

Klassen und Objekte



Ceci n'est pas une pipe.

Retrospektive

- Für die Modellierung von Dingen oder Sachverhalten nutzt man in C Strukturen.
- Strukturen sind dabei benutzerdefinierte Datentypen, die Komponenten verschiedenen Datentypes zu einem neuen Datentyp vereinen.
- Von diesem Datentyp können (beliebig) viele Objekte erzeugt werden, die dann jeweils ein Ding oder einen konkreten Sachverhalt modellieren.

- In diesem Sinn wird ein konkreter Student, z.B. Hans Huckebein, durch ein Objekt zugegebenermaßen abstrahiert beschrieben.
- Alle Objekte, die in dieser Weise Studenten beschreiben, werden ihrerseits durch den Datentyp `tStudent` beschrieben.
- Die Deklaration eines solchen benutzerdefinierten Datentyps erfolgt in den meisten Fällen in einem Headerfile, in dem auch die Prototypen der Funktionen, die zur Arbeit mit eben diesem Datentyp bereitgestellt werden, angegeben sind.

Als Beispiel ein struct btime

```
#ifndef _H_BTIME_
#define _H_BTIME_
#include "btime.h"
```

```
typedef struct
{
    int h;
    int m;
}btime;
```

```
btime btimeGetCurrentTime();
void btimeShow(btime* pt);
btime btimeIncrement(btime t);
```

```
#endif
```

Datenstruktur

Funktionsprototypen dazu

- Die struktur `btime` dient der Modellierung einer Uhrzeit, bestehend aus Stunde(h) und Minute(m).
- Es gebe beispielhaft drei Funktionen zur Arbeit mit dem „Zeit-struct“.
- `btime btimeGetCurrentTime();`
 - Besorgt die aktuelle Systemzeit und trägt davon die Stunde und die Minute in eine neue Variable unserer Struktur und gibt diese als Wert zurück.
- `void btimeShow(btime* pt);`
 - Gibt die Werte als Zeit im Format hh:mm aus
- `btime btimeIncrement(btime t);`
 - Berechnet die Zeit der nächsten Minute

- Implementiert sind die drei Funktionen in der c-Datei btime.c wie folgt:

```
#include <stdio.h>
#include <time.h>
#include "btime.h"

btime btimeGetCurrentTime()
{
    btime tmp={};
    time_t lc = time(NULL);
    struct tm*plc = localtime(&lc);
    int h,m;
    tmp.h=plc->tm_hour;
    tmp.m=plc->tm_min;
    return tmp;
}
```

```
void btimeShow(btime* pt)
{
    printf("%02d:%02d",
           pt->h,pt->m);
}

btime btimeIncrement(btime t)
{
    t.m++;
    if (t.m==60)
        {t.m = 0; t.h++;}
    if (t.h==24)
        {t.h = t.m = 0;}
    return t;
}
```

- Verwendung findet unser keiner btime – Modul in der c-Datei btimeMain.c

```
#include <stdio.h>
#include <time.h>
#include "btime.h"

int main(int argc, char*argv[])
{
    btime bt={};
    if (argc==3)
        {bt.h=atoi(argv[1]); bt.m=atoi(argv[2]);}
    else bt=btimeGetCurrentTime();
    printf("time is now ");
    btimeShow(&bt);
    putchar('\n');
    bt=btimeIncrement(bt);
    printf("next time is ");
    btimeShow(&bt);
    putchar('\n');
}
```

- Das Ganze können wir nun mit
gcc btime.c btimemain.c
- Oder
gcc btime*.c
- Nach a.out compilieren und linken.

```
./a.out  
time is now 18:04  
next time is 18:05  
beck@U330p:~/SS2016/cpp/class$ ./a.out 19 00  
time is now 19:00  
next time is 19:01
```


Zusammenfassend ist festzustellen:

- Es gibt einen Programmmodul btime bestehend aus
 - Einer öffentlichen Schnittstelle, die die Struktur und die dazugehörige Funktionalität in Form der Funktionen beschreibt (btime.h)
 - Einer Implementationsdatei btime.c, die die Funktionalität implementiert. Dieser Teil könnte auch in Form einer vorkompilierten Datei btime.o oder als Library libbtime.a oder Teil einer Library vorliegen.
 - Die, im Headerfile enthaltene Struktur kann ggf. vor dem Benutzer auch noch verborgen werden, so wäre der Nutzer definitiv auf die öffentlichen Funktionen angewiesen, um mit dem Modul zu arbeiten.

