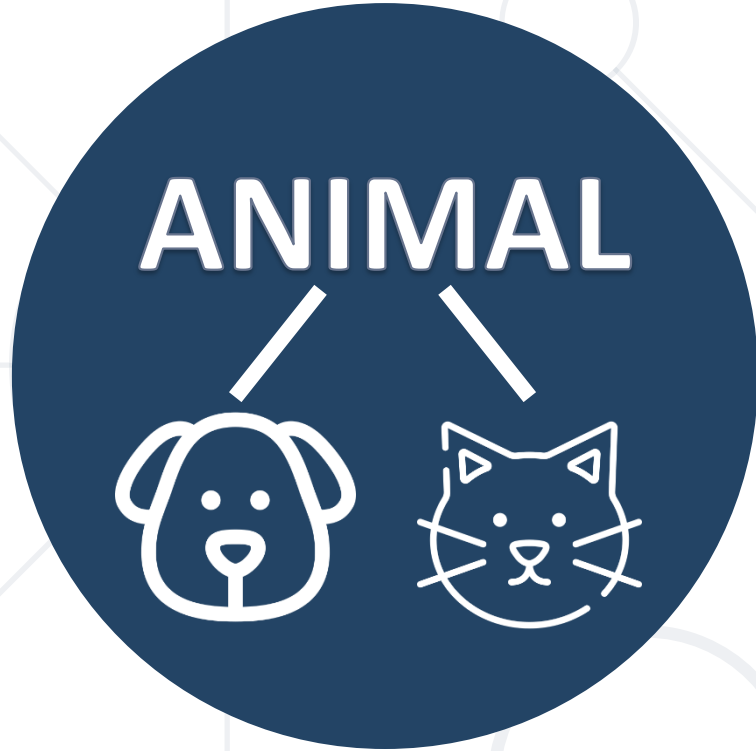
C++ PITFALL: DELETE CALL ON BASE CLASS POINTER TO DERIVED CLASS, WHERE BASE HAS NO VIRTUAL DESTRUCTOR

Undefined behavior:

- **delete** a base class pointer
- to a derived class object
- if base has no virtual destructor


Most compilers will just call the destructor of the base class – which won't do deallocation for the derived class



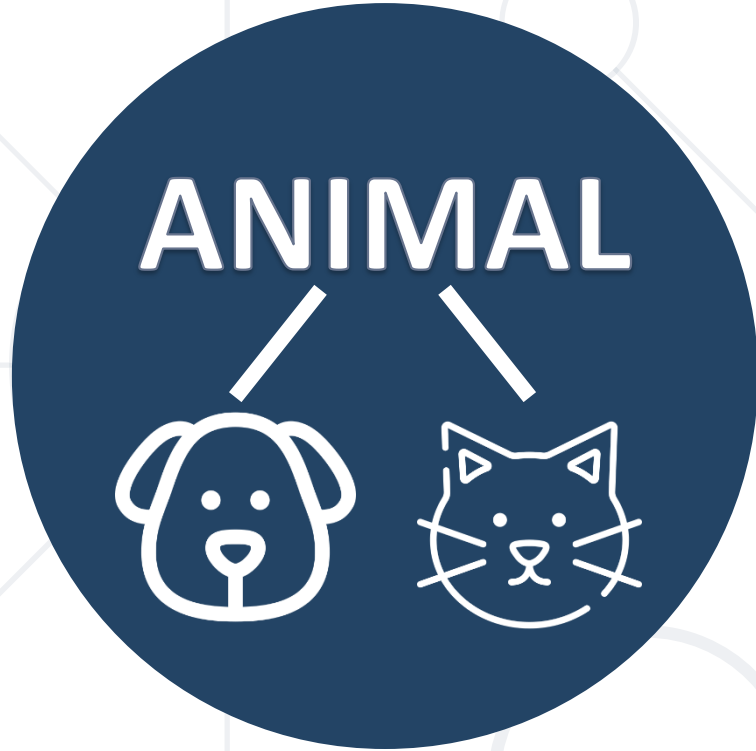


Using Polymorphism

Polymorphism

- 
- **Base** class and **derived** class
 - **virtual** methods in **base**, with **overrides** in **derived**
 - **Base** pointers/references to **derived** objects, calling **overrides**
 - **virtual** destructor in a base class

```
vector<Vehicle*> vehicles{  
    new Airplane(...), new Car(...), new PlaygroundTrain()  
};  
  
for (auto vehiclePtr : vehicles) vehiclePtr->stop();
```

