Programming II

Abstract Class

Multiple Inheritance

Lecture Outline

- Polymorphism - reminder
- Abstract class
- Multiple inheritance

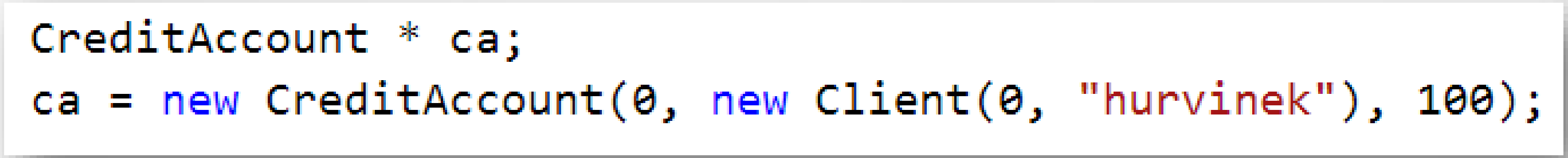
Polymorphism - reminder

Polymorphism

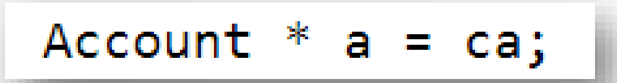
- *Polymorphism* is the ability of an object to play many roles (forms)…
- *Early* and *late* binding. When to use?
- It is related to the substitution principle (the substitutability an ancestor by a descendant).
- In C++, the polymorphism is related to inheritance!!!

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);
```

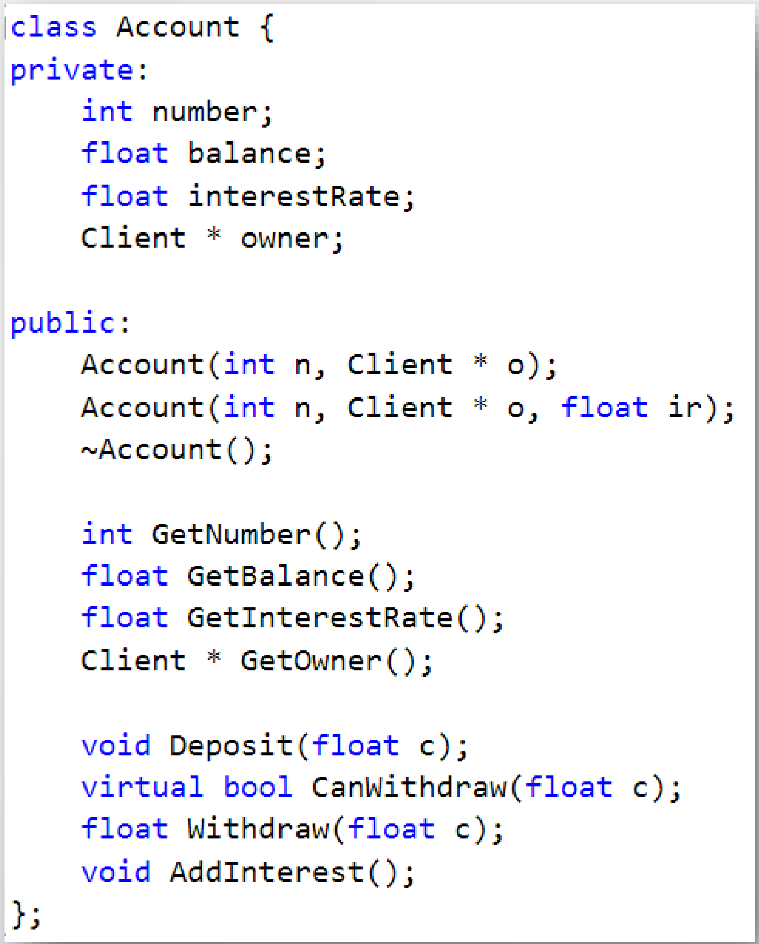


```
Account * a = ca;
```

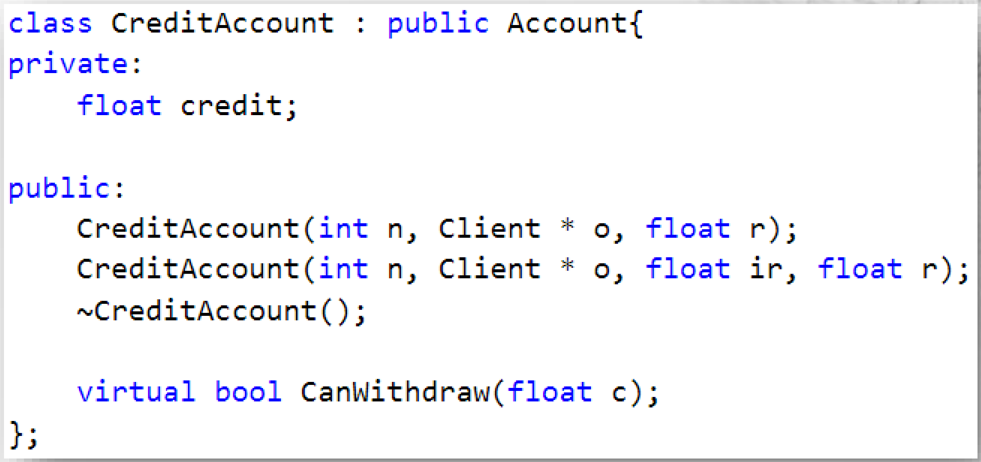
Virtual Method

- When a method is virtual, all descendants have the method also virtual.
- We can call *virtual* methods inside the constructors. However, these virtual methods will be executed in a *non-virtual* mode.
- We need to use a virtual destructor in a case of polymorphic attachment.

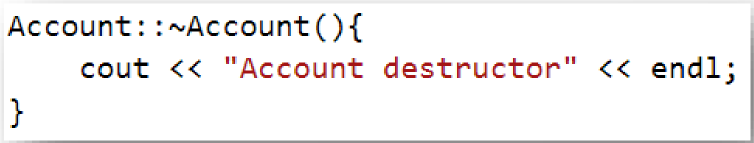
Virtual Destructor



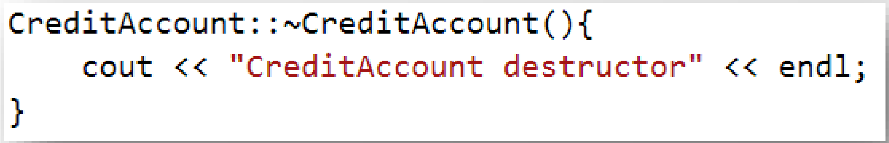
```
class Account {
private:
    int number;
    float balance;
    float interestRate;
    Client * owner;

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

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

    void Deposit(float c);
    virtual 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);
    ~CreditAccount();

