Script for the vhb lecture

Programming in C++
Part 1

Prof. Dr. Herbert Fischer
Deggendorf Institute of Technology

Table of Contents

9 Inheritance

Chapter 9 teaches you the concept of class inheritance.

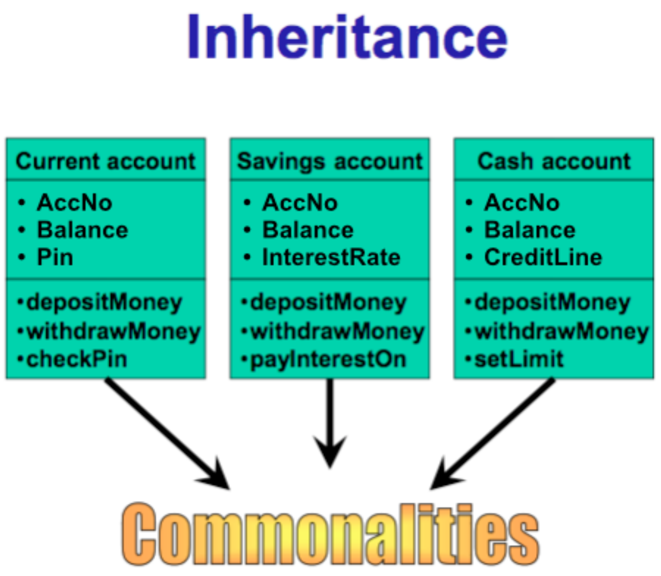
We already know the concept of inheritance in object-oriented software development. With the use of inheritance, we can reuse existing program parts by passing the elements of a class on to other classes. This means that you “don´t have to reinvent the wheel” with each class you create. If we recognize generally valid concepts in our objects, we should define them only once in so-called base classes.

In this chapter we will get to know the C++ techniques of inheritance and use them as examples. We will answer the following question: How do you pass the general properties of an account on to specialised accounts?

9.1 Motivation

For reasons of an economically efficient program development and higher software quality, the software industry demands the following of its programs or program parts: **Reuse, customization and extension options.**

Therefore, OO software development enables us to build so-called **class hierarchies**. This hierarchy is generated by creating new classes; to this end, existing, more general class declarations are used on the basis of the concepts of **inheritance** or **derivation**.

