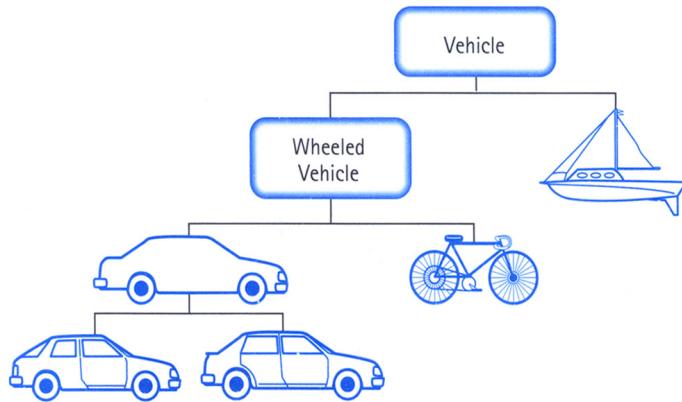


Objekt-Orientierte Programmierung

VORLESUNG 3





Overview

- Speicher
- Über C++
- Grundlagen Objekt Orientierung
- Objekte und Klassen in C++

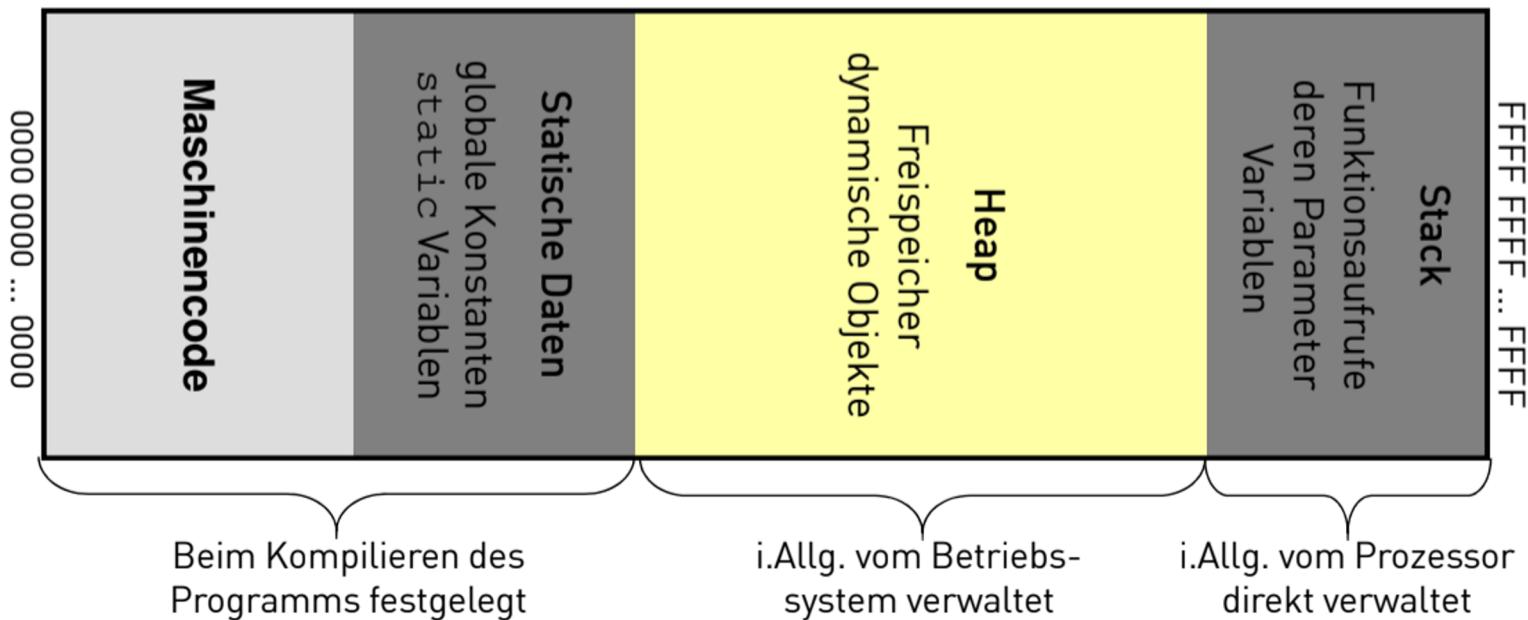


Speicher §
Zeiger



Speicher-Layout

Ein C++ Programm benutzt den Speicher in etwa so





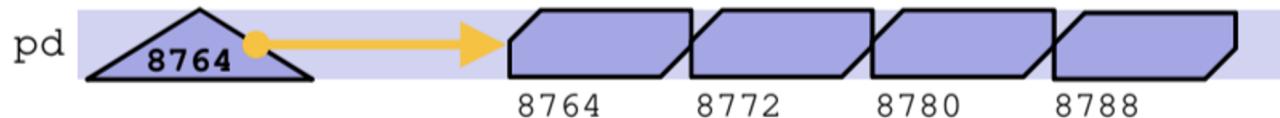
Verschiedene Arten von Speicher

- **statischer Speicher** wird reserviert und besteht so lange, wie das Programm ausgeführt wird
- **Stackspeicher** wird beim Aufruf einer Funktion reserviert und wieder freigegeben, wenn aus der Funktion zurückgekehrt wird (automatisch)
 - der Stackspeicher wird oft durch den Prozessor direkt verwaltet
- **Heapspeicher** kann im Programm durch spezielle Syntax ausdrücklich reserviert und wieder freigegeben werden (dynamisch)
 - der Heap- bzw. Freispeicher wird i. Allg. vom Betriebssystem verwaltet
- Stack und Heap sind oft so organisiert, dass sie an beiden Enden des adressierbaren Speichers liegen und aufeinander zuwachsen
 - wenn sie sich treffen, ist der Speicher erschöpft.

Dynamisch Speicher am Heap reservieren

- In C++ gibt es den Operator `new` zur Anforderung von Speicher im Heap

```
double* pd { new double[4] };
```
- Mit dieser Anweisung fordert das Programm zur Laufzeit vom Betriebssystem
 - im Heap zusammenhängenden Speicherplatz für vier double Werte
 - und die Adresse der ersten double zurückzuliefern
