Programming II

Polymorphism

Lecture Outline

- Overloading, overriding, *protected* access

- What is the polymorphism?

- Example

Overloading, overriding, *protected*

Overloading x Overriding

- By overloading, the new behavior is added. It is an extension, although the method has the same name.
- Overriding solves a real change of behavior.
- *Polymorphism* is something more.

Problem of behavior changes

- Usually, we need access to the details of implementation.
- However, implementation details should be hidden.
- Can we have access to private items ancestor (ancestors)?

Access to State and Behavior

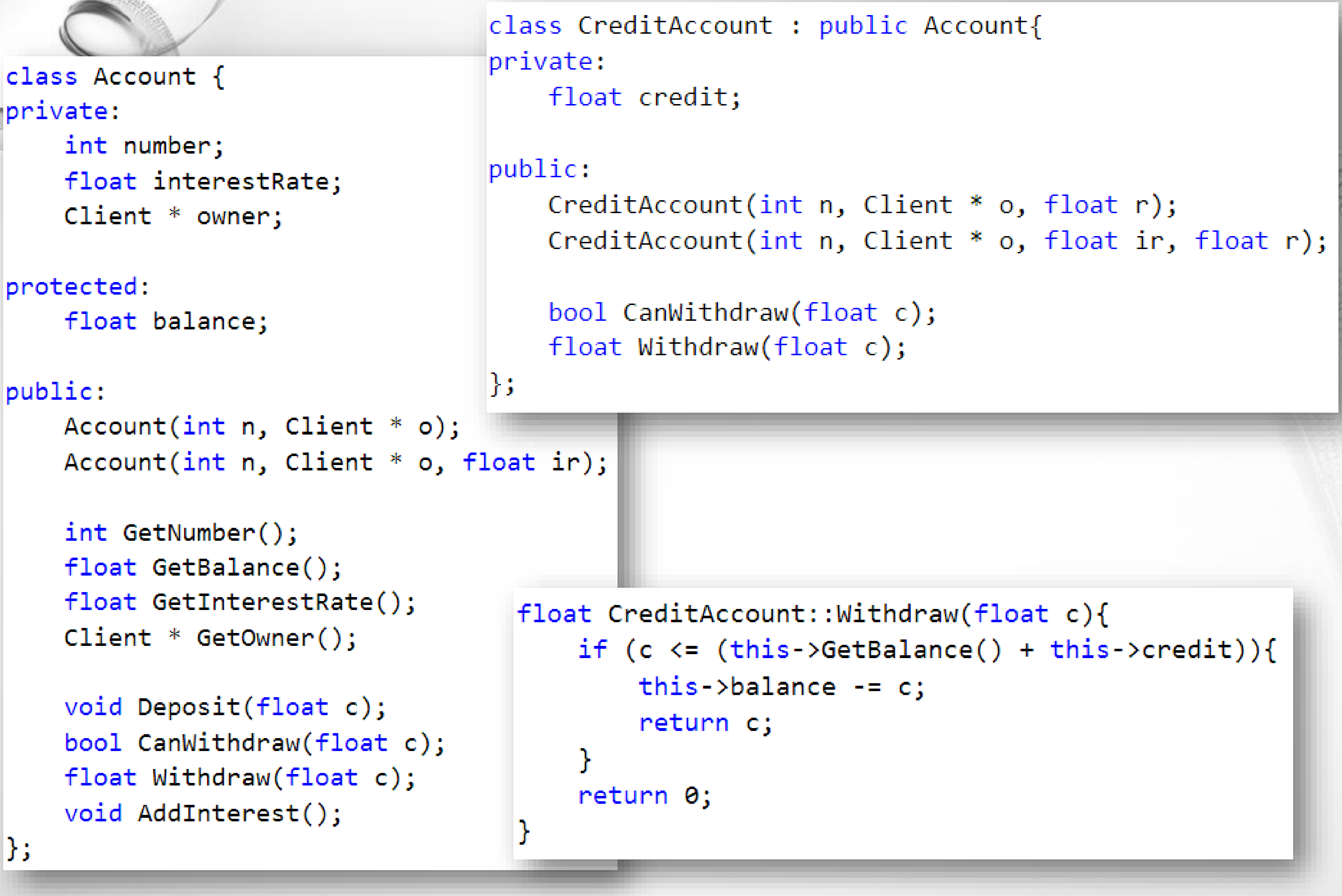
	public	private	protected
klient	x	-	-
třída	x	x	x
potomek	x	-	x

Protected Access

- Access to details of implementation can be solved by using the "*protected*".

- However, is it correct?

- Is it wrong? Why?



```
class Account {
private:
    int number;
    float interestRate;
    Client * owner;

protected:
    float balance;

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

    int GetNumber();
    float GetBalance();
    float GetInterestRate();
    Client * GetOwner();

    void Deposit(float c);
    bool CanWithdraw(float c);
    float Withdraw(float c);
    void AddInterest();
};
```