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

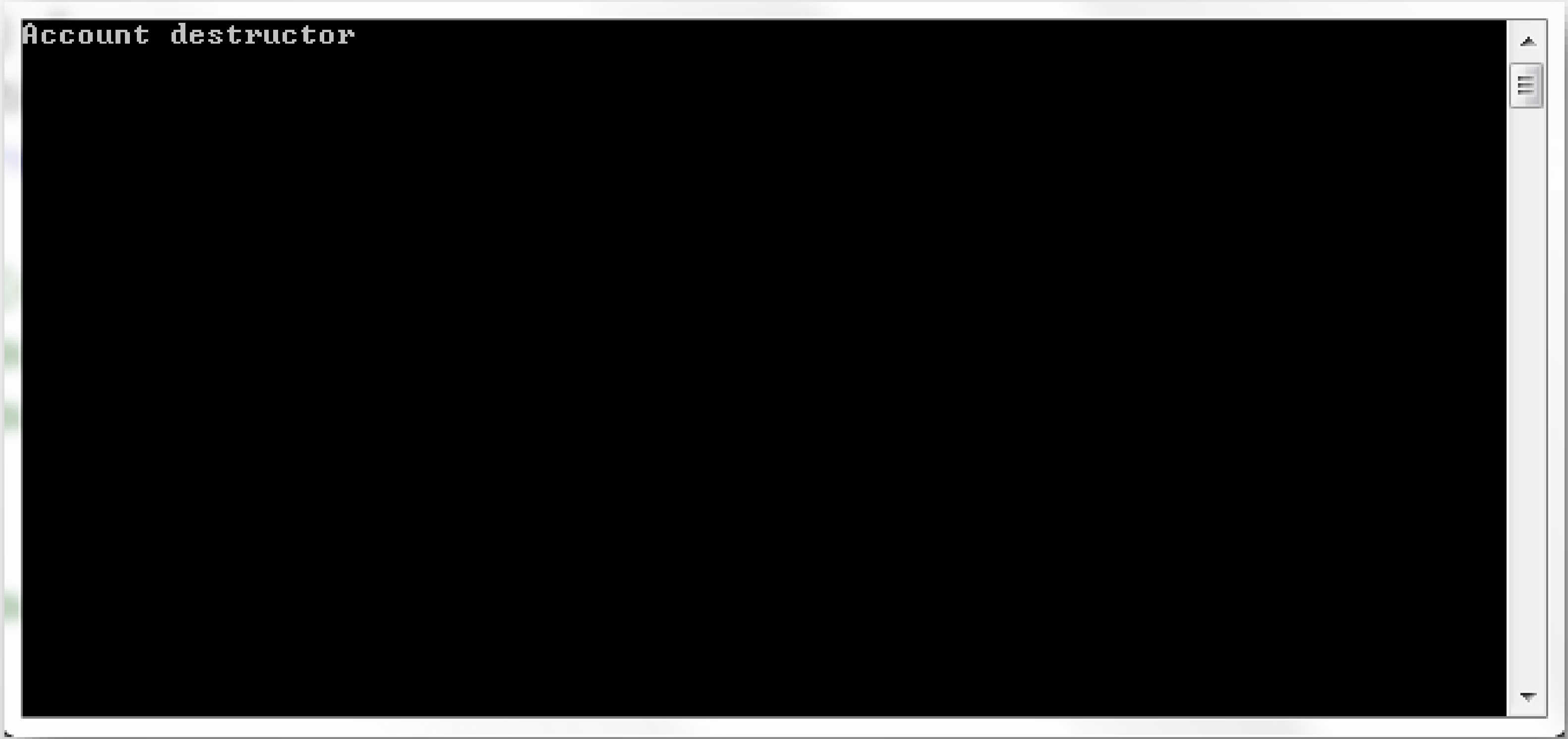


```
Account::~Account(){
    cout << "Account destructor" << endl;
}
```

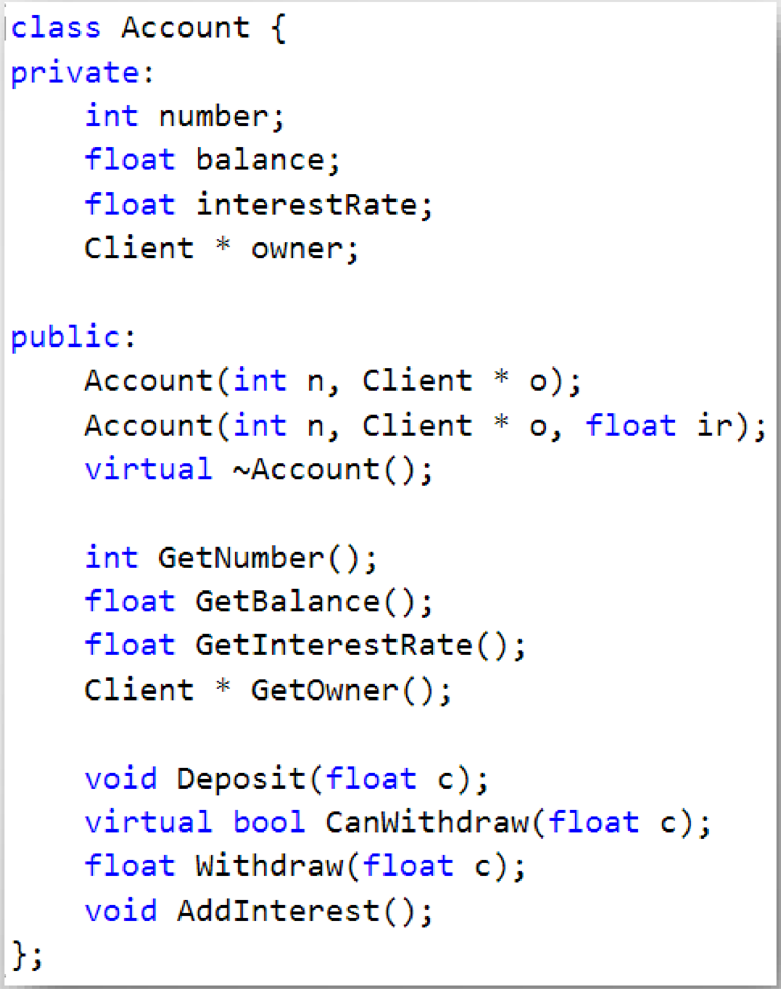


```
CreditAccount::~CreditAccount(){
    cout << "CreditAccount destructor" << endl;
}
```

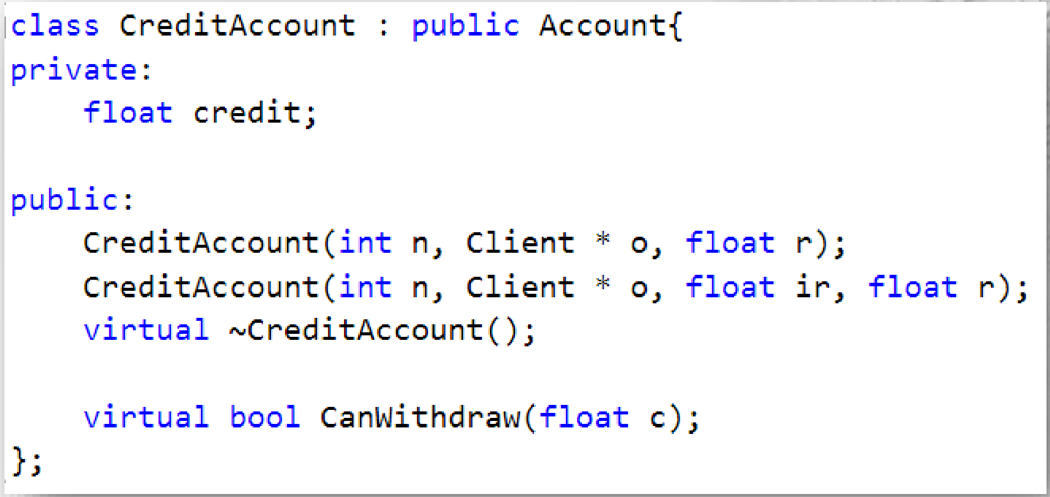
```cpp
CreditAccount * ca;
ca = new CreditAccount(0, new Client(0, "hurvinek"), 100);

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



