Abstract Classes

Pure-Virtual Members

SoftUni Team

Technical Trainers

Software University

SoftUni

Software University

https://softuni.bg

Have a Question?

Software University

sli.do

#cpp-oop

Table of Contents

1. Pure-virtual Methods and Abstract Classes
2. OOP Interfaces
3. Multiple Inheritance
4. Runtime Type Checking

Pure-virtual Methods and Abstract Classes

Pure-virtual Methods

- **Virtual methods** are just pointers
  - To function code in memory
  - Pointers can point to `0` / `NULL` / `nullptr`
- **Pure-virtual method**
  - Points to no code
  - Function pointer to `NULL`
  - Syntax: append = `0;` to virtual method signature

```
virtual void write(string s) = 0;
```

Abstract Classes

- Abstract class – that either defines or inherits at least one pure virtual function
  - Can not be **instantiated**
  - Can not **create objects**

```
class Writer
{
protected: ostringstream log;
public:
  Writer() {}
  virtual void write(string s) = 0;
  string getLog() const {
    return this->log.str();
  }
};
```

```
class FileWriter : public Writer
{
  ofstream fileOut; string filename;
public: FileWriter(string file)
  : fileOut(file), filename(file) {}

  void write(string s) override {
    this->fileOut << s;
    this->log << "wrote " << s.size()
      << " bytes to " << filename;
  }
};
```

```
Writer writer; // compilation error
FileWriter writer("out.txt"); // ok
writer.write("hello");
```

Abstract Classes and Polymorphism

Base declares => Derived defines / implements => Code uses Base

- Usable methods **accessible** from base pointer / reference
- Pointers **guaranteed to point** to derived
- Guaranteed override access – **derived must have override**

```
void writeHello(Writer* writer)
{
  writer->write("hello");
}
```

```
void writeHello(Writer& writer)
{
  writer.write("hello");
}
```

```
FileWriter fileWriter("out.txt");
writeHello(&fileWriter);
```

```
FileWriter fileWriter("out.txt");
writeHello(fileWriter);
```

OOP Interfaces

Declaring Functionality for Others to Implement

OOP Interfaces

- Abstract classes that **only declare public methods**
  - Don't have implementation
  - Derived classes required to implement methods (or be abstract)

```
class Writer
{
 public:
   virtual void write(string s) = 0;
};
```

```
// struct avoids typing public:
struct Writer
{
  virtual void write(string s) = 0;
};
```

OOP Interface - Common Usage

- Derived classes with:
  - **Common methods**
  - **No common base members**
- Extract interface
  - Contains common methods as **pure-virtual methods**
  - Derived **classes inherit** it in addition to their members

OOP Interface - Example

```
class HasInfo {
public:
  virtual string getInfo() const = 0;
};
```

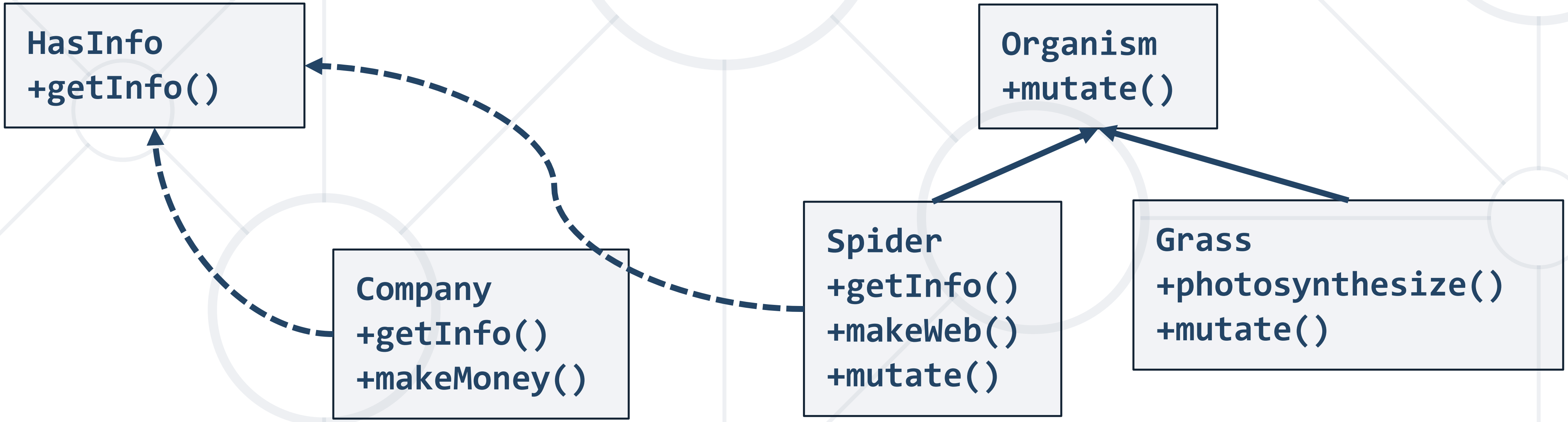
```
class Spider : public Organism, public HasInfo {
...
string getInfo() const override {
...
```

```
class Company : public HasInfo {
...
string getInfo() const override {
...
```

