Warum besteht bei `private`-Vererbung die IST EIN - Relation nicht ?

```
class A {                       class B : private A {
public:                                 ....
        int i;                  };
};

B b;
b.i = 1; // ERROR: `class B' has no member named `i'
// Wenn ein B ein A wäre:
A* pa = &b;
pa->i = 1; // sollte aber gerade geschützt werden !
// ergo, b ist kein A
A* pa = &b; // ERROR: `A' is an inaccessible base of `B'
```

Friends

oftmals ist die Entscheidung zwischen Alles (`public`) oder Nichts (`private`) zu restriktiv --> Möglichkeit, speziellen Klassen/Funktionen Zugriff einzuräumen, indem diese als `friend` deklariert werden

```
class B { public:void f(class A*); };
class A {
        int secret;
public:
        friend void trusted_function(A& a) // globale funktion !!!
        {... a.secret .... }               // inline !!!
        friend B::f(A*);
};
void B::f(A* pa) { .... pa->secret ....}
```

Friends

`friend`-Funktionen sind keine Memberfunktionen der Klasse, die die `friend`-Rechte einräumt

macht man eine ganze Klasse zum `friend`, werden alle Memberfunktionen dieser zu `friend`**s**

Vorsicht bei unterschiedlichen Kontexten für `inline`- und „outline"-Funktionen

```
typedef char* T;
class S {
        typedef int T;
        friend void f1(T) { .... } // void f1(int);
        friend void f2(T);         // void f2(int);
};
void f2(T) { .... } // void f2(char*); also kein friend !
```

Friends

`friend`-Funktionen sind keine Memberfunktionen der Klasse, die die `friend`-Rechte einräumt

macht man eine ganze Klasse zum `friend`, werden alle Memberfunktionen dieser zu `friend`**s**

Vorsicht bei unterschiedlichen Kontexten für `inline`- und „outline"-Funktionen

```
typedef char* T;
class S {
        typedef int T;
        friend void f1(T) { .... } // void f1(int);
        friend void f2(T);         // void f2(int);
};
void f2(T) { .... } // void f2(char*); also kein friend !
```

Friends

Die `friend`-Relation ist nicht symmetrisch, nicht transitiv & nicht vererbbar

```
class ReallySecure {
        friend class TrustedUser;
        ....
};
class TrustedUser {
        // can access all secrets
};
```

```
class Spy: public TrustedUser {
        // if friend relation would be inherited: aha !
};
```

Die Position einer `friend`-Deklaration in einem Klassenkörper (`private/protected/public`) ist ohne Bedeutung, dennoch sollte man `friend`-Deklarationen in einem `public` Abschnitt unterbringen (Schnittstelle der Klasse!)

```
// intermezzo: what's wrong with this code ?
#include <string>
#include <iostream>

class A{
public:
  const std::string& txt;
  A(const char *);
};
A::A(const char* chr) : txt(chr) {}
int main(){
  const char * foo = "foo";
  A test(foo);
  std::cout << test.txt << std::endl; // doesn't print "foo"
}
```

```
$ /opt/intel/compiler70/ia32/bin/icc -o z z.cpp
$ z

$ /opt/intel/compiler70/ia32/bin/icc -o z z.cpp -Wall
z.cpp(9): remark #383: value copied to temporary,
reference to temporary used
A::A(const char* chr) : txt(chr) {}
                              ^
```

Häufig verwendete Muster (unter Ausnutzung von Zugriffsrechten)

Unikate - Objekte, die man nicht kopieren kann:

```
class U { // wie Unikat
        U(const U&); // 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 (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/
```

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):

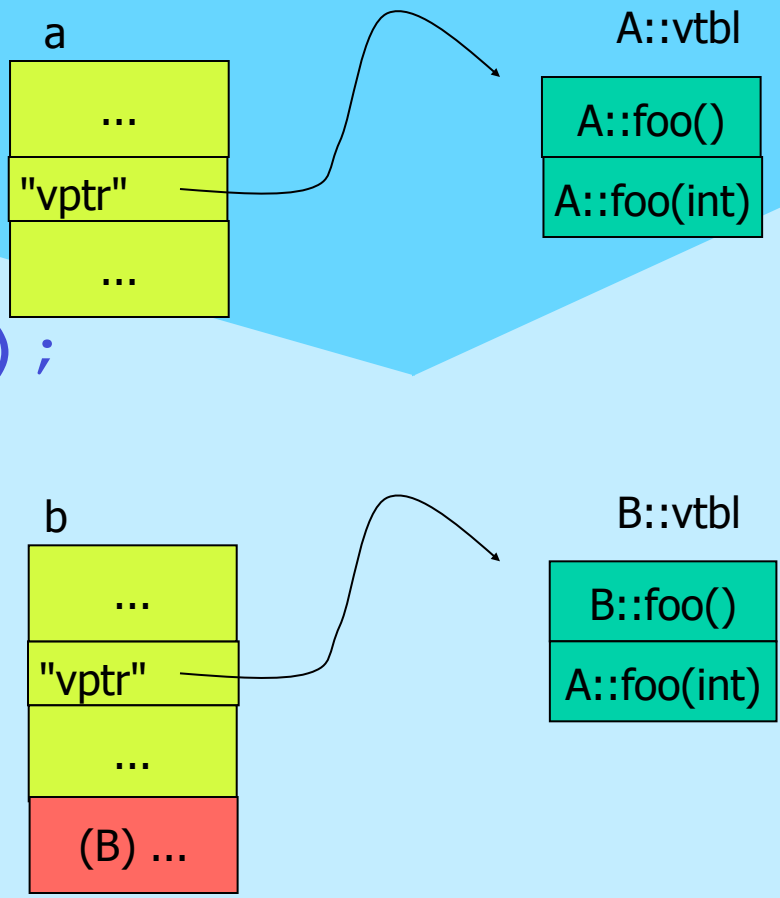
- ein (verborgener) Zeiger (**`vptr`**) pro Objekt +
- eine Adress-Substitution beim Aufruf virtueller Funktionen

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()
```