Geändertes Beispiel mit verborgenem struct Typ

btimep.h

```
#ifndef _H_BTIME_
#define _H_BTIME_

struct btimes;
typedef struct btimes btime;

btime* btimeCreate();
void btimeGetCurrentTime(btime*);
void btimeShow(btime* pt);
void btimeIncrement(btime* t);

#endif
```

Leere struct-forward deklaration
Allerdings kann der Benutzer
nun keine Variable, nur Pointer
des struct-Types anlegen!

btimep.c

```
#include <stdio.h>
#include <time.h>
#include <malloc.h>
#include "btimep.h"
```

```
typedef struct
{
    int h;
    int m;
}btime;
```

Hier ist jetzt die
vollständige
Strukturdeklaration

```
btime* btimeCreate()
{
    btime*pbt=malloc(sizeof (btime));
    pbt->h=pbt->m=0;
    return pbt;
}
```

```
void btimeGetCurrentTime(btime*bt)
{
    time_t lc = time(NULL);
    struct tm*plc = localtime(&lc);
    int h,m;
    bt->h=plc->tm_hour;
    bt->m=plc->tm_min;
}
```

```
void btimeShow(btime* pt)
{
    printf("%02d:%02d",pt->h,pt->m);
}
```

```
void btimeIncrement(btime* t)
{
    t->m++;
    if (t->m==60)
        {t->m = 0; t->h++;}
    if (t->h==24)
        {t->h = t->m = 0;}
}
```

```
#include <stdio.h>
#include <time.h>
#include "btime.h"

int main(int argc, char*argv[])
{
    btime *pbt=btimeCreate();
    btimeGetCurrentTime(pbt);
    printf("time is now  ");
    btimeShow(pbt);
    putchar('\n');
    btimeIncrement(pbt);
    printf("next time is  ");
    btimeShow(pbt);
    putchar('\n');
}
```

Vom c-struct zur Klasse

- Eine Klasse dient in gleicher Weise, wie ein struct-Datentyp der Modellierung.
- Betrachtet man die Funktionen zum c-Struct genauer, stellt man fest, dass alle einen Parameter von dem struct-Typ (Pointer) haben, mit dem sie die Operation durchführen sollen.
- Eine Klasse ist nun ein Verbund der Daten eines struct-Types, wie wir ihn von c kennen und der Funktionen dazu.
- Da Funktionen und Daten nun eine Symbiose bilden, kann der erste Parameter auf magische Weise entfallen.

Schritte vom c-Struct zur Klasse

- Im Headerfile:
 - typedef entfernen.
 - Die Funktionsprototypen in den struct vor die Daten kopieren.
 - Zwischen Funktionen und Daten ggf. private ergänzen
 - Den ersten Parameter bei allen Funktionen entfernen.
 - Die create-funktion entfernen.
- Das Ergebnis könnte wie folgt aussehen:

Voila! Unsere erste Klassendeklaration!

```
#ifndef _H_BTIME_
#define _H_BTIME_

struct clbtime
{
    void getCurrentTime();
    void show();
    void increment();
private:
    int h;
    int m;
};

#endif
```

Öffentliche Funktionen

Private Daten

Nun zum Implementationsfile

- Streichen des jeweils ersten Parameters.
- Vor den Funktionsnamen den Klassennamen, gefolgt von `::` setzen (hier `clbtime::`).
- Den ersten Parameter, der im Beispiel immer ein Pointer vom Typ `btime*` war, im Anweisungsteil durch `this->` ersetzen.
- Noch ein paar Kleinigkeiten:
 - `malloc` braucht jetzt einen `cast`
 - Includes auf `cpp-includes` ändern
 - `printf` auf `cout<<` ändern


```
#include <iostream>
#include <iomanip>
#include <cstdio>
#include <ctime>
#include <cstdlib>
using namespace std;

#include "clbtime.h"

void clbtime::getCurrentTime()
{
    time_t lc = time(NULL);
    struct tm*plc = localtime(&lc);
    int h,m;
    this->h=plc->tm_hour;
    this->m=plc->tm_min;
}
```

```
void clbtime::show()
{
    cout<<setw(2)<<setfill('0')
    <<this->h
    <<':'<<setw(2)<<setfill('0')
    <<this->m;
}

void clbtime::increment()
{
    this->m++;
    if (this->m==60)
        {this->m = 0; this->h++;}
    if (this->h==24)
        {this->h = this->m = 0;}
}
```