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

public:
    CreditAccount(int n, Client * o, float r);
    CreditAccount(int n, Client * o, float ir, float r);

    bool CanWithdraw(float c);
    float Withdraw(float c);
};
```

```
float CreditAccount::Withdraw(float c){
    if (c <= (this->GetBalance() + this->credit)){
        this->balance -= c;
        return c;
    }
    return 0;
}
```

By using protected…

- …the encapsulation is violated
- Consequences:
  - If we decide to change the ancestor implementation, it may influence the implementation of the child.
  - Descendant becomes implementation-dependent on ancestor (and the reverse is also true).

What is the polymorphism?

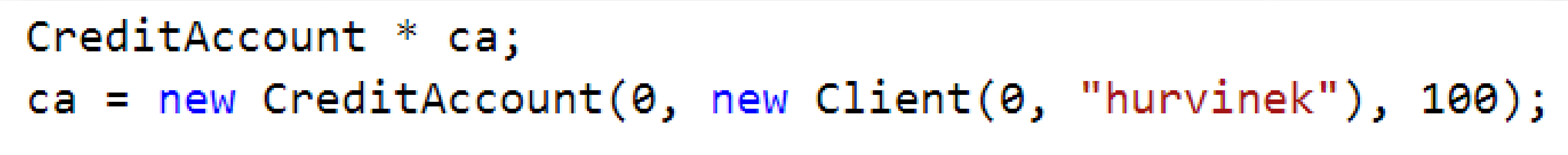
Polymorphism

- *Polymorphism* is the ability of an object to play many roles (forms)…
  - …and behaves accordingly.

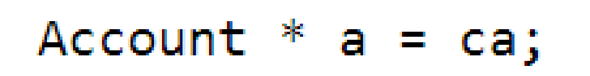
- It is related to the substitution principle (the substitutability an ancestor by a descendant).

Polymorphic Attachment (assignment)

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

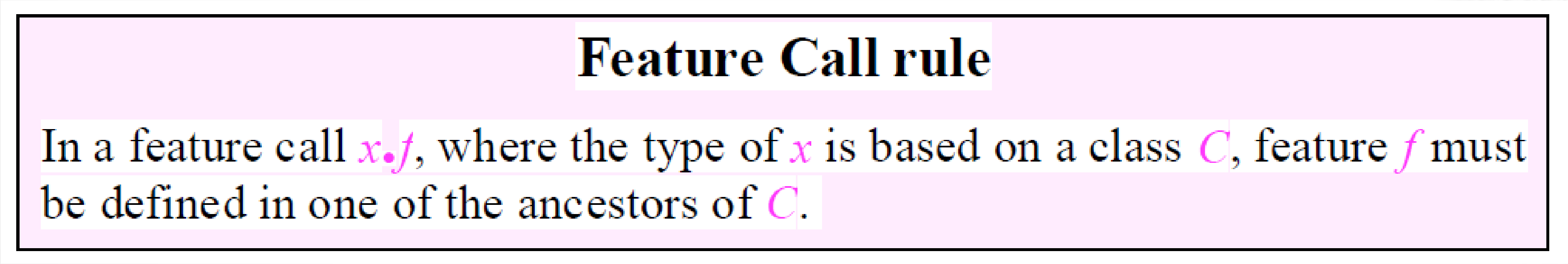


