Zeiger und Referenzen können polymorph sein (Objekte NICHT) !

```
Stack* sp = new CountedStack;
Stack& sr = *sp;
Stack s = *sp; // slicing
```

beim Aufruf (nicht-virtueller) Memberfunktionen entscheidet die statische Qualifikation (Eintrittspunkt zur wird zur Compile-Zeit ermittelt --> early binding)

```
sp->push (42); // Stack::push ! ???  --> Stack.h
sr .push (42); // Stack::push ! ???
// obwohl es ein eigenes CountedStack::push gibt und
// in beiden Fällen CountedStack-Objekte vorliegen
```

Memberfunktionen können jedoch (in der Basisklasse) als virtuell deklariert werden

dann entscheidet die dynamische Qualifikation (Eintrittspunkt wird zur Laufzeit ermittelt --> late binding)

```
class Stack' {...
public: virtual void push(int); ...};

Stack* sp = new CountedStack;
Stack& sr = *sp;
sp->push (42); // CountedStack::push !!!
sr .push (42); // CountedStack::push !!!
```

Um die Entscheidung in die Laufzeit vertagen zu können, muss eine Typinformation im Objekt hinterlegt werden

Ziel für C++: Mechanismus mit hoher Zeit- und Platzeffizienz

Realisierung (nicht normativ aber de facto Standard):

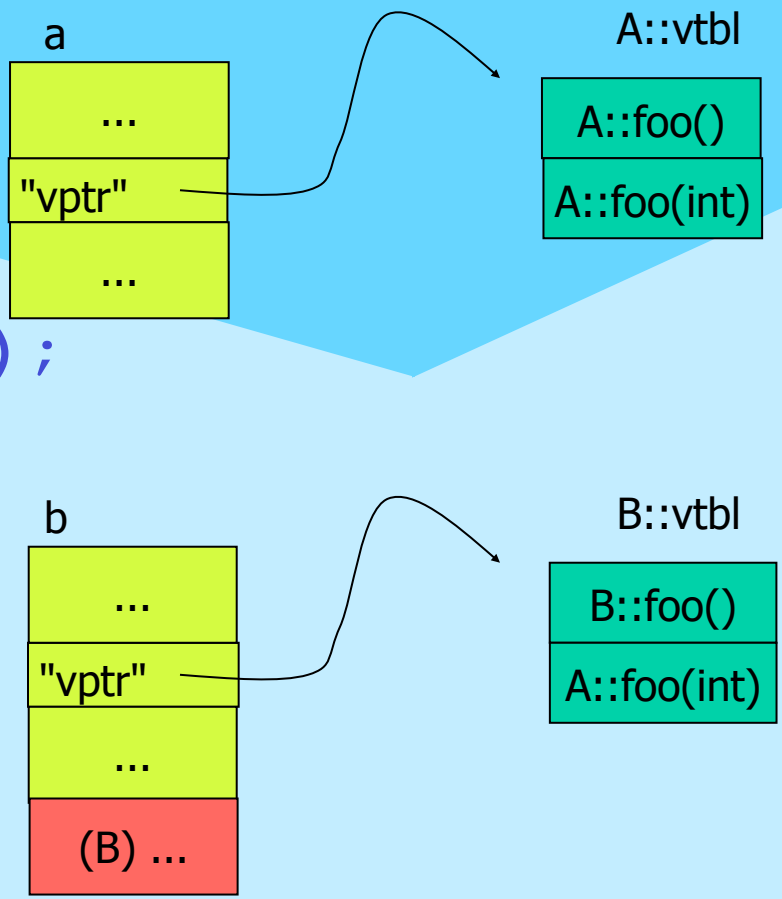
- ein (verborgener) Zeiger (**`vptr`**) pro Objekt +
- eine Adress-Substitution beim Aufruf virtueller Funktionen

damit ist *late binding* (geringfügig) teurer -- wie immer gilt das Prinzip *»Aufpreis nur auf Anfrage«*

Beispiel

```
struct A {
 void bar();
 virtual void foo();
 virtual void foo(int);
} a;

struct B : public A {
 void bar();
 virtual void foo();
} b;
```



Beispiel (Fortsetzung)

```
A ao;
A *ap = new A;
B bo;
B *bp = new B;
A *app = new B;

ao.foo();      // A::foo (&ao); NOT LATE!
ap->foo();     // (ap->vptr[0])(ap); LATE BINDING
bo.foo();      // B::foo (&bo); NOT LATE!
bp->foo();     // (bp->vptr[0])(bp); LATE BINDING
app->foo();    // (app->vptr[0])(app); LATE BINDING
app->foo(1);   // (app->vptr[1])(app); LATE BINDING
bp->foo(1);    // Warning W8022 ab.cpp 14: 'B::foo()' hides virtual function 'A::foo(int)'
               // Error E2227 ab.cpp 30: Extra parameter in call to B::foo() in function main()
```

Wie gelangt der richtige `vptr` in ein Objekt, der die korrekte dynamische Typinformation widerspiegelt ?