Anmerkungen zu this

- Hinter this verbirgt sich in den Memberfunktionen ein Pointer auf das Objekt, zu dem die Memberfunktion aufgerufen worden ist.
- Man kann this-> beim Zugriff auf Member auch weglassen. In manchen Situationen braucht man this aber, z.B. bei Namensgleichheit von Funktionsparameter(n) und Member(n).

```
this->p=p;
```

Änderungen im Hauptprogramm

- Änderungen der includes
- Statt des Pointers definieren wir uns eine Variable `clbtime bt`;
- Bei allen Funktionsaufrufen, die unsere Klasse betreffen, streichen wir den ersten Parameter.
- Vor alle Funktionsaufrufe, die unsere Klasse betreffen, setzen wir `bt .` , damit verdeutlichen wir, dass wir die Funktion zu dem Objekt `bt` aufrufen, und bewirken dass der interne Pointer `this` genau auf dieses Objekt zeigt.

Verwendung unserer ersten Klasse

```
#include <iostream>
#include <ctime>
#include <cstdlib>
#include "btime.h"

using namespace std;

int main(int argc, char*argv[])
{
    btime bt;
    bt.btimeGetCurrentTime();

    bt.btimeShow();
    cout<<endl;

    return 0;
}
```

Klassen und Objekte

- Eine Klasse definiert einen benutzerdefinierten Datentyp.
- In der Regel stellt eine Klasse einen Verbund aus Daten (Membervariablen) und Funktionalität (Memberfunktionen/Methoden) dar.
- Instanzen einer Klasse bezeichnet man als Objekte.
- Objekte sind gekennzeichnet durch:
 - State (Status): Gesamtheit der Werte der Membervariablen
 - Behavior (Verhalten): Bestimmt durch die Gesamtheit der Memberfunktionen

Klassen und Objekte

- Es gibt public Member und private Member.
- In C++ gibt es in Klassen public- und private Bereiche. Sie werden durch `public:` bzw. `private:` eingeleitet und gelten bis zur nächsten Festlegung oder dem Ende der Klassendeklaration. Public und private regeln die Sichtbarkeit von Membern.
- Wird eine Klasse mit dem Schlüsselwort `struct`, wie in der Beispielklasse `cbtime` eingeleitet, so sind die Member, solange nichts anderes festgelegt ist, `public`. Beginnt eine Klasse mit dem Schlüsselwort `class`, so sind die Member vor einer definierten Festlegung `private`.
- `struct` oder `class` nennt man `Klasskey`.
- Die Klasse `cbtime` könnte auch, wie folgt lauten:

Klassen und Objekte

```
#ifndef _H_BTIME_
#define _H_BTIME_

class clbtime
{
public:
    void getCurrentTime();
    void show();
    void increment();
private:
    int h;
    int m;
};

#endif
```

Initialisierung

- Zur Initialisierung von Objekten gibt es spezielle Funktionen – Constructoren.
- Ein Constructor trägt den Namen der Klasse als Funktionsname und hat keinen Returntyp.
- Ein Constructor der Klasse `clbtime` könnte z.B. folgenden Prototyp haben:

```
clbtime(); //in der Classdeklaration
clbtime::clbtime(){...} // im Implementationsfile
```
- Constructoren können überladen werden, eine Klasse kann also mehrere Constructoren haben.
- Die Daten eines Objektes sollen immer valide sein. Dafür haben der u.a. Constructoren zu sorgen.

