Inheritance & Polymorphism

OOP Principles, Inheritance, Virtual Members, Polymorphism

SoftUni Foundation

Georgi Georgiev

A guy that knows C++

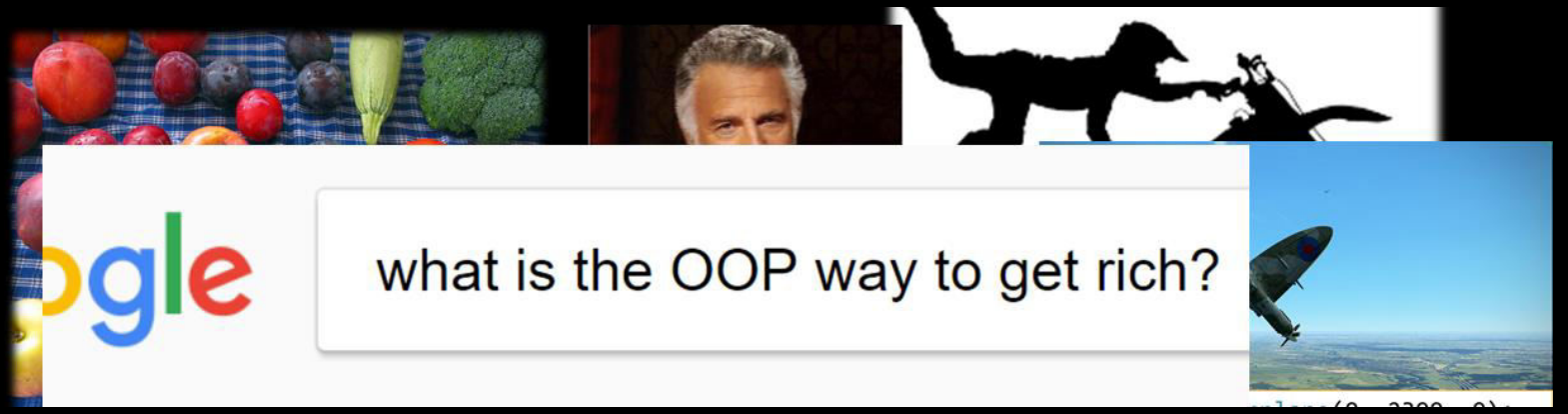


Table of Contents

SoftUni Foundation

Questions

SoftUni Foundation

sli.do

#cpp-softuni

SoftUni Foundation

OOP Principles

Encapsulation, Inheritance, Polymorphism, Abstraction

Encapsulation – *Reduces Complexity*

SoftUni Foundation

- Classes have internal state (e.g. `vector`'s capacity)
  - `private`/`protected` – state inaccessible to outside code
  - `public` members interact with outside code, keep state correct

```
class IntArray {
private:
  int* data; int size;
public:
  IntArray(int size) : data(new int[size]), size(size) {}
  ~IntArray() { delete[] this->data; }
  ...
};
```

Can't be modified from outside, so class can:
- **assume `data` needs a `delete[]` in destructor**
- **assume last index in `data` is `size-1`**
- **rename `size` to `length` without checking for outside usages**

Inheritance – *Allows Member Reuse*

SoftUni Foundation

- **Derived** classes **inherit** a **base** class to reuse its members

```
class Vehicle { private: double speed;
public: Vehicle(double speed) : speed(speed) {}
        void setSpeed(double speed) { this->speed = speed; }
};
```

```
class Car : public Vehicle {
private: bool parkingBrakeOn;
public:
Car(double spd, bool park)
  : Vehicle(spd)
  , parkingBrakeOn(park) {}
};
```

```
class Airplane : public Vehicle {
private: double altitude;
public:
Airplane(double spd, double alt)
  : Vehicle(spd)
  , altitude(alt) {}
};
```

Polymorphism – virtual Members

- Base class can have **virtual** members
  - Derived classes **override** them to have different behavior

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

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

```
class Airplane : public Vehicle {
  ...
  virtual void stop() override {
    Vehicle::stop();
    this->altitude = 0;
  }
};
```

Polymorphism – Base Class Pointers

SoftUni Foundation

- Base class pointers/references can point to any derived class
  - Normal members access **base** class member
  - **virtual** members access **override** member in **derived**

```
std::vector<Vehicle*> vehicles{
  new Car(90, false),
  new Airplane(700, 10000, 242),
  new Car(0, true)
};
vehicles[0]->stop(); // calls Car::stop()
vehicles[1]->stop(); // calls Airplane::stop()
vehicles[2]->stop(); // calls Car::stop()
```