```
Account destructor
```

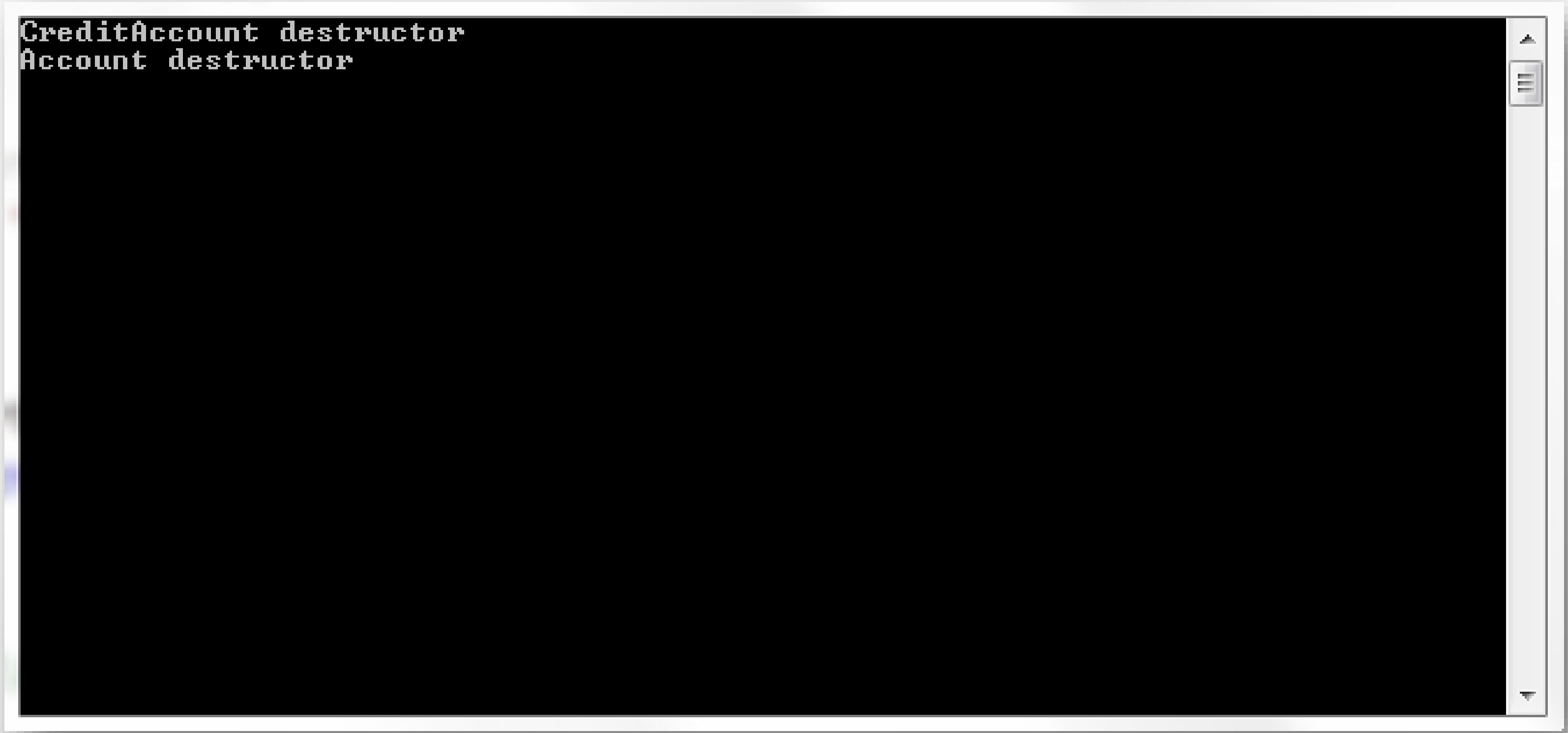


```cpp
class Account {
private:
    int number;
    float balance;
    float interestRate;
    Client * owner;

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

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

    void Deposit(float c);
    virtual 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);
    virtual ~CreditAccount();

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

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

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



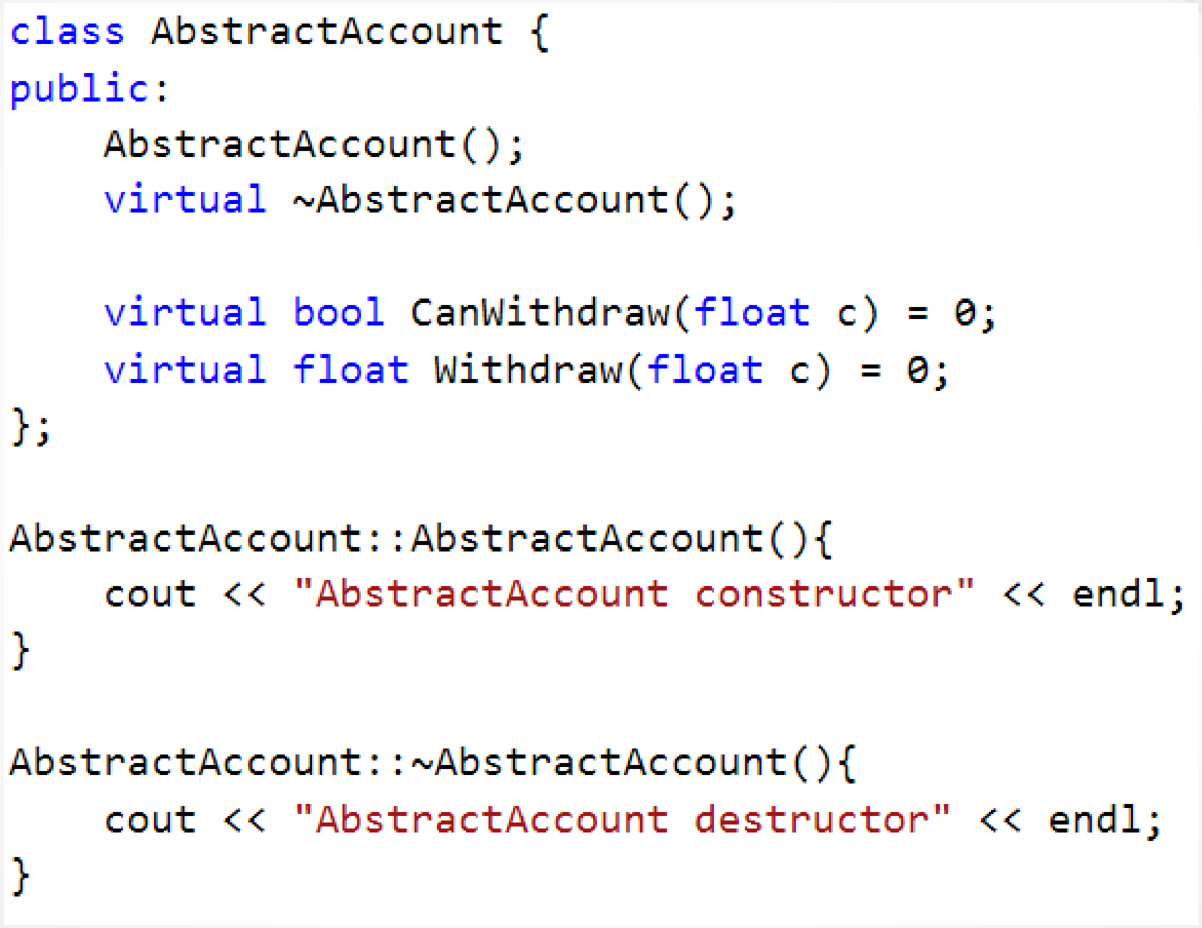
```
CreditAccount destructor
Account destructor
```

