Pure-Virtual Members & Multiple Inheritance

How Inheritance Works, Pure-virtual, Multiple Inheritance

SoftUni Foundation

Georgi Georgiev

A guy that knows C++

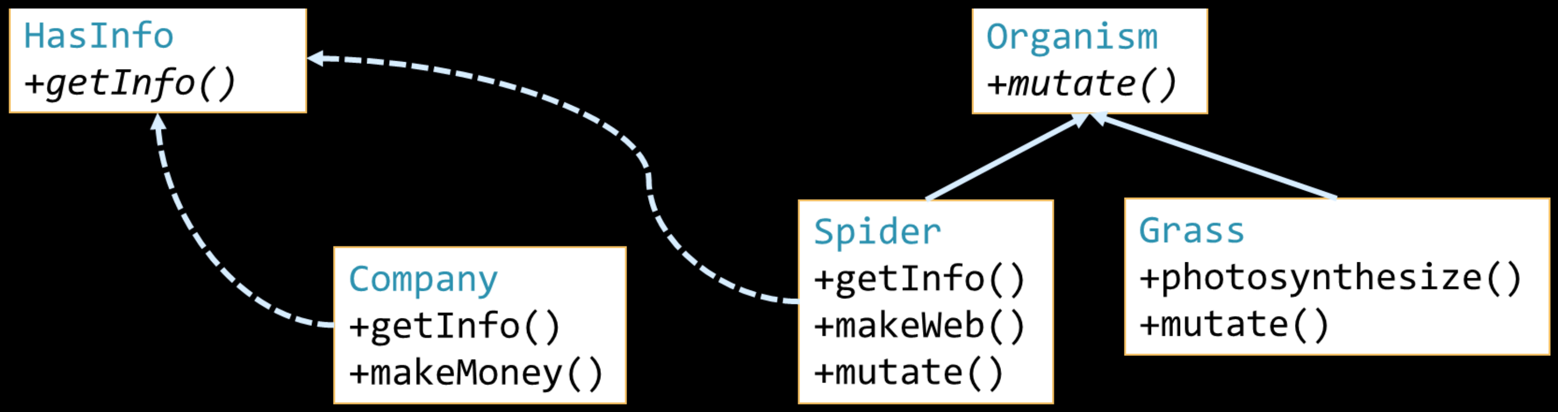


Table of Contents

Questions

sli.do

#cpp-softuni

Inheritance in Memory

Why Base Pointers Work

Objects in Memory

- Fields in memory follow declaration order (usually)
  - “Padding” is auto-added to make size a power of **2**

```
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);
```

Organism o

		weight	eatsPlants	eatsAnimals			
Address	...	0x6afe4c…0x6afe4f	0x6afe50	0x6afe51	0x6afe52	0x6afe53	...
Byte$_{(binary)}$	…	42	true	false	*padding*	*padding*	…