Using Polymorphism

LIVE DEMO

- Smart pointers can be used as raw pointers in order to achieve polymorphism
- They are used in the same fashion

```
std::unique_ptr<Vehicle> v = std::unique_ptr<Vehicle>(new Car(50, false));  
v->stop();
```

- Smart pointers could also be part of a container
 - This is the most common polymorphic approach

```
vector<Vehicle*> vehicles{
    new Airplane(...), new Car(...), new PlaygroundTrain()
};

std::vector<std::unique_ptr<Vehicle>> vehicles;
vehicles.push_back(std::unique_ptr<Vehicle>(new Car(50, false)));
...
for (const auto &vehiclePtr : vehicles) {
    vehiclePtr->stop();
}
```



Polymorphism and Smart Pointers

LIVE DEMO



Practice

Live Exercise in Class

Problem 1: Particle System

- Implement a particle system on the console simulating:
 - Raindrops (fall straight down)
 - Snowflakes (fall down & move sideways)
 - Meteorites (fall diagonally, leaving a fixed-length trace behind)
 - Lightning bolts (random downward pattern of particles, disappears as fast as each of the others does a move)
- Loop iterating **list** of **Particle***, calls **update()** on each
 - Inherit **Particle** (**position**, **symbol**, **exists**) with the above



Specifics and Good Practices

- The **override** keyword is just a safeguard
 - No effect if a **virtual** base method exists
 - Compilation error if NO **virtual** base method
 - *Good practice: use always when intending an override*
- If class can be a base, declare a **virtual** destructor
 - **virtual ~ClassName() {}**
 - or **virtual ~ClassName() = default;**
 - *There are some exceptions to this, but it's an ok beginner rule*

- Inheriting from a final classes is forbidden

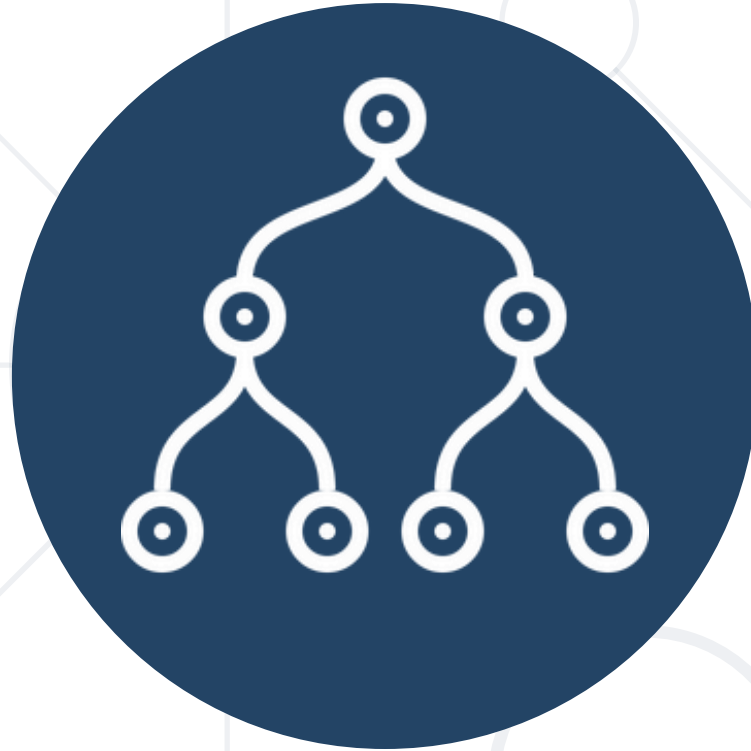
```
class Animal final {  
    ...  
}
```

```
public class Dog :public Animal { } // Error...
```