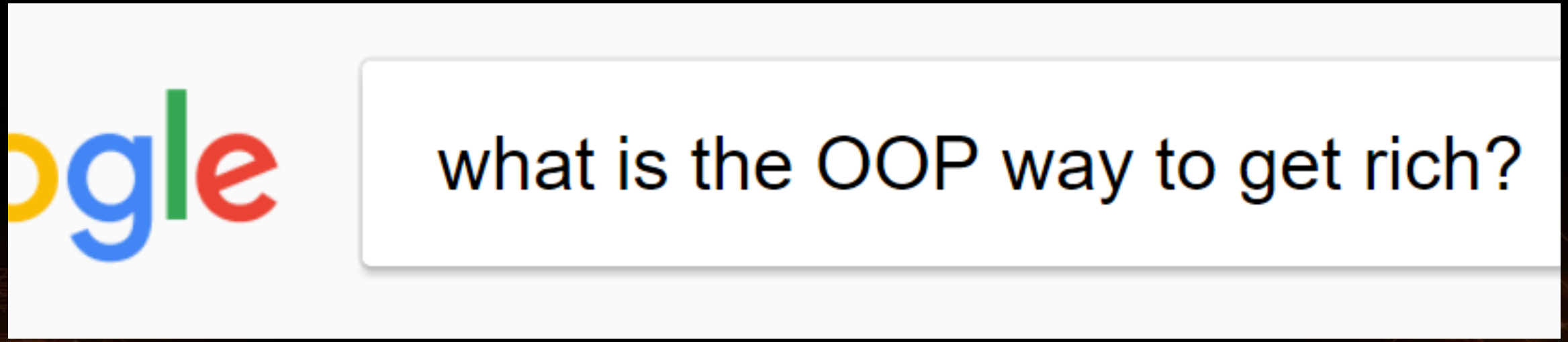
Abstraction – *Allows Generalizing Logic*

SoftUni Foundation

- Abstraction – using base virtual members
  - So allowing any class with **overrides** for them
- **ostream& operator<<(ostream& out, const Person& p)**
  - Allows any **ostream** – **ostringstream**, **ofstream**, ***cout***

```
void stopIfOverLimit(Vehicle* v, double limit) {
  if (v->getSpeed() > limit) {
    v->stop();
  }
}
```

SoftUni Foundation



Inheritance

Syntax, Protected Members, Accessing Base

Class Code Reuse

SoftUni Foundation

- Code reuse patterns:
  - Repeated code -> extract function
  - Functions using similar parameters/globals -> extract **class**
  - Repeated members in multiple classes -> extract **base class**
- Inheritance – sharing member definitions
  - A class declares/defines members
  - Other classes inherit it – get all members of inherited class

C++ Inheritance

SoftUni Foundation

- `class Derived : access-modifier Base { ... }`
  - `access-modifier` – one of `public`/`protected`/`private`
- Members of `Base` added to `Derived`
  - Access limited to inheritance `access-modifier`
  - `public`: doesn't change `Base` modifiers
  - `protected`: `public` from `Base` -> `protected` in `Derived`
  - `private`: any from `Base` -> `private` in Derived