Durch eine initiale Operation bei der die Typzugehörigkeit des Objektes bekannt ist: **Konstruktoren** 'wissen, was sie gerade konstruieren'

```
A::A() // impliziter default-ctor
{ » this -> vptr = &A::vtbl; « }
B::B() : A() // impliziter default-ctor
{ » this -> vptr = &B::vtbl; « }
```

nicht jeder Aufruf einer virtuellen Funktion wird spät gebunden:

- Aufruf an einer Objektvariablen (s.o. `ao.foo();`)
- Aufruf mit scope resolution: --> CountedStack::push
- Aufruf in einem Konstruktor/Destruktor !

```
inline virtual void foo();
```

erlaubt, aber `inline` xor `virtual` pro Aufruf

```
static virtual void foo();
```
nicht erlaubt

Die »Planung« von austauschbarer Funktionalität muss in einer Basisklasse erfolgen, unterhalb dieser Basis ist die Funktionalität nicht verfügbar

Eine Redefinition einer virtuellen Funktion liegt nur vor, wenn die Signatur exakt mit dem ursprünglichen Prototyp übereinstimmt

Ausnahme: kovariante Ergebnistypen

```
class X {
public:
        virtual X* clone () { return new X(*this);}
};
class Y: public X {
public:
        virtual Y* clone () { return new Y(*this);}
};
int main()
{
        X x, *px=x.clone();
        Y y, *py=y.clone();
}
```

`virtual <returntyp> fkt` und `<returntyp> virtual fkt` sind synonym (bevorzugt 1. Variante)

»einmal virtuell, immer virtuell« (sofern die gleiche Funktion vorliegt), erneute `virtual` Deklaration in Ableitungen eigentlich redundant, aber empfohlen

Vorsicht: virtuelle Funktionen können u.U. »überdeckt « werden

```
#define O(X) std::cout<<#X<<std::endl;

struct A {
        virtual void foo() { O( A::foo() ); }
};
struct B : public A {
        void foo (int=0)    { O( B::foo(int) ); } // non virtual
};
struct C : public B {
                void foo()   { O( C::foo() ); }    };
```

```
int main()
{
        C c;
        B* p = &c;

        c.foo();
        p->foo();
}
```

```
C:\tmp>bcc32 hide.cpp
...
Warning W8022 hide.cpp
15:
'B::foo(int)' hides
virtual
function 'A::foo()'
...
C:\tmp>hide
C::foo()
B::foo(int)
```

g++ (auch 4.x) warnt nicht [nicht mal bei -Wall]

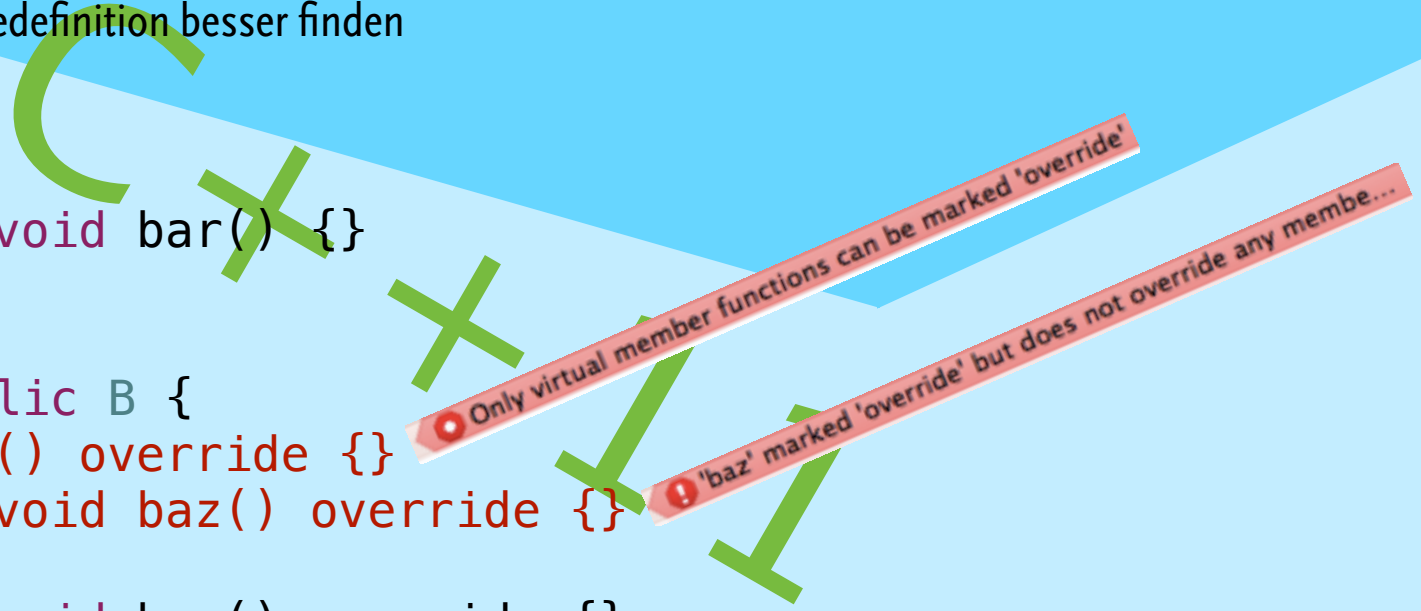
Neu in C++11 **override** (kein! reservierter Bezeichner - ein kontextsensitives Schlüsselwort)