- Zum Zugriff auf diesen Speicher speichern wir dessen Adresse.
- Wir initialisieren also einen double-Zeiger namens `pd` mit der zurückgegebenen Adresse.





Dynamisch Speicher am Heap reservieren

- der Operator `new` gibt einen Zeiger auf das bereitgestellte Objekt zurück
 - ist das zur Verfügung gestellte Objekt vom **Typ T**, so ist der von `new` zurückgegebene Zeiger vom **Typ T***
- der Wert des Zeigers ist die Adresse des (ersten) Speicherbytes
- `new` reserviert mit `[]` mehrere Objekte eines Typs
 - Array von Objekten
 - es wird ein Zeiger auf das erste dieser Objekte zurückgegeben
- die Anzahl der Objekte, die angelegt werden sollen, kann durch eine Variable angegeben werden
- **der Zeiger weiß nicht, auf wie viele Objekte er zeigt**

```
int* pi { new int[4] };  
double* pd { new double[n] };
```
- **der Zeiger weiß aber immer, auf welchen Typ er zeigt**

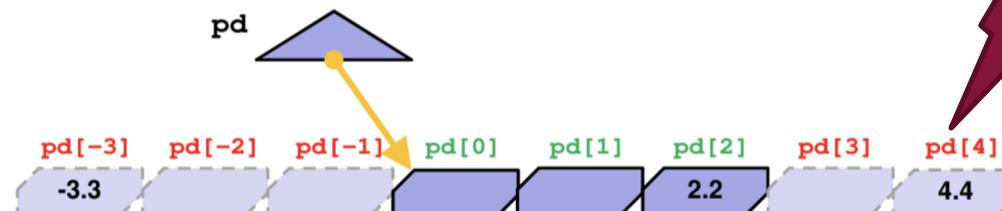
Zugriff über Zeiger

- für Arrays ist `*p` identisch mit `p[0]` (der Indexoperator beginnt bei 0)
- es gibt keinerlei Überprüfung der Zugriffe auf Pointer und der Indexe

```
double* pd { new double[4] };
double x { *pd };      // oder: double x { pd[0] };
double y { pd[2] };    // oder: double y { *(pd+2) };
*pd = 8.8;             // schreibt in das (erste) Objekt, auf das pd zeigt
pd[2] = 9.9;            // schreibt in das dritte Objekt, auf das pd zeigt
pd[3] = 4.4;
double z { pd[3] };
```



`pd[2] = 2.2; * (pd+4) = 4.4; pd[-3] = -3.3;`



Dringend Abgeraten!