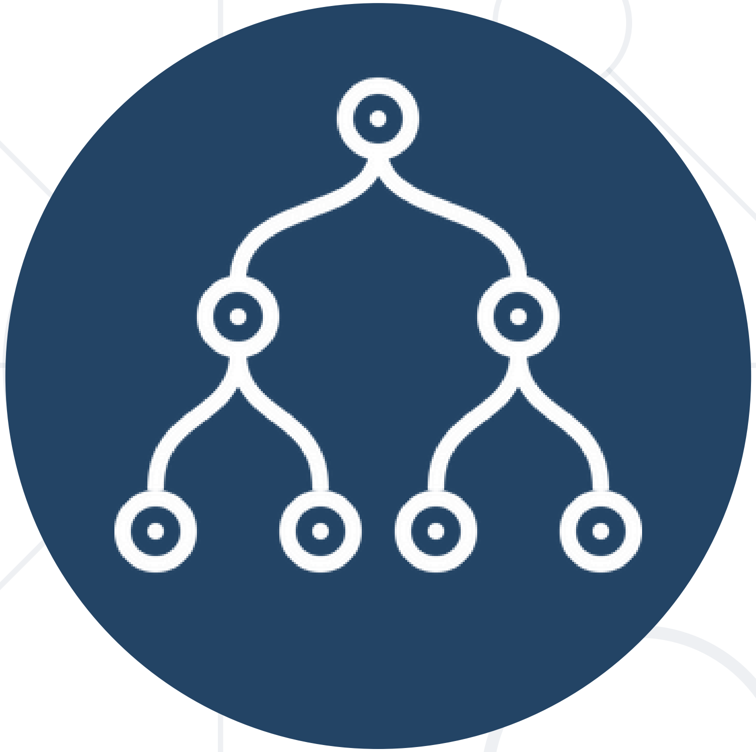
```
Spider spider(...);
Company company(...);
spider.getInfo();
company.getInfo();
```

OOP Interface - Usage Diagram

- **Company** and **Spider** are in different "trees"
  - **Company** is a "root", **Spider** is "under" the **Organism** "root"
  - Share members through **HasInfo** interface



- OOP hierarchies are often described with diagrams



Multiple Inheritance

Multiple Inheritance

- In the previous slides, we demonstrated multiple inheritance
  - But we used the "safe" way – interfaces
- C++ allows a derived class to have multiple bases

```
class Derived : public Base1, public Base2, ...
```