Abstract class

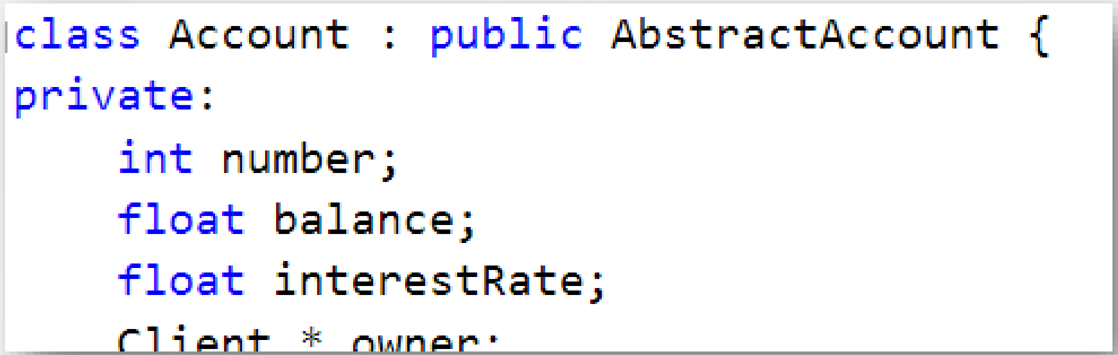
Pure Virtual Method

- The method which has only a declaration.
- The method has no implementation (definition).
- Why and when to use it?
  - We require a good design of our programs.

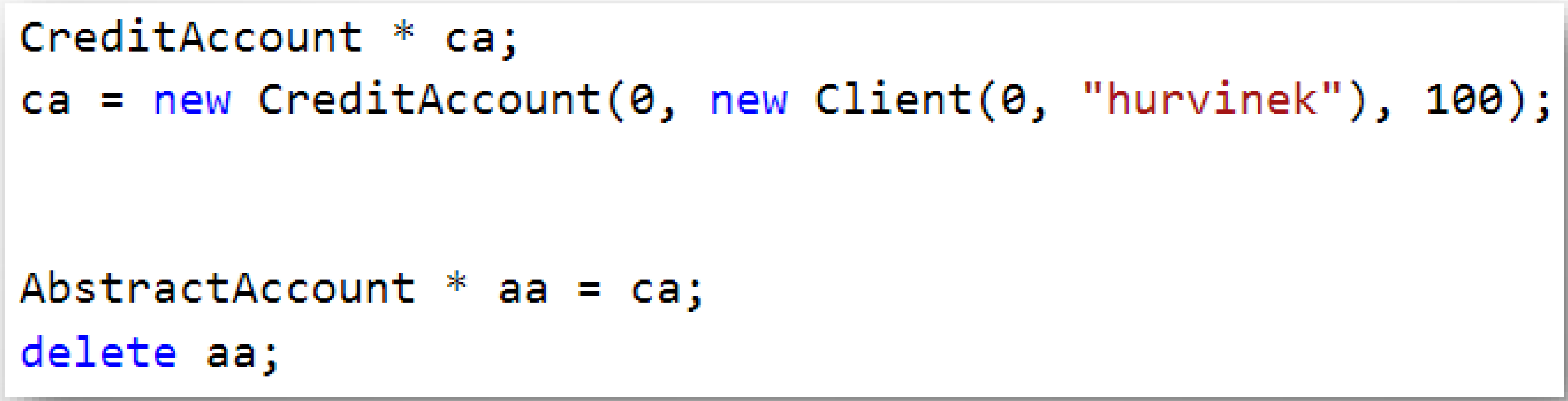
Virtual methods



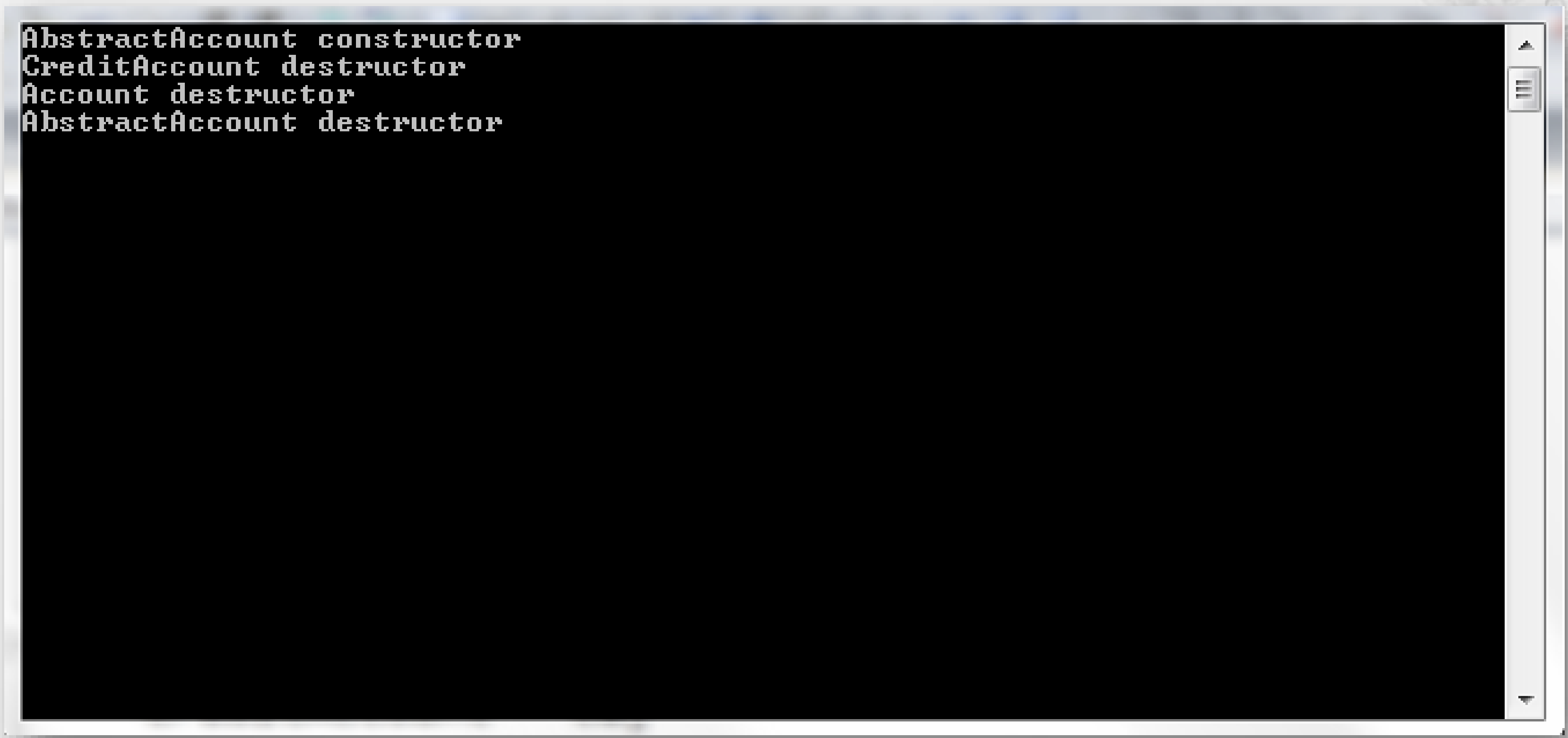
```cpp
class AbstractAccount {
public:
    AbstractAccount();
    virtual ~AbstractAccount();

    virtual bool CanWithdraw(float c) = 0;
    virtual float Withdraw(float c) = 0;
};

AbstractAccount::AbstractAccount(){
    cout << "AbstractAccount constructor" << endl;
}

AbstractAccount::~AbstractAccount(){
    cout << "AbstractAccount destructor" << endl;
}
```