Inheritance in Memory

- **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);
```

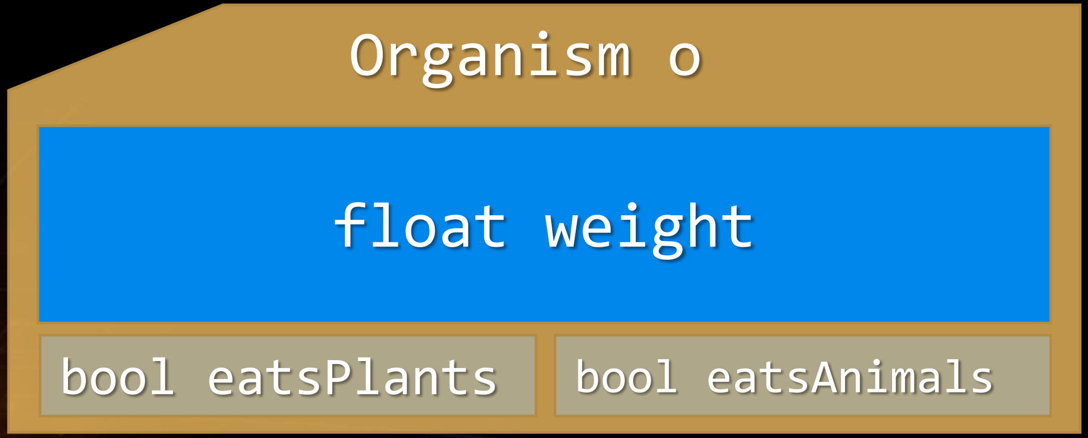
Spider s

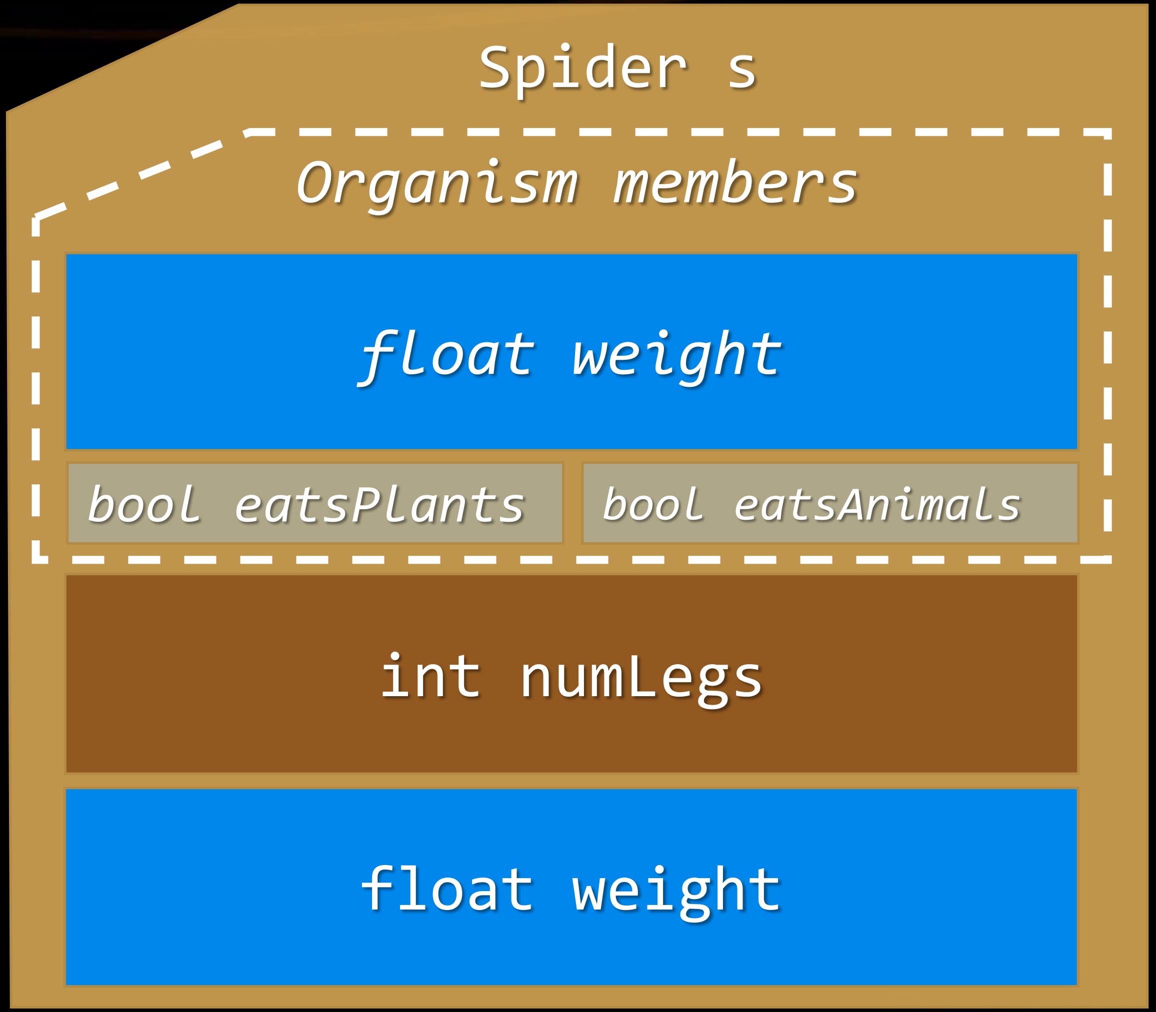
		Organism					numLegs	weight	
