Polymorphism

Iuliana Bocicor

Polymorphism

Static and dynamic binding

Virtual methods

Upcasting and downcasting

Abstract classes

Polymorphism

Iuliana Bocicor
maria.bocicor@ubbcluj.ro

Babes-Bolyai University

2024

Overview

1. Polymorphism
2. Static and dynamic binding
3. Virtual methods
4. Upcasting and downcasting
5. Abstract classes

Primary OOP features

- **Abstraction**: separating an object's *specification* from its *implementation*.
- **Encapsulation**: grouping related data and functions together as objects and defining an interface to those objects.
- **Inheritance**: allowing code to be reused between related types.
- **Polymorphism**: allowing an object to be one of several types, and determining at runtime how to "process" it, based on its type.

Polymorphism

Iuliana Bocicor

Polymorphism

Static and dynamic binding

Virtual methods

Upcasting and downcasting

Abstract classes

Definitions

- Polymorphism is the property of an entity to react differently depending on its type.
- Polymorphism is the property that allows different entities to behave in different ways to the same action.
- Polymorphism allows different objects to respond in different ways to the same message.
- Polymorphism - Greek meaning: "having multiple forms".

Polymorphism II

- Usually, polymorphism occurs in situations when one has classes related by inheritance.
- A call to a member function will cause the execution of a different code, depending on the type of object that invokes the function.
- The code to be executed is determined dynamically, at run time.
- The decision is based on the actual object.

Polymorphism III

Figure source: https://www.youtube.comwatchv=ng98qapa4Sw

Polymorphism IV

? In the code below, why are we allowed to write:
Animal a2 = &p1;* ?

```
Animal a1{"black", 20};
Penguin p1{"black and white", 7, "Magellanic"};
Animal* a2 = &p1;
cout<<a2->toString(); //which toString function?
    The one from the class Animal, or the one
   from the class Penguin?
```

Static binding

- The choice of which function to call is made at compile time.
- The object *a2* is declared as a pointer to Animal $\Rightarrow$ at compile-time it is decided that the function *Animal::toString()* will be called.
- Static binding is also called **early binding**.

Dynamic binding I

- In the presented case, the behaviour we expect is a call to the function *Penguin::toString()*.
- At runtime, the actual type of the object is determined.
- The decision of using *Animal::toString()* or *Penguin::toString()* should be taken only after determining the actual type of the object $\Rightarrow$ at runtime.
- This is **dynamic binding**: take the decision of which function body to execute according to the actual type of the object.

Dynamic binding II

- Dynamic binding is also called **late binding**.
- When a message is sent to an object, the code being called in not determined until runtime.
- Dynamic binding **only works with non-value types**: references and pointers.
- In C++, dynamic binding is achieved using virtual functions.

Virtual methods I

Declaration

virtual function_signature

- If a function is declared virtual **in a base class** and then overriden in a derived class $\Rightarrow$ dynamic binding is enabled.
- The actual function that is called depends on the content of the pointer (or reference).
- The function becomes polymorphic by being designed virtual.

Virtual methods II

- The function in the derived class that is overriding the function in the base class can use the override specifier to ensure that the function is overriding a virtual function from the base class.

- override is an identifier with a special meaning when used after member function declarators and otherwise, it is not a reserved keyword.

DEMO

Polymorphic function *toString* (Animal - Penguin, Dog) (*Lecture6_polymorphism*).

C++ mechanism I

- In memory, for an object with no virtual functions, only its own data is stored.
- Member functions or pointers to them are **not** stored in the object. They are stored in a code memory section, and are known to the compiler.
- When a member function is called, the pointer to the current object (this) is passed as an invisible parameter so the functions know on which object to operate on when they are called.

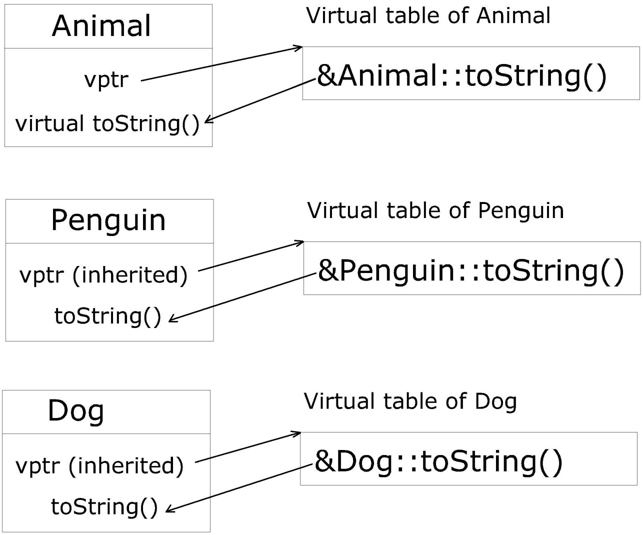
C++ mechanism II

- Things are different when virtual functions come into play.
- In the case of a derived class with virtual functions, the compiler creates a table of function addresses called the **virtual table** - a static array set up by the compiler at compile time.
- Every class that uses virtual functions (or is derived from a class that uses virtual functions) will have its own virtual table.
- Each entry in the virtual table is a function pointer that points to the most derived function accessible by the class.