- **final** – defines a method that **can't be overridden**

```
class Animal {  
    public:  
    virtual void eat() final { ... }  
}
```

```
class Dog : public Animal {  
    public:  
        void eat() override {}  
    // Error...  
}
```



Inheritance in Memory

Why Base Pointers Work

- Fields in memory follow declaration order (usually)
 - "Padding" is auto-added to make size multiple of the biggest primitive data type used in the object.

```
class Organism {  
float weight; bool eatsPlants; bool eatsAnimals;  
public:  
Organism(float w, bool p, bool a) : weight(w), eatsPlants(p), eatsAnimals(a) {}  
};
```

```
Organism o(42, true, false);
```

Address	...	0x6afe4c...0x6afe4f	0x6afe50	0x6afe51	0x6afe52	0x6afe53	...
Byte	...	42	true	false	padding	padding	...

- Base class members inserted at start of **derived** object

```
class Spider : public Organism {  
    int numLegs; float weight; // NOTE: hiding weight field from Organism  
public:  
    Spider(int l, float w) : Organism(w, false, true), numLegs(l), weight(w) {}  
};
```

```
Spider s(6, 0.1);
```

		Organism					numLegs	weight		
Address	...	0x6afe4c...0x6a53					0x6afe54...0x6afe57		0x6afe58...0x6afe61	
Byte	...	0.1	false	true	6		0.01	

Inheritance & Hidden Fields - Memory

```
class Organism {  
    float weight; bool eatsPlants;  
    bool eatsAnimals; ...  
};
```

```
class Spider : public Organism {  
    int numLegs; float weight;  
    ...  
};
```

