

2. Klassen in C++

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& ????`

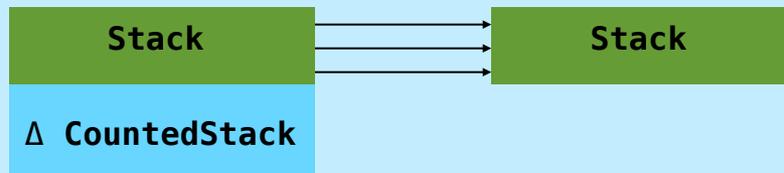
2. Klassen in C++

- 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

2. Klassen in C++

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 !

2. Klassen in C++

Zugriffsrechte in C++

`class A`

benutzbar
in A

`class B : public A`

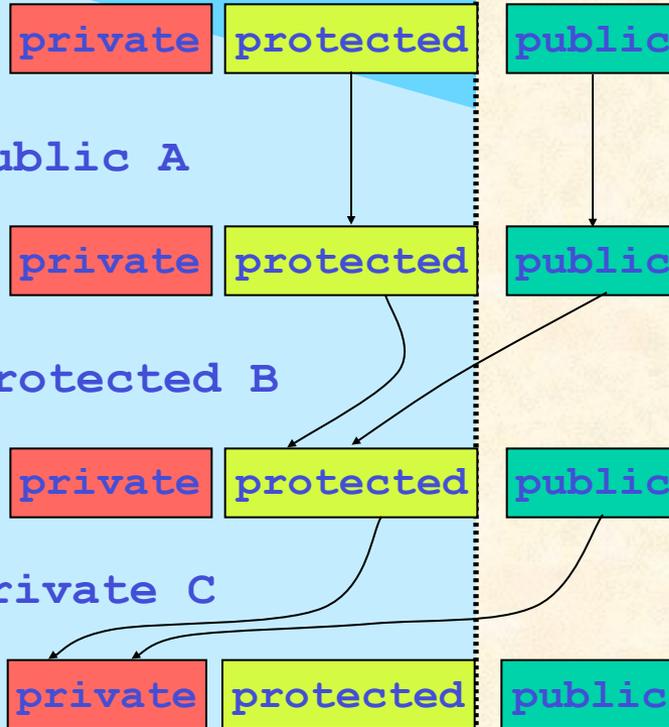
benutzbar
in B

`class C : protected B`

benutzbar
in C

`class D : private C`

benutzbar
in C



benutzbar
von
außen

2. Klassen in C++

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

Deprecated:

`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
}
```

2. Klassen in C++

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

```
class A {
public:
    int i;
};

class B : private A {
    ....
};

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'
```

2. Klassen in C++

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 .... }
```

2. Klassen in C++

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 **friends**

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 !
```

2. Klassen in C++

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 **friends**

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 !
```

2. Klassen in C++

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!)

2. Klassen in C++

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
```

2. Klassen in C++

Häufig verwendete Muster (a la C++11)

Unikate - Objekte, die man nicht kopieren kann:

```
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
```

2. Klassen in C++

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/>

2. Klassen in C++

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;
```

2. Klassen in C++

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();
```

2. Klassen in C++

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.)

2. Klassen in C++

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
```

2. Klassen in C++

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 !!!
```

2. Klassen in C++

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

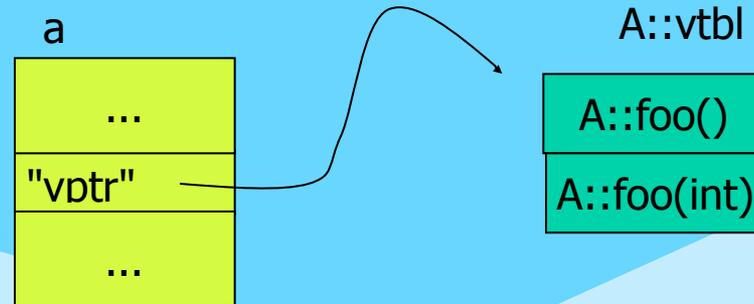
damit ist *late binding* (geringfügig) teurer -- wie immer gilt das Prinzip »*Aufpreis nur auf Anfrage*«

2. Klassen in C++

Beispiel

```

struct A {
    void bar();
    virtual void foo();
    virtual void foo(int);
} a;
  
```



```

struct B : public A {
    void bar();
    virtual void foo();
} b;
  
```