Constructoren

```
class clbtime
{
public:
    clbtime();
    clbtime(int h, int m);
    static clbtime* create();
    void getCurrentTime();
    void show();
    void increment();
private:
    int h;
    int m;
};
```

include und defines
Wie gehabt, deshalb hier
weggelassen

Constructoren

```
#include "clbtime.h"
```

Ausschnitt aus dem
Implementationsfile

```
// Defaultconstructor
```

```
clbtime::clbtime()
```

```
{
```

```
    h=m=0;
```

```
}
```

```
// Ueberladener Constructor
```

```
clbtime::clbtime(int h, int m)
```

```
{
```

```
    if (h>0 && h<24)    this->h=h;
```

```
    else                this->h=0;
```

```
    if (m>0 && m<59)    this->m=m;
```

```
    else                this->m=0;
```

```
}
```

```
. . .
```

Constructoren

```
int main(int argc, char*argv[])
{
    clbtime bt1;
    clbtime bt2(12,00);
    cout<<"time bt1:  ";
    bt1.show();
    cout<<endl;
    cout<<"time bt2:  ";
    bt2.show();
    cout<<endl;
}
```

Includes, wie gehabt

Aufruf des
Defaultconstructors

Aufruf des
Überladenen
Constructors

```
./a.out
time bt1:  00:00
time bt2:  12:00
```

Constructoren

- Ein Constructor wird automatisch bei der Erzeugung eines Objektes aufgerufen.
- Gibt es mehrere, überladene Constructoren, so wählt der Compiler auf Grundlage der angegebenen Parameter den passenden Constructor aus.
- Constructoren können auch Defaultargumente haben, es dürfen nur keine Mehrdeutigkeiten entstehen.
- Hat eine Klasse keinen Constructor, so generiert der Compiler einen Defaultconstructor ohne jedwede Funktionalität. Die Member sind in diesem Falle mit undefinierten Werten belegt!

Constructoren

- Ein Constructor tätigt normalerweise keine Ein- oder Ausgaben.
- Constructoren müssen public sein, sonst sind sie sinnlos.
- die Parameter einer Instanzierung einer Klasse (Objekterzeugung) sind die Parameter des entsprechenden Constructors.

Memberfunktionen/Methoden

- Memberfunktionen können auch überladen werden und ebenfalls Defaultparameter haben.
- Memberfunktionen einer Klasse können sich gegenseitig durch `this->func()` aufrufen, wobei `func` hier für die aufzurufende Funktion steht und natürlich Parameter haben kann. Die Angabe von `this->` kann aber in der Regel auch entfallen.
- Sehr kleine Memberfunktionen (getter/setter - funktionen) können in der Klassendeklaration ausprogrammiert werden. Sie werden dann als Inlinefunktionen übersetzt.

Memberfunktionen/Methoden

```
#ifndef _H_BTIME_  
#define _H_BTIME_
```

```
class clbtime  
{  
public:  
    clbtime();  
    clbtime(int h, int m);  
    static clbtime* create();  
    void setH(int h){if (h>0 && h<24)this->h=h;else h=0;}  
    void setM(int m){if (m>0 && m<59)this->m=m;else m=0;}  
    int getM(){return m;}  
    int getH(){return h;}  
    void getCurrentTime();  
    void show();  
    void increment();  
private:  
    int h;  
    int m;  
};
```

```
#endif
```

Diese Funktionen tauchen dann im Implementationsfile nicht mehr auf

Memberfunktionen/Methoden

- Memberfunktionen sollen so implementiert sein, dass die Daten eines Objektes immer valide Werte beinhalten.
- Memberfunktionen kann man funktional in Verwaltungsfunktionen, Implementierungsfunktionen, Hilfsfunktionen und Zugriffsfunktionen einteilen.
- Gültigkeitsbereiche in Memberfunktionen
 - lokaler Block, in dem der Bezeichner verwendet wird,
 - umfassende Blöcke innerhalb der Funktion, in der der Bezeichner verwendet wird,
 - Klasse, in der die Funktion als Memberfunktion deklariert worden ist,
 - Quelldatei (Filescope),
 - wird ein Bezeichner in einem eingeschlossenen Gültigkeitsbereich erneut vereinbart, so verdeckt diese Vereinbarung die ursprüngliche Vereinbarung.

Der Destructor

- Eine Klasse kann einen Destructor definieren.
- Der Destructor dient dem Abbau eines Objektes, insbesondere dann, wenn es, meist über Pointer, andere Ressourcen bindet oder verwaltet.
- Der Name des Destructors wird aus dem Klassennamen und einer vorgestellten Tilde (~) gebildet, er hat keinen Returntyp, keine Parameter und kann demzufolge auch nicht überladen werden.
- Er wird automatisch aufgerufen, wenn die Lebenszeit eines Objektes, auf welche Weise auch immer, endet.
- Nicht alle Klassen brauchen einen Destructor.