Organism o

float weight

bool eatsPlants

bool eatsAnimals

Spider s

Organism members

float weight

bool eatsPlants

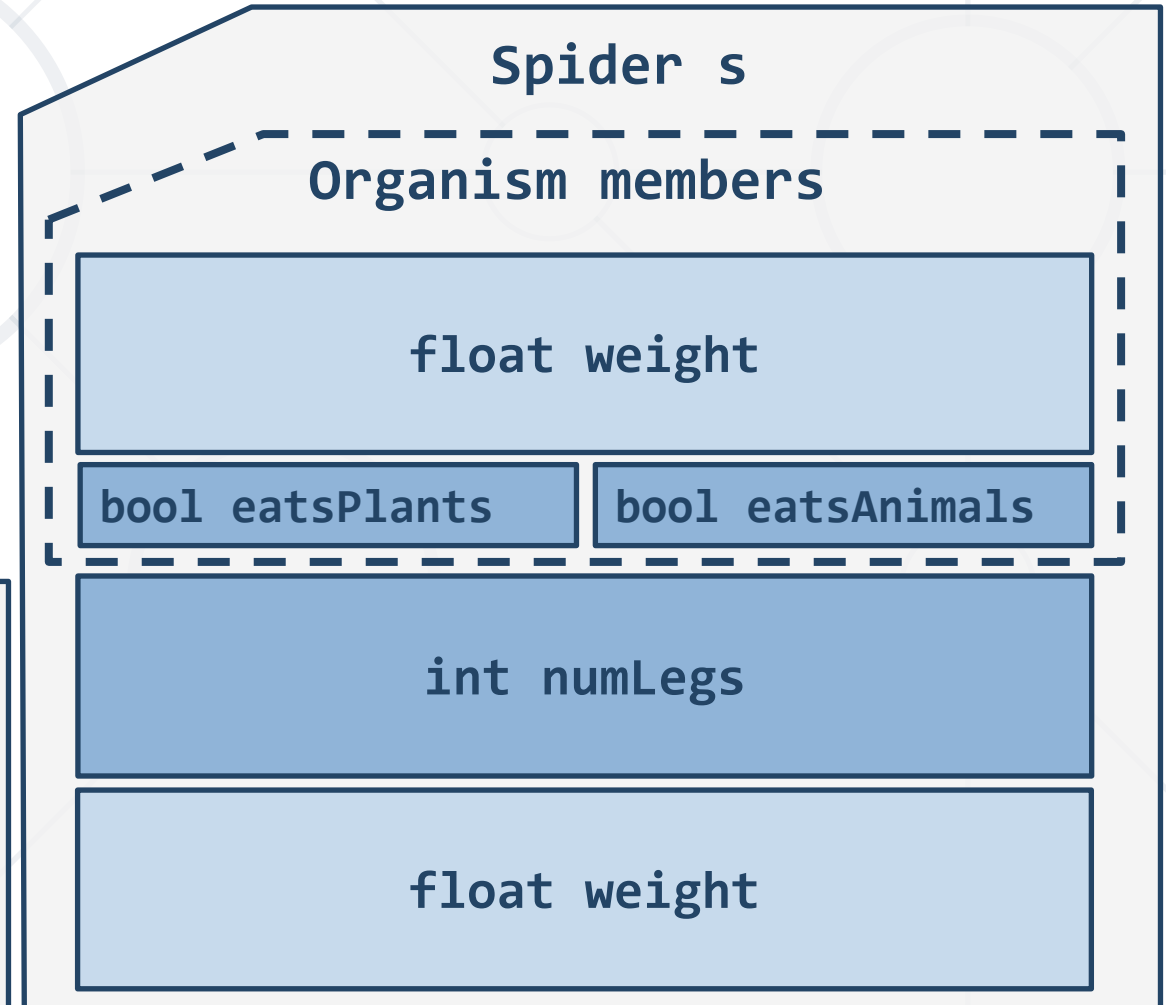
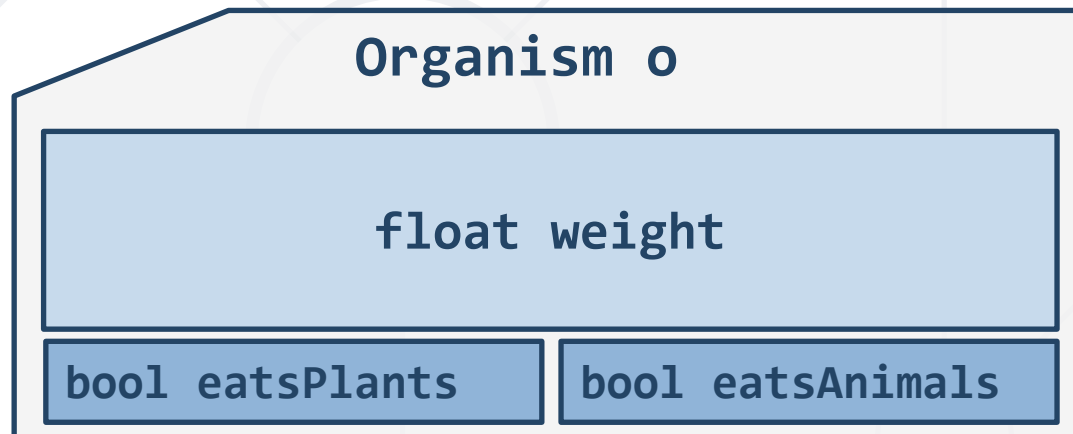
bool eatsAnimals