Zeiger und Typen

- Zuweisungen zwischen unterschiedlichen Zeigertypen sind nicht erlaubt
- **Verletzung der Typsicherheit**

```
double* pd { new int[4] }; // Fehler, falscher Typ  
int* pi { new double[4] }; // Fehler, falscher Typ
```

- der Inhaltsoperator `*` und der Indexoperator `[]` benötigen die Größe des Elementtyps, um berechnen zu können, wo im Speicher sich ein bestimmtes Element befindet
 - der Wert `pi[2]` liegt beispielsweise im Speicher genau zwei int-Blöcke, d.h. zweimal `sizeof(int)`, hinter dem Wert `pi[0]`
 - der Wert `* (pd+4)` liegt genau vier double-Blöcke, d.h. viermal `sizeof(double)` hinter dem Wert `*pd`



Initialisierung von Arrays

```
#define LEN 50

int ai[] {2,4,6,8};

// int-Datenfeld aus vier int

// die Datenfelder hier liegen im Stack (kein new verwendet)

int ai2[LEN] {0,1,2,3};

// die letzten 46 int im Datenfeld sind mit 0 initialisiert

double ad[100] {};

// alle double im Datenfeld sind mit 0.0 initialisiert

ad - 1 + ( sizeof( ad ) / sizeof( *ad ) )

// Adresse des letzten double im Datenfeld
```



Initialisierung von Variablen

```
#include <string>

int n; // n = global/static n = 0

int main() {
    int n;           // non-class: the value is undeterminate
    std::string s;   // calls default constructor,
                      // the value is "" (empty string)
    std::string a[2]; // calls default onstruc,
                      // creates two empty strings "", ""
    int m = n;       // undefined behavior
}
```





Initialisierung von Zeigern

```
double* p; // nichts ist initialisiert  
*p = 7.5; // grober Fehler
```



```
double* p1 { new double }; // Zeiger OK, Zielwert nicht  
double* p2 { new double{5.5} }; // Zeiger und Zielwert OK  
double* p3 { new double[5] }; // Zeiger OK, Zielwerte nicht  
double* p4 { new double[10]{} } // Zeiger und Zielwerte OK  
  
double* pd { new double[5] { 12.4, 5.3, 15.2, 8.1, 24.0 } };
```



Speicher im Heap freigeben

- mit `new` erzeugte Objekte unterliegen nicht den Gültigkeitsregeln, sie existieren so lange, bis sie wieder gelöscht werden (Operation `delete`)
- `delete` wird auf die Zeiger angewendet, die von `new` zurückgeliefert wurden und gibt den reservierten Speicher wieder frei
- es gibt zwei Varianten, es muss die jeweils richtige verwendet werden
 - `delete` gibt den mit `new` reservierten Speicher für ein einzelnes Objekt frei
 - `delete []` gibt den mit `new` reservierten Speicher für ein Array von Objekten frei
- der Compiler prüft nicht, ob die richtige Variante verwendet wurde
- der Compiler prüft nicht, ob der Pointer nach Freigabe noch benutzt wird





Memory Errors



- Ungültiger Speicherzugriff
 - Zugriff auf nicht initialisierten oder freigegebener Speicher
- Speicherlecks
 - Speicher wird reserviert, aber niemals freigegeben
 - Visual Studio: <crtDBG.h> und CrtDumpMemoryLeaks();
- nicht übereinstimmende Reservieren-Freigabe
 - alloc - free (wie in C)
 - new - delete
- Freigeben von Speicher, der nie reserviert wurde
- Wiederholte Freigabe
 - Freigeben von Speicher, der bereits freigegeben wurde

Dangling pointer

- ein Zeiger, der nicht auf gültige Daten verweist
 - die Daten wurden möglicherweise freigegeben
 - der Speicher, auf den verwiesen wird, enthält undefinierten Inhalt
- Dereferenzierung eines solchen Zeigers führt zu undefiniertem Verhalten!

