C++

Grundlagen und Objektorientierte Programmierung

Matthias Pospiech

IQO

1 Grundlagen
- Variablen, Referenzen und Pointer
- Dateiaufbau
- Programmierstil

2 Klassen
- vector Klasse als Beispiel
- Aufbau einer Klasse
- Nutzung von Klassen
- Member Funktionen
- Beispiele
- Ableiten von Klassen
- Funktionen überschreiben
- Beispiele

Nomenklatur

- Pointer Zeigt auf den Speicherort einer Variable
- Referenz Aliasname für eine bereits existierende Variable
- Instanz Erzeugtes Klassenobject
  (Synonym: Objekt, Implementierung)
- Definition Deklaration der Aufrufweise einer Funktion/Klasse
- Deklaration Inhalt der Funktionen/Klassen
- Member (einer Klasse) Funktion die zu einer Klasse gehört
- Stack Arbeitsspeicher der zum Programm gehört
- Heap freier Arbeitsspeicher

Datentypen

Eingebaute Typen

- `bool` (true, false)
- `char` (einzelnes 8 bit ASCII Zeichen)
- `int` ($-2.147.483.648$ - 2.147.483.647)
- `long` ($-9.223.372.036.854.775.808$ - 9.223.372.036.854.775.807)
- `float` ($1,5 \cdot 10^{-45}$ - $3,4 \cdot 10^{38}$)
- `double` ($5 \cdot 10^{-324}$ - $1,7 \cdot 10^{308}$)

STL

- `std::string` Zeichenkette (String) speichern
- `std::vector<double>` Dynamisches Datenarray
- `std::complex<double>` Komplexe Zahlen
- ...

sowie beliebige weitere selbst- und automatisch definierte Typen

Heap & Stack

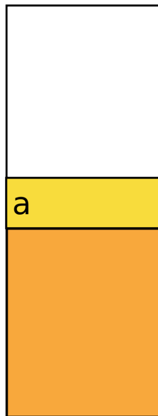
- Stack Arbeitsspeicher der zum Programm gehört
- Heap freier Arbeitsspeicher

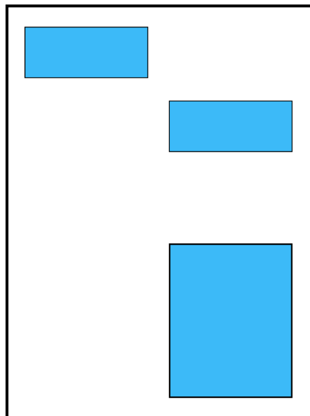
```
void function()
{


}
```

Heap & Stack

- Stack Arbeitsspeicher der zum Programm gehört
- Heap freier Arbeitsspeicher

```
void function()
{
 int a;



}
```

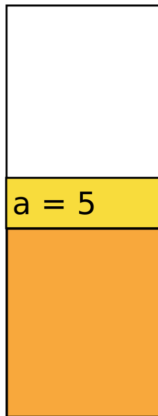


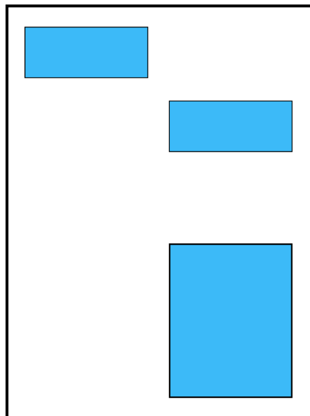


Heap & Stack

- Stack Arbeitsspeicher der zum Programm gehört
- Heap freier Arbeitsspeicher

```
void function()
{
 int a;
 a = 5;


}
```