C++ Inheritance – Extracting Base Class

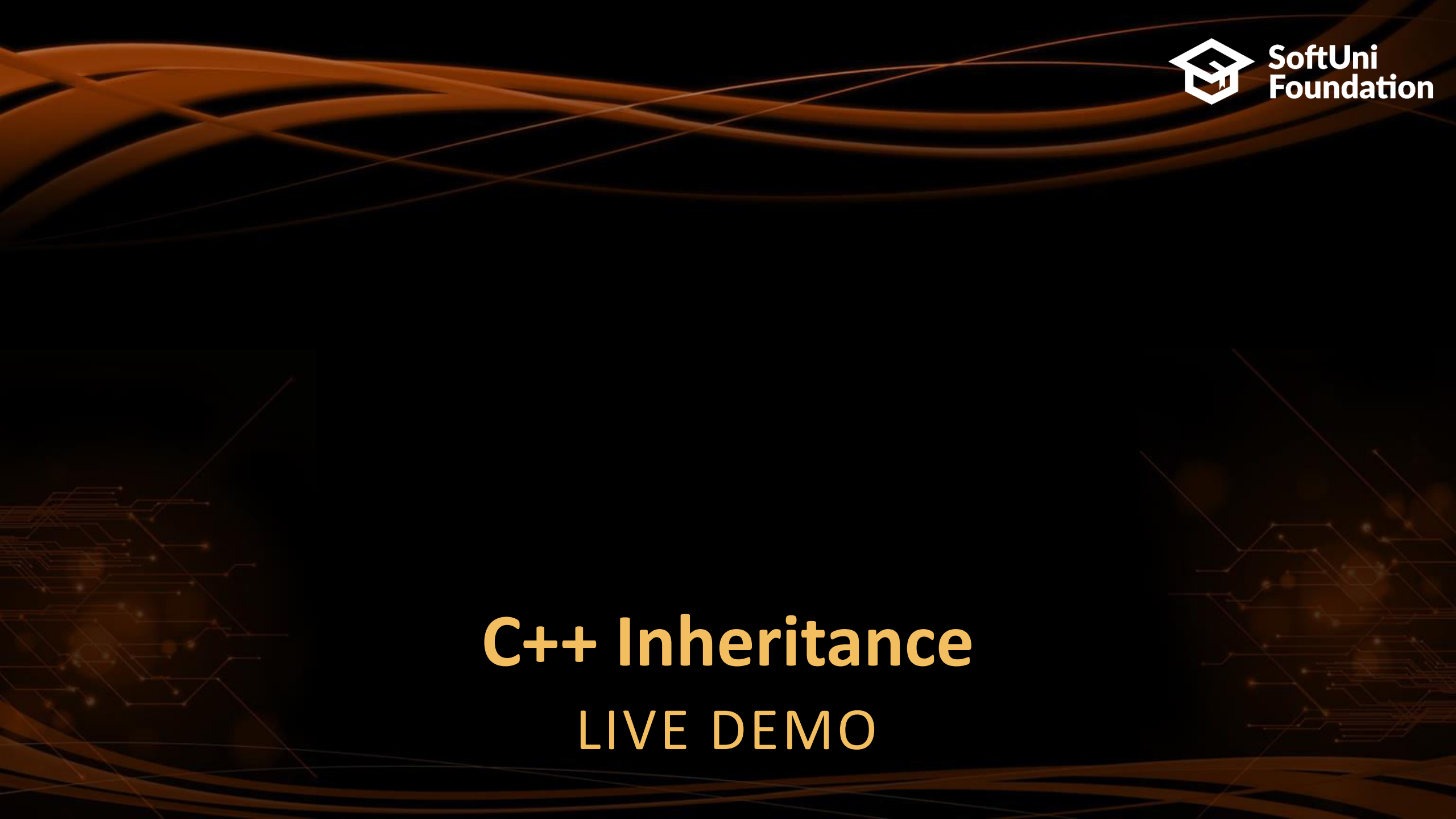
SoftUni Foundation

- Extract common members into a base class
  - NOTE: can't use initializer-list for base class fields

```
class Vehicle {
public: double speed;
};
```

```
class Car : public Vehicle {
  bool parkingBrakeOn;
public:
  Car(double speed, bool parked)
  : parkingBrakeOn(parked) {
    this->speed = speed;
  }
};
```

```
class Airplane : public Vehicle {
double altitude; double heading;
public:
  Airplane(double spd,
    double alt, double hdg)
  : altitude(alt)
  , heading(hdg) {
    this->speed = spd;
  }
};
```


SoftUni
Foundation
C++ Inheritance
LIVE DEMO

Share Access with Derived – protected

- *Previous example had **public speed** – breaking encapsulation*
  - *Can't use **private**, because we lose access to **speed***
- **protected** members – accessible to inheriting class

```
class Car : public Vehicle { ...
public:
  Car(...) {
    this->speed = speed; // ok
  }
};
```

```
class Vehicle {
protected:
  double speed;
};
```

```
Car car(90, false);
cout << car.speed << std::endl; // compilation error
```


