- Klassen können auch sogenannte static Member enthalten, diese werden nur einmal pro Klasse angelegt !
- `static` Memberfunktionen dürfen (implizit) nur auf static Memberdaten zugreifen, (sie haben keinen `this`-Zeiger!)
- `static` Memberdaten sind nicht Teil des Objekt-Layouts
- `static` Memberdaten sind (einmalig) zu initialisieren !

```
$ s
Es gibt jetzt 3 A-Objekte
```

```
class A {
   static int count;
public:
   static int c(){ return count; }
   static const double A_specific_const; // NOT HERE = 123.456;
   A() {count++;}
   A(const A&) {count++; /* and copy */} // Kopien mitzählen !
   ~A(){count--;}
} a1, a2, a3;
int A::count = 0; // hier erst definiert !
const double A::A_specific_const = 123.456; // dito
int main() {
   double x = A::A_specific_const; // class access
   // A::A_specific_const = 1.23; // Fehler: const !
   cout << "Es gibt jetzt "<< a1.c()<<" A-Objekte\n";
// a1.count ist private,auch a2.c() oder a3.c() oder A::c() möglich
}
```

- neben den traditionellen C-Zeigern gibt es in C++ auch spezielle Zeigertypen für Zeiger auf Member(-daten und -funktionen)

```
class X { public: int p1,p2,p3; };
void foo() {
  X x; X* pp=&x;      // ein C-Zeiger auf ein X
  int X::*xp=&X::p2; // xp ist ein Zeiger auf ein int in X
// xp = &x.p2;
// error: bad assignment type: int X::* = int *
  int *p;
// p = &X::p2;
// error: bad assignment type: int * = int X::*
  p = &(x.*xp); // ok, ohne Klammern falsch: (&x).*xp
  pp->*xp = 1; } // .* und ->* sind neue Operatoren
```

```
class Y {
public:
  void f1(){cout<<"Y::f1()\n";};
  void f2(){cout<<"Y::f2()\n";};
  static void f3(){cout<<"static Y::f3()\n";}
  typedef void (Y::*Action)();
  void repeat(Action=&Y::f1, int=1);//...(void(Y::*)(),int)
};
void Y::repeat (Action a, int count) {
  while (count--) (this->*a)();
}
int main() {
  Y y; Y* pp=&y;
  void (Y::*yfp)();
  // Zeiger auf Memberfkt. in Y mit Signatur void->void
```

```
yfp=&Y::f1; // nicht yfp =Y::f1 !(trotz vc++6.0, bcc32, icc)
// yfp();
// object missing in call through pointer to memberfunction
(y.*yfp)(); // Y::f1()
yfp=&Y::f2;
(pp->*yfp)(); // Y::f2()
// yfp=&Y::f3;
// bad assignment type: void (Y::*)() = void (*)()static
// aber:
void (*fp)()=&Y::f3;
fp(); // besser (*fp)();
y.repeat(yfp, 2);
}
```

```
$ mp
Y::f1()
Y::f2()
static Y::f3()
Y::f2()
Y::f2()
```

# Vererbung: Grundprinzip von OO

- Übernahme von Eigenschaften aus einer Klasse
- Erweiterung / Modifikation

## Beispiel: ein Stack mit Buchführung

```
class CountedStack : public Stack // IST EIN STACK
{
   int min, max, n, sum; // zusätzliche Attribute
public:
   CountedStack(int dim = 100);
   void push (int i); // redefined !
   int minimum(); // neu
   int maximum(); // neu
   double mean(); // neu
   double actual_mean();// neu
// pop, empty, full aus der Basisklasse !
};
```

```
CountedStack::CountedStack(int dim):Stack(dim),n(0),sum(0){}

void CountedStack::push(int i) {
   sum+=i;
   if (!n++) { min = max = i; }
   else { min = (i<min) ? i : min; max = (i>max) ? i : max; }
   Stack::push(i); // use base functionality NOT push(i)
}

double CountedStack::actual_mean() {
   if (top) { int s=0;
       for (int i=0; i<top; i++) s += data[i];
       return double(s)/top; // direct access to base members
   } else std::exit(-4);
}
```

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 */}
```

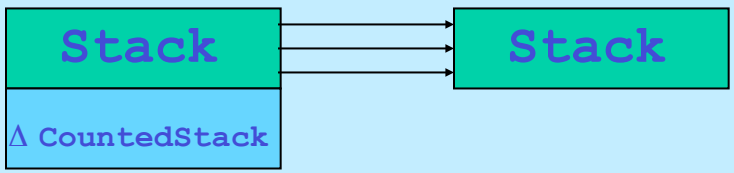
Der (nutzerdefinierte) `Stack`-Copy-K. erwartet allerdings eine
`const Stack& ????`

- 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