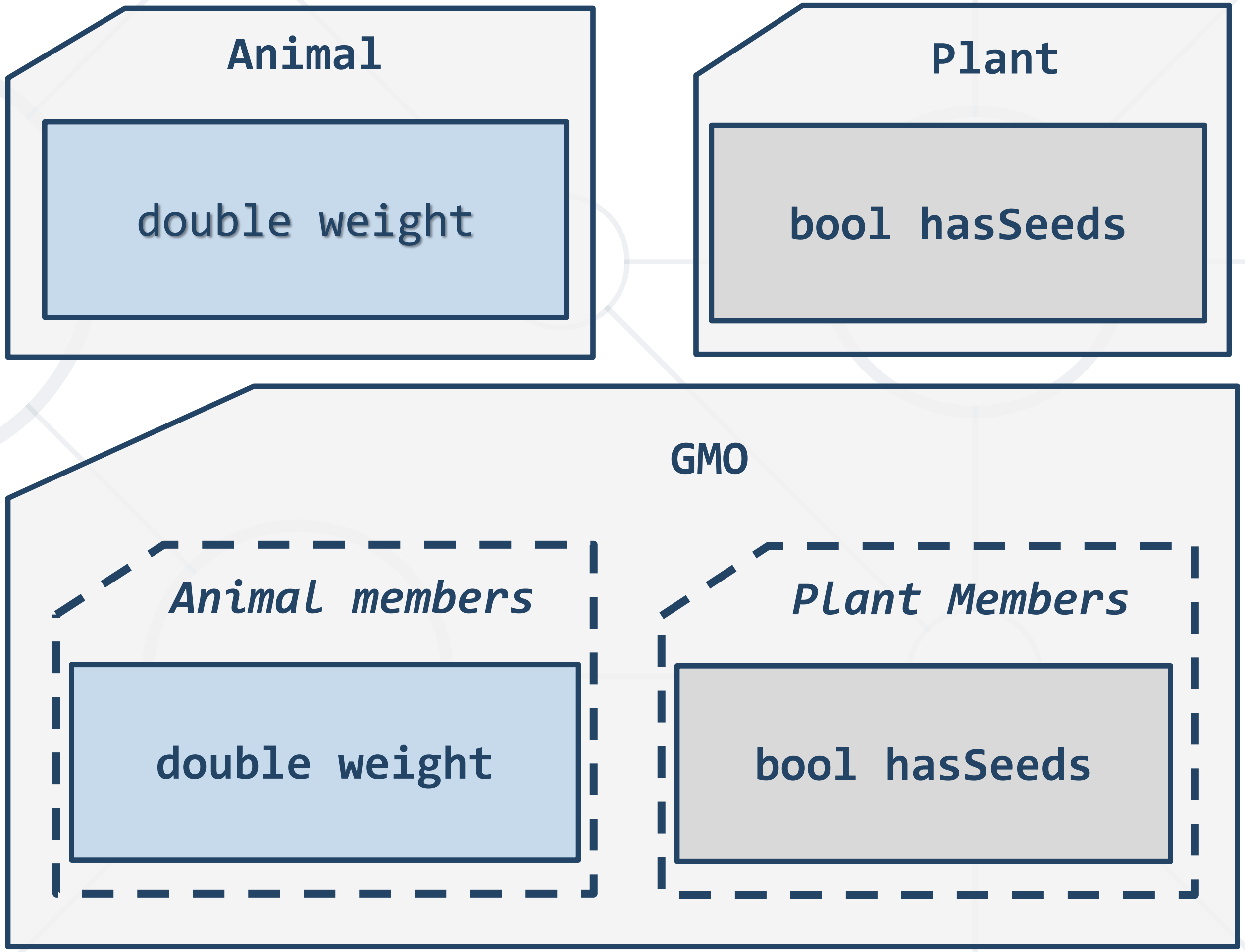
- Can cause member conflicts – if member names match
  - Internal code uses `Base1::member` vs. `Base2::member`
  - External code can be cast to `(Base1*)` or `(Base2&)`, etc.

Multiple Inheritance - Example

```cpp
class Animal {
  double weight;
};

class Plant {
  bool hasSeeds;
};

class GMO : public Animal,
            public Plant {
};
```



Multiple Inheritance - Error Prone

- With C++ multiple inheritance come multiple pitfalls
  - Name conflicts, casting, base member calls, memory, ...
  - Interfaces are mostly **immune** to the above (except name conflicts)
- The diamond problem – the root of most pitfalls
  - `class Top;`
  - `class Left : Top; class Right : Top;`
  - `class Bottom : Left, Right;`
  - Bottom has 2 copies of each Top member