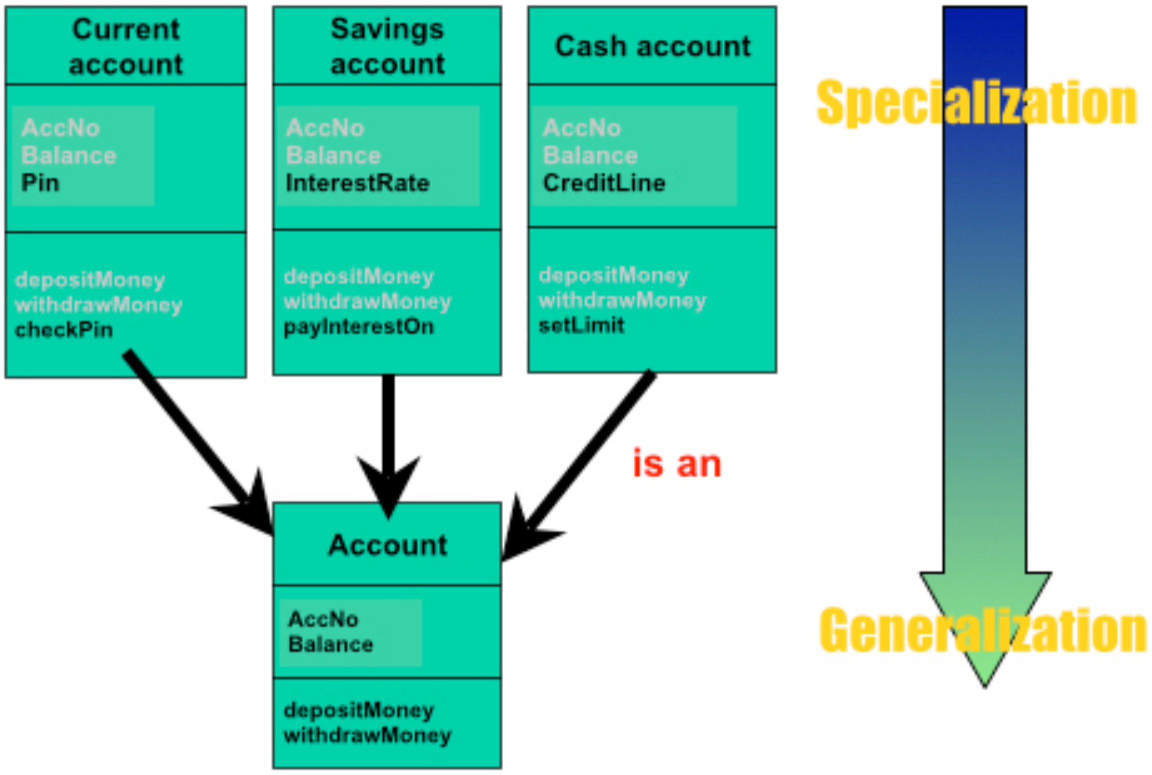


An example in which inheritance can be used advantageously are the various types of bank accounts. There are current accounts, savings accounts, share portfolios, and cash accounts - to name only a few. All these different types of bank accounts have something in common.

We model these commonalities in a general account class and refer to this class as the **base class**. Using the concept of inheritance, we want to reuse these general account properties in more specific account classes. This saves a lot of work because basic data members and member functions that apply to all accounts only need to be developed and tested once. This approach also prevents errors that can occur when developing the same functionality several times.

Arrows in the class diagram represent the relationship between the **base class** and the **derived classes.** The arrowhead points in the direction of **generalization**. In the other direction, we therefore have a **specialization**. In the special accounts we then only have to realize "the specialities" of the respective accounts. For example, for the current account: a pin number and a "check" member function that compares the number entered with the actual pin number of the account. Or for the savings account: a retention period, …

The concept of inheritance ensures that the data members and member functions of the base class are also available in the derived classes - we also say data members and member functions **get inherited**. The relationship between a derived class and a base class can be read as "is a/is an". So, for example, current account "is an" account; just like cash account "is an" account or savings account "is an" account.



When should a class be built using inheritance?

- A class is a specialization or enhancement of an existing class (current account).
- A class combines the properties of several existing classes (multiple inheritance).
- Two classes developed in parallel contain a common core that can be used as base class.

Advantages:

- The derived (new) class (subclass, child class) inherits (almost) all properties
  - data members
  - member functions
  from exisiting classes (base classes, superclasses, parent classes) without having to touch or compile them again.

- These properties can be added or changed in the derived class:
  - add further data members or member functions
  - modify member functions
  Different classes in a hierarchy can have a *common, uniform interface*, which simplifies the use of objects in this class.

Constraints:

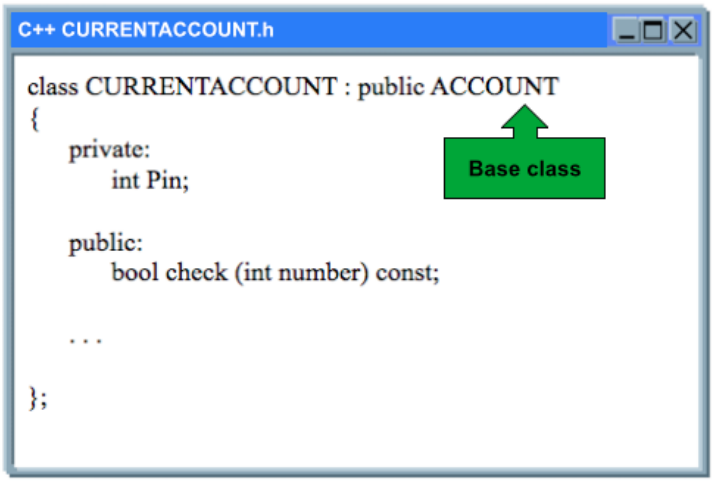
- Not inherited are
  - Friendship with another *friend* class
  - A class-specific assignment operator operator=()
  - Constructors and destructors

9.2 Declaration und access rights

The general C++ syntax to derive a class is:

```
class DerivedClass : [private|public|protected] BaseClass, ...
{
        ...
        // Declaration of the DerivedClass
        ...
};
```