Address	...	0x6afe4c…0x6a53					0x6afe54…0x6afe57	0x6afe58…0x6afe61	...
Byte (binary)	…	0.1	false	true	…	…	6	0.01	…

Inheritance & Hidden Fields – Memory

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

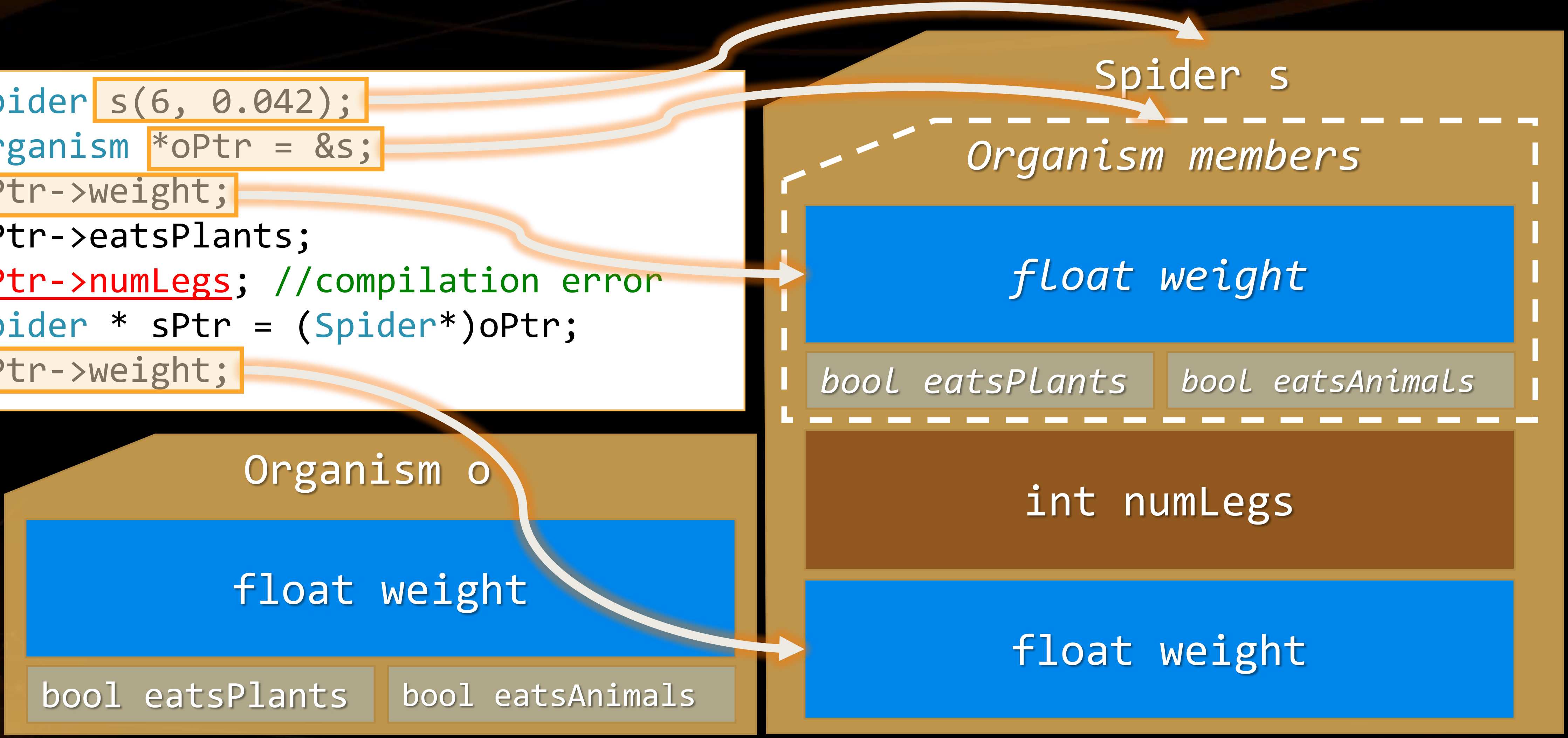
```