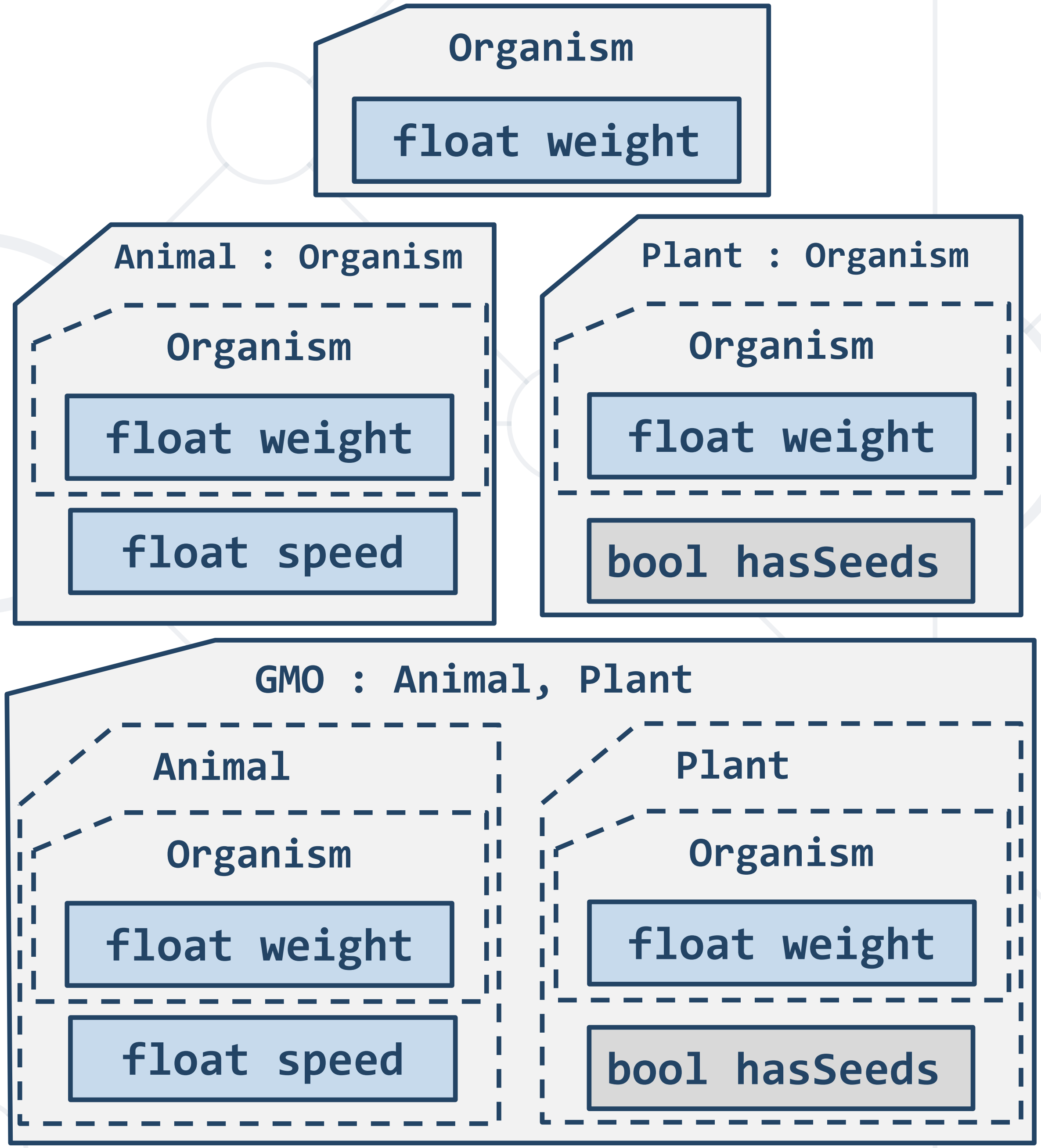
The Diamond Problem

```
class Organism {
  double weight;
};

class Animal : Organism {
  double movementSpeed;
};

class Plant : Organism {
  bool hasSeeds;
};

class GMO : Animal, Plant {
};
```



Virtual Inheritance - Solving the Diamond

- Virtual Inheritance – "override" instead of copy same members
  - `class Top;`
  - `class Left : virtual Top`
  - `class Right : virtual Top`
  - `class Bottom : Left, Right`
  - `Bottom` gets single `Top`, that both `Left` and `Right` point to