Klasse Fraction

```
#include <cstdlib>
#include <cmath>
#include <iostream>

using namespace std;
class Fraction
{
    public:
    Fraction () { Numerator = 0; Denom = 1; };
    Fraction ( long Num, long Den);
    void display() const;
    Fraction add (const Fraction & Second) const;
    private:
    long gcf(long First, long Second);
    long Numerator,
    Denom;
};
```

Klasse Fraction

```
Fraction Fraction :: add (const Fraction & Second) const
{
    long Factor = gcf (Denom, Second.Denom);
    long Mult1  = Denom / Factor;
    long Mult2  = Second.Denom / Factor;
    long NumRes = Numerator * Mult2 + Second.Numerator * Mult1;
    long DenRes = Denom * Mult2;
    return Fraction(NumRes, DenRes);
}
long Fraction :: gcf (long First, long Second)
{
    if (First < 0) First = -First;
    if (Second < 0) Second = -Second;
    while (Second)
    {
        int tmp = First % Second;
        First   = Second;
        Second  = tmp;
    }
    return First;
}
```

- Ergänzen Sie die weiteren Funktionen
- Brüche sollen in gekürzter Form vorliegen

Referenzen als Returnwert

- Getter sollen keine Referenzen auf Memberdaten zurückgeben.
- Eine get-Funktion, die eine Referenz zurückgibt, eröffnet die Möglichkeit, Daten im Objekt unkontrolliert zu manipulieren.

- Eine get-Funktion der nachfolgenden Form:

```
int& getH(){return h;}
```

ermöglicht nachfolgende Anweisung:

```
bt1.getH()=88;
```

- Lösung: Referenzen auf Memberdaten als Returnwert vermeiden oder const-Referenzen verwenden.

```
./a.out  
time bt1: 88:00  
time bt2: 12:00
```

Const Objekte

- C++ gestattet es, auch von Klassen konstante Objekte zu definieren

```
const clbtime start(12,00);
```

- Für solche Const-Objekte gelten Einschränkungen.
- Da Memberdaten durch Memberfunktionen verändert werden können, dies für const-Objekte aber nicht zulässig ist, verbietet der Compiler den Aufruf aller Memberfunktionen (zunächst).

Const Objekte

- Einzelne Memberfunktionen, die die Memberdaten nicht verändern, können durch ein, dem Funktionskopf nachgestellten `const`, davon ausgenommen werden.
- Der Compiler lässt diese Funktionen dann auch für Aufrufe zu `const`-Objekten zu.
- Der Compiler überprüft aber auch, dass keine Member in diesen Funktionen als LValue verwendet werden.
- Funktionen mit/ohne `const` können bei sonst gleicher Parameterliste überladen werden.

Const Objekte

```
class clbtime
{
public:
    clbtime();
    clbtime(int h, int m);
    void setH(int h){if (h>0 && h<24)this->h=h;}
    void setM(int m){if (m>0 && m<59)this->m=m;}
    int getM()const {return m;}
    int getH()const {return h;}
    void setCurrentTime();
    void show()const;
    void increment();
private:
    int h;
    int m;
};
```

Const Objekte

```
void clbtime::show() const
{
    cout<<setw(2)<<setfill('0')<<this->h
         <<': ' <<setw(2)<<setfill('0')<<this->m;
}
```

- Alle Memberfunktionen, die die Memberdaten nicht modifizieren, sollten mit dem Attribut `const` versehen werden.