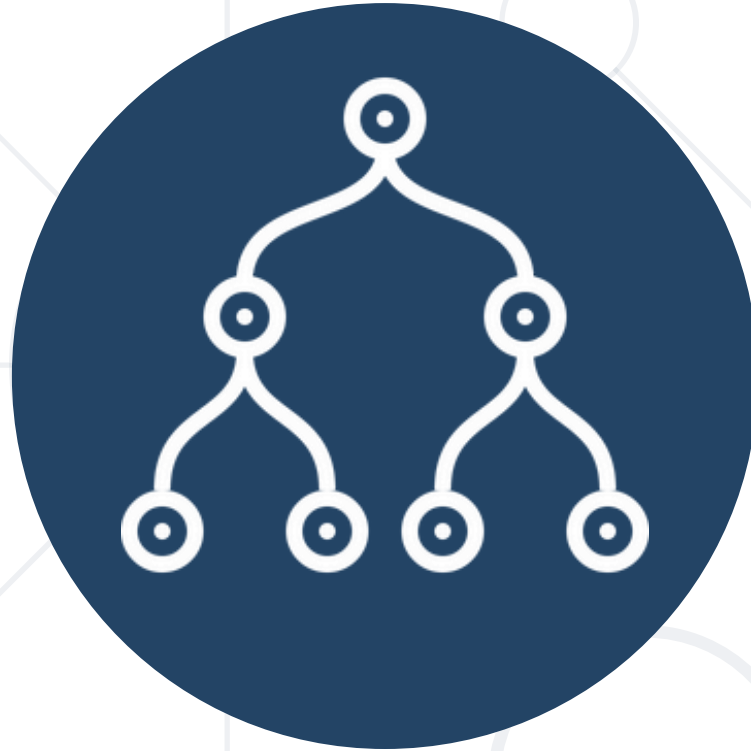
int numLegs

float weight

Inheritance & Hidden Fields - Pointers

```
Spider s(6, 0.042);  
Organism *oPtr = &s;  
oPtr->weight;  
oPtr->eatsPlants;  
oPtr->numLegs; //compilation error  
Spider * sPtr = (Spider*)oPtr;  
sPtr->weight;
```





Inheritance in Memory

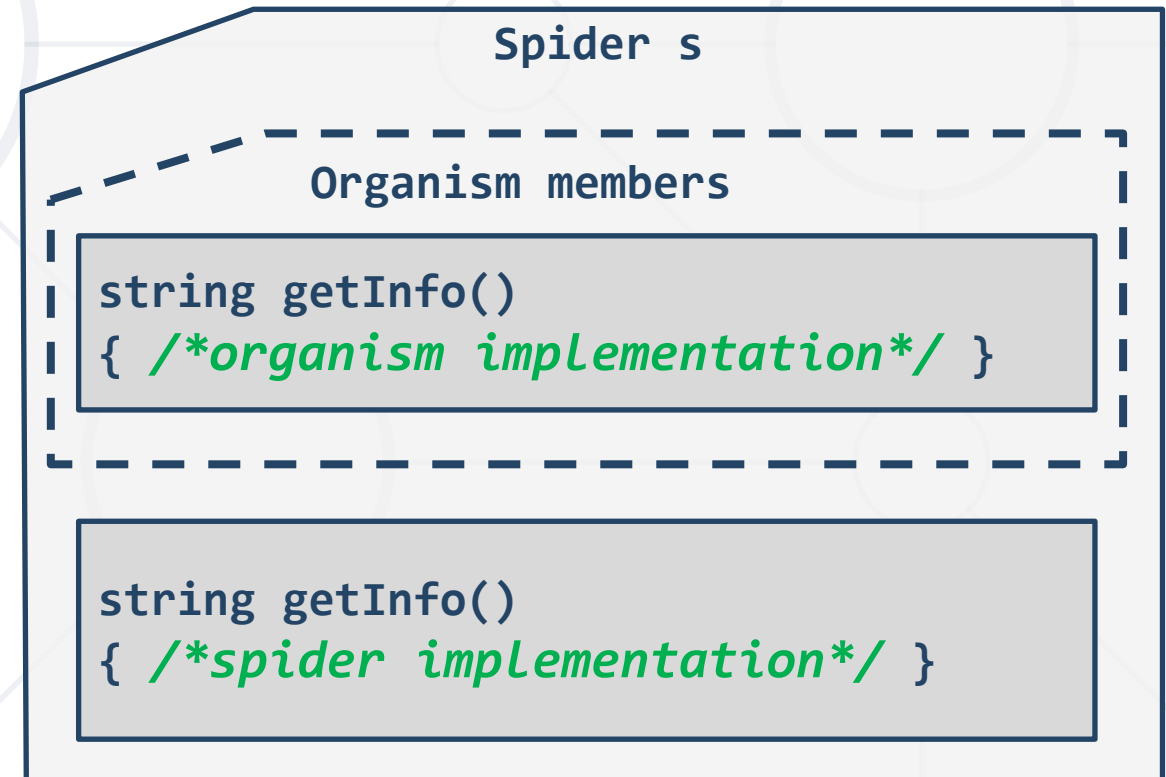
LIVE DEMO

Hidden Methods in Memory - NO virtual

```
class Organism { ...  
    string getInfo() const {  
        ...  
    }  
};
```

```
class Spider : public Organism { ...  
    string getInfo() const {  
        ...  
    }  
};
```

```
Spider s;  
Organism *oPtr = &s;  
oPtr->getInfo();  
Spider *sPtr = &s;  
sPtr->getInfo();
```