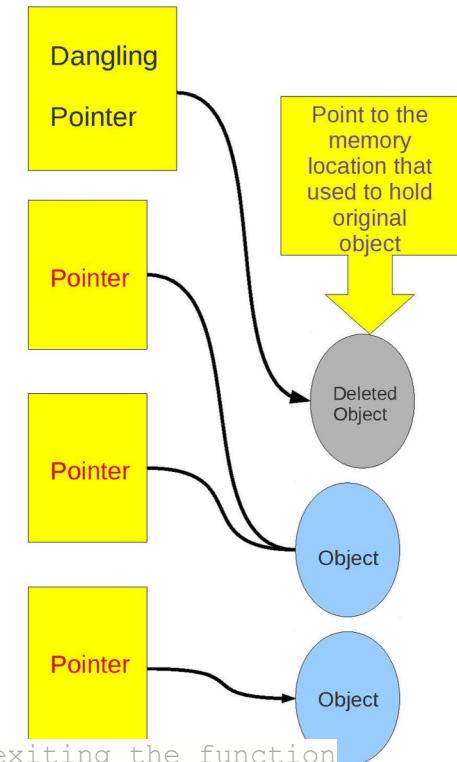
```

Class *object = new Class();
Class *object2 = object;

delete object;
object = nullptr;
// now object2 points to something which is not valid anymore

Object *method() {
    Object object;
    return &object;
} // a method in a class

Object *object2 = method();
// object2 points to an object which has been removed from stack after exiting the function
  
```





Fine!



über C++



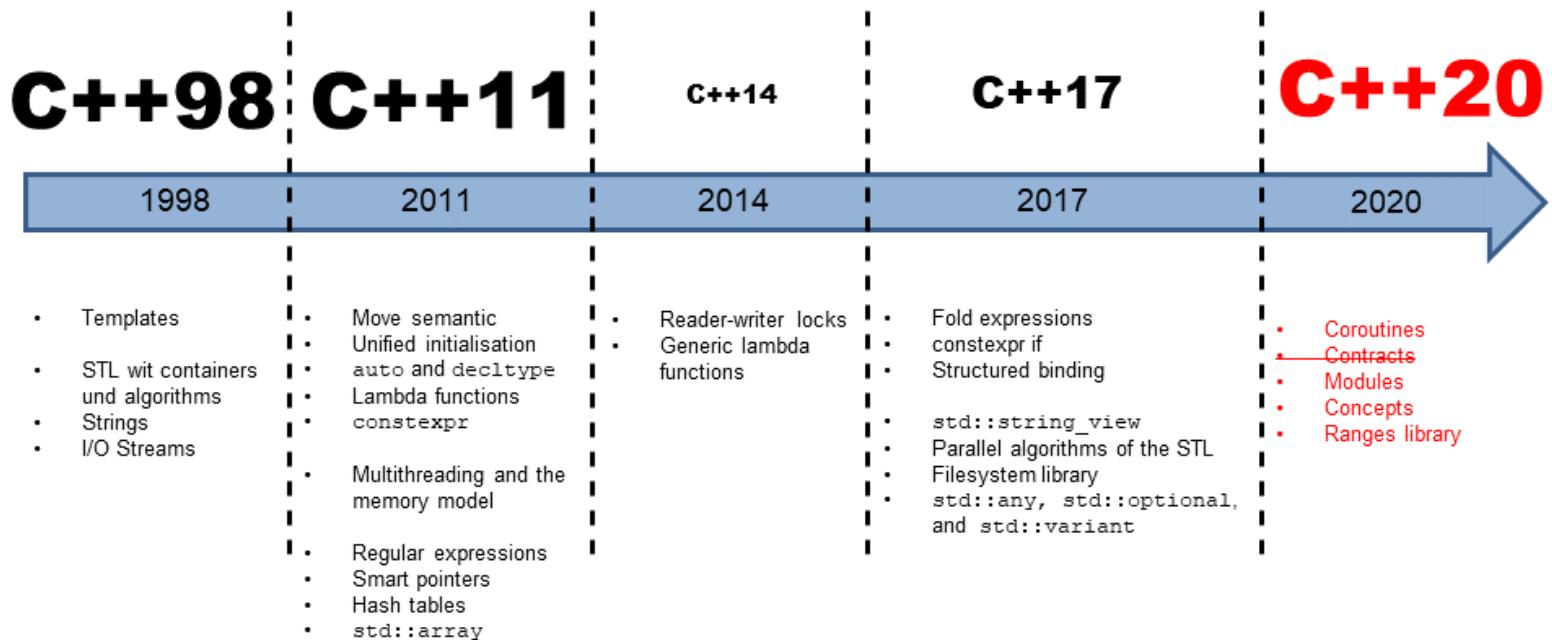
Unterschiede zu C - wesentliche Unterschiede