Declaration: Current Account

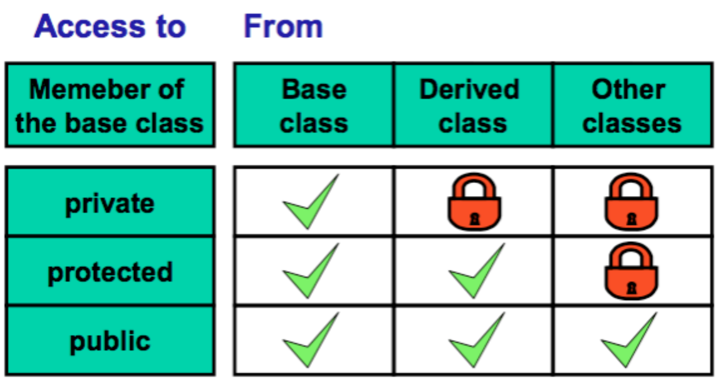


```
class CURRENTACCOUNT : public ACCOUNT
{
    private:
        int Pin;

    public:
        bool check (int number) const;

    . . .
};
```

“public“ derivation



Access to	From		
Memeber of the base class	Base class	Derived class	Other classes
private	✓	🔒	🔒
protected	✓	✓	🔒
public	✓	✓	✓

The name of the derived class is separated from the name of the base class(es) by a colon (:).

Access rights after derivation::

Basically, DerivedClass inherits all components of BaseClass, i.e. each instance of DerivedClass contains an (anonymous) object of the BaseClass type.

This sub-object is created before the additional components of DerivedClass by implicitly calling the base class constructor.

The access right in the base class and the access specifier during derivation determine the access right to components of BaseClass in DerivedClass.

private components of BaseClass are generally not accessible in DerivedClass. Access to private members is still only possible using public member functions of BaseClass. By deriving from a class, you cannot gain access to private elements.

public inheritance: class DerivedClass : public BaseClass

protected members of BaseClass are also accessible in DerivedClass, but not generally accessible.

public members of BaseClass are also public members in DerivedClass.

- **protected inheritance:** class DerivedClass : protected BaseClass

public members of BaseClass become protected members of DerivedClass. They can no longer be accessed from “outside” the class.

- **private inheritance:** class DerivedClass: [private] BaseClass

All public and protected members of BaseClass are private members of DerivedClass. This type of inheritance is used whenever a new interface is to be created in DerivedClass, i.e. the implementation is inherited (models not "*is a*" but "***is implemented by***").

Access specifier in the base class	Access specifier during derivation		
	private	protected	public
private	No access	No access	No access
protected	private	protected	protected
public	private	protected	public
	Modified access right to members of the base class in the derived class		

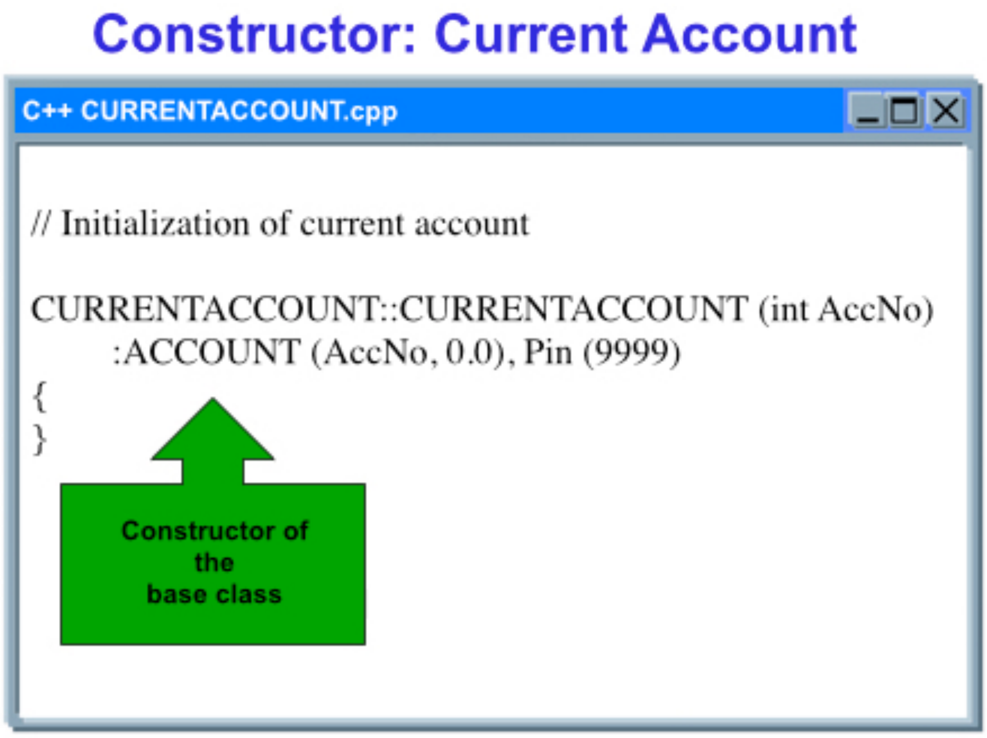
Note: It is always the more restrictive right of access that applies!