```
CreditAccount * ca;
ca = new CreditAccount(0, new Client(0, "hurvinek"), 100);
```



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

Property of Polymorphic Attachment



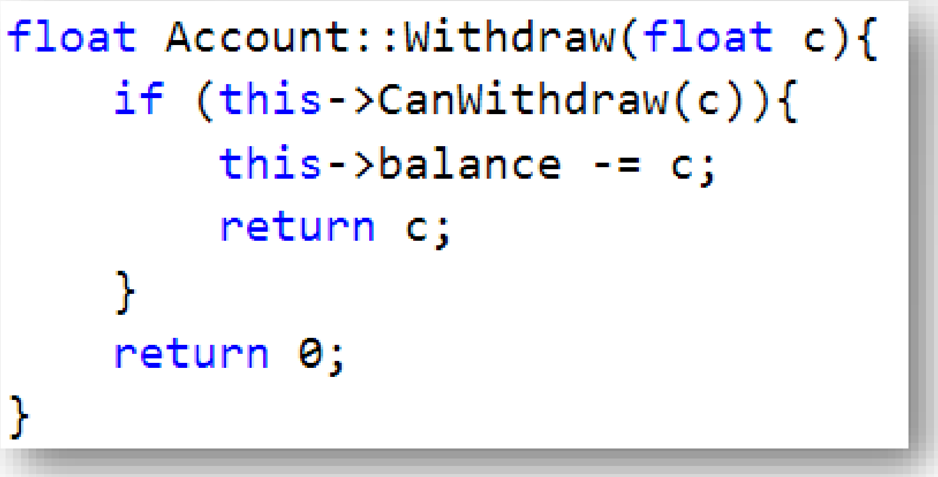
Feature Call rule

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.

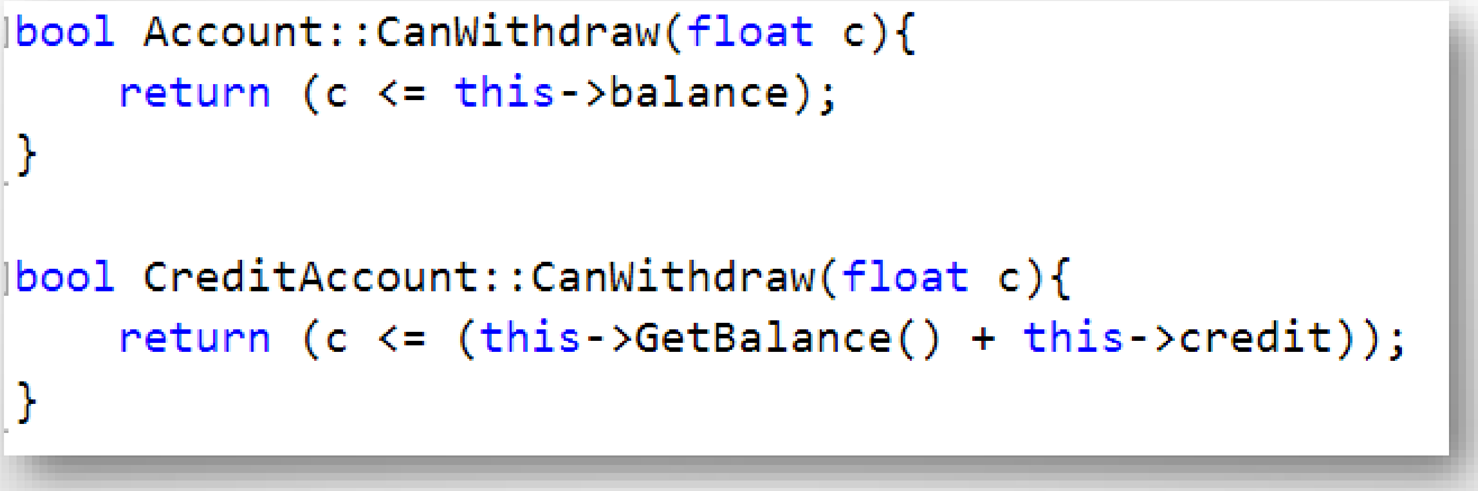
Overriding x Polymorphism

- Is the overriding the same as polymorphism?
- NO!
- Why?

???



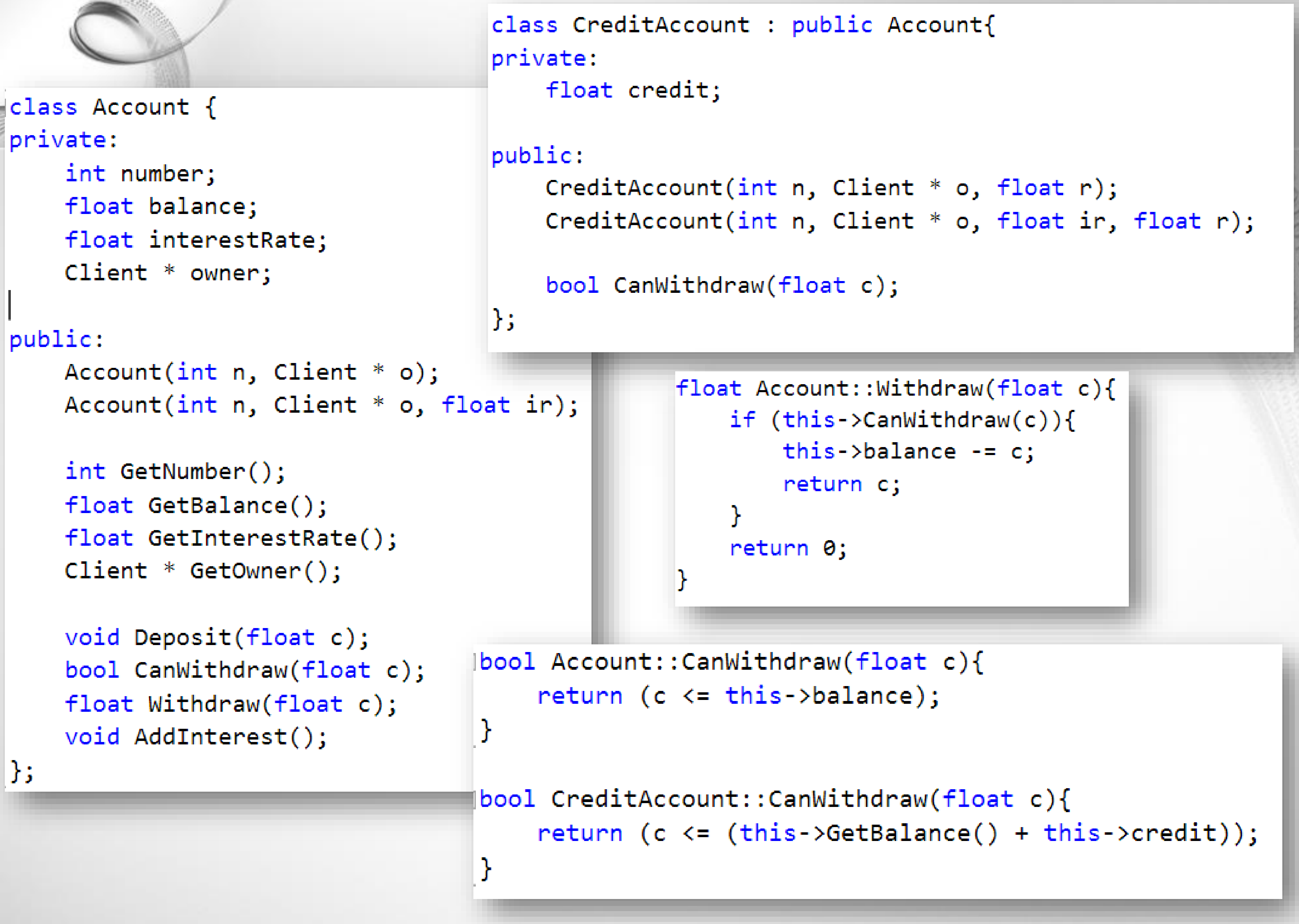
```
float Account::Withdraw(float c){
    if (this->CanWithdraw(c)){
        this->balance -= c;
        return c;
    }
    return 0;
}
```