- C++ ist eine Erweiterung von C
- Abwärtskompatibilität, nahtlose Integration
- Elegante Erweiterung des Typsystems
- Objektorientierung Sprache (Klassen)
- Generische und generative Programmierung
(Templates)



Standards

- Important: compatibility of different Compilers
- Standard C++ Foundation <https://isocpp.org/>





Code Guidelines

- C++ Core Guidelines, initiative led by Bjarne Stroustrup, the inventor of C++, and Herb Sutter, the convener and chair of the C++ ISO Working Group
- The main aim is to efficiently and consistently write type and resource safe C++, using best practices for the language standards C++14 and newer
- to help developers of compilers and static checking tools to create rules for catching bad programming practices.

Objektorientierte Programmierung

I





Hintergrund und Motivation

- Vor OOP war prozedurale Programmierung
 - Code (Anweisungen, Funktionen)
 - Daten (Strukturen)

→ sind meist eng gekoppelt!
- Menschliches Verständnis von der Welt unterscheidet Objekte und Prozesse
 - Objekte (physisch oder informationell)
 - Prozesse verändern/verwandeln Objekte



OOP Grundlagen I

Software dient zur Lösung einer Aufgabe/Problem

- Entwicklern verwenden Kategorien des Problemraums
- das Problem wird in eine Menge von Objekten zerlegt
- die Wirklichkeit wird modelliert
 - Objekte werden abstrahiert in Klassen
 - Prozesse werden abstrahiert in Methoden / Klassen
- Die Lösung des Problem besteht aus der geordneten Interaktion der Objekte



OOP Grundlagen II

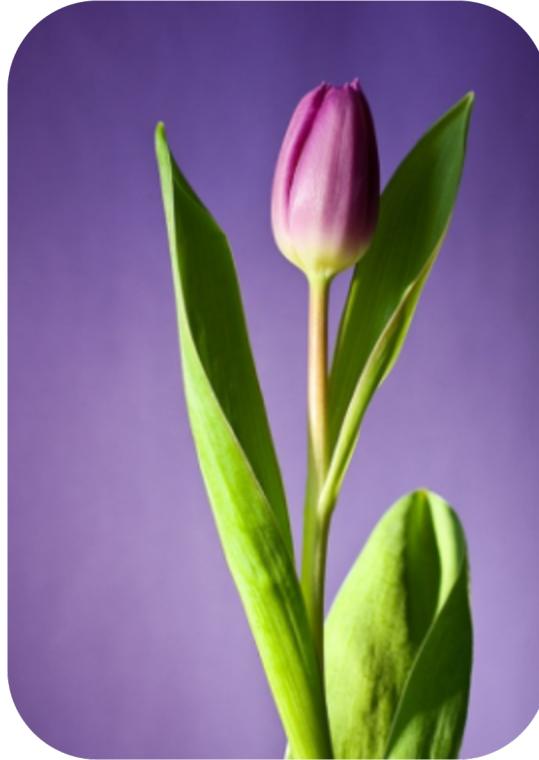
Softwareentwicklung löst eine Aufgabe

- es werden **Objekte** identifiziert
- die **Attribute** der Objekte werden definiert
- das Verhalten der Objekte wird beschrieben, also wie Objekte interagieren (**Methoden**)

Vorteile

- Programm erhält kompakte Struktur (**Modularity**)
- Es ergeben sich Teile die wiederverwendbar sind (**Reuse**)