Hint: The access rights of *individual members* of BaseClass can be influenced systematically in the definition of DerivedClass:

```
class DerivedClass : <mod> BaseClass
{
 ...
 [private|public|protected]:
   BaseClass::MemberName;           // old spelling
   using BaseClass::MemberName;     // new spelling
 ...
};
```

9.3 Initialization

Creating an object of a derived class - for example a current account - raises the question how the initialization of the inherited data members of the Account class works. There are precise rules for the use of constructors of derived classes.



Constructor: Current Account

```
// Initialization of current account

CURRENTACCOUNT::CURRENTACCOUNT (int AccNo)
      :ACCOUNT (AccNo, 0.0), Pin (9999)
{
}
```

In a test we want to make the call of the constructors and destructors visible during the program run by built-in outputs:

```
#include <iostream>
#include <cstdlib>

using namespace std;

class ACCOUNT
{
  private:
    int AccNo;
    //...

  public:
    // Default constructor
    ACCOUNT() : AccNo(999)
    { cout << "1: ACCOUNT(),AccNo: " << AccNo << endl;  }

    // Constructor with parameters
    ACCOUN (int aNo) : AccNo (aNo)
    { cout << "2: ACCOUNT(int), AccNo: " << AccNo << endl; }

    // Destructor
    ~ACCOUNT()
    { cout << "3: ~ACCOUNT(),AccNo: " << AccNo << endl; }
};
```

```
class CURRENTACCOUNT : public ACCOUNT
{
 private:
   float Overdraft;

 public:
   CURRENTACCOUNT() : Overdraft(2000)
   { cout << "4: - CURRENTACCOUNT(), Overdraft.: " << Overdraft << endl;  }

   CURRENTACCOUNT(float o) : Overdraft()
   { cout << "5: - CURRENTACCOUNT (float), Overdraft.: " << Overdraft << endl;  }

   CURRENTACCOUNT(int aNo, float o) : ACCOUNT(aNo), Overdraft(o)
   { cout << "6: - CURRENTACCOUNT(int,float), Overdraft.: " << Overdraft << endl;  }

   ~ CURRENTACCOUNT()
   { cout << "7: - ~ CURRENTACCOUNT(), Overdraft.: " << Overdraft << endl;  }
};

int main()
{
 CURRENTACCOUNT a, b(5500.0), c(123,1000.0);
 cout << "Press any key to continue ........." << endl;
 system("pause");
}
```

Output:

```
1: ACCOUNT(),AccNo: 999
4: - CURRENTACCOUNT(), Overdraft.: 2000
1: ACCOUNT(),AccNo: 999
5: - CURRENTACCOUNT(float), Overdraft.: 5500
2: ACCOUNT(int), AccNo: 123
6: - CURRENTACCOUNT(int,float), Overdraft.: 1000
Press any key to continue .........
7: - ~ CURRENTACCOUNT (), Overdraft.: 1000
3: ~ ACCOUNT(),AccNo: 123
7: - ~ CURRENTACCOUNT (), Overdraft.: 5500
3: ~ ACCOUNT(),AccNo: 999
7: - ~ CURRENTACCOUNT (), Overdraft.: 2000
3: ~ ACCOUNT(),AccNo: 999
```

Conclusions:

- Members of base classes can only be initialized using the constructors of the base classes.

- The constructor is called either:
  - *implicit*: default constructor (if available) or
  - *explicit*: in the member initialization list

- An *explicit* constructor call must be made if parameters have to be passed on.

- The direct initialization of inherited members (= data members or member functions) via the initialization list is *not possible*.

- Order of construction:
  - First the *Base classes* in the order of their declaration in the derivation list (not in the member initialization list), then the
  - *Members* of the derived class in the order of their declaration.

Order of destruction: the other way around