```
bool Account::CanWithdraw(float c){
    return (c <= this->balance);
}

bool CreditAccount::CanWithdraw(float c){
    return (c <= (this->GetBalance() + this->credit));
}
```

Without „*protected*“?

- How can we access private items of an ancestor?
- When they are "*private*" for the descendant…



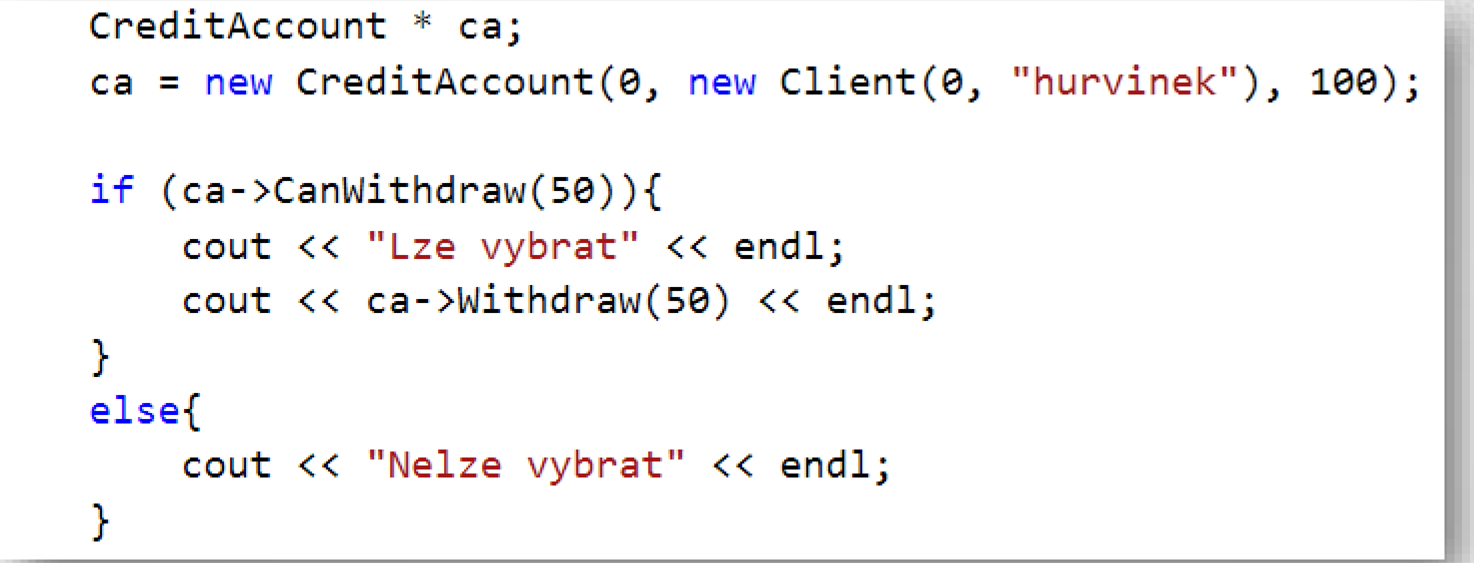
```cpp
class Account {
private:
    int number;
    float balance;
    float interestRate;
    Client * owner;

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

    int GetNumber();
    float GetBalance();
    float GetInterestRate();
    Client * GetOwner();

    void Deposit(float c);
    bool CanWithdraw(float c);
    float Withdraw(float c);
    void AddInterest();
};
```

```cpp
class CreditAccount : public Account{
private:
    float credit;

public:
    CreditAccount(int n, Client * o, float r);
    CreditAccount(int n, Client * o, float ir, float r);

    bool CanWithdraw(float c);
};
```

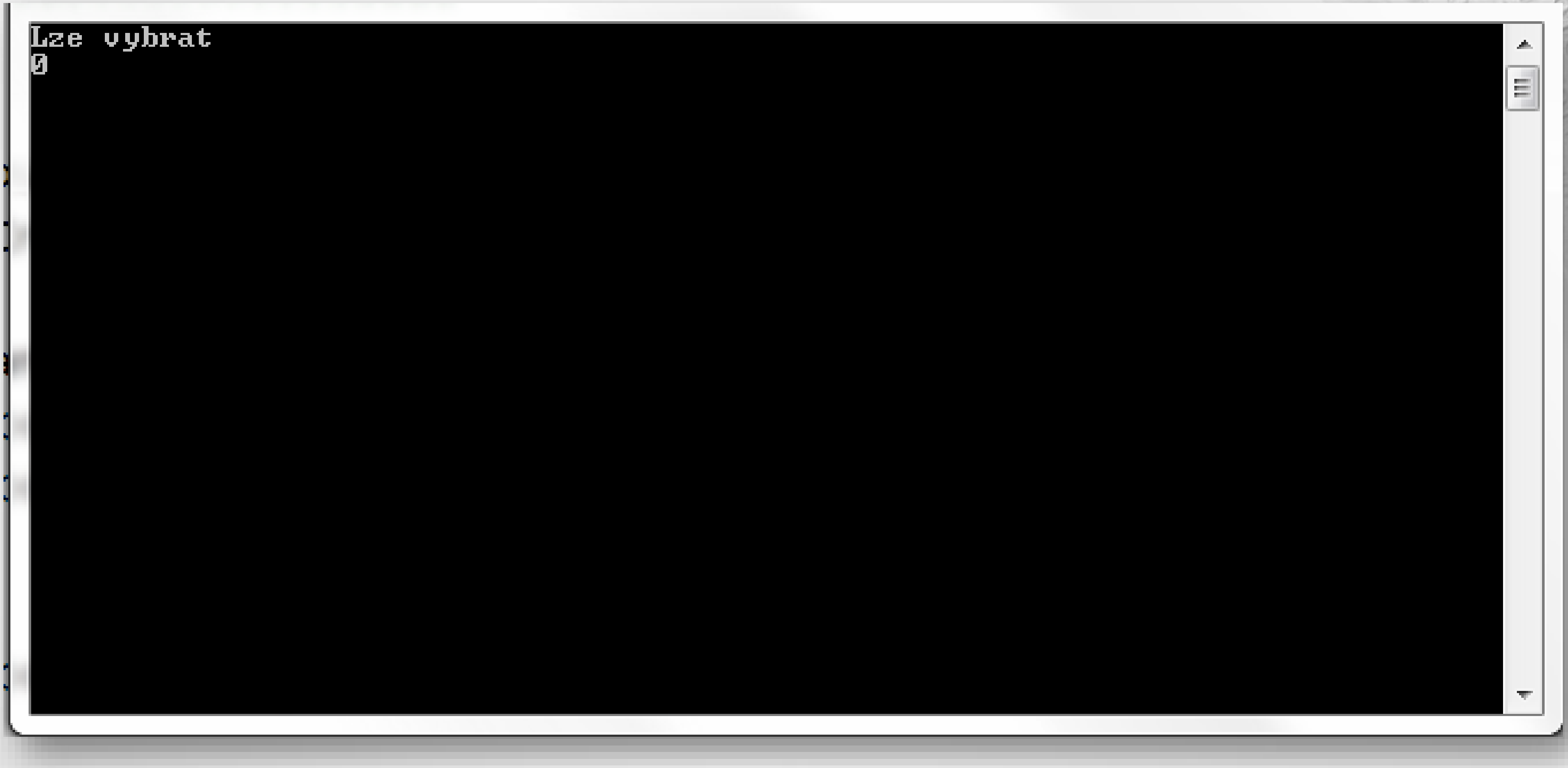
```cpp
float Account::Withdraw(float c){
    if (this->CanWithdraw(c)){
        this->balance -= c;
        return c;
    }
    return 0;
}
```