Ziel: Fehler bei der Redefinition besser finden

```
class B {
public:
    virtual void bar() {}
};

class D: public B {
    void bac() override {}
    virtual void baz() override {}

    virtual void bar() override {}
} override; // OK no keyword in this context
```



Konstruktoren können nicht virtuell sein (warum nicht?)

Destruktoren können virtuell sein (und sollten dies auch sein, wenn in der Klasse ansonsten mindestens eine andere virtuelle Methode vorkommt)

```
class X {                          class Y: public X {
public:                            public:
        ...                                ...
        ~X();                              ~Y();
};                                 };
X* px = new Y;    delete px; // undefined behaviour (meist nur X::~X())
```

```
class X {                          class Y: public X {
public:                            public:
        ...                                ...
        virtual ~X();                      /*virtual*/ ~Y();
};                                 };
X* px = new Y;    delete px; // ruft Y::~Y() !!!
```

Überladung wird auch bei abgeleiteten Klassen lokal zu einer Klasse berechnet (ausgehend vom Bezugspunkt !):

```
#define O(X) std::cout<<#X<<std::endl;

struct X {
       void foo(int){O(X::foo(int));}
       void foo(char){O(X::foo(char));}
};

struct Y : public X {
       void foo(int){O(Y::foo(int));}
       void foo(double){O(Y::foo(double));}
};
```

Überladung wird auch bei abgeleiteten Klassen lokal zu einer Klasse berechnet (ausgehend vom Bezugspunkt !):

```
int main () {
                X x;
                x.foo(1);
                x.foo('1');
//              x.foo(1.0); Ambiguity between 'X::foo(int)' and 'X::foo(char)
                Y y;
                y.foo(1);
                y.foo('1');
                y.foo(1.0);
}
```

```
C:\tmp>lookup
X::foo(int)
X::foo(char)
Y::foo(int)
Y::foo(int)
Y::foo(double)
```

```
class X1 {
public:
        f(int);
};
// chain of derivations Xn : Xn-1 without f
class X9 : public X8 {
public:
        void f(double);
};
void g(X9* p) { p->f(2); } // X9::f or X1::f ? X9::f !
```

ARM: Unless the programmer has an unusually deep understanding of the program, the assumption will be that p->f(2) calls X9::f - and not X1::f declared deep in the base class. Under the C++ rules, this is indeed the case. Had the rules allowed X1::f to be chosen as a better match, unintentional overloading of unrelated functions would be a distinct possibility.

Wenn aber doch `X1::f` gemeint ist?

```
class X1 {
public:
      f(int);
};
// chain of derivations Xn : Xn-1 without f
class X9 : public X8 {
public:
      void f(double);
      void f(int i) { X1::f(i); } // inline !
};
void g(X9* p) { p->f(2); } // X1::f
```

Java dagegen betrachtet bei Überladung alle Funktionen aus der gesamten Vererbungslinie !

```
class X {
      void O(String s){System.out.println(s);}
      public void foo(int i) {O("X::foo(int)");}
      public void foo(char c){O("X::foo(char)");}
};

class Y extends X {
      public void foo(int i){O("Y::foo(int)");}
      public void foo(double d){O("Y::foo(double)");}
};
```

```
public class lookup {
        public static void
        main (String s [])
        {
                X x = new X();
                x.foo(1);
                x.foo('1');
                // x.foo(1.0); cannot find symbol foo(double)
                Y y = new Y();
                y.foo(1);
                y.foo('1'); // bis 1.4 Fehler:
                        // Reference to foo is ambiguous
                y.foo(1.0);
        }
}
```

```
C:\tmp>java lookup
X::foo(int)
X::foo(char)
Y::foo(int)
X::foo(char)
Y::foo(double)
```