SoftUni
Foundation
Protected Members
LIVE DEMO

Using Base Constructors

SoftUni Foundation

- Inheriting class can call **base** constructor
  - In initializer list, like field, BUT with base class name
  - Syntax: `Derived(...) : Base(...), ... { ... }`

```
class Vehicle { protected:
double speed;
Vehicle(double speed) : speed(speed) {}
```

```
class Car : public Vehicle {
...
Car(double speed, bool park)
: Vehicle(speed)
, parkingBrakeOn(park) {}
```

```
class Airplane : public Vehicle {
...
Airplane(double s, double a, double h)
: Vehicle(s)
, altitude(a), heading(h) {}
```

Hiding Methods

SoftUni Foundation

- Methods are inherited just like any member
- Hiding – using same signature in **derived** as in **base**
  - E.g. **base** has `void f()`, **derived** hides with `int f()`
    – calling `f()` in **derived** calls **derived** version (same for objects)
- Explicit access to base member (field/method/...)
  - Prefix member with **base** class name and `operator::`
  - E.g. `Base::f()` calls `f()` of inherited class `Base`

Example: Hiding & Calling Base Methods

SoftUni Foundation

- Example: Let's make a **toString()** for **Vehicle**
  - Reuse it in **Car**'s **toString()**

```
class Vehicle { ...
  string toString() const {
    ostringstream stream;
    stream << "speed: "
      << this->speed;
    return stream.str();
  }
}
```

```
class Car { ...
  string toString() const {
    ostringstream stream;
    stream << Vehicle::toString()
      << " parking brake: "
      << this->parkingBrakeOn;
    return stream.str();
  }
}
```

```
Car car(90, false); cout << car.toString();
```


SoftUni
Foundation
Calling Base Constructors & Methods
LIVE DEMO

C++ Object Slicing

- Base objects can be assigned with derived objects
  - Implicit cast, called **upcasting**
  - Fields from derived object are "sliced off"
  - *Should generally be avoided*
- `Base x = Derived();`
  - `x` can only access `Base` fields


```
Vehicle v = Airplane(0, 2300, 0);
```

C++ Object Slicing

LIVE DEMO

Quick Quiz TIME:

SoftUni Foundation

- What will this code do?

```
Vehicle v =
  Airplane(250, 10000);

cout << v.speed << endl;
```

a) Print **250**

b) Print **0**

c) Compilation error

d) Behavior is undefined

```
struct Vehicle {
public:
  double speed;
  Vehicle() : speed(0) {}
};

class Airplane : public Vehicle {
public:
  double speed; double altitude;

  Airplane(double speed, double altitude)
  : speed(speed), altitude(altitude) {}
};
```

C++ PITFALL: SLICING A HIDDEN FIELD

Fields can be hidden just like methods can.

The derived class in the example initializes its own field, not the base field. That field gets sliced-off.

