Abstrakte Klassen

- Es gibt Anwendungen, bei denen von vornherein eine Basisklasse nur als **Vorlage** für abgeleitete Klassen dienen soll. Nicht alle Methoden müssen deshalb implementiert werden. Derartige Methoden, die den Zusatz **virtual** tragen müssen und im Methodenkopf mit **=0** enden, werden "**rein virtuell**" genannt:

```
class object {                          // abstrakte Klasse
    public: virtual void aus() = 0;  // rein virtuelle Methode
}
```

- Klassen mit wenigstens einer **rein virtuellen Methode** werden "**abstrakt**" genannt.

- Abstrakte Klassen können **nicht instantiiert** werden.

- **Zeiger** und **Referenzen** vom Typ einer **abstrakten Klasse** sind dagegen möglich. Diesen Zeigern oder Referenzen sind Adressen bzw. Objekte abgeleiteter Klassen zuweisbar.

- **Konstruktoren** und **Destruktoren** abstrakter Klassen sind sinnvoll, da Konstruktoren abgeleitete Klassen die Basisklassenkonstruktoren benötigen.

- Besitzt eine abstrakte Klasse Zeiger-Member, dann sind auch **Kopierkonstruktoren** und überladene **Zuweisungsoperatoren** sinnvoll.

- Spätestens in der abgeleiteten Klasse, die für die Objektinstantiierungen vorgesehen sind, müssen auch alle rein virtuellen Methoden konkret implementiert werden bzw. in übergeordneten Klassen implementiert worden sein.

- Klassen mit ausschließlich rein virtuellen Methoden werden als "**abstrakte Datentypen**" bzw. **Schnittstellen** oder **Interfaces** bezeichnet.

- **Schnittstellen** bzw. **Interfaces** stellen ein wichtiges Softwarearchitekturkonzept im **.NET-Framework** und der Sprache **C#** dar.
  Beispiel für Interface mit **struct** bzw. **class**:

```
struct IInterface { virtual void f1()=0;
                    virtual void f2()=0;}

// alternativ:
class IInterface { public: virtual void f1()=0;
                           virtual void f2()=0; }

// Implementierung (polymorph) einer Schnittstelle:
class CImplementation : public IInterface {
                        public: void f1(){ // ... }
                                void f2(){ // ... }};
```

Eine Vererbung ohne virtuelle Methoden wird auch **Implementierungsvererbung** genannt. Die Nutzung von Interfaces wird **Schnittstellenvererbung** genannt. **Multiple Vererbung** ist praktisch nur mit **Schnittstellenvererbung** beherrschbar. Aufgrund der in Basisklassen definierten Methoden und deren Vererbung in Richtung abgeleiteter Klassen führen Änderungen an diesen Methoden häufig zu unüberschaubaren Fehlern und großen Testaufwendungen in den erbenden abgeleiteten Klassen. Dagegen bewirkt die Schnittstellenvererbung eine optimal an die Klasse angepaßte, polymorphe und fehlerarme Implementierung

Das folgende Beispiel **abstractbase_late.cpp** besitzt die abstrakte Basisklasse **class abstrakt** mit der **rein virtuellen** Methode **virtual void f()=0**;. In den Klassen **konkret1** und **konkret2** wird **f()** polymorph implementiert, wobei **f_imp()** aus den jeweils zweiten Basisklassen gerufen werden. Über die Zeiger **k1** und **k2** erfolgt die späte Bindung von **f()** zur Laufzeit. Damit werden indirekt die Methoden **f_imp()** aus den beiden Basisklassen **class imp1** und **class imp2** gerufen:

```
#include <iostream>   // abstrakte Schnittstelle:  abstractbase_late.cpp
using namespace std;

class abstrakt {
    public: abstrakt(){ cout<<"Konstruktor abstract"
                           <<endl;
            }
            virtual void f()=0; //abstrakte Methode
};

// Implementation 1
class imp1 {
    public: void f_imp(){ cout<<"f_imp() aus imp1"
                          <<endl;
            }
};

// Implementation 2
class imp2 {
    public: void f_imp(){ cout<<"f_imp() aus imp2"
                          <<endl;
            }
};

// Konkrete Klasse, Version 1
class konkret1 : public abstrakt, private imp1 {
    public: void f(){ f_imp(); } // Redef. virt. Meth.
};

// Konkrete Klasse, Version 2
class konkret2 : public abstrakt, private imp2 {
    public: void f(){ f_imp(); }
};

void main(){
     abstrakt *k1 = new konkret1;
     abstrakt *k2 = new konkret2;
     k1->f();                    // late binding
     k2->f();                    // late binding
}
```

Ausgabe:

```
Konstruktor abstract
Konstruktor abstract
f_imp() aus imp1
f_imp() aus imp2
```