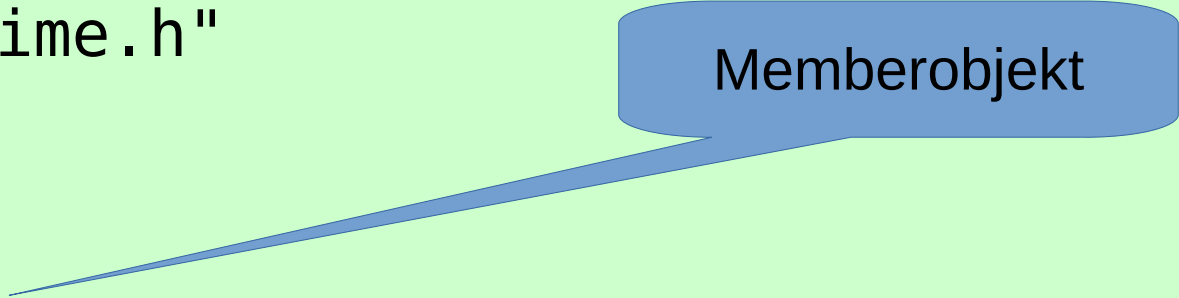
Memberobjekte

- Memberdaten können von beliebigem Datentyp sein, so gesehen auch von einem anderen Klassendatentyp.
- Wird eine solche Klasse instanziiert, so entstehen Memberobjekte, Member, die selbst Objekt(Instanz) einer Klasse sind.
- Wird ein Objekt, das Memberobjekte enthält, instanziiert, so wird für die Memberobjekte der Defaultconstructor (Constructor ohne Parameter) aufgerufen.
- Das ist nicht immer sinnvoll.

Memberobjekte

```
#ifndef _H_DEPART_  
#define _H_DEPART_  
#include "btime.h"
```

```
class depart  
{  
    btime time;  
    char dest[30];  
public:  
    depart(int h, int m, const char* dest);  
    void show();  
};  
#endif
```



Memberobjekt

Memberobjekte

```
#include <iostream>
#include <cstdlib>
#include <cstring>
#include "btime.hpp"
#include "depart.h"
using namespace std;
```

Hiwr wird der
Defaultconstructor von
Btime ausgeführt

```
depart::depart(int h, int m, const char* dest)
{
    strncpy(this->dest, dest, 30);
    time.setH(h);
    time.setM(m);
}
void depart::show()
{
    time.show();
    cout<<" to: " << dest<<endl;
}
```

Memberobjekte

- Soll ein bestimmter Constructor zur Initialisierung eingebetteter Memberobjekte aufgerufen werden, so ist dies über einen MemberInitialisierer möglich.
- MemberInitialisierer werden hinter der schließenden Klammer des Funktionskopfes der Implementierung des Constructors angegeben.
- Mehrere MemberInitialisierer werden durch Komma getrennt nacheinander aufgeführt.
- Die Reihenfolge der Ausführung ist undefiniert.

Memberobjekte

```
#include <iostream>
#include <cstdlib>
#include <cstring>
#include "btime.hpp"
#include "depart.h"
using namespace std;

depart::depart(int h, int m, const char* dest)
                :time(h,m)
{
    strncpy(this->dest,dest,30);
}
void depart::show()
{
    time.btimeShow();
    cout<<" to: "<< dest<<endl;
}
}
```



Memberinitialisierer

Memberobjekte

```
#include <iostream>
#include <cstdlib>
#include "btime.h"
#include "depart.h"

int main(int argc, char* argv[])
{
    depart d[]={ {7,23, "Tharandt"},
                 {7,59, "Hof"},
                 {8,44, "Wien"}};
    for (int i=0; i<3; i++)
        d[i].show();
    return 0;
}
```

static Member

- Statische Member haben Klassenbezug.
- Sie sind durch das attribut static gekennzeichnet.

```
static int count;
```

- Statische Datenmember existieren nur in einer Instanz für alle Objekte der Klasse. Sie müssen explizit in einem Implementationsfile definiert werden:

```
int StaticMember::count=0;
```

- StaticMember ist hier der Klassenname.

static Member

```
#include <iostream>
#include <iomanip>
using namespace std;
#include "StaticMemb.h"

int StaticMember::count=0;

int main()
{
    StaticMember *array[10];

    cout<<"count:"<<StaticMember::getCount()<<endl;

    for (int i=0; i<10; i++)
    {
        array[i]=new StaticMember;
        cout<<"count:"<<StaticMember::getCount()<<endl;
    }
    for (int i=0; i<10; i++)
    {
        delete array[i];
        cout<<"count:"<<StaticMember::getCount()<<endl;
    }
}
```

```
class StaticMember
{
private:
    static int count;
public:
    StaticMember() {count++;}
    ~StaticMember() {count--;}
    static int getCount()
        {return count;}
};
```


static Member

- Statische Methoden haben auch Klassenbezug.
- Typischer Weise werden sie unter Angabe des Klassennamens und des Selektors :: aufgerufen.
- Im Beispiel: **StaticMember::getCount()**
- Ein Aufruf über einen Pointer (Pfeilselektor) oder einer Referenz auf ein Objekt über das Objekt selbst (Punktselektor) ist auch möglich, aber für statische Funktionen eher untypisch.