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]

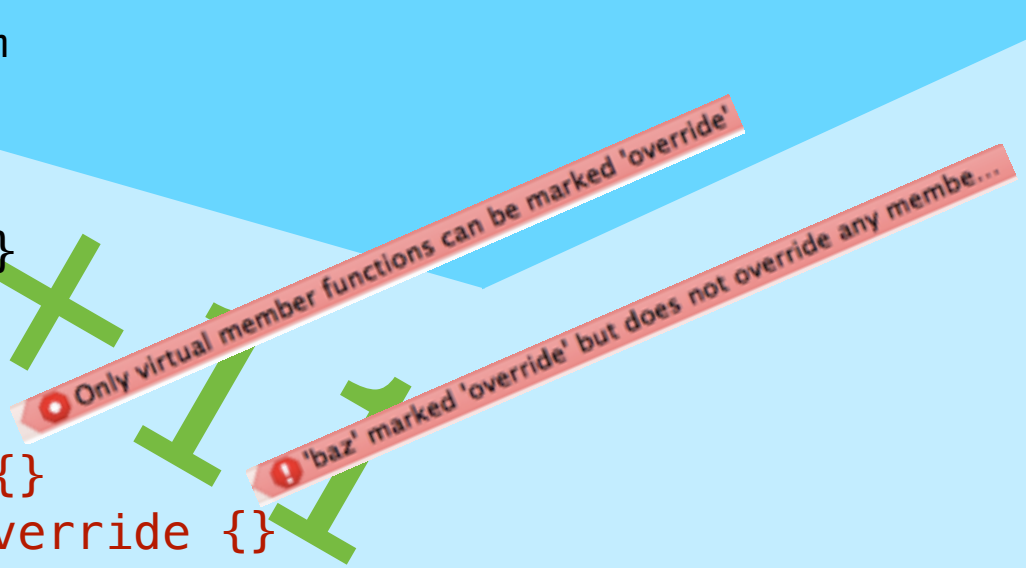
Neu in C++11 **override** (kein! reservierter Bezeichner - ein kontextsensitives Schlüsselwort)

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:
        void 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)
```

Ziel: maximales Code-Sharing -- Weg: gemeinsame (aber ggf. in Ableitungen variierende) Funktionalität in Basisklassen festlegen

Problem: die so entstehenden Basisklassen sind oft so rudimentär, dass Objekterzeugung nicht sinnvoll und Implementation einiger Memberfunktionen (noch nicht) möglich ist:

abstract base class (ABC)

pure virtual function

Beispiel:

```
struct AbstractShape {
        virtual void draw() = 0;
        virtual void erase()= 0;
};
// no objects allowed:
// AbstractShape aShape; ERROR
AbstractShape *any; // ok
any = new Circle (Point(0,0), 100);
```

```
struct Circle : // real Shape
 public AbstractShape {
  virtual void draw() {...}
  virtual void erase() {...}
};
```

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

finale Klassen: keine Ableitung möglich

finale Methoden: keine Redefinition in Ableitungen

```
class X {
public:
    virtual void foo();
};

class Y final : public X {
public:
    virtual void foo() override {}
};

void call (Y* p)
{
    p->foo(); // can bind statically !
}

// class Z : public Y {}; // not possible
```

finale Methoden müssen virtuell sein !

```
class Z {
    void virtual foo() const {}
};

class ZZ : public Z {
    void foo() const final override {};
} final, override;
```

2. Klassen in C++

```
class abstractBase { public:
        virtual void pure() = 0;
        void notPure() { pure(); }
        abstractBase() { notPure(); }
        virtual ~abstractBase() { notPure(); }
};

class concrete: public abstractBase { public:
        void pure() {}
        concrete() {}
};

int main() {
        cout<<"buggy:"<<endl;
        concrete c;
/*
        g++: pure virtual method called
        terminate called without an active exception
        Abort
*/
}
```

Scott Meyers, Effective C++ :
Item 9: "Never call virtual functions during construction or destruction."

Neu in C++11 **deleted/defaulted functions** (in Anlehnung an die Syntax von pure virtual functions)

```
class X {
public:
    X() = default;
    virtual ~X() = default;
    X(const X&) = delete;
    void foo(int);
    void foo(double) = delete;
};

X x;
X x1(x);
x.foo(1);
x.foo(1.0);

// delete auch für globale Funtionen
void bar(double);
void bar(int) = delete;
bar(1.9);
bar(19);
```

Call to deleted constructor of 'X'

Call to deleted member function 'foo'

Call to deleted function 'bar'

Im Kontext von Klassen können Operatoren mit nutzerdefinierter Semantik implementiert werden:

```
//Complex.h:                ∃  std::complex<T>
#include <iosfwd>
class Complex {
        double re, im;
public:
   Complex(double r = 0.0, double i = 0.0) : re(r), im(i) {}
   friend Complex operator+(const Complex&, const Complex&);
   friend Complex operator*(const Complex&, const Complex&);
   friend bool operator==(const Complex&, const Complex&);
   friend bool operator!=(const Complex&, const Complex&);
   Complex& operator+=(const Complex&); // Member !
   Complex operator-(); // Member !
   friend std::ostream& operator<<(std::ostream&, const Complex&);
   friend std::istream& operator>>(std::istream&, Complex&);
....};
```

```
//complex.cpp: Auswahl
Complex operator+(const Complex& c1, const Complex& c2) {
        return Complex(c1.re+c2.re, c1.im+c2.im);
}
bool operator==(const Complex& c1, const Complex& c2) {
        return (c1.re==c2.re && c1.im==c2.im);
}
Complex& Complex::operator+=(const Complex& c) {
        re += c.re; im += c.im;
        return *this;
}
Complex Complex::operator-() {
        return Complex(-re, -im);
}
std::ostream& operator<< (std::ostream& o, const Complex &c) {
        return o << c.re << "+i*" << c.im;
}
```