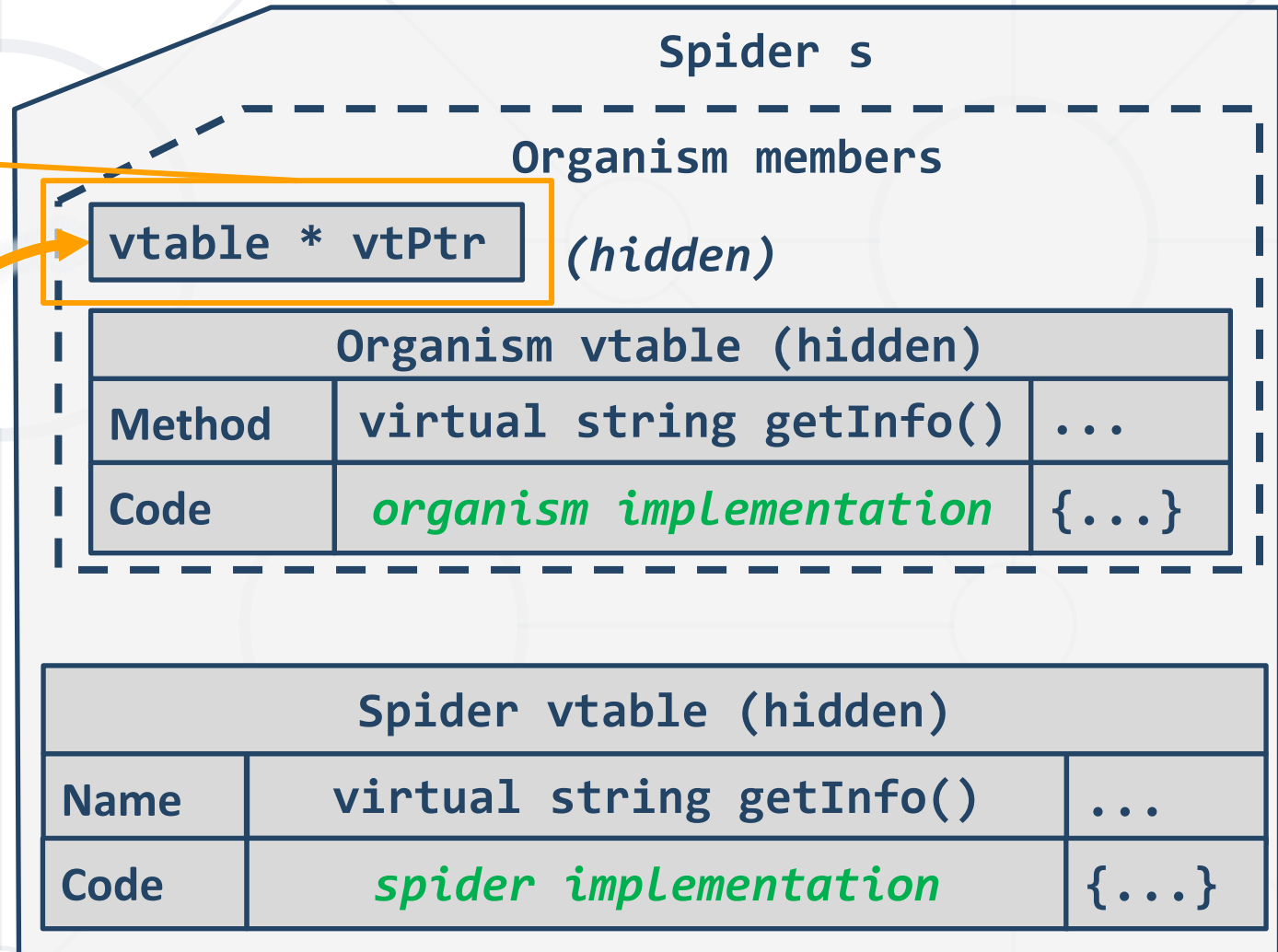


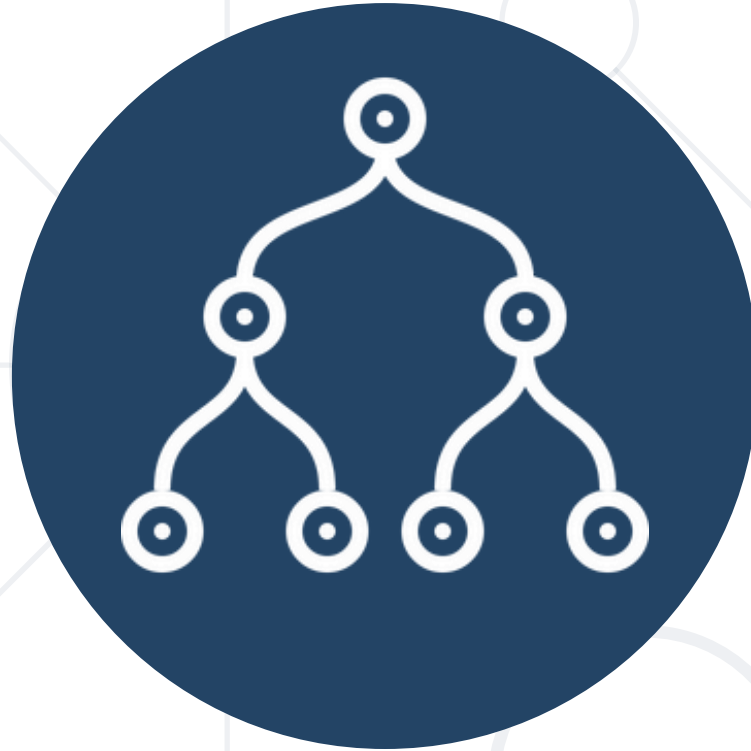
virtual Methods in Memory

```
class Organism { ...  
    virtual string getInfo() const {  
        ...  
    }  
};
```

```
class Spider : public Organism {  
    virtual string getInfo() const {  
        ...  
    }  
};
```

```
Spider s;  
Organism *oPtr = &s;  
oPtr->getInfo();  
Spider *sPtr = &s;  
sPtr->getInfo();
```