class Spider : public Organism {
  int numLegs; float weight;
  ...
};
```





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

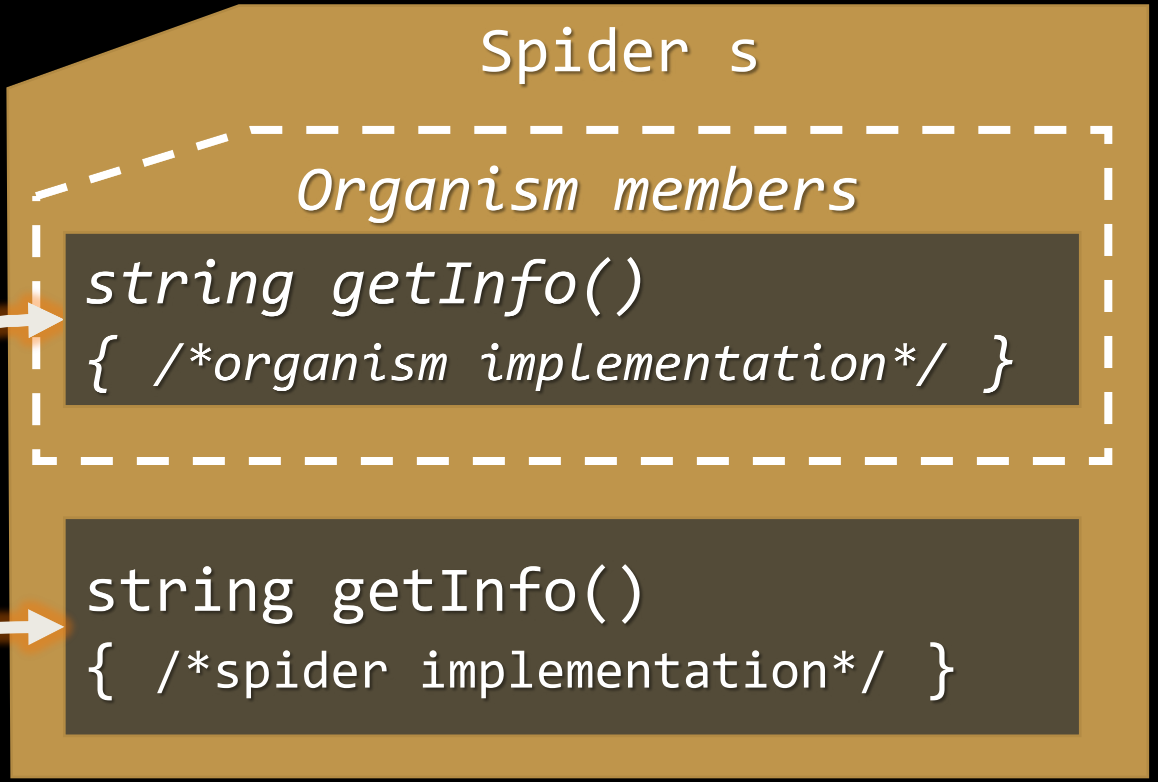
LIVE DEMO

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();
```