```cpp
bool Account::CanWithdraw(float c){
    return (c <= this->balance);
}

bool CreditAccount::CanWithdraw(float c){
    return (c <= (this->GetBalance() + this->credit));
}
```

Does it work?



```
    CreditAccount * ca;
    ca = new CreditAccount(0, new Client(0, "hurvinek"), 100);

    if (ca->CanWithdraw(50)){
        cout << "Lze vybrat" << endl;
        cout << ca->Withdraw(50) << endl;
    }
    else{
        cout << "Nelze vybrat" << endl;
    }
```



```
Lze vybrat
0
```

Early Binding

- Compiler normally uses so-called *early binding.* The type of instance is known at compile time.

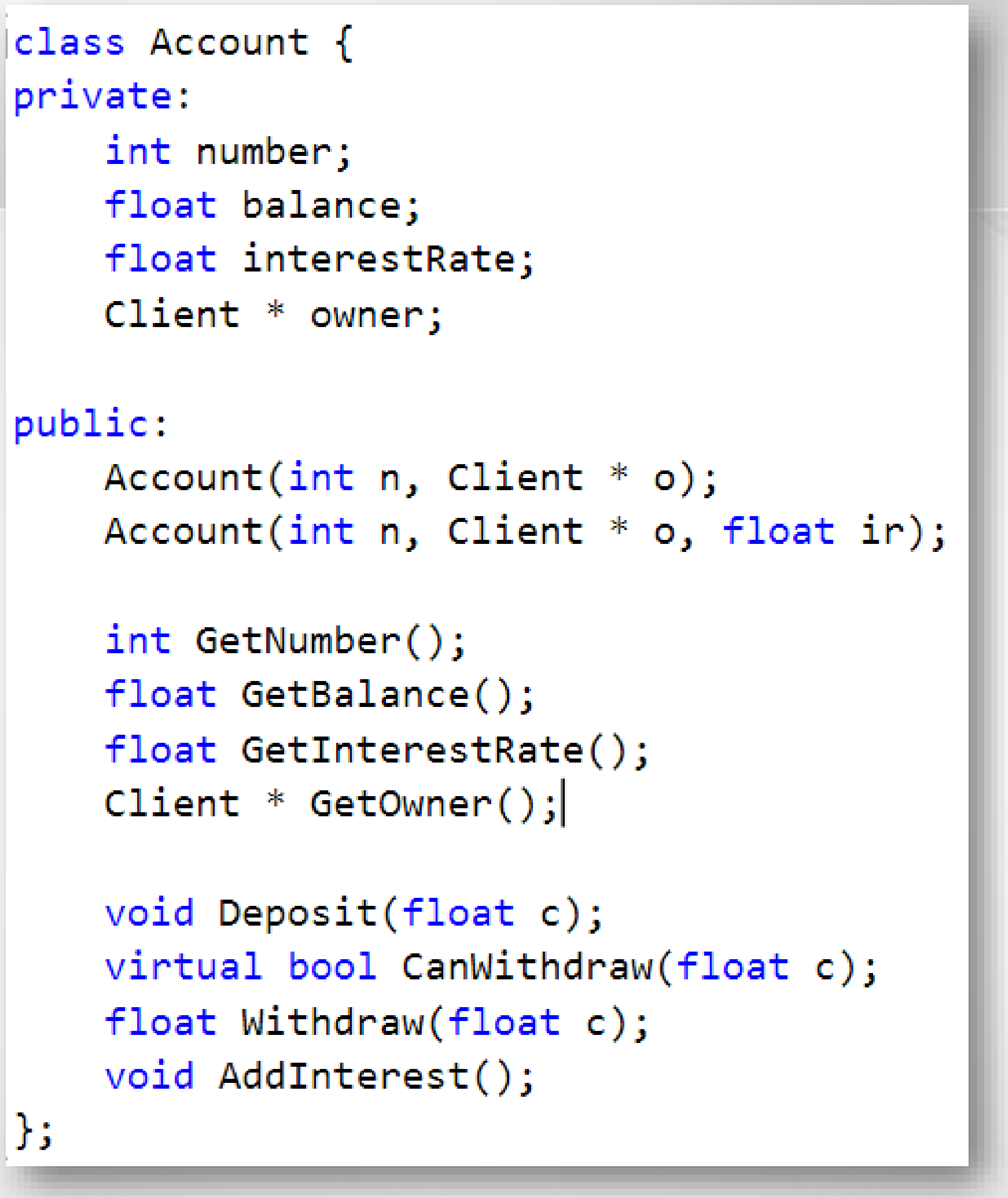
- The method *Withdraw* calls ancestor method *CanWithdraw*.

Late Binding

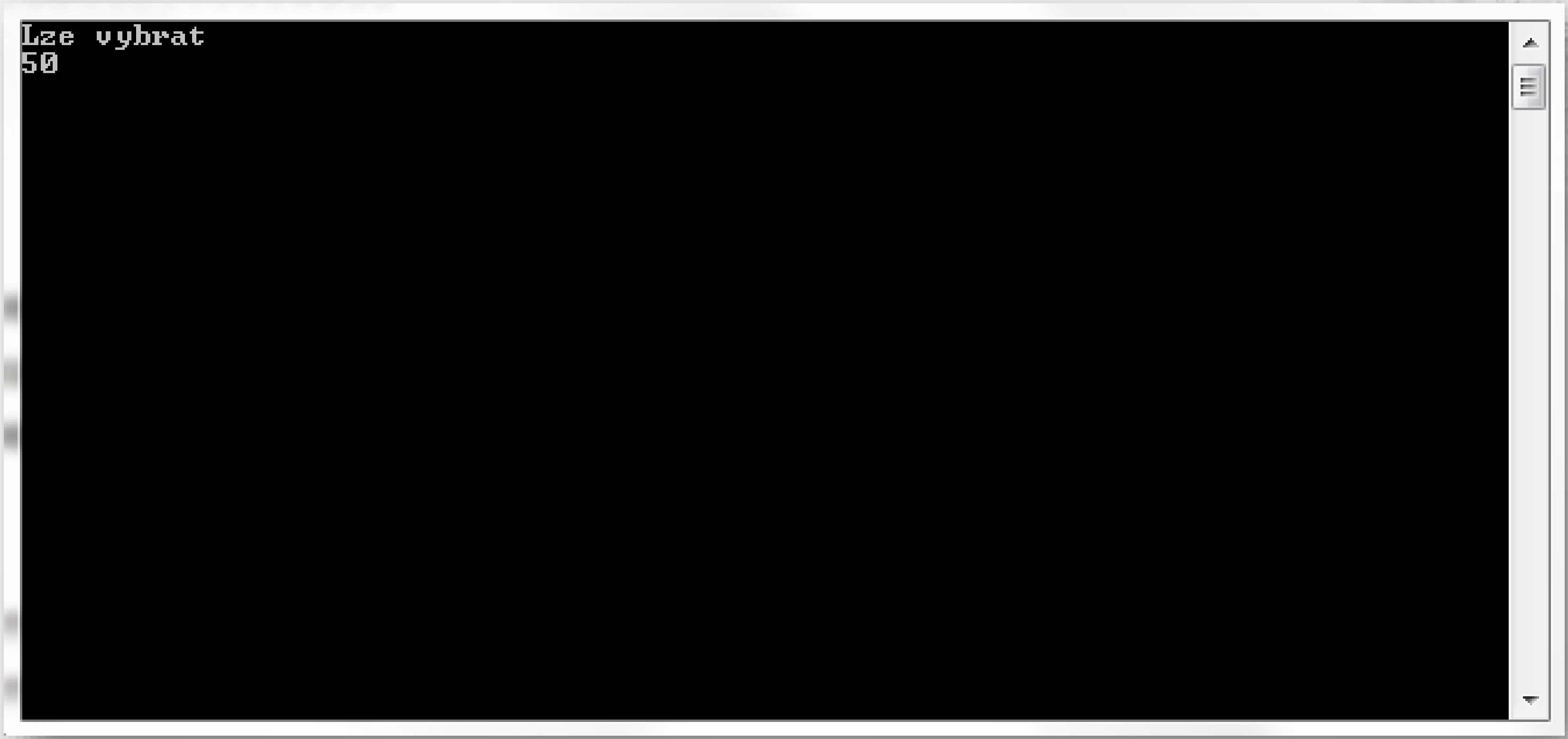
- We need to know who asks method, but at the moment of the call.

- In our case, it is not possible because the *early binding* is used.

- For this purpose, we have to use so-called *late binding*.