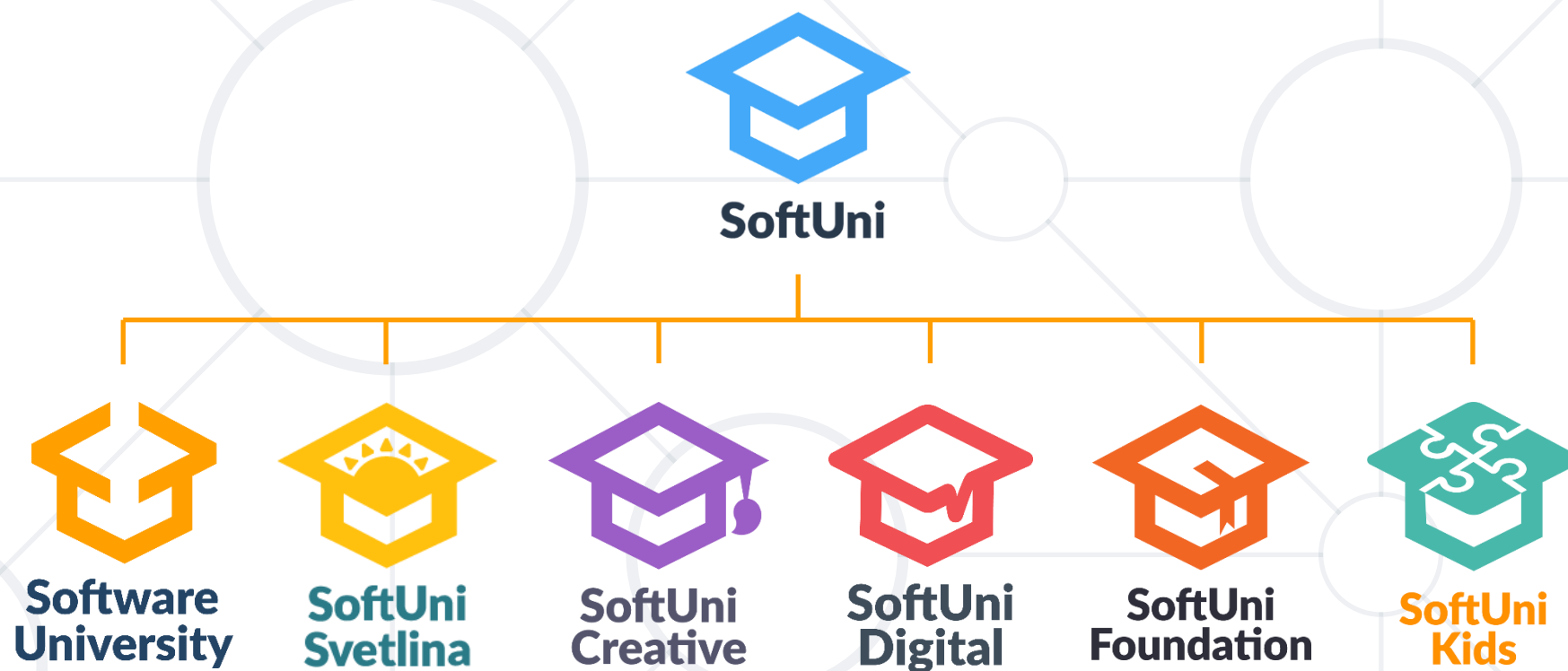
Plymorphism in Memory

LIVE DEMO

- Polymorphism - **Definition** and **Types**
- Override Methods
- Overload Methods
- **Virtual members** allow polymorphism
 - Treating objects as base pointers/references
 - Objects behave according to their overrides



Questions?



SoftUni Diamond Partners

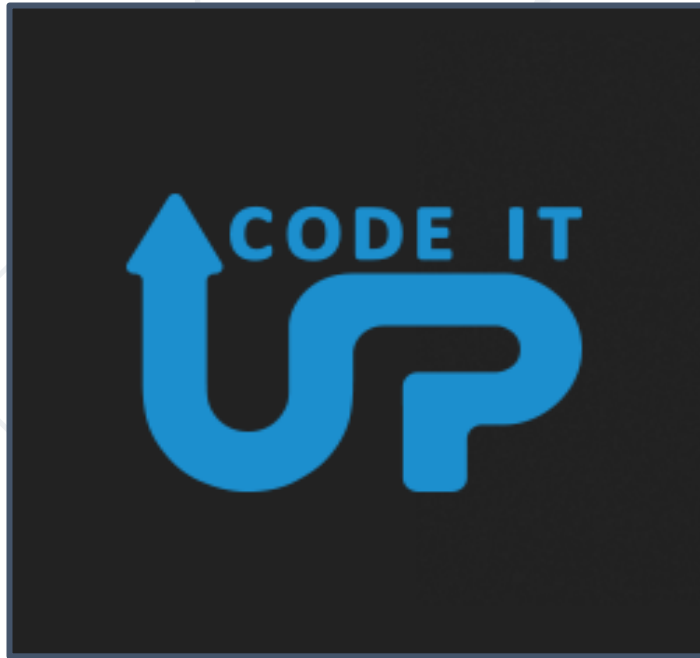


SCHWARZ



**SUPER
HOSTING
.BG**





VIRTUAL RACING SCHOOL



- This course (slides, examples, demos, exercises, homework, documents, videos and other assets) is **copyrighted content**
- Unauthorized copy, reproduction or use is illegal
- © SoftUni – <https://about.softuni.bg/>
- © Software University – <https://softuni.bg>



- Software University – High-Quality Education, Profession and Job for Software Developers

- softuni.bg, about.softuni.bg

- Software University Foundation

- softuni.foundation

- Software University @ Facebook

- facebook.com/SoftwareUniversity

- Software University Forums

- forum.softuni.bg

