Object Oriented Programming

Inheritance - polymorphism

2023/24

Lecture Outline

- Relationship between overloading, method hiding, and protected access.
- Types of behavior change, polymorphism.
- Example.

Overloading, shadowing, *protected*

Overloading x Behavior Change

- Overloading solves the addition of behavior. This is an extension, although the method has the same name.
- Shadowing solves a static change in behavior.
- Polymorphism is something more…

Behavior Change Problem

- We often need to access details of the implementation.
- However, implementation details should be hidden.
- Is it possible to access the private items of the ancestor(s)?

State and Behavior Access

	public	private	protected
klient	x	-	-
třída	x	x	x
potomek	x	-	x

Protected Access

- Access to implementation details can be solved by using *"protected.„*
- Is it correct?
- Or is it wrong? *And why?*

```
class Account
{
private:
    int number;
    double interestRate;

    Client *owner;

protected:
    double balance;

public:
    Account(int n, Client *o);
    Account(int n, Client *o, double ir);

    int GetNumber();
    double GetBalance();
    double GetInterestRate();
    Client *GetOwner();
    bool CanWithdraw(double a);

    void Deposit(double a);
    bool Withdraw(double a);
    void AddInterest();
};
```

```
class CreditAccount : public Account
{
private:
    double credit;

public:
    CreditAccount(int n, Client *o, double c);
    CreditAccount(int n, Client *o, double ir, double c);

    bool CanWithdraw(double a);
    bool Withdraw(double a);
};
```

```
bool CreditAccount::CanWithdraw(double a)
{
    return (this->GetBalance() + this->credit >= a);
}
```

```
bool CreditAccount::Withdraw(double a)
{
    bool success = false;
    if (this->CanWithdraw(a))
    {
        this->balance -= a;
        success = true;
    }
    return success;
}
```

Using *protected*…

- …violates encapsulation
- Consequences:
  - If we decide to change the implementation of the ancestor, it may affect the implementation of a descendant.
  - The descendant becomes implementation-dependent on the ancestor (and vice versa).

What is polymorphism?

Polymorphism

- *Polymorphism* is the ability of an object to appear in different roles (forms)…
- …and behave accordingly;
  - it combines its behavior with the behavior of an ancestor; otherwise, it is not actual polymorphism…
- This is related to the substitution principle, i.e., the substitutability of the ancestor by the descendant.

Polymorphic Assignment

- The source of assignment is of a different type than the target of the assignment.

```
Client *o = new Client(0, "Smith");
CreditAccount *ca = new CreditAccount(1, o, 1000);
```

```
Account *a = ca;
```

- Assigning an inherited class pointer to a pointer of its base (ancestor) class.

Key Property of Polymorphic Assignment

Feature Call rule

In a feature call $x.f$, where the type of x is based on a class C, feature f must be defined in one of the ancestors of C.

Shadowing x Polymorphism

- Does shadowing ensure that it is a polymorphism?
- **NO!**
- Why?
- Because the descendant in the role of ancestor behaves exactly like this ancestor (behavior is not combined).

Without „*protected*“?

- How else to get access to ancestor's private member items?
- When these items are hidden for the descendant due to "private“ in its ancestor class.

What do we want?

Let us take a step back…

```
class Account
{
private:
    int number;
    double balance;
    double interestRate;

    Client *owner;

public:
    Account(int n, Client *o);
    Account(int n, Client *o, double ir);

    int GetNumber();
    double GetBalance();
    double GetInterestRate();
    Client *GetOwner();
    bool CanWithdraw(double a);

    void Deposit(double a);
    bool Withdraw(double a);
    void AddInterest();
};
```

```
class CreditAccount : public Account
{
private:
    double credit;

public:
    CreditAccount(int n, Client *o, double c);
    CreditAccount(int n, Client *o, double ir, double c);

    bool CanWithdraw(double a);
};
```

```
bool Account::Withdraw(double a)
{
    bool success = false;
    if (this->CanWithdraw(a))
    {
        this->balance -= a;
        success = true;
    }
    return success;
}
```