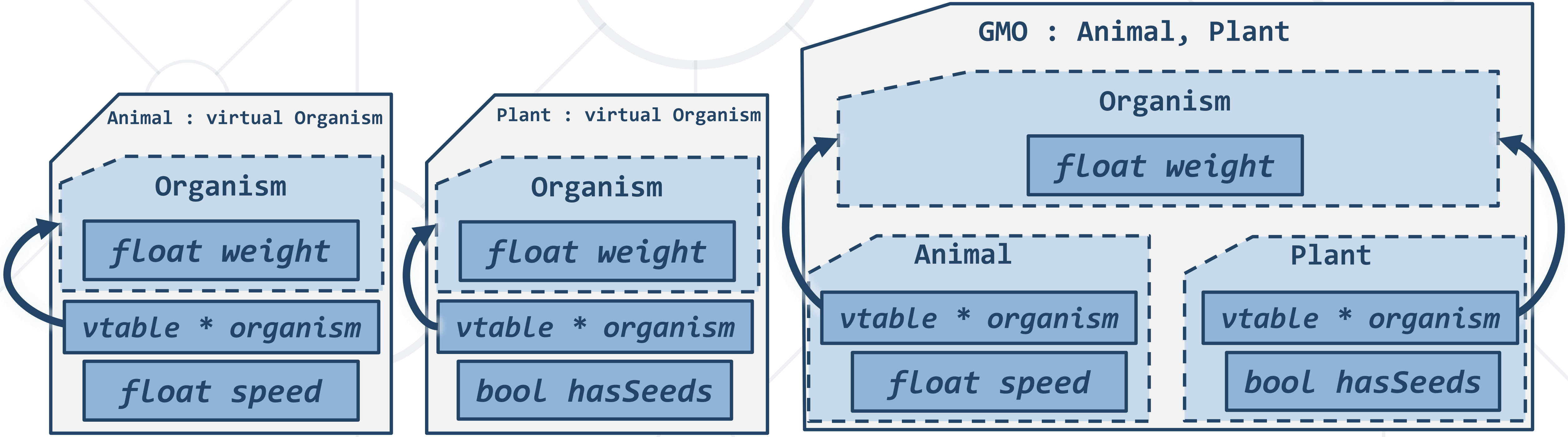
```
class Animal : public virtual Organism
```

```
class Plant  : public virtual Organism
```

```
class GMO : public Animal, public Plant
```

Solving the Diamond - Diagram

```
class Organism { ... };
class Animal : virtual Organism { ... };
class Plant : virtual Organism { ... };
class GMO : Animal, Plant { ... };
```



Runtime Type Checking

Dynamic Casting

dynamic_cast<T>(value)

- Casts **value** to **T**, **value** must be a pointer / reference
- **T** must be a pointer/reference to a class

- If a cast is not possible – returns **nullptr** if casting to pointer
  - Runtime error if casting to reference

std::dynamic_pointer_cast<T>(smartPtr)

- Similar to **dynamic_cast<T>**, but used for smart pointers

Runtime Type Checking

- `dynamic_cast` allows type checking of base pointers
  - Cast and check if the result is non-null

```
Spider spider(...);
Organism* upcast1 = dynamic_cast<Organism*>(&spider);
Company* toCompany = dynamic_cast<Company*>(&spider); // null
Organism* upcast2 = dynamic_cast<Organism*>(&spider);
```

Avoiding Runtime Type Checking

- Needing **runtime type checks** may indicate bad design
- Prefer using **overrides to define special behavior**
  - If not possible, why?
  - Do we need more classes?
  - Do we need "wider" or better base classes?
  - Is the function handling more than it is responsible for?

Summary

- C++ uses memory layout to handle inheritance
  - Base is at beginning of the memory block
  - Derived continues after base in memory
- Pure-virtual methods force implementation
  - Derived defining them guaranteed to be called due to virtual
  - Allows pure-virtual classes – OOP Interfaces
- Multiple inheritances allows combining multiple bases

Software University

Questions?

SoftUni

Software University

SoftUni Creative

SoftUni Digital

SoftUni Foundation

SoftUni Kids

СофтУни БУДИТЕЛ

Finance Academy

SoftUni Diamond Partners

Software University

SmartIT

Coca-Cola HBC Bulgaria

SUPER HOSTING .BG

createX

Postbank

Решения за твоето утре

DRAFT KINGS

VIVACOM

AMBITIONED

INDEAVR

Serving the high achievers

SOFTWARE GROUP

PHAR VISION

Trainings @ Software University (SoftUni)

- 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

License

- 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