Häufig verwendete Muster (unter Ausnutzung von Zugriffsrechten)

Unikate - Objekte, die man nicht kopieren kann:

```
class U { // wie Unikat
        U(const U&); // privat und ohne Definition
        U& operator=(const U&); // dito
public:
        ...
};

U u1; // ein Unikat
U u2; // noch eines
U u3 (u1); // ERROR U::U(const U&)' is private within this context
void foo(U);
void bar () { foo(u1); }  // ERROR dito
```

Häufig verwendete Muster (a la C++11)

Unikate - Objekte, die man nicht kopieren kann:

C++11

```
class U { // wie Unikat
public:
        U(const U&) = delete;
        U& operator=(const U&) = delete;
        ...
};

U u1; // ein Unikat
U u2; // noch eines
U u3 (u1); // ERROR Call to deleted constructor of ´U
void foo(U);
void bar () { foo(u1); }  // ERROR dito
```

Häufig verwendete Muster (unter Ausnutzung von Zugriffsrechten)

Singletons - Objekte, die es nur einmal gibt

```
class S { // wie Singleton, mit lazy creation
        S( some parameters ) { .... }
        S(const S&);                // inhibit copy
        S& operator=(const S&);     // inhibit assign
        static S *it_;
public:
        static S& instance() {
                if (! it_) it_ = new S( parms );
                return *it_;
        }
};  // in S.h
S* S::it_ = 0; // in S.cpp, so nötig obwohl privat !

S::instance(); // gibt stets eine Referenz auf dasselbe Objekt

// Attn.: NOT thread safe
http://www.devarticles.com/c/a/Cplusplus/C-plus-in-Theory-Why-the-Double-Check-Lock-Pattern-Isnt-100-ThreadSafe/
```

Thread-safe Singletons

```
class Singleton {
    static std::shared_ptr<Singleton> instance_;
    static std::once_flag oflag;
    Singleton();// private !
    static void safe_create()
    { instance_.reset(new Singleton()); }
public:
    static std::shared_ptr<Singleton> instance() {
        std::call_once(oflag, safe_create); // variadic args
        return instance_;
    }
};
// in some cpp-File
std::shared_ptr<Singleton> Singleton::instance_;
std::once_flag Singleton::oflag;
```

Häufig verwendete Muster (unter Ausnutzung von Zugriffsrechten)

Factory - Objekte, die andere Objekte am Fließband produzieren

```
class P { // … wie Produkt
        // alles privat
public:
        friend class P_Factory;
};

class P_Factory { // sinnvollerweise zugleich singleton
public:
        P* generate () { .... return new P; }
};
....
P_factory::instance().generate();
```

Häufig verwendete Muster (unter Ausnutzung von Zugriffsrechten)

'No' - Objekte, die es (an sich) nicht gibt

```
class No { // keine Objekte sind erzeugbar
protected:
        No::No() { .... }
public: ...
};

No n; // ERROR NO::No() not accessible
```

Besseres Sprachfeature, um dies auszudrücken sind abstract base classes
- Klassen die sich nur für Vererbung, nicht für Objekterzeugung eignen (s.u.)

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):

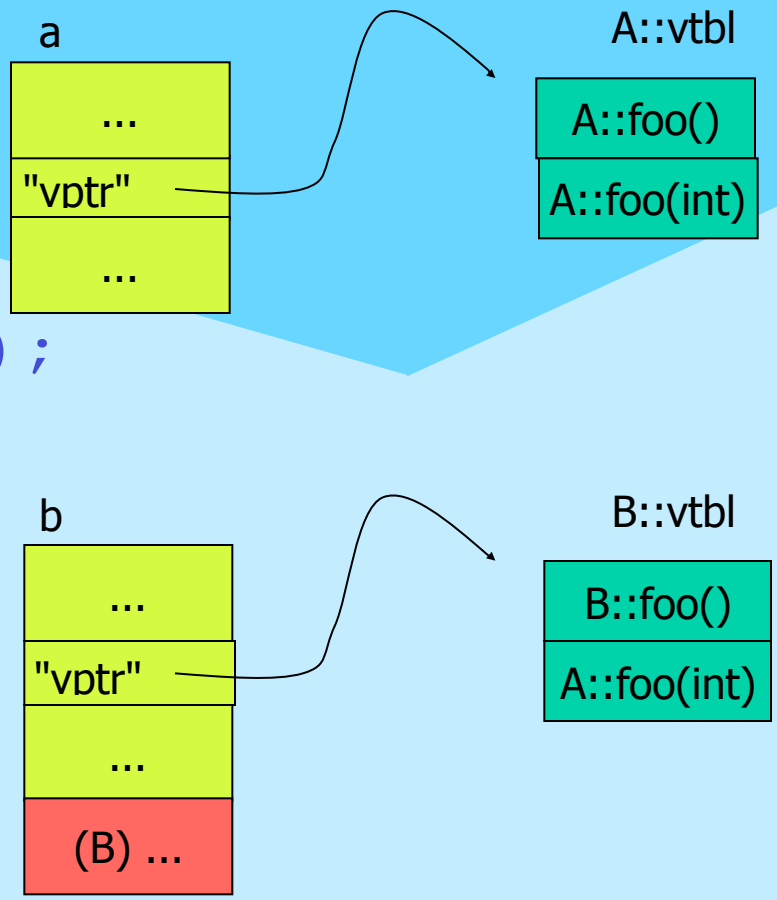
- ein (verborgener) Zeiger (**`vptr`**) pro Objekt +
- eine Adress-Substitution beim Aufruf virtueller Funktionen

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
```