```
class Account {
private:
    int number;
    float balance;
    float interestRate;
    Client * owner;

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

    int GetNumber();
    float GetBalance();
    float GetInterestRate();
    Client * GetOwner();

    void Deposit(float c);
    virtual bool CanWithdraw(float c);
    float Withdraw(float c);
    void AddInterest();
};
```



```
Lze vybrat
50
```

Virtual Methods

- If we want to postpone our decisions which method will be called during the program, we need to use a virtual method.
- The compiler knows we wish to use dynamic or late binding.
- When a method is virtual, all descendants have the method also virtual.

Virtual Method Table

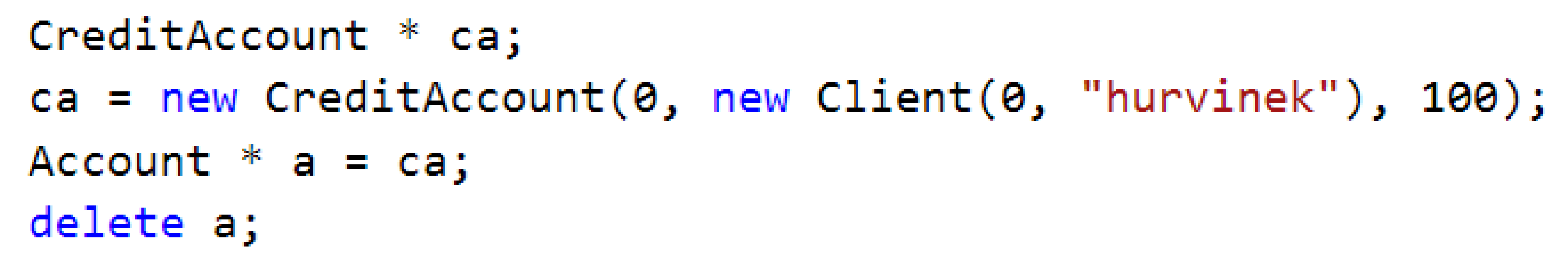