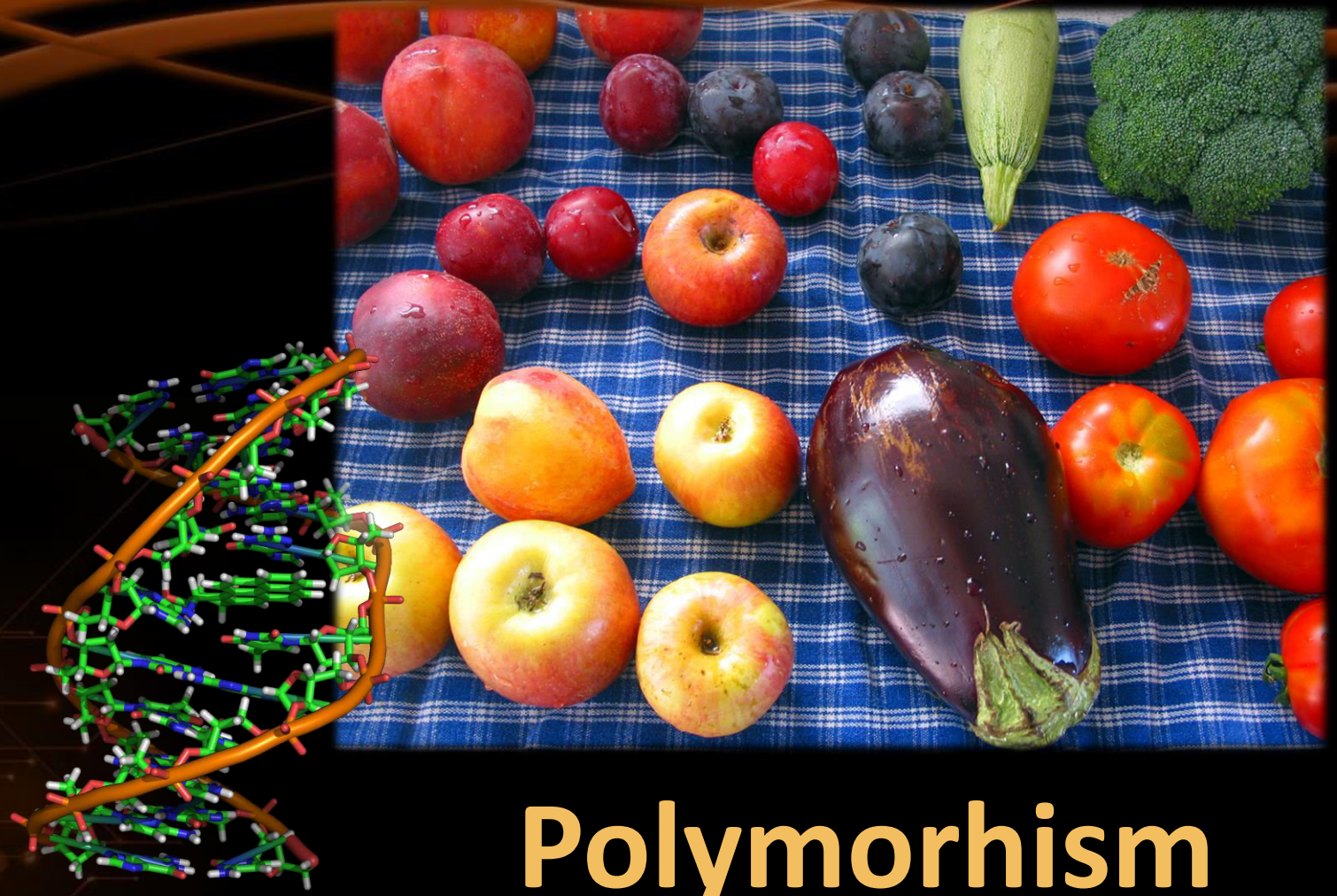
The base class has a default constructor – it gets called on derived initialization, hence `0`

Constructors & Assignments

SoftUni Foundation

- If **base** has no default constructor
  - **Derived** must define constructor calling **base** constructor
- Assignment operator is always hidden in a **derived** class
  - Signature not the same, but implicitly the same as base (**upcast**)
- Constructors aren't inherited – can't be used externally
  - Only used internally in initializer list
  - This also applies to copy/move constructors

SoftUni Foundation



Polymorhism

Using Objects Through Their Base Class

Base Pointers to Derived Objects

SoftUni Foundation

- **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 override`**s


SoftUni
Foundation
Base Pointers to Derived Objects
LIVE DEMO

virtual Members and override

SoftUni Foundation

- **virtual** methods – allow **derived** to change implementation
- **override** – placed after same-signature **virtual** in **derived**
  - E.g. **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;
  }
};
```

Virtual Members and Base Pointers

SoftUni Foundation

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

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

```
class Airplane : public Vehicle
virtual string toString() const override { ... }
virtual void stop() override { ... }
```

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

SoftUni Foundation

virtual, override, and Base Pointers

LIVE DEMO

Quick Quiz TIME:

SoftUni Foundation

- Will this leak memory?

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