Spider s

Organism members

vtable * vtPtr *(hidden)*

Organism vtable (hidden)		
Method	*virtual string getInfo()*	...
Code	*{/*organism implementation*/}*	{...}

Spider vtable (hidden)		
Name	virtual string getInfo()	...
Code	{/**spider implementation**/}	{...}

Polymorphism in Memory

LIVE DEMO

Pure-virtual Methods

Base Declares Methods, Derived Implements Them

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
  - i.e. function pointer to `NULL`
  - Syntax: append `= 0;` to virtual method signature
  - E.g.: `virtual void write(string s) = 0;`

C++ Abstract Classes

- Abstract class – class containing pure-**virtual** methods
  - Can't be instantiated
  - i.e. can't 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

SoftUni Foundation

- Base declares, Derived defines/implements, Code uses Base
  - Usable methods accessible from base pointer/reference
  - Pointers guaranteed to point to derived (*can't instantiate base*)
  - Guaranteed override access – derived must have override

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

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

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

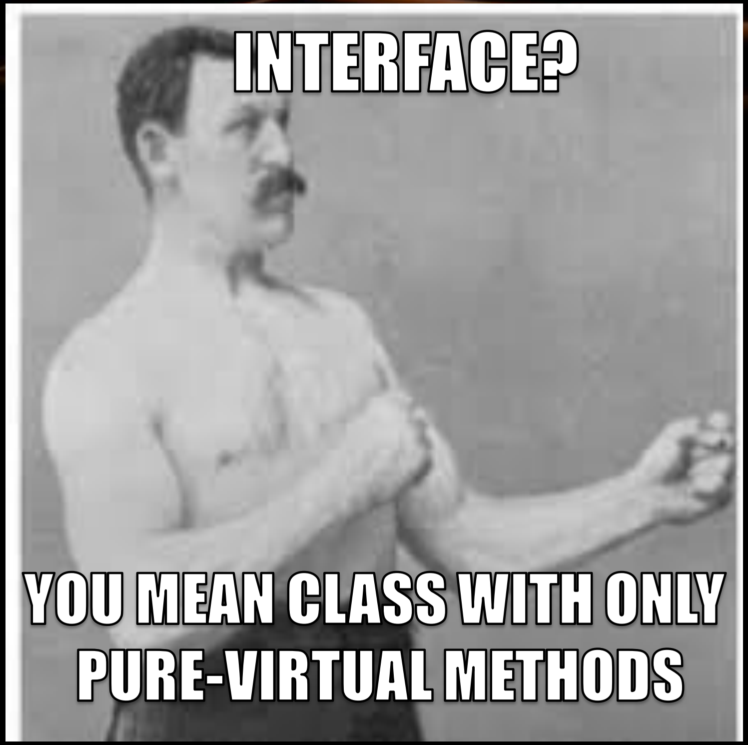
```
FileWriter fileWriter(“out.txt");
writeHello(fileWriter);
```

Pure-Virtual Methods

LIVE DEMO

Exercise 1: Zoo

- Example: Zoo of `Organisms`
  - Can act (move, stop, …), have position, image (sequence of `char`s)
  - Code provided for `Cat`, `Mouse`
  - Task: edit the code to initialize and animate objects of the above
- Approach: Several classes have common methods
  - One or more methods behave differently per class
  - Make base abstract class with common members
  - Pure-virtual for the ones with unique implementations per class



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)
- In C++ – pure-virtual classes – all methods are pure-virtual