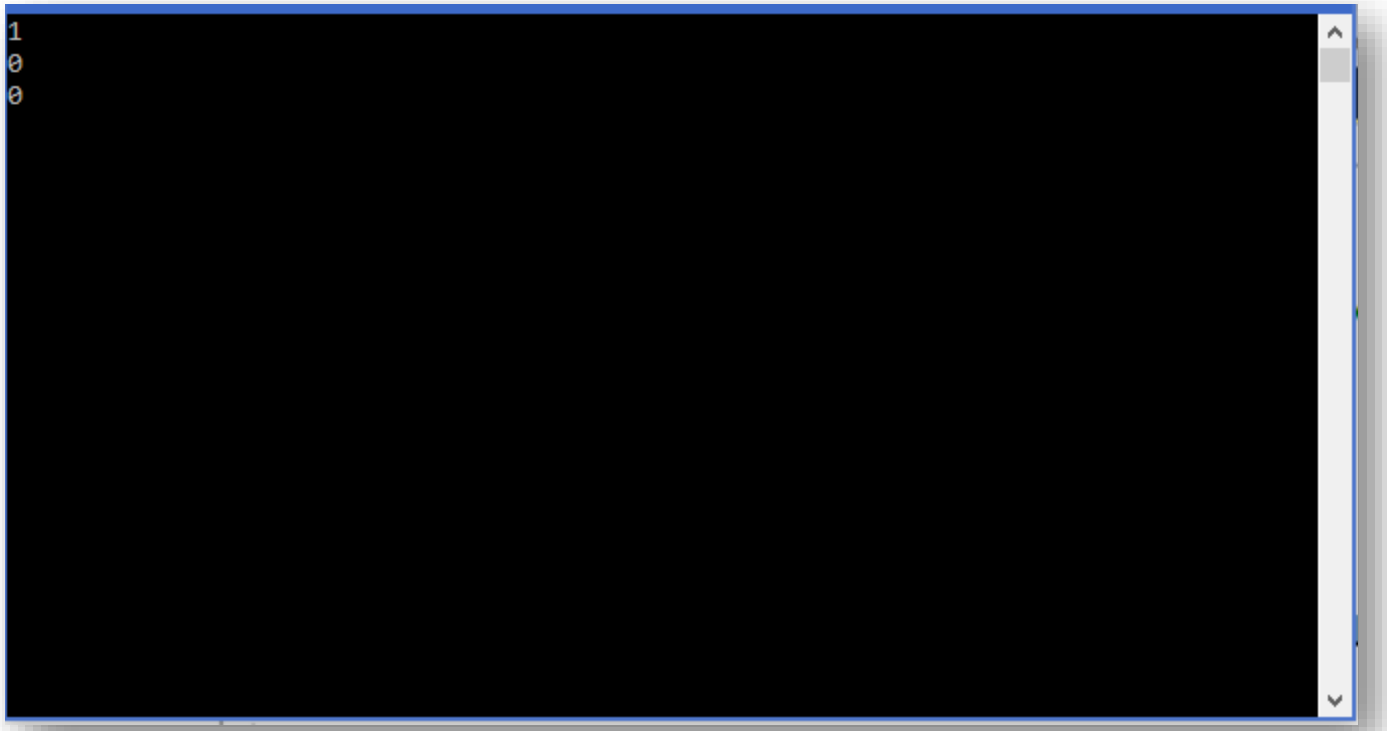
```
bool Account::CanWithdraw(double a)
{
    return (this->balance >= a);
}
```

```
bool CreditAccount::CanWithdraw(double a)
{
    return (this->GetBalance() + this->credit >= a);
}
```

Does it work?

```
int main()
{
    Client *o = new Client(0, "Smith");
    CreditAccount *ca = new CreditAccount(1, o, 1000);

    cout << ca->CanWithdraw(1000) << endl;

    Account *a = ca;

    cout << a->CanWithdraw(1000) << endl;
    cout << ca->Withdraw(1000) << endl;

    a = nullptr;
    delete ca;

    getchar();
    return 0;
}
```

Not as we would expect…



Early Binding

- The compiler normally uses so-called *early binding*, which evaluates the type of instance when calling the method already at the compilation time.