- If a method is defined as virtual, the compiler adds to the class a hidden pointer that points to a special table called virtual method table (VMT).
- For each class with at least one virtual method, the compiler creates a virtual method table.
- The table is common to all instances of the class.

Virtual Constructors?

- **NO!**
- Before their call, the pointer to the VMT is not created.
- We can call virtual methods inside the constructors. However, these virtual methods will be executed in a non-virtual mode.

Virtual Destructors?

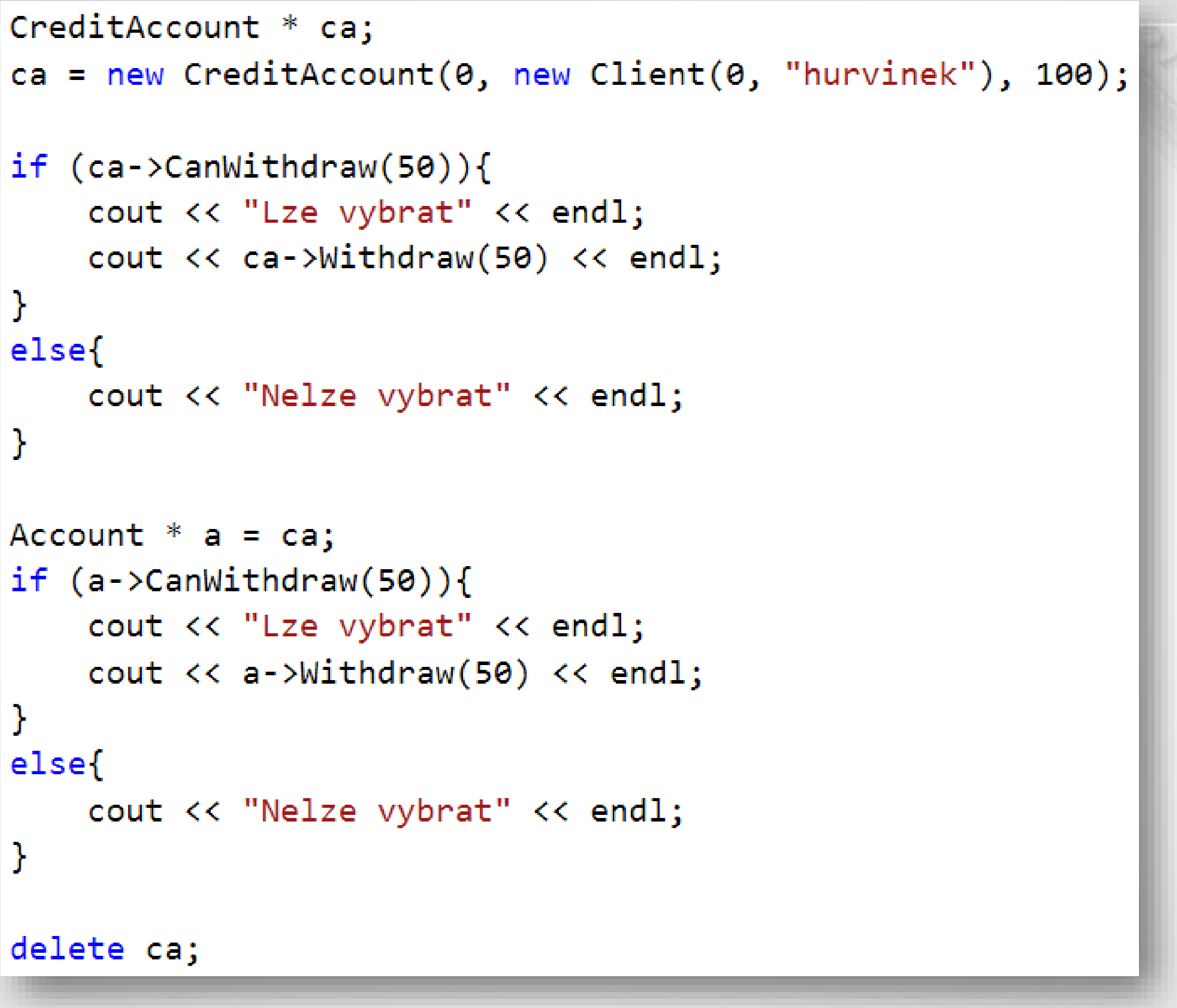
- **YES!**



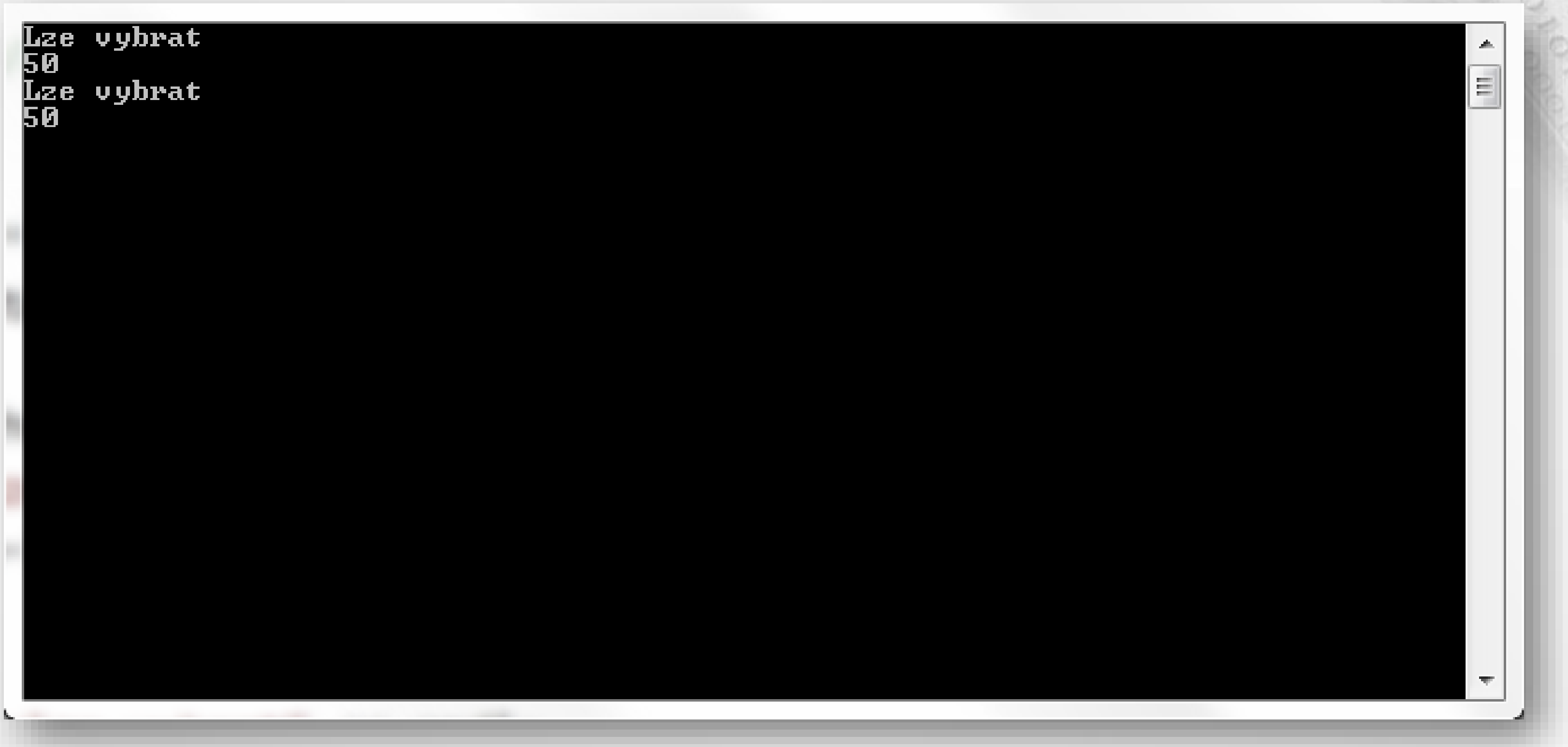
```
CreditAccount * ca;
ca = new CreditAccount(0, new Client(0, "hurvinek"), 100);
Account * a = ca;
delete a;
```

Example

How it works?