Objekte in der realen Welt



Objekt = Eigenschaften + Verhalten

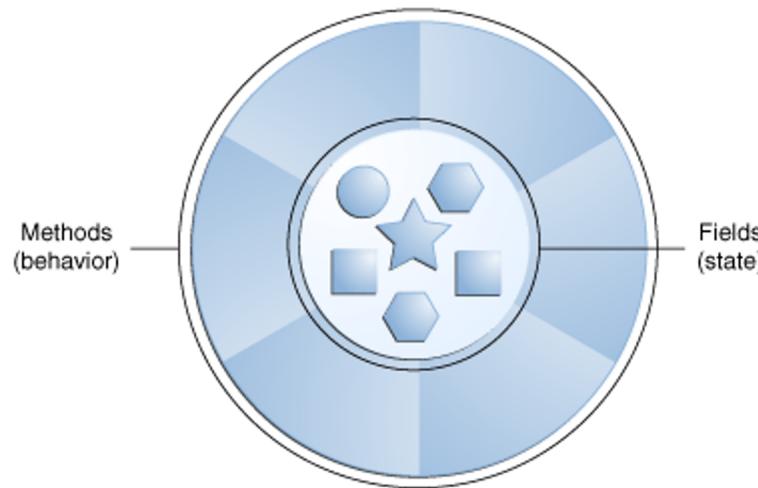


OOP Grundbegriffe

- **Kapselung:** Gruppierung von Daten und Funktionen als Objekte.
- **Abstraktion:** Klassen für Typen von ähnlichen Objekten. Trennung der Spezifikation eines Objekts von seiner Implementierung
- **Vererbung:** Erlaubt Code zwischen verwandten Typen wiederzuverwenden
- **Polymorphismus:** Ein Objekt kann einer von mehreren Typen sein. Abhängig von seinem Typ wird seinem Verhalten zur Laufzeit bestimmt

Software Objekte

- Abbild von Objekten der realen Welt
- **der Zustand** - wird in Feldern (**Daten/Attribute**) gespeichert;
- **das Verhalten** - wird durch **Methoden** umgesetzt.



Beispiel - 2D Vector

- Attribute
 - x-Wert
 - y-Wert
- Verhalten
 - addieren
 - multiplizieren
 - subtrahieren
 - drehen

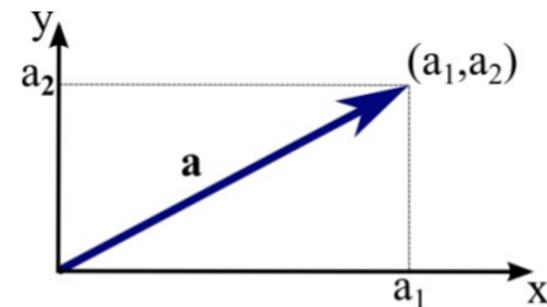


Figure source: http://mathinsight.org/vectors_cartesian_coordinates_2d_3d



Objekte und Klassen in C++



Klassen-Deklaration (in Headerdatei)

```
class Vector2D
{
public:
    double xCoordinate;
    double yCoordinate;

    /*
        Rotate the given vector by the given angle (in degrees).
    */
    void rotate (double angle);

    /*
        Multiplies the 2D Vector with the given scalar.
    */
    void multiplyByScalar (double scalarValue);
...
};
```



Unterschied zu Python

```
class Vector2D:  
  
    def __init__(self):  
        self.__xCoordinate = 0.0  
        self.__yCoordinate = 0.0
```

```
class Vector2D  
{  
public:  
    double xCoordinate;  
    double yCoordinate = 0.0  
  
    /*  
     * Rotate the given vector by the given angle (in degrees).  
     */  
    void rotate (double angle);  
  
    /*  
     * Multiplies the 2D Vector with the given scalar.  
     */  
    void multiplyByScalar (double scalarValue);  
...  
}
```

```
def rotate (self, angle):  
    pass  
  
def multiplyByScalar (self, scalarValue):  
    pass
```



Methoden-Implementierung

(in einer separaten CPP-Datei)

- man verwendet den **Scope-Operator** `::`, um anzugeben, dass die Funktion die Methode einer Klasse implementiert

```
#include "Vector2D.h"
#include <cmath>

void Vector2D :: add(Vector2D v)
{
    xCoordinate += v.xCoordinate;
    yCoordinate += v.yCoordinate;
}

void Vector2D :: rotate(double angle)
{
    xCoordinate = xCoordinate * cos(angle) - yCoordinate * sin(angle);
    yCoordinate = xCoordinate * sin(angle) + yCoordinate * cos(angle);
}
```



Methoden-Implementierung

- Methoden können auch in der Klassendeklaration (Headerdatei) implementiert werden
- dies ist gut geeignet für **Inline-Methoden**
 - solche ersetzt der Compiler durch den tatsächlichen Code der Funktion (daher performanter in der Ausführung)
 - am besten für kurze Funktionen geeignet
- Die Headerdatei dient als Definition der Schnittstelle der Klasse
 - Sollte sich daher seltener ändern
 - Änderungen machen eine erneute Kompilierung aller Verwender/Aufrufer notwendig