- In the *Withdraw* method, the *CanWithdraw* method of the ancestor is called.

Late Binding

- We need to find out who is requiring the method, however, at the moment of the call.
- In our case, this is not possible because the early binding is used.
- The *late binding* must be used for this.

Shadowing x Overriding

- *Shadowing* (method hiding). This is a static solution, where the new descendant method "shadows" the ancestor method.
  - The partial behavior of the object, therefore, corresponds to the class in whose role it acts.
- *Overriding*. It is a dynamic solution, where the descendant method is always used (even in the role of an ancestor) if it has it implemented.
  - Therefore, the partial behavior of the object corresponds to the class of which this object is an instance.

Example

```
class Account
{
private:
    int number;
    double balance;
    double interestRate;

    Client *owner;

public:
    Account(int n, Client *o);
    Account(int n, Client *o, double ir);

    int GetNumber();
    double GetBalance();
    double GetInterestRate();
    Client *GetOwner();
    virtual bool CanWithdraw(double a);

    void Deposit(double a);
    bool Withdraw(double a);
    void AddInterest();
};
```


1
1
1

Virtual Method

- If we want to decide which method will be called during the program execution (*overriding*), we must mark the method with the keyword virtual.

- Then we indicate to the compiler that we wish to use dynamic or *late binding*.

- Once marked as **virtual,** the method remains **virtual** in all descendants!!!

Virtual Method Table (VMT)

- Once we define a method as virtual, the compiler adds an "invisible pointer" to the class that points to a particular table called the Virtual Method Table *(VMT)*.
- For each class with at least one virtual method, the compiler creates one VMT.
- The table contains pointers to all virtual methods.
- The table is common to all instances of the class.

Virtual Constructor?

- **No!**

- The pointer to the VMT has not yet been created before calling the constructor for the first time.

- We can call virtual methods inside constructors, but they will behave non-virtually.

Virtual Destructor?

- **YES!**

```
CreditAccount *ca = new CreditAccount(1, o, 1000);
Account *a = ca;
delete a;
```

- Which destructor is called if it is not virtual? Is it correct? And why?

- And which destructor is called if the destructor is virtual?

Polymorphism

- Polymorphism is associated with inheritance.
- There is no actual polymorphism if we do not use virtual methods (*overriding*).
- It is still a matter of substitutability of the ancestor by the descendant.

Virtual Methods

- The descendant uses the virtual method in various contexts:
  - In cases where this virtual method is used in the body of a method of an ancestor.
  - Unlike the *shadowing,* even in the case of polymorphic assignment.

Polymorfic Data Structures

- A structure that contains objects of different classes.
  - E.g., array, list,…, which is of type "Ancestor" (a pointer).
- We can only call common ancestor methods for objects stored in these structures.
- How to call other methods of an object returned in an ancestor type?
  - It needs to be retyped (casting) - this is one of the limitations of polymorphism.

Seminar Assignments

- Implement examples from the presentation, focus on using the virtual methods, and understand how it works with the polymorphic assignment.

- Design and implement a simple inheritance hierarchy of geometric figures that will share the "Area" and "Perimeter" virtual methods. Use a polymorphic data structure (e.g., an array of pointers) and analyze the behavior when using the substitution principle (especially when comparing with shadowing).

Seminar Questions

- What is the difference between shadowing and overriding? Give examples
- What do we mean by polymorphism, and what is it related to?
- What do we mean by polymorphic assignments?
- What is early binding? Give examples.
- What is late binding? Give examples.
- Describe what a virtual method is and its properties.
- Describe what a virtual method table is and how it works.
- Can the constructor be virtual? And why?
- Can the destructor be virtual? And why?
- When are we speaking about polymorphism in C++, and how will this be reflected in the design?
- What is a polymorphic data structure, and what do we use it for?
- When do we need a virtual destructor? What is it related to?

Sources

- Bertrand Meyer. *Object-Oriented Software Construction*. Prentice Hall 1997. [467-472]