C++ mechanism III

- A pointer to the virtual table (vptr) is added to the base class - and inherited by the derived classes.
- When a class object is created, the pointer to the virtual table is set to point to the virtual table for that class.
- When a call is made through a pointer or reference, the compiler generates code that dereferences the pointer to the object's virtual table and makes an indirect call using the address of a member function stored in the table.

C++ mechanism IV



C++ mechanism V


```
Animal* p = new Penguin{ "black and white",
    7, "Magellanic" };
```

p	0x005bab78 {type="Magellanic" }
Animal	{colour="black and white" weight=7.0000000000000000 }
__vfptr	0x01232488 {Lecture6_demo_virtual_functions.exe!const Penguin::`vftable'} {0x012212cb {Le
[0]	0x012212cb {Lecture6_demo_virtual_functions.exe!Penguin::toString(void)const }
colour	"black and white"
weight	7.0000000000000000
type	"Magellanic"

C++ mechanism VI

- The virtual function mechanism works only with pointers and references, **but not** with value-types (objects).

- Calling a virtual function is slower than calling a non-virtual function:
  1. Use the vptr to access the correct virtual table.
  2. Find the correct function to call in the virtual table.
  3. Call the function.

- Declare functions as virtual only if necessary.

DEMO

Virtual functions. (*Lecture_6_polymorphism*).

Virtual constructors?

- Constructors **cannot** be virtual.
- When creating an object, one must know exactly what type of object one is creating.
- Usually, the virtual table pointer is initialized in the constructor.

Virtual destructors I

- A derived class object contains data from both the base class and the derived class.
- The destructor's responsibility is to dealocate resources (memory).
- In the case of a derived class object, it is essential that both the destructor of the base class and the destructor of the derived class are called.

Virtual destructors II

- The correct destructor must be invoked based on the actual type of the object, not the type of the pointer holding the reference to the object.
- Therefore, the destructor must have a polymorphic behaviour $\Rightarrow$ the base class destructor must be virtual.

DEMO

Virtual destructor. (*Lecture_6_polymorphism*).

Upcasting and downcasting I

Upcasting

- Casting an object/reference/pointer of a derived class to an object/reference/pointer of the base class.
- Casting up the hierarchy.
- Allows us treating a derived type as though it were its base type.
- Is always allowed for public inheritance, without an explicit cast, as a a derived class object has all the members of the base class (and more).

Upcasting and downcasting II

Downcasting

- Casting a base-class pointer/reference to a derived-class pointer/reference.
- Casting down the hierarchy.
- Is **not** allowed without an explicit type cast (requires explicit casting from the user). **?** Why?
- For explicit casting: static_cast, dynamic_cast.

Upcasting and downcasting III

- static_cast:
  - converts a reference/pointer to a specified type;
  - will check, at compile time, if the types are compatible (in the same inheritance hierarchy);
  - does not perform runtime checking; does not check if the object being converted is "complete" $\Rightarrow$ bad casts can lead to runtime errors.

Upcasting and downcasting IV

- dynamic_cast:
  - converts a reference/pointer to a specified type;
  - will check, at runtime, if the object can be converted and if it cannot, it returns nullptr or an error;
  - only works with pointers or references.

DEMO

Upcasting and downcasting (*Lecture6_demo_virtual_functions - upCasting() and downCasting()*).

Pure virtual functions I

- The definition of a method from a class can be omitted by making the function **pure virtual**.

Syntax

virtual function_signature =0;

- A **pure virtual** function is a function with no body.
- All the derived classes will have to define the function.

Pure virtual functions II

- The compiler will reserve a slot for the pure virtual function in the virtual table, but will not add any address in that particular slot.
- A destructor can be declared pure virtual, but if any objects of that class or any derived class are created in the program, the destructor shall be defined.

Abstract classes I

- A class containing *at least* one pure virtual function is called an **abstract class**.
- An abstract class cannot be instantiated (one cannot create objects of that type).
- There are cases in which one needs the base class only as a starting point for derived classes.
- In reality, there are penguins and dogs and koala bears, but no generic animals.

Abstract classes II

- When a class is abstract, the compiler will not allow the creation of objects of that class.