```
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
  { ... }
};
```

a) Yes

b) No

c) Maybe

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

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

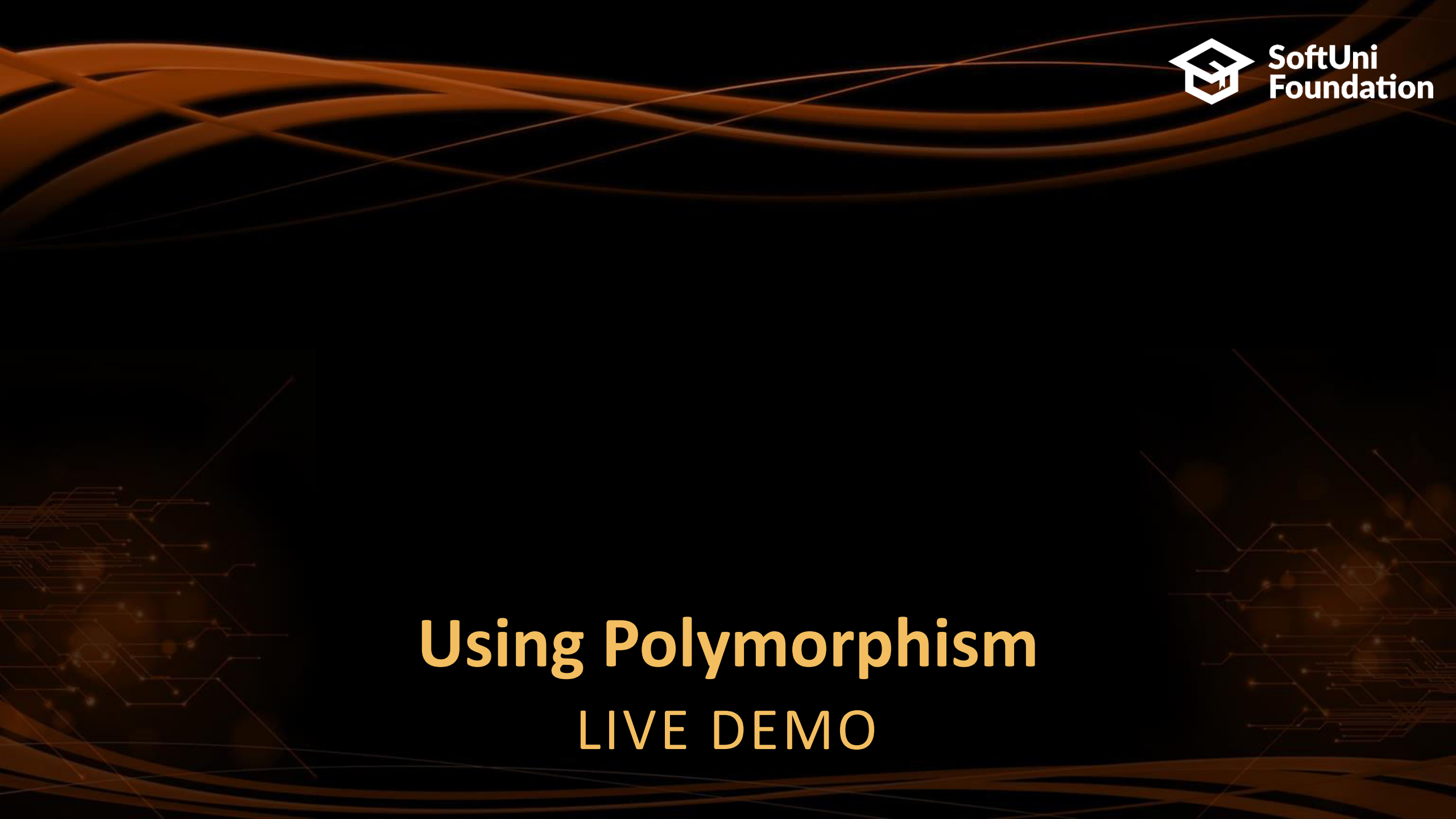
Polymorphism

SoftUni Foundation

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

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

for (auto vehiclePtr : vehicles) vehiclePtr->stop();
```


SoftUni
Foundation
Using Polymorphism
LIVE DEMO

Exercise: Particle System

SoftUni Foundation

- Implement a particle system on the console simulating:
  - Raindrops (fall straight down)
  - Snowflakes (*get offen… no, no :D* – fall down & move sideways)
  - Meteorites (fall diagonally, leaving 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 & Good Practices (1)

SoftUni Foundation

- The `override` keyword is just a safeguard
  - No effect if `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*

Good Practices (2)

SoftUni Foundation

- Use inheritance for "is-a" relationships
  - A `Car` is a `Vehicle`, a `FileWriter` is a `Writer`, etc.
  - A `Car` is NOT a `Wheel`, it contains wheels – composition
- Use composition for "has-a" relationships
  - Using another class for fields – e.g. `Person` with `string name`
  - `Car` has a `Wheel`, `FileWriter` has a `string filename`, etc.
- Prefer `public`–access inheritance
  - Use composition to achieve others

Summary

- OOP Principles – Encapsulation, Inheritance, Abstraction
  - improve reusability and reduce complexity
- Inheritance reuses class members
  - Extract multiple-usage code into base class
  - Inherit base to extend functionality
- Virtual members allow polymorphism
  - Treating objects as base pointers/references
  - Objects behave according to their overrides

Inheritance & Polymorphism

SoftUni Foundation

Questions?