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

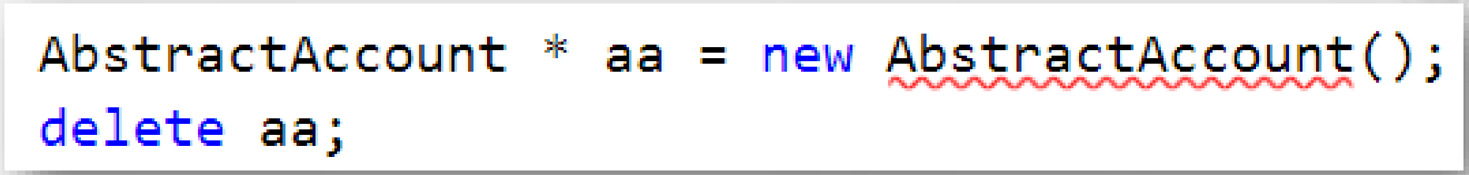


```
CreditAccount * ca;
ca = new CreditAccount(0, new Client(0, "hurvinek"), 100);



AbstractAccount * aa = ca;
delete aa;
```



```
AbstractAccount constructor
CreditAccount destructor
Account destructor
AbstractAccount destructor
```

But…



```
AbstractAccount * aa = new AbstractAccount();
delete aa;
```

- Why???

Abstract Class

- Class with at least one pure virtual method is called *abstract class*.
- Abstract, because we cannot create an instance.
- May, but need not, have a member variable and methods implemented.
- It has a constructor and destructor; for its descendants.

Pure Abstract Class

- Class with only pure virtual methods.
- Why we need such a class?
  - As an "empty" pattern of descendants.
- It declares but does not define the future common behavior of the descendants.

Accounts Inheritance

AbstractAccount – abstraktní třída

Account

PartnerAccount ***CreditAccount***

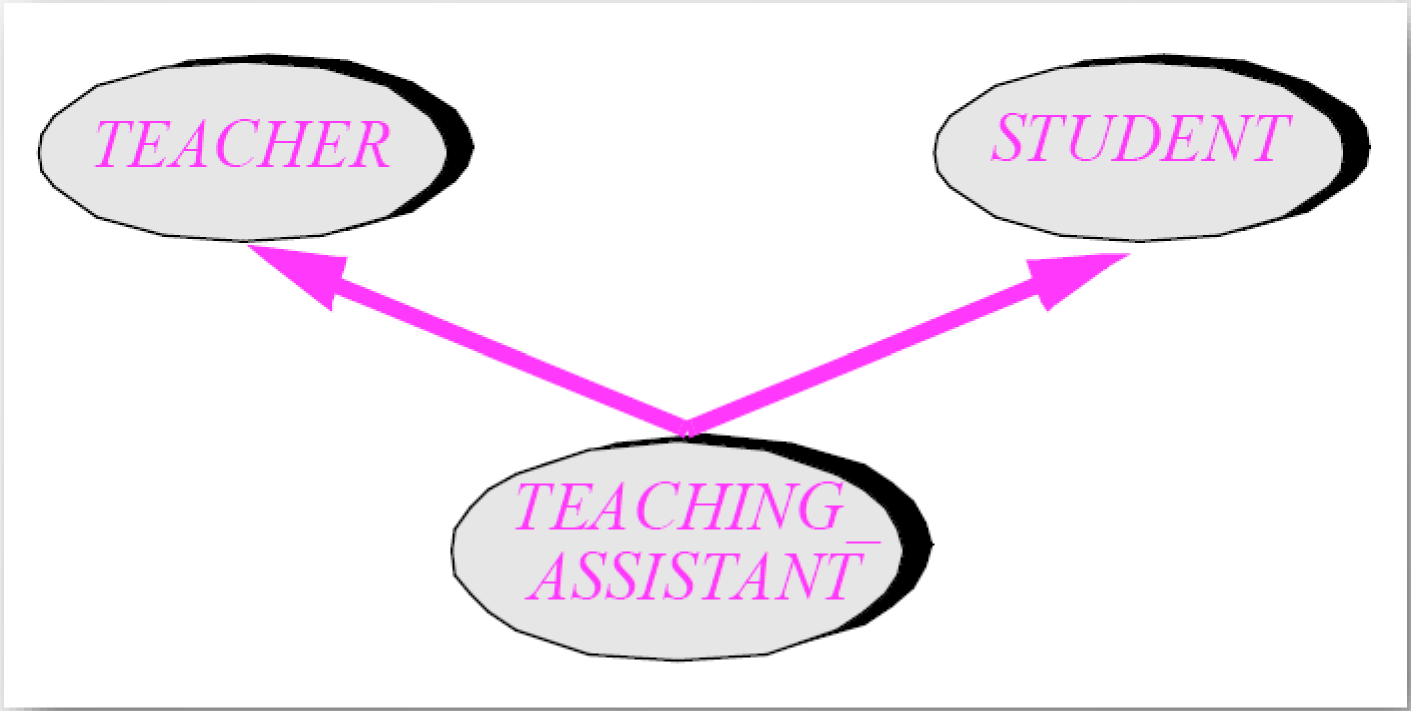
- Who should implement (define) a pure virtual method? **A descendant!!!**