Beispiel

```
class Punkt {  
    private:  
        short x; short y;  
    public:  
        short get_x() { return x; }  
        short get_y() { return y; }  
        void set(short sx, sy) { x = sx; y = sy; }  
};  
  
int main () {  
    int x = 1;  
    Punkt P;  
    P.set(x, 5);  
    P.set(P.get_x(), 123);  
    return(0);  
}
```



Attribute

Attribute haben

- einen Namen
- einen Typ
- einen Zugriffsmodifikator

```
private: std::string farbe;
```



Attribute

- Zugriffsmodifikatoren definieren, von wo und wie die Attribute und Methoden einer Klasse zugegriffen werden können
- `public` Felder/Methoden können von überall zugegriffen werden
- `private` Felder/Methoden können nur innerhalb der Klasse (und über Friend-Funktionen) zugegriffen werden
- der Standard Zugriffsmodus für Klassen ist `private`
 - `public` für `struct`



Attribute

- getters
- setters
 - Attribute sollten nur über getter- und setter-Methoden zugänglich sein
- `this` - ein Zeiger auf die aktuelle Instanz
- dieser Zeiger ist **implizit** definiert und in jeder Methode **vorhanden**

```
double getXCoordinate() { this->xCoordinate; }
```
- Hilfreich wenn ein Methodenparameter den selben Namen hat wie ein Klassenfeld hat

```
double setXCoordinate(double xCoordinate)  
{ this->xCoordinate = xCoordinate; }
```

const Methoden

- Methoden können als const deklariert werden
- solche Methode darf die Daten der Klasse nur lesen
- der Compiler kontrolliert dies!

```
class Complex{  
  
public:  
  
    double real() const {return re;} // ok, re wird kopiert  
    const double& real() const {return re;} // ok, nur konstante Referenz  
    double& real() const {return re;} // Fehler: re könnte verändert  
  
    void addToReal(double x) const {re += x;} // Fehler: re verändert  
  
private:  
  
    double re, im;  
  
};
```



Konstruktor

- Der Konstruktor ist eine spezielle Methode, die aufgerufen wird wenn eine Instanz einer Klasse angelegt wird
- Konstruktoren tragen den Namen der Klasse
- Wird vom Benutzer keine eigene Implementierung vorgenommen, dann wird automatisch ein Standardkonstruktor erzeugt

```
class MyClass {  
    public:  
        // Standardkonstruktor  
        MyClass() {  
            // wird beim Anlegen eines Objekts aufgerufen  
        }  
};
```



Konstruktor

- eine Klasse kann einen, keinen oder auch mehrere Konstruktoren haben
- gibt es mehrere Konstruktoren, so müssen sich diese anhand der Funktion-Signatur unterscheiden
- im Konstruktor werden dann alle notwendigen Initialisierungen durchgeführt. Das aufrufende Programm kann diesen Prozess durch Parameter steuern.



Standardkonstruktor

- Ein Standardkonstruktor
 - kann ohne Argumente aufgerufen werden;
 - hat keine Argumente oder
 - alle Argumente sind implizit
- **beim Erstellen eines Arrays von Objekten wird für jedes Element der Standardkonstruktor aufgerufen**
- Benutzerdefinierter Konstruktor vorhanden...
 - Ja. Der Compiler verwendet diesen.
 - Nein. Der Compiler definiert den Standardkonstruktor



Kopierkonstruktor

- Kopierkonstruktoren werden aufgerufen, wenn eine Kopie des aktuellen Objekts benötigt wird
 - bei der Zuweisung einer Klasseninstanz zu einer anderen
 - beim Übergeben von Objekten als Argumente (**pass by value**)
 - wenn eine Funktion ein Objekt zurückgibt
- der Eingabeparameter muss eine (const) Referenz auf ein Objekt des gleichen Typs sein

```
Vector2D( const Vector2D& v );
```



Kopierkonstruktor

- der Compiler generiert automatisch einen Kopierkonstruktor, wenn keiner definiert wurde
- der automatisch generierte Kopierkonstruktor kopiert alle Attribute des Originals in das neue Objekt (**bytewise copy**)
 - Zeiger werden kopiert, nicht aber die referenzierten Objekte **shallow copy**)
- Wenn die Klasse Zeiger hat die dynamischen Speicher verweisen, muss explizit ein Kopierkonstruktor erstellt werden.