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

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

OOP Interface – Common Usage

SoftUni Foundation

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

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

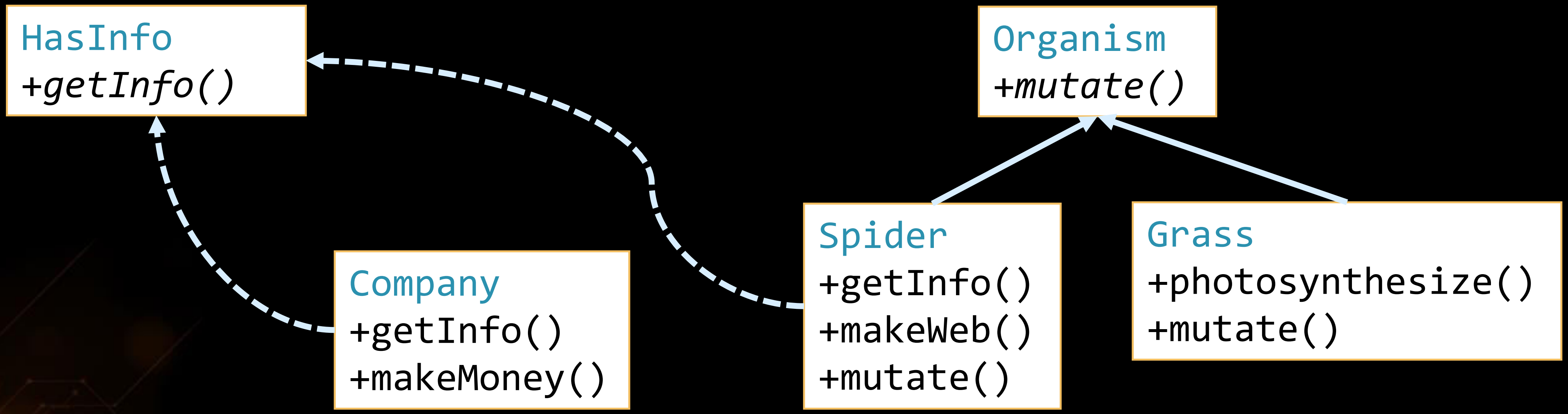
```
class Spider : public Organism
             , public HasInfo {
...
string getInfo() const override {
...
```

```
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

OOP Interfaces Usage

LIVE DEMO

Multiple Inheritance

Inheriting from Multiple Base Classes

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 cast to `(Base1*)` or `(Base2&)`, etc.

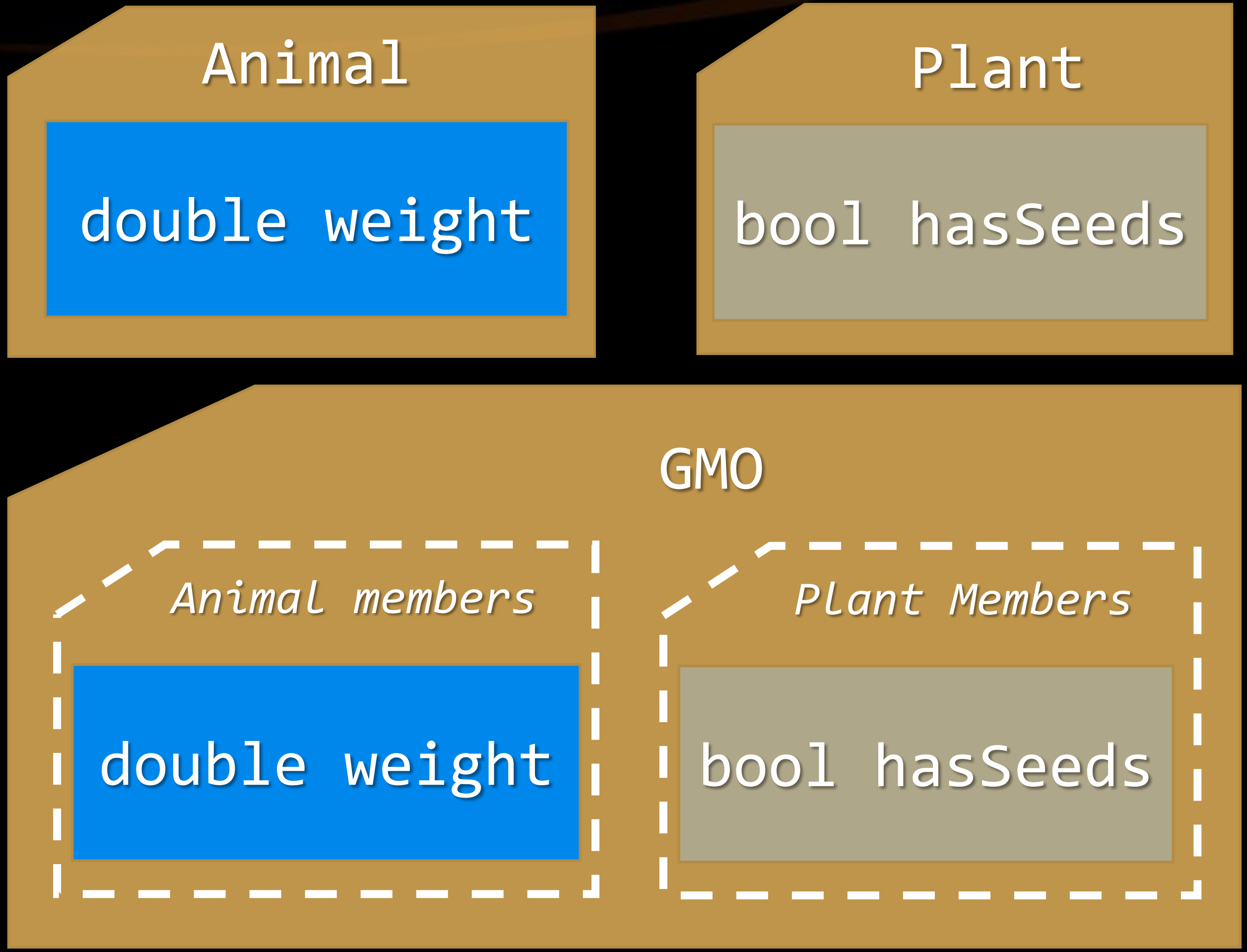
Multiple Inheritance – Example

SoftUni Foundation