Multiple inheritance

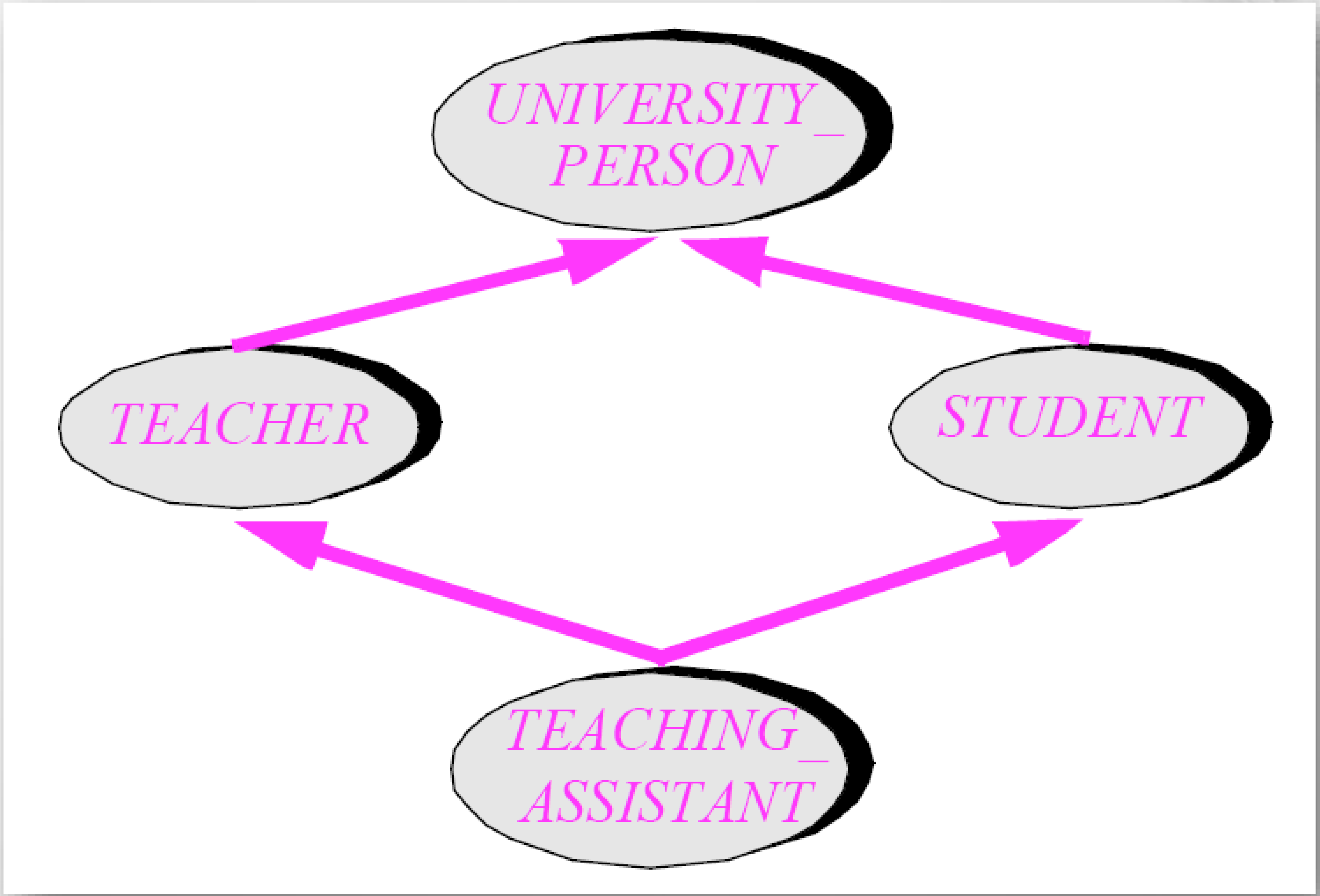
Multiple Inheritance

- Can the child inherit from multiple classes?
- Why not?
- Why yes?
- It is a nice concept, but rather a dangerous one and often difficult to understand...

Bad Example?



Why it is bad?



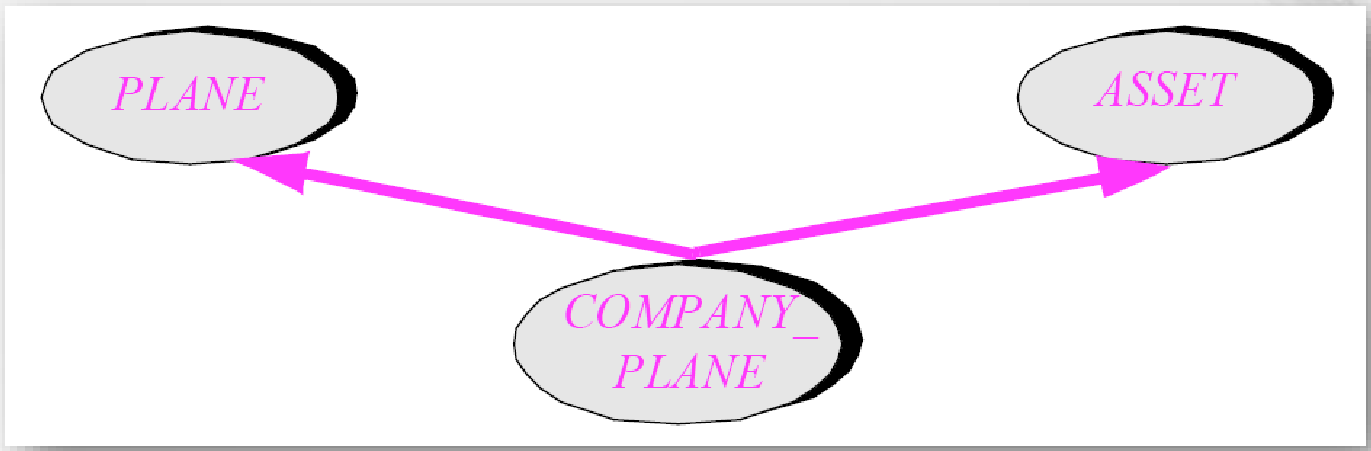
It is not for beginners…

- …and advanced developers may lack it.
- The problem is that *Teacher* and *Student* are not different abstractions.
- They share common features of *University_Person*.
- There are also technical problems…

Does it make sense?

