non-`public` Vererbung

```
class Deriv1 : private Base { .... };
```

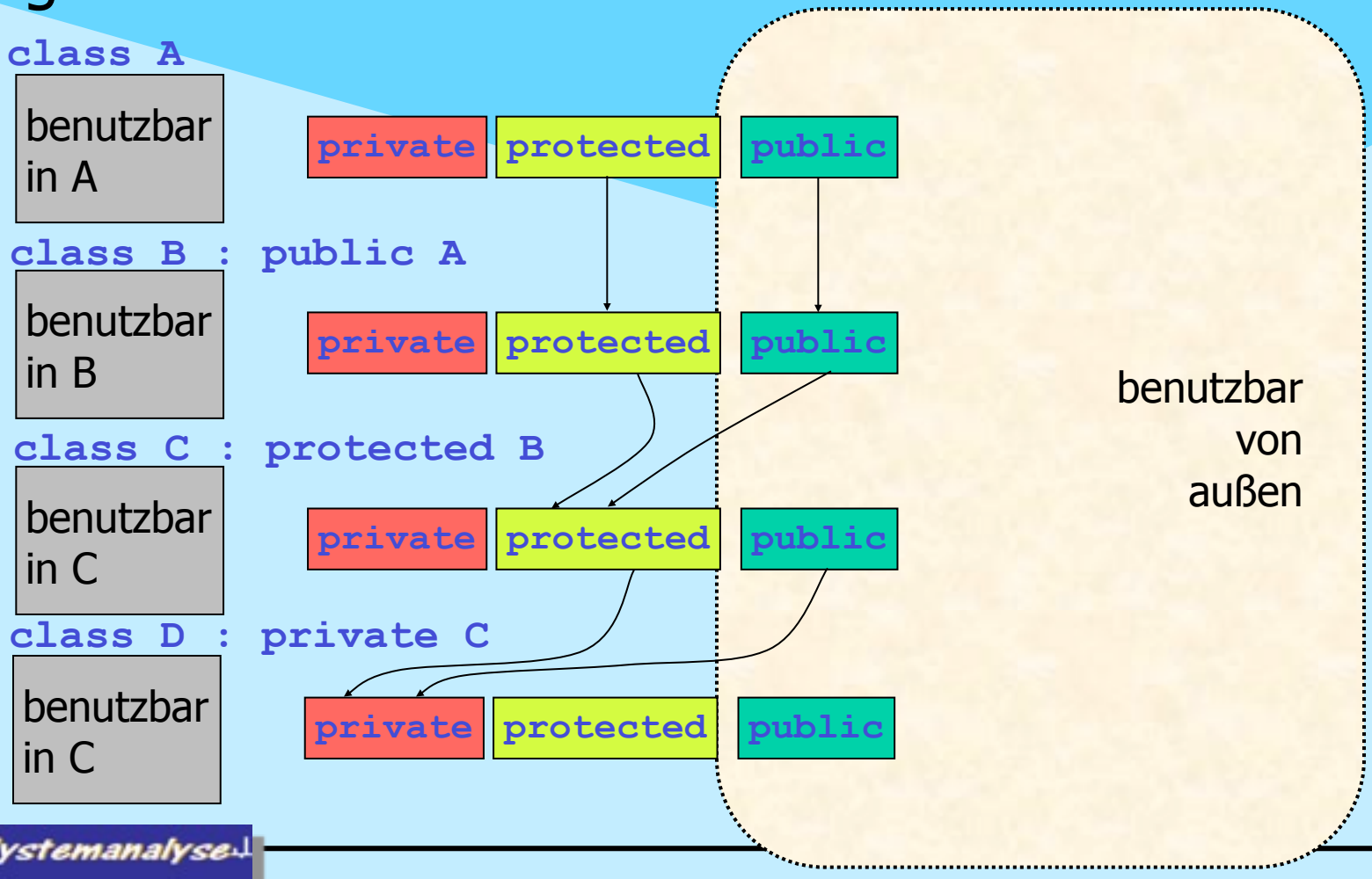
`Deriv1` IST nirgends EIN `Base` == die Vererbung ist ein (nicht erkennbares) Implementationsdetail

```
class Deriv2 : protected Base { .... };
```

`Deriv2` IST nur in Ableitungen von `Deriv2` EIN `Base` == die Vererbung ist nur Ableitungen `Deriv2` von bekannt

das Layout von Objekten abgeleiteter Klassen wird von der Art der Vererbung **NICHT** beeinflusst !

Zugriffsrechte in C++



`struct` ist implizit `public`, `class` ist implizit `private`

Depricated:

`struct` erbt implizit `public`, `class` erbt implizit `private`

Beim lookup von Funktionsnamen erfolgt
overload resolution **VOR** access check !

```
class X {
        foo(int);
public:
        foo(int, int = 0);
};

int main(){  X x;
             x.foo(1); //call of overloaded `foo(int)' is ambiguous
}
```

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!)

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/
```

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.)