

12. Vererbung

Prof. Dr. Markus Gross

Informatik I für D-ITET (WS 03/04)

- Vererbung – Konzept
- Protected Section
- Virtuelle Mitgliedsfunktionen
- Verwendung von Vererbung

Vererbung

- *Vererbung* ist ein Mechanismus, der es erlaubt, aus bestehenden Klassen, neue spezialisierte Klassen abzuleiten
 - ◆ Zusätzliche Funktionalität
 - ◆ Zusätzliche Daten
 - ◆ Veränderung von Methoden der übergeordneten Klasse
- Die übergeordnete Klasse heisst auch *Basisklasse*
- Die untergeordnete Klasse heisst auch *abgeleitete Klasse*
- Beispiel: Bank-Account Klasse aus Buch S. 605ff

Beispiel_1: Bank-Klasse

```
// bankacct.h -- a simple BankAccount class

#ifndef _BANKACCT_H_
#define _BANKACCT_H_

class BankAccount
{
private:
    enum {MAX = 35};
    char fullName[MAX];
    long acctNum;
    double balance;
public:
    BankAccount(const char *s = "Nullbody", long an = -1,
                double bal = 0.0);
    void Deposit(double amt);
    void Withdraw(double amt);
    double Balance() const;
    void ViewAcct() const;
};

#endif
```

Beispiel_1: Bank-Klasse

```
// bankacct.cpp -- methods for BankAccount class - Auszug
#include <iostream>
using namespace std;
#include "bankacct.h"
#include <cstring>

BankAccount::BankAccount(const char *s, long an, double bal)
{
    strncpy(fullName, s, MAX - 1);
    fullName[MAX - 1] = '\0';
    acctNum = an;
    balance = bal;
}

void BankAccount::Deposit(double amt)
{
    balance += amt;
}

void BankAccount::Withdraw(double amt)
{
    if (amt <= balance)
        balance -= amt;
    else
        cout << "Withdrawal amount of $" << amt
              << " exceeds your balance.\n"
              << "Withdrawal canceled.\n";
}

double BankAccount::Balance() const
{
    return balance;
}
```

Beispiel_1: Bank-Klasse

```
// usebank.cpp
// compile with bankacct.cpp

#include <iostream>
using namespace std;

#include <cstring>
#include "bankacct.h"

int main()
{
    BankAccount Porky("Porcelot Pigg", 381299, 4000.00);

    Porky.ViewAcct();
    Porky.Deposit(5000.00);
    cout << "New balance: $" << Porky.Balance() << endl;
    Porky.Withdraw(8000.00);
    cout << "New balance: $" << Porky.Balance() << endl;
    Porky.Withdraw(1200.00);
    cout << "New balance: $" << Porky.Balance() << endl;

    return 0;
}
```

Basisklasse

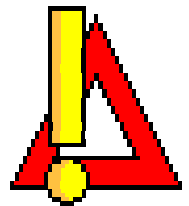
- Von der Basisklasse **BankAccount** kann nun eine abgeleitete Klasse definiert werden
- Diese erbt die Methoden und Daten der Basisklasse
- C++ kennt 3 Arten von Vererbung: **public**, **private**, **protected**
- Public Vererbung (Inheritance) modelliert eine *Is-Beziehung*
- Die abgeleitete Klasse hat vollen Zugriff auf alle Methoden und Daten der Basisklasse
- Beispiel: Banane ist-eine Frucht (*is-a* relation)
- Gegenbeispiel: Lunch und Frucht (*has-a* relation)
- Kann durch Einbeziehung von anderen Klassen erfolgen (*Containment*)

Abgeleitete Klassen

- Bei Deklaration einer abgeleiteten Klasse wird Klassenname der Basisklasse sowie Vererbungstyp nachgestellt

```
class Overdraft : public BankAccount;
```

- Public Members der Basisklasse werden public Members der abgeleiteten Klasse
- Private Daten der Basisklasse können über public Interface verwendet werden
- Ohne Angabe des Typs ist die Vererbung private
- Bei der Instantiierung wird zunächst der Basiskonstruktor aufgerufen, dann derjenige, der abgeleiteten Klasse
- Umgekehrt bei Destruktorenaufruf



Abgeleitete Klassen

- Bei *private Inheritance* gehen Daten und Methoden der **public** und **protected** Section der Basisklasse in die **private** Section der abgeleiteten Klasse über
Class Overdraft : private BankAccount;
 - ◆ Methoden werden also nicht Teil des public Interface der Klasse
 - ◆ Es wird jedoch die Implementierung vererbt
 - ◆ Abgeleitete Klasse kann somit Methoden der Basisklasse für eigene Implementierungen verwenden, aber nicht nach aussen sichtbar machen
 - ◆ Diese Methoden können NICHT weitervererbt werden

Abgeleitete Klassen

- Bei *protected Inheritance* gehen Daten und Methoden der **public** und **protected** Section der Basisklasse in die **protected** Section der abgeleiteten Klasse über
 - ◆ Wie **private**, jedoch weitervererbbar
 - ◆ Bedeutend bei Mehrfachvererbung
- Public oder protected Members sind von der abgeleiteten Klasse aus zugreifbar
- **friend** Funktionen können nicht vererbt werden
- Methoden der Basisklasse können dabei *überschrieben* werden

Konstrukturen

- Eine abgeleitete Klasse benötigt geg. eigene Konstruktoren
- Vor deren Aufruf, muss jedoch der Basisklassenkonstruktor aufgerufen werden
- Problem: Korrekte Initialisierung
- Beispiel: Kontruktor zu Overdraft soll auch Mitgliedsvariablen der Basisklasse initialisieren

```
Overdraft(const char s* = „Nullbody“, long an = -1,  
          double bal = 0.0, double ml = 500, double r = 0.10);
```