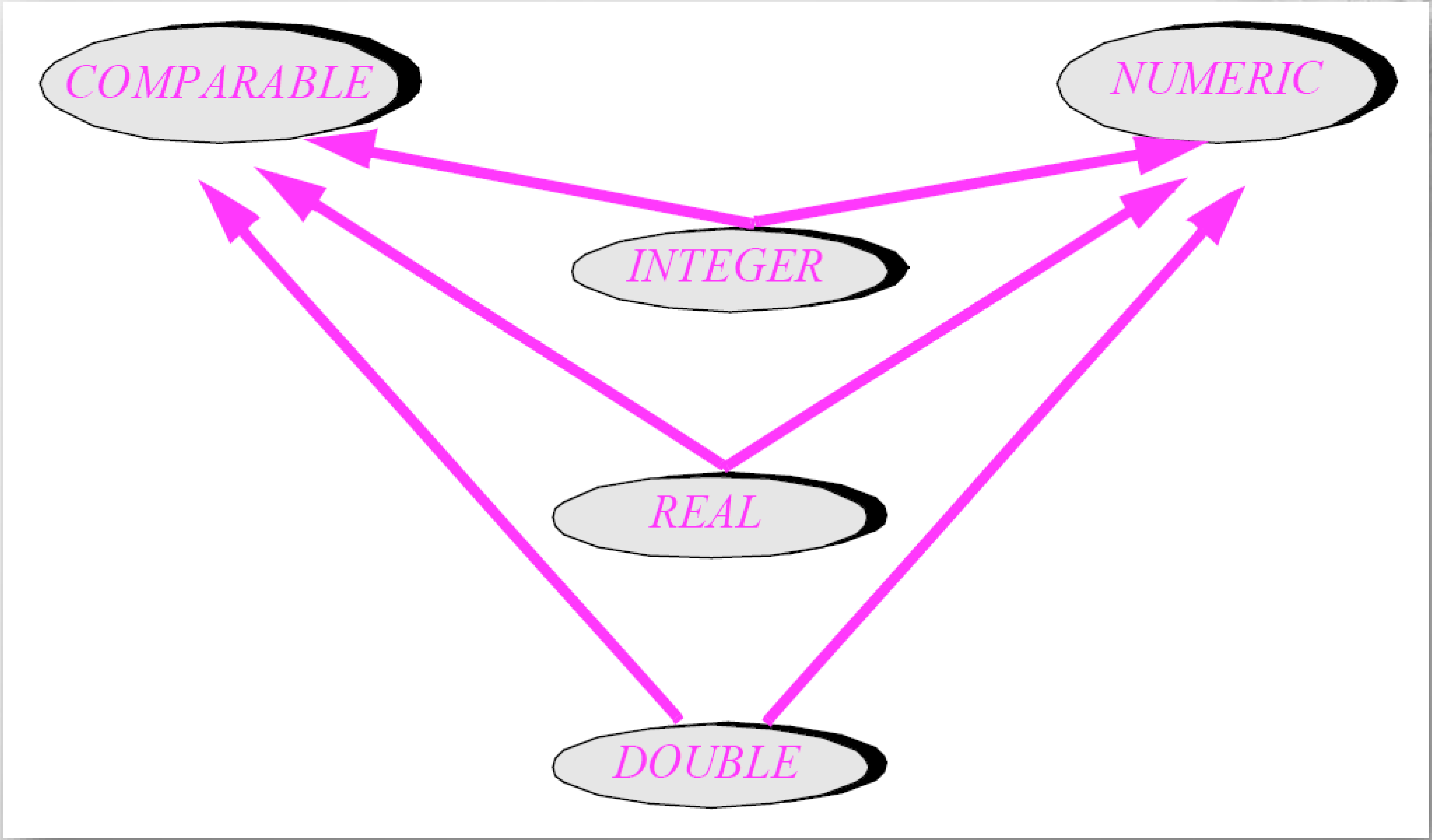
- The ancestors must be different abstractions.
- Different abstractions can be seen as not having a common state or behavior.
- Then it makes sense to consider multiple inheritance.

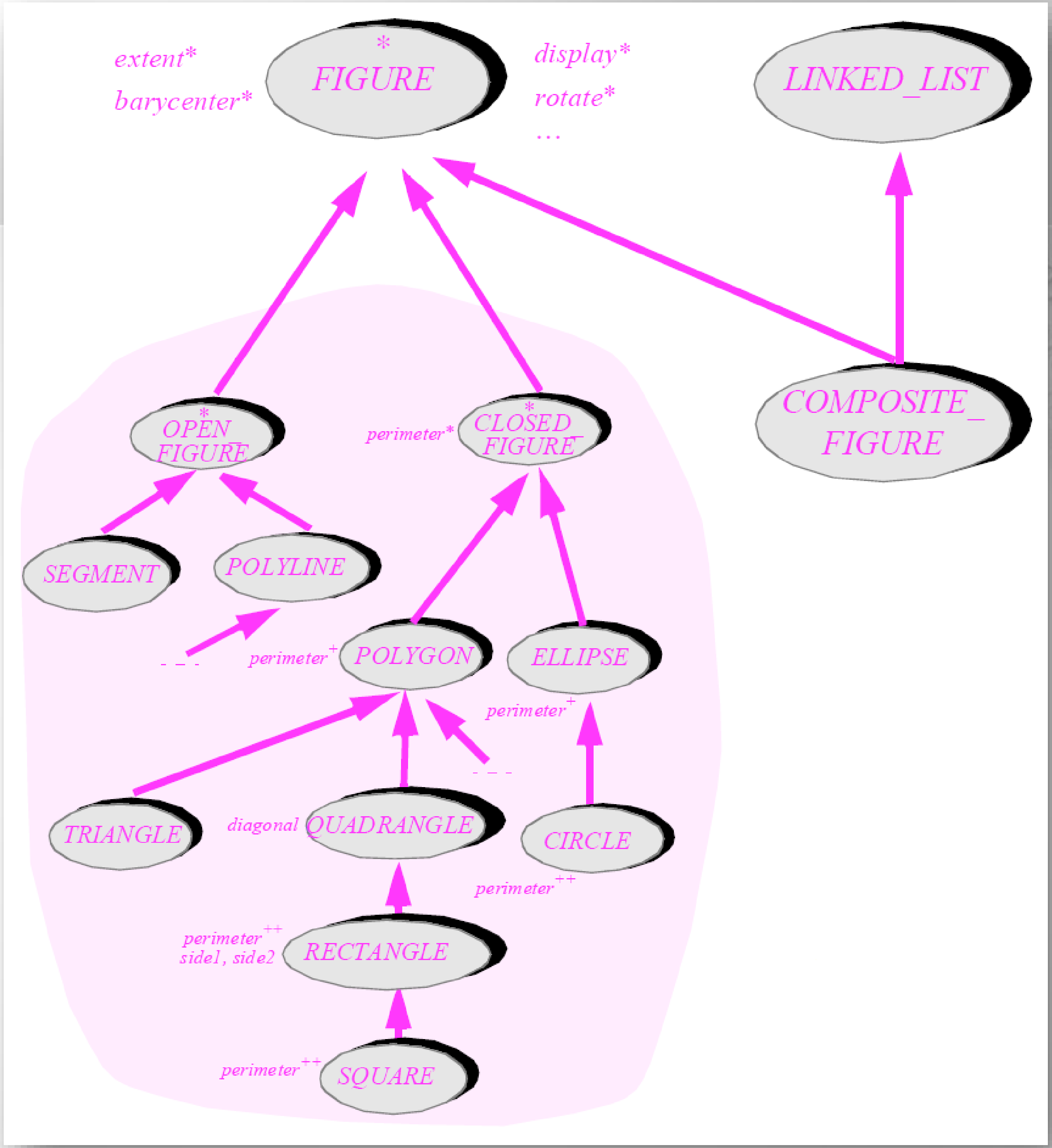
Good example



- A company plane is:
  - A plane with its technical data and functions associated with them.
  - Property with registration data and corresponding functionality.