```
CreditAccount * ca;
ca = new CreditAccount(0, new Client(0, "hurvinek"), 100);

if (ca->CanWithdraw(50)){
    cout << "Lze vybrat" << endl;
    cout << ca->Withdraw(50) << endl;
}
else{
    cout << "Nelze vybrat" << endl;
}

Account * a = ca;
if (a->CanWithdraw(50)){
    cout << "Lze vybrat" << endl;
    cout << a->Withdraw(50) << endl;
}
else{
    cout << "Nelze vybrat" << endl;
}

delete ca;
```



```
Lze vybrat
50
Lze vybrat
50
```

Polymorphism

- Polymorphism is associated with inheritance.
- It makes no sense to talk about polymorphism unless we do not use virtual methods.
- It still about the substitutability ancestor-descendant.

Polymorphic data structures

- The structure that contains objects of different classes.
  - e.g. array, list, etc. which stores the ancestor type

- We can use (call) only common method of the ancestor.

- How to use (call) other methods?
  - We must cast (re-type) - it is one of the limitations of polymorphism.

Sources

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

Questions

- What do we mean by the polymorphism, and what does this relate?
- What do we mean by the polymorphic attachment (assignment)?
- What is early binding? Give examples.
- What is late binding? Give examples.
- Describe what the virtual method is and explain its properties.
- Describe what is the virtual method table and how it works.
- Can a constructor be virtual? Explain why?
- Can a destructor be virtual? Explain why?
- When should we use polymorphism in C++?
- What is a polymorphic data structure and when to use it?
- When do we need virtual destructor? When should we use it?