- 3 der Argumente sind für Basisklasse bestimmt
- Basisklassenkonstruktor (default constructor) wird vorab aufgerufen

```
Overdraft::Overdraft(const char s*, long an, double  
                    bal, double ml, double r) : BankAccount(s, an, bal)
```

Beispiel_2: Overdraft

```
// overdraft.cpp -- Overdraft class methods
#include <iostream>

using namespace std;
#include "overdraft.h"

Overdraft::Overdraft(const char *s, long an, double bal,
                    double ml, double r) : BankAccount(s, an, bal)
{
    maxLoan = ml;
    owesBank = 0.0;
    rate = r;
}

Overdraft::Overdraft(const BankAccount & ba, double ml, double r)
    : BankAccount(ba) // uses default copy constructor
{
    maxLoan = ml;
    owesBank = 0.0;
    rate = r;
}
```

Beispiel_2: Overdraft

```
// redefine how ViewAcct() works

void Overdraft::ViewAcct(int n) const
{
    cout << n << endl;
    // set up ###.## format
    ios_base::fmtflags initialState =
        cout.setf(ios_base::fixed, ios_base::floatfield);
    cout.setf(ios_base::showpoint);
    cout.precision(2);

    BankAccount::ViewAcct(); // display base portion
    cout << "Maximum loan: $" << maxLoan << endl;
    cout << "Owed to bank: $" << owesBank << endl;
    cout.setf(initialState);
}
```

Beispiel_2: Overdraft

```
// redefine how Withdraw() works

void Overdraft::Withdraw(double amt)
{
    // set up ###.## format
    ios_base::fmtflags initialState =
        cout.setf(ios_base::fixed, ios_base::floatfield);
    cout.setf(ios_base::showpoint);
    cout.precision(2);

    double bal = Balance();
    if (amt <= bal)
        BankAccount::Withdraw(amt);
    else if ( amt <= bal + maxLoan - owesBank)
    {
        double advance = amt - bal;
        owesBank += advance * (1.0 + rate);
        cout << "Bank advance: $" << advance << endl;
        cout << "Finance charge: $" << advance * rate << endl;
        Deposit(advance);
        BankAccount::Withdraw(amt);
    }
    else
        cout << "Credit limit exceeded. Transaction
        cancelled.\n";
    cout.setf(initialState);
}
```

Beispiel_2: Overdraft

```
// useover.cpp -- test Overdraft class
// compile with bankacct.cpp and overdrft.cpp

#include <iostream>
using namespace std;
#include "overdrft.h"

int main()
{
    BankAccount Porky("Porcelot Pigg", 381299, 4000.00);
    // convert Porcelot to new account type
    Overdraft Porky2(Porky);
    Porky2.ViewAcct(1);
    cout << "Depositing $5000:\n";
    Porky2.Deposit(5000.00);
    cout << "New balance: $" << Porky2.Balance() << "\n\n";
    cout << "Withdrawing $8000:\n";
    Porky2.Withdraw(8000.00);
    cout << "New balance: $" << Porky2.Balance() << "\n\n";
    cout << "Withdrawing $1200:\n";
    Porky2.Withdraw(1200.00);
    Porky2.ViewAcct(3);
    cout << "\nWithdrawing $500:\n";
    Porky2.Withdraw(500.00);
    Porky2.ViewAcct(3);
    Porky2.ResetOwes();
    cout << endl;
    Porky2.ViewAcct(4);

    return 0;
}
```

Dynamic Binding

- Im Beispiel wird die Methode `ViewAccount()` von der abgeleiteten Klasse überschrieben
- Beispiel: Pointer auf Basisklassenobjekt

```
BankAccount *bp = &bretta;
bp->ViewAcct(); // BankAccount::ViewAcct()
bp = &ophelia;
bp->ViewAcct(); // unklar, welche Funktion
```

 - ◆ Die Entscheidung über den korrekten Aufruf von `bp->ViewAcct()` kann zur Compilezeit nicht getroffen werden
 - ◆ Hängt vom Pointer ab
- Wir brauchen eine *dynamische Anbindung* der aufzurufenden Funktion zur Laufzeit

Dynamic Binding

- Feste Anbindung des Funktionscodes an den Aufruf zur Compilezeit heisst *statische Anbindung (static, early binding)*
- Anbindung des Funktionscodes an den Aufruf zur Laufzeit heisst *dynamische Anbindung (dynamic, late binding)*
- Bei statischer Anbindung wird im Beispiel die Basisklassenfunktion aufgerufen

```
bp = &ophelia;  
bp->ViewAcct(); // BankAccount::ViewAcct()
```

- Bei dynamischer Anbindung wird im Beispiel die Funktion der abgeleiteten Klasse aufgerufen

```
bp = &ophelia;  
bp->ViewAcct(); // Overdraft::ViewAcct()
```


Dynamic Binding

- Dynamische Anbindung wird mit dem Schlüsselwort **virtual** aufgerufen (in Basisklasse)
`virtual void ViewAcct() const;`
- Aus Effizienzgründen sollte dynamische Anbindung sorgsam verwendet werden
- Faustregel: Wenn Funktion in abgeleiteter Klasse redefiniert wird, dann deklariere sie **virtual**, andernfalls nicht!
- Dynamische Anbindung nicht mit Funktionsüberladung verwechseln!!