- An abstract class serves as a base class for a collection of related derived classes and it provides:
    - a common public interface (or pure virtual member functions);
    - any shared representation;
    - any shared member functions.

Abstract classes III

Extending an abstract class

- A concrete class that extends an abstract class inherits its public interface.
- A concrete class is expected to have instances.
- Override "abstract" functions to provide specific implementation (otherwise the derived classes will also be abstract classes).

Homogeneous containers and polymorphism

- One example of use of polymorphism is on containers holding elements of the "same type" (from the same class hierarchy).

- A message is sent to each of the objects in the container and they must respond in their specific way.

DEMO

Abstract classes. (*Lecture6_demo_abstract_classes*).

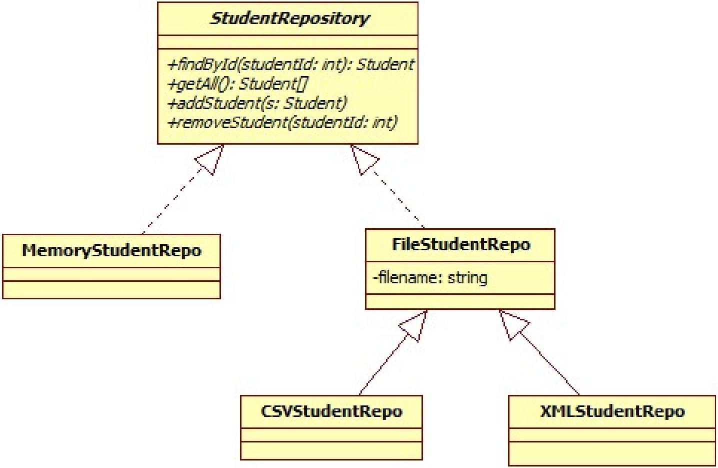
Pure abstract classes I

- A **pure abstract class** contains nothing but pure virtual methods.
- A pure abstract class is also called an **interface**.
- An interface describes the capabilities of a class without committing to a particular implementation of that class.
- The UML representation for abstract entities (functions or classes): *italic font*.

Pure abstract classes II

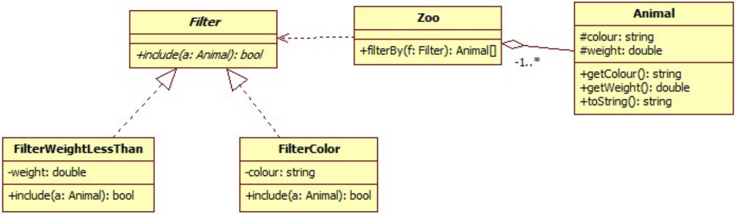
- Remember *MemoryRepository* and *FileRepository* in Fundamentals of Programming?
- We started with an in-memory repository, and added a file-based one.
- Defining an interface would allow us to use any class that implements it.

Pure abstract classes III



Example - pure abstract class I

- **Requirement**: given a list of animals, display, in turns, the animals having:
  - the weight smaller than a given value;
  - the colour equal to a given colour.



Example - pure abstract class II

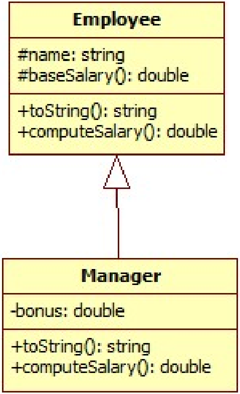
DEMO

Abstract classes. (*Lecture6_demo_abstract_classes* - Filter.h and filterAnimals()).

Exercise I

Write the C++ code corresponding to the following UML class diagram related to companies and their employees.

Exercise II



Employee
#name: string
#baseSalary(): double
+toString(): string
+computeSalary(): double
Manager
-bonus: double
+toString(): string
+computeSalary(): double

Exercise III

- The company has several employees, some of them are managers.
- The **toString** method from Employee returns a string with the name of the employee.
- The **toString** method from Manager returns a string with the word "Manager" and the name of the employee.
- The **computeSalary** method from Employee returns the base salary.
- The **computeSalary** method from Manager returns the base salary, to which the manager bonus is added.

Exercise IV

- Write a test program that creates several employees (both regular employees and managers), add all the employes into a list (vector).
- Create a function that for a list of employees will print out the proper name and salaries for all the employes, using the values returned by the **toString** and **computeSalary** methods.

Inheritance and polymorphism - benefits

- code reuse:
  - derived classes inherit from the base class;
  - code duplication is avoided $\Rightarrow$ better maintanance, evolution, understanding.
- extensibility:
  - generic code;
  - new functionalities can be added without modifying the existing code;
  - extension points are provided for further evolution.