```
class Animal {
  double movementSpeed;
};

class Plant {
  bool hasSeeds;
};

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



Multiple Inheritance

LIVE DEMO

Multiple Inheritance – Error Prone

SoftUni Foundation

- 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

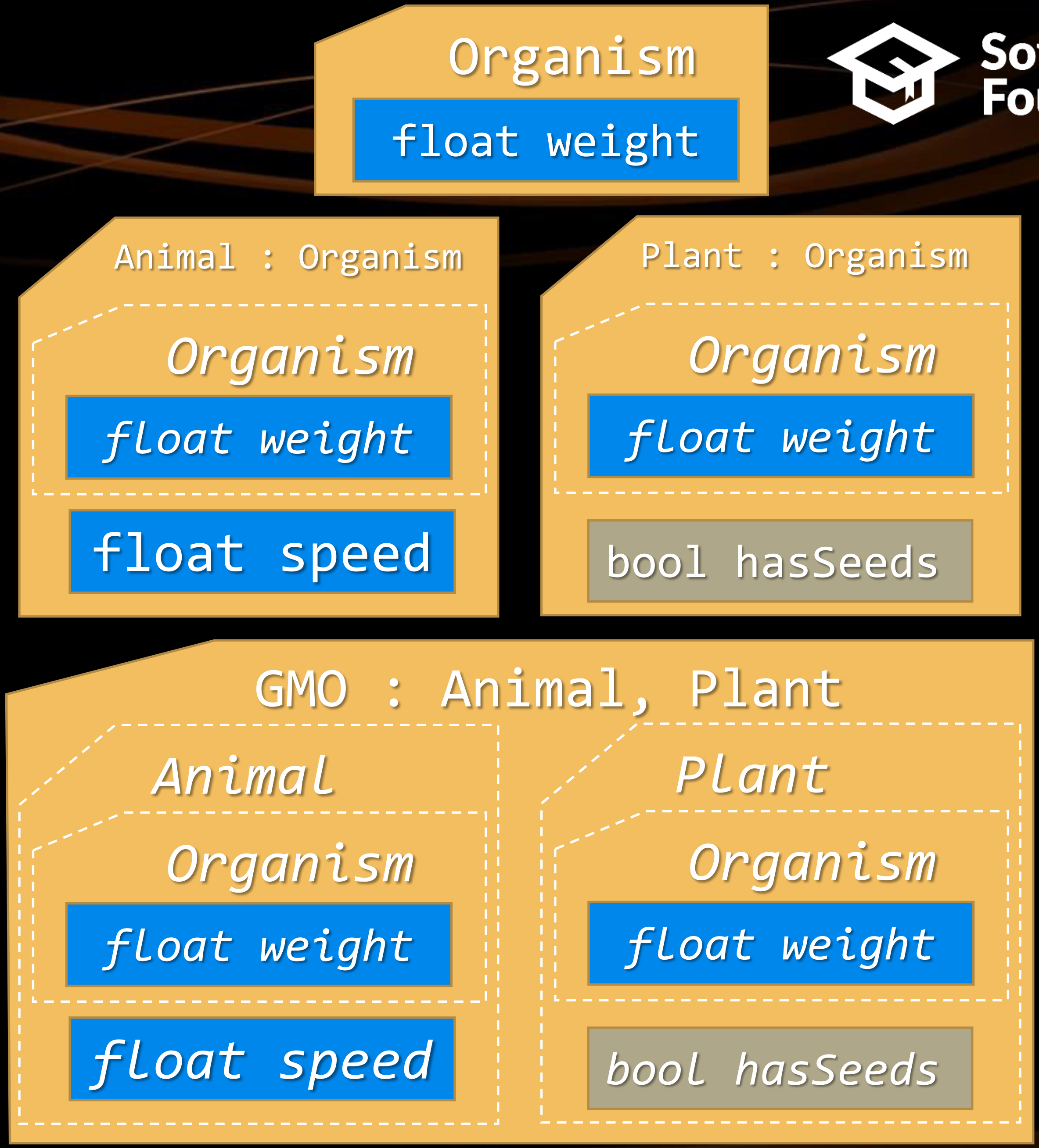
The Diamond Problem

```
class Organism {
  double weight;
};

class Animal : Organism {
  double movementSpeed;
};

class Plant : Organism {
  bool hasSeeds;
};

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



Virtual Inheritance – Solving the Diamond

SoftUni Foundation

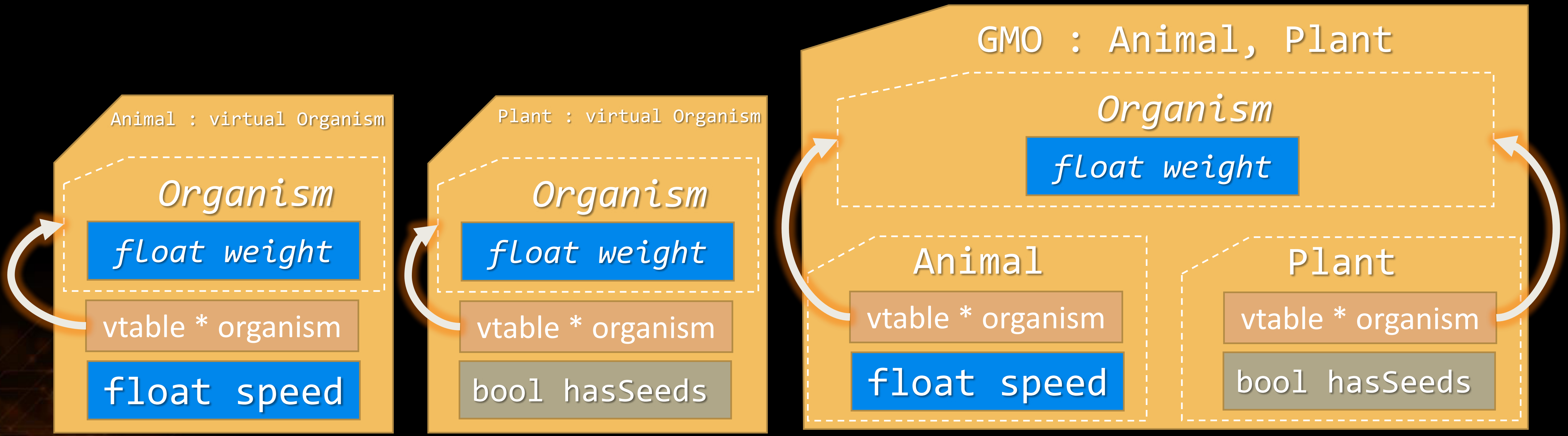
- Virtual Inheritance – "override" instead of copy same members
  - `class Top;`
  - `class Left : virtual Top`
  - `class Right : virtual Top`
  - `class Bottom : Left, Right`
  - **Bottom** gets single **Top**, that both **Left** and **Right** point to

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



Virtual Inheritance

LIVE DEMO

Runtime Type Checking

Using `dynamic_cast` for Type-Specific Handling

Dynamic Casting

- C++ has `dynamic_cast<T>(value)`
  - Casts `value` to `T`, `value` must be a pointer/reference
  - `T` must be pointer/reference to a class
- If 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 result is non-**null**

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

`dynamic_cast`

LIVE DEMO

Avoiding Runtime Type Checking

SoftUni Foundation

- *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 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 inheritance allows combining multiple bases
  - Has issues, but mostly safe with interfaces

Pure-Virtual Members & Multiple Inheritance

SoftUni Foundation

xssoftware

SmartIT

NETPEAK
SEO and PPC for Business

SUPERHOSTING®.BG

INDEAVR
Serving the high achievers

Questions?

telenor

SOFTWARE GROUP

INFRAGISTICS™
DESIGN / DEVELOP / EXPERIENCE

https://softuni.bg/courses/