Next example…




extent*
barycenter*
*
FIGURE
display*
rotate*
...
LINKED_LIST
*
OPEN_FIGURE
perimeter*
*
CLOSED_FIGURE
COMPOSITE_FIGURE
SEGMENT
POLYLINE
- - -
perimeter+
POLYGON
ELLIPSE
perimeter+
- - -
TRIANGLE
diagonal
QUADRANGLE
CIRCLE
perimeter++
perimeter++
side1, side2
RECTANGLE
perimeter++
SQUARE

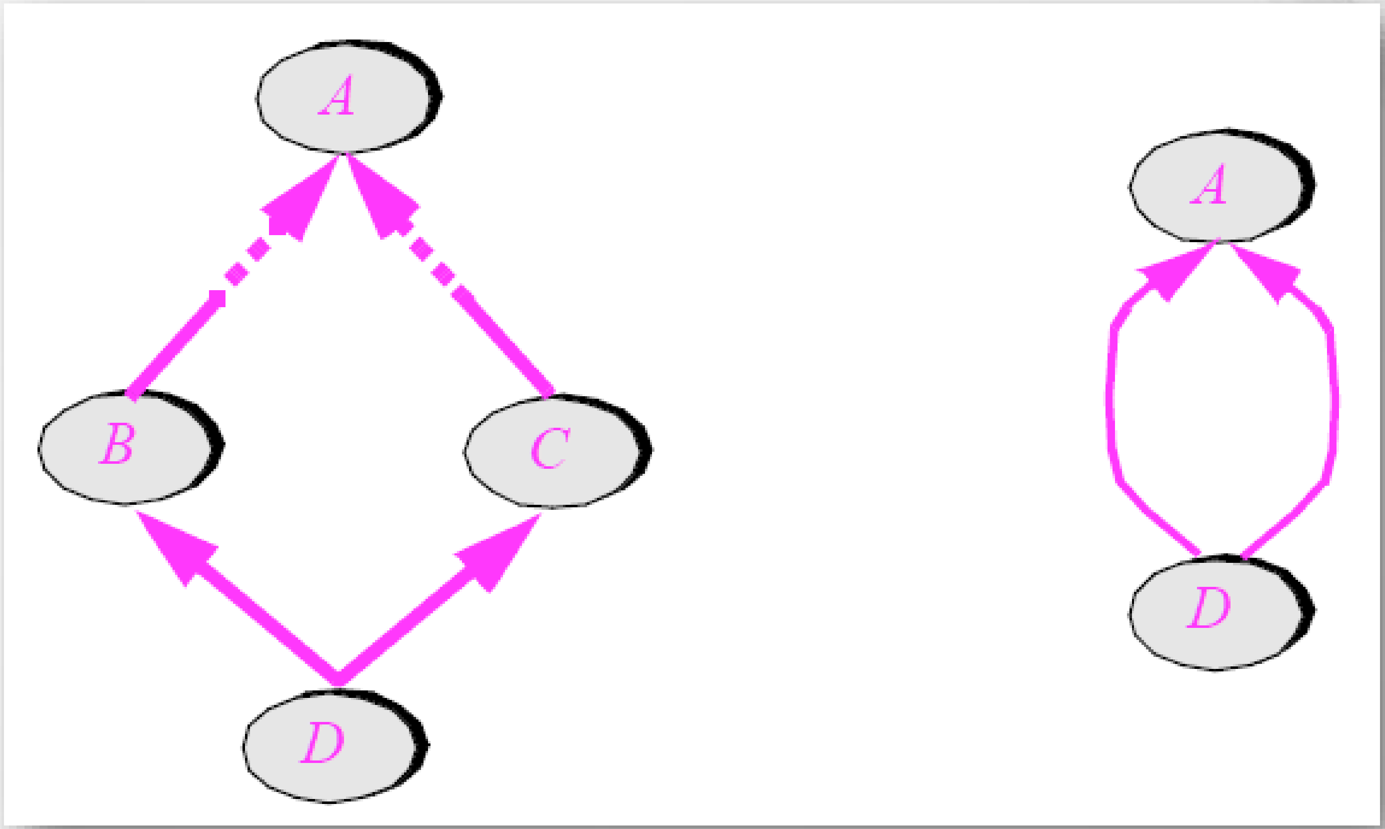
Do we need it?

- Yes, but…
- Sometimes we need a class with properties beyond the basic abstraction that is described by one class.
- It is again about the descendant-ancestor substitutability.
- In this case, however, the descendant play a role its behavior differs significantly.

Problems

- Conflicting features (name)
  - Base classes may have members (variables and methods) with the same names.
  - It can be resolved in various ways.
- Repeated inheritance (sharing ancestors)
  - Is it possible to recognize the multiple inheritance?
  - Here is a little worse...

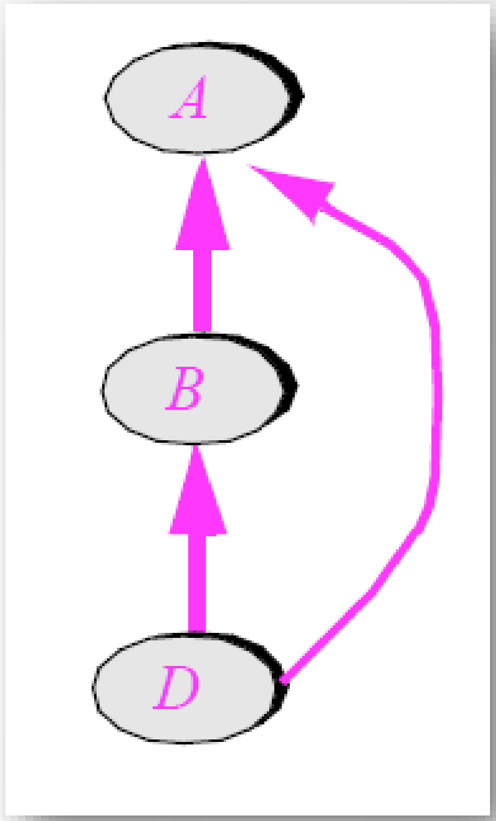
Repeated Inheritance



Indirect

Direct

Why on earth?



Redundant inheritance

Why do we talking about?

- Multiple inheritance is a useful concept, especially because the object can represent two different abstractions.
- However, it must be used wisely.
- When?

Usage of Multiple Inheritance

- We need objects which represent different abstractions in different situations.
  - Differences should prevent conflicts of features (names).
  - It must be related to ancestor-descendant substitutability.

- If possible, the ancestors should be pure abstract classes (with no data).
  - Then it is the same like "interface“ in modern object-oriented languages.
  - The interface is a concept that replaces the multiple inheritance.

Sources

- Bertrand Meyer. *Object-Oriented Software Construction.* Prentice Hall 1997. [486-490, 519-529]

Questions

- What is a pure virtual method?
- When is it appropriate to use pure virtual method? Give an example.
- What is an abstract class?
- When is it appropriate to use an abstract class? Give an example.
- Does the abstract class need constructor and destructor? And why?
- May have an abstract class member data and functions (methods)?
- What is a pure abstract class?
- What is multiple inheritance?
- When it is not appropriate to use multiple inheritance? Give an example.
- When it is possible to use multiple inheritance? Give an example.
- What problems can arise when using multiple inheritance? Give an example.
- What is repeated inheritance? Give examples.
- Why do we need multiple inheritance?