Definition späte Bindung (late binding, dynamische Bindung zur Laufzeit)**:**

Ist eine **nicht statische**, **gleichnamige** Methode

- sowohl in der **Basisklasse** als auch in der **abgeleiteten Klasse** definiert und

- besitzt die Methodendefinition wenigstens in der Basisklasse den Zusatz **virtual** und

- stimmen bei beiden Methodendefinitionen sowohl die **return-Typen** als auch die **Signaturen** überein und

- wird über **Zeiger** oder **Referenz** auf die Methoden zugegriffen,

dann wird in Abhängigkeit vom aktuellen Objekt, auf welches der Zeiger bzw. die Referenz verweist, die für den **Typ des Objektes zugehörige Methode zur Laufzeit** (spät, late, dynamisch) ausgewählt.

Mit der späten Bindung wird die **Polymorphie** (Vielgestaltigkeit) von gleichnamigen Methoden praktiziert, da jede der Methoden eine andere Funktionalität (anderen Anweisungsteil) besitzen kann.

Realisiert wird die späte Bindung, indem einem Basisklassenzeiger bzw. einem Basisklassenobjekt die Adresse bzw. das Objekt einer abgeleiteten Klasse zugewiesen wird. In der Regel gibt es mehrere abgeleitete Klassen und auch Ableitungsebenen. Vgl. hierzu die Beispiele **inh4.cpp**, **virt1.cpp**, **virt2.cpp**, **virt0.cpp** (alles späte Bindung), **virt0_old.cpp** (frühe Bindung)

Sind eine oder mehrere Voraussetzungen der späten Bindung **nicht** erfüllt, dann erfolgt die **frühe Bindung** (early binding, Bindung zur Compiletime). Beispielsweise ist das in **virt3.cpp** der Fall, da die Signaturen von **setvol** in **class musiker** und **class trompeter** nicht übereinstimmen. Die Methode **setvol** in der Klasse **trompeter** wird damit **nicht-virtuell** überschrieben, d.h. es existieren 2 Methoden **setvol** in **class trompeter**. Der Zugriff auf **setvol** der Basisklasse **musiker** seitens der Klasse **trompeter** geschieht beispielsweise in **setvol(long value)** über **musiker::setvol((int)value)**;. Über den Zeiger **pm** erfolgt der Zugriff auf **setvol** der Klasse **musiker** mit **pm->setvol(5)**. Auf das **setvol** der Klasse **trompeter** wird über **pt->setvol(10UL)**; zugegriffen:

```
#include <iostream>              // virt3.cpp
using namespace std;
class musiker { public: virtual void setvol(int value); };

class trompeter : public musiker { public : void setvol(long value); };

void musiker::setvol(int value){ musiker::setvol((int)value);
                                 cout<<"Musiker: "<<value<<endl; }

void trompeter::setvol(long value){ cout<<"Trompeter: "<<value<<endl; }

void main(){
            musiker *pm=new trompeter;
            trompeter *pt=(trompeter *)pm;  // expliziter cast
            pm->setvol(5);                  // Musiker: 5 , fruehe Bindung !
            pt->setvol(10L);                // Trompeter: 10

            delete pm; pm=0; cin.get();
}
```

Wird beispielsweise mit Objekten auf die Methoden zugegriffen, dann wird immer die frühe Bindung realsiert.

Bei der frühen Bindung (zur Compiletime) wird immer die Methode ausgewählt, die dem **Typ des Zeigers** bzw. der **Referenz** entspricht.

Der Zusatz **virtual** für gleichnamige Methoden kann auch in den abgeleiteten Klassen erfolgen, damit ist der Quelltext besser lesbar.

Kandidaten für virtuelle Methoden sind diejenigen, deren Implementation sich in abgeleiteten Klassen ändern kann.

