Ist ein (nutzerdefinierter) Copy-Konstruktor erforderlich ?

Nein, weil der implizite Copy-K. die Copy-K.en aller Basisklassen ruft und für die Erweiterung `CountedStack` shallow copy ausreichend ist:

```
// implizit bereitgestellt:
CountedStack::CountedStack(const CountedStack& other)
  :
  Stack(other) {/* real copy */}
```

Der (nutzerdefinierte) `Stack`-Copy-K. erwartet allerdings eine

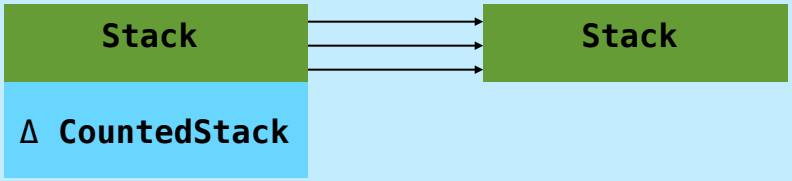
```
const Stack& ????
```

- Jedes **CountedStack** - Objekt **IST EIN** **Stack**-Objekt

```
CountedStack cs; ... cs.pop();
void foo (Stack&); ... foo (cs);
```

- von der Ableitung zur Basisklasse ist implizit eine Projektion definiert

```
void bar (Stack); ... bar(cs); // slicing
```



- nur bei `public` Vererbung gilt die **IST EIN** Relation

non-`public` Vererbung

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

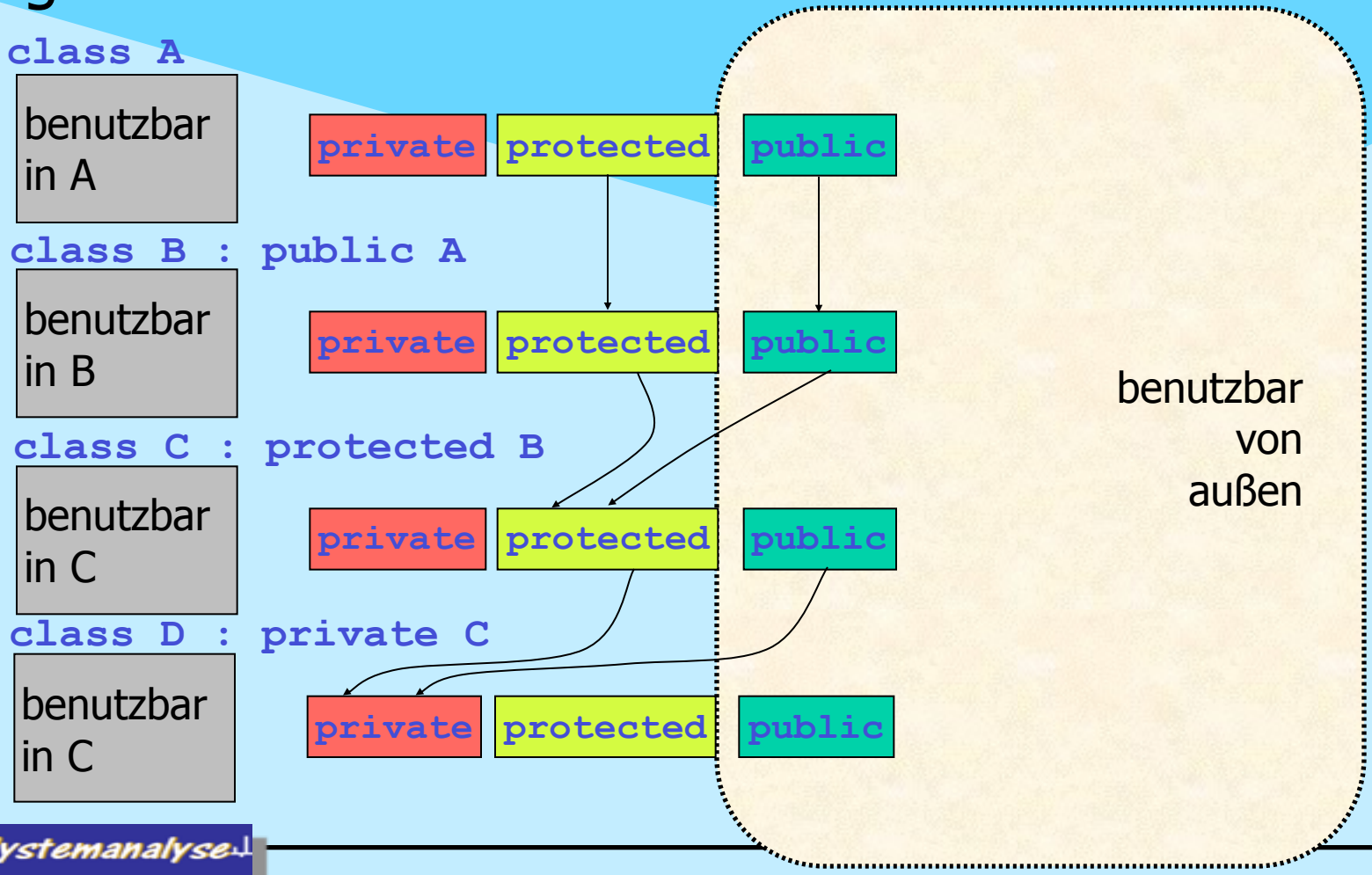
`Deriv1` IST nirgends EIN `Base` == die Vererbung ist ein (nicht erkennbares) Implementationsdetail

```
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!)