Kopierkonstruktor Beispiel 1

```
class T {  
private:  
    int* p;  
  
public:  
    int* get_p() const{ //getter. NEEDS CONST  
        return p;  
    }  
  
    T () { //constructor  
        p = new int;  
        *p = 0;  
  
        cout << "ctor" << endl;  
    }  
  
    ~T () { //destructor  
        delete p;  
        cout << "dctor" << endl;  
    }  
}
```



Kopierkonstruktor Beispiel 2

```
T (const T& t) { //copy constructor
    p = new int;
    *p = *(t.get_p());
    cout << "copy ctor" << endl;
}

T& operator = (const T& t) { //assignment operator
    if (this == &t)
        return *this; //self assignment

    if (p != NULL)
        delete p;

    p = new int;
    *p = *(t.get_p());

    cout << "ass oper" << endl;
    return *this;
}

};
```



Kopierkonstruktor Beispiel 3

```
int main() {
    T t; //constructor

    T t1(t); //copy constructor

    T t2 = t1; //copy constructor

    T t3;
    t3 = t1; //assignment operator

    std::cout << *(t3.get_p()) << endl;
    return 0;
}
```



Destruktor

- neben einem Konstruktor kann eine Klasse einen (und nur einen!) Destruktor haben
- er ist parameterlose Funktion, die wie der Konstruktor keinen Rückgabewert haben
- als Name dient der Klassennamen, mit einer vorangestellten Tilde ~
- er dient dazu, zur Beseitigung des Objekts anfallende Aufräumarbeiten zu erledigen (Speicherfreigabe)



Konstruktoren und Destruktor

Konstruktoren werden aufgerufen

- wenn eine neue Variable auf dem Stack deklariert wird
- wenn wir die Instanz mit `new` (auf dem Heap) erstellen
- wenn eine Kopie der Instanz erforderlich ist (Kopierkonstruktor):
 - Zuweisung
 - Funktionsargumente Übergabe
 - pass by value
 - Zurückgeben eines Objekts in Funktionen

Der Destruktor wird aufgerufen:

- wenn `delete` verwendet wird, um den Speicher freizugeben
- wenn eine auf dem Stack definierte Instanz den Scope verlässt



Rule of Three

man muss immer definieren

- Kopierkonstruktor
- Assignment Operator
- Destruktor

http://en.cppreference.com/w/cpp/language/rule_of_three

Für Move Semantik: **Rule of Five**



Statische Elemente I

Statische Attribute (Klassenattribute)

- statische Attribute gehören zur Klasse
- sie repräsentieren keinen Objektzustand
- sie sind "global" für alle Objekte der Klasse. D.h. sie werden von allen Objekten geteilt
- Zugriff auf solche Variable erfolgt mit den **Klassenname** und dem **Scope-Operator** (::)



Statische Elemente II

Statische Methoden (Klassenmethoden)

- gehören zur Klasse
- keine vorhandene Instanz der Klasse ist erforderlich
- können nur auf andere statische **Attribute/Methoden** bzw. Funktionen außerhalb der Klasse zugreifen
- haben keinen Zugriff auf den **this** Zeiger
- Zugriff auf solche Methoden erfolgt mit den **Klassenname** und dem **Scope-Operator (::)**

Friend Elemente I

- Friend-Funktionen werden verwendet, wenn man einer Funktion Zugriff auf **private** und **protected** Attribute einer Klasse gewähren möchte.
- In diesem Fall muss der Prototyp der Funktion zusammen mit dem Schlüsselwort **friend** innerhalb der Klasse definiert sein.

```
class Vector2D
{
// ...
public:
    // ...

    // friend function
    friend void printVectorData(const Vector2D& v);
};
```



Friend Elemente II

Eine Klasse kann auch ein Freund einer anderen Klasse sein:

die gesamte Klasse und alle seine Methoden sind Freunde der ersten Klasse

```
class Vector2D
{
// ...
public:
    // ...

    // friend class
    friend class Graphics;
};

class Graphics
{
    // ...
};
```



Fragen und Antworten