Die Motivation für virtuelle Methoden ergibt sich insbesondere daraus, daß ohne virtuelle Methoden umfassende Tests mit **dynamic casts** erfolgen müßten, um den gleichen Effekt zu erzielen, vgl. **inh4dyncast.cpp** :

```
#include <iostream>    // inh4dyncast.cpp
using namespace std;   // Ersatz fuer virtual mittels Nutzung des
                       // dynamic_cast<derived *>(pb)
class base {

     public:  base(int i=0):a(i){ cout<<"Konstruktor base\n"; }

              ~base(){ cout<<"Destruktor base\n"; }

              void print(){ cout<<"base print a = "<<a<<endl; };
              void give(){ cout<<"base give a = "<<a<<endl; };
              int geta(){ return a; }
              virtual void dummy(){};  // notw. wegen dynamic_cast
     private: int a;
};

class derived : public base {
    public:  derived(int i=0):base(i){ cout<<"Konstr. derived\n"; }

             ~derived(){ cout<<"Destruktor derived\n"; }

             void print(){cout<<"derived print a = "<<geta()<<endl;} // override
             void give(){cout<<"derived give a = "<<geta()<<endl;}   // override
};

void main(){
             base b1(1), *pb=&b1;
             pb->print();           // base print(), Zeiger auf base-Objekt, 1

         // Ersatz fuer late binding: Testen des dynamic_cast<derived *>(pb)
             dynamic_cast<derived *>(pb)?(dynamic_cast<derived *>(pb))->give():
                                         (dynamic_cast<base *>(pb))->give();

             derived d1(2);
             pb = &d1;
             pb->print(); // base print(), Zeiger auf derived-Objekt, 2

         // Ersatz fuer late binding: Testen des dynamic_cast<derived *>(pb)
             if( dynamic_cast<derived *>(pb) )
                  ((derived *)pb)->give();
             else
                  pb->give();
}
```

Ausnahmen vom **identischen return-Typ** bei virtuellen Methoden sind erlaubt, ohne daß die späte Bindung beeinträchtigt wird:

Liefert die virtuelle Methode der Basisklasse eine **Adresse** oder eine **Referenz** auf eine Klasse **X**, so darf eine neue Version der virtuellen Methode in der abgeleiteten Klasse eine **Adresse** bzw. **Referenz** auf eine **von X abgeleitete Klasse Y** zurückgeben (vgl. virt_ret_exception.cpp).

Es ist auch möglich, daß eine virtuelle Methode in der Basisklasse **public** und in der abgeleiteten Klasse **private** ist. Für ein Objekt der abgeleiteten Klasse ist dann die **private** Methode nicht mehr im Zugriff.

Beim Zugriff mittels **später** Bindung über einen **Zeiger** oder **Referenz** der Basisklasse, wobei der Zeiger bzw. die Referenz auf ein abgeleitetes Objekt verweist, bleibt das **public**-Basisklasseninterface erhalten, womit der Zugriff auf die **private**-Methode der abgeleiteten Klasse "spät" erfolgt (vgl. **virt_ret_exception.cpp**):

```
#include <iostream>        // virt_ret_exception.cpp
using namespace std;

class X {};
class Y: public X {};

class A { public:   virtual X &f(X &x){ cout<<"A.f\n"; return x; }
class B: public A { private: virtual Y &f(X &x){ cout<<"B.f\n"; return (Y &)x; }

void main(){
            Y y;
            X &x = y;      // Referenz x mit y initialisieren
            B b;
            A &a = b;      // Referenz a mit b initialisieren
            a.f((Y &)x);   // spaete Bindung, Aufruf b.f, obwohl private
          // b.f(y);        // Compile-Error, da private
            cin.ignore();
}
```

Virtuelle Destruktoren

Destruktoren werden in Hierarchien abgeleiteter Klassen immer von der am weitesten abgeleiteten Klasse, beginnend in Richtung der allgemeinsten Klasse, aufgerufen, das ist die umgekehrte Richtung im Gegensaz zur Ruffolge der **Konstruktoren**.

Dynamisch erzeugte Objekte werden über Zeiger angesprochen, wobei ein Basisklassenzeiger u.U. auf ein Objekt einer abgeleiteten Klasse zeigt. Ein **delete** auf einen Basisklassenzeiger ruft den Destruktur der Basisklasse, nicht jedoch den Destruktor der abgeleiteten Klasse auf, auf den eigentlich verwiesen wird, was zur Nichtfreigabe von einzelnen Objektmembern und damit zu Fehlern führt. (vgl. **virt4.cpp**)

Um ein dynamisches, spätes Binden der Destruktoren an die Objekte zu erreichen, muß die Destruktordefinition den Zusatz "**virtual**" erhalten. Damit werden im Falle des Zugriffs mit Zeigern oder Referenzen auf mit **delete** freizugebende Objekt die korrekten Destruktoren gerufen (vgl. **virt5.cpp**)

Allgemein wird empfohlen, Destruktoren immer mit dem Zusatz **virtual** zu versehen.