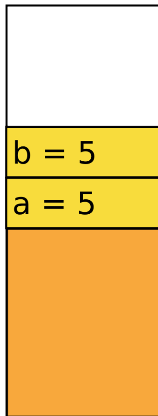


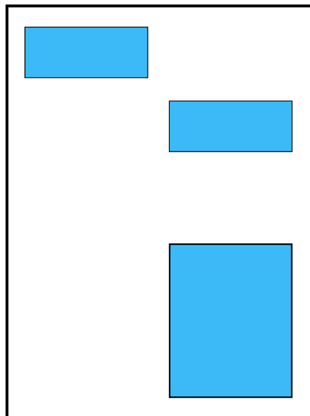


Heap & Stack

- Stack Arbeitsspeicher der zum Programm gehört
- Heap freier Arbeitsspeicher

```
void function()
{
 int a;
 a = 5;
 int b=a;



}
```

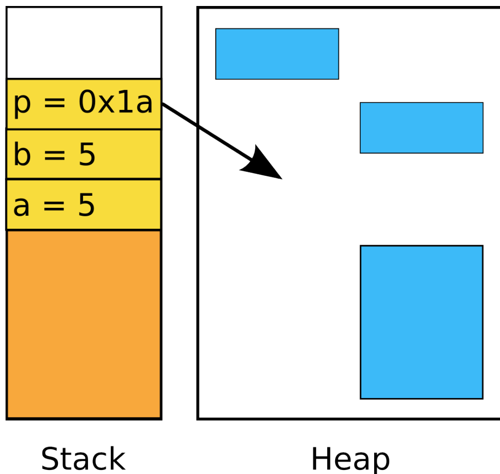




Heap & Stack

- Stack Arbeitsspeicher der zum Programm gehört
- Heap freier Arbeitsspeicher

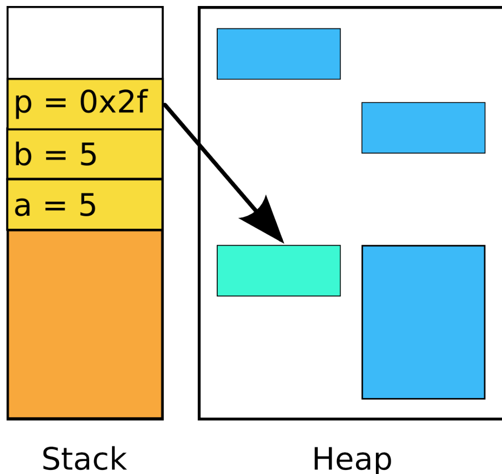
```
void function()
{
 int a;
 a = 5;
 int b=a;
 int * p;



}
```



Heap & Stack

- Stack Arbeitsspeicher der zum Programm gehört
- Heap freier Arbeitsspeicher

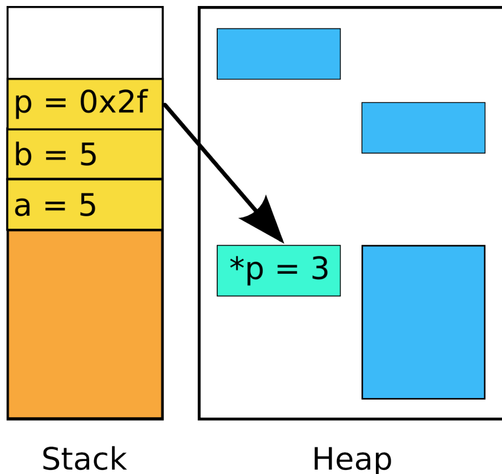
```
void function()
{
 int a;
 a = 5;
 int b=a;
 int * p;
 p = new int;


}
```



Heap & Stack

- Stack Arbeitsspeicher der zum Programm gehört
- Heap freier Arbeitsspeicher

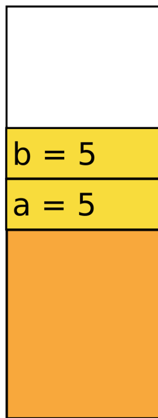
```
void function()
{
 int a;
 a = 5;
 int b=a;
 int * p;
 p = new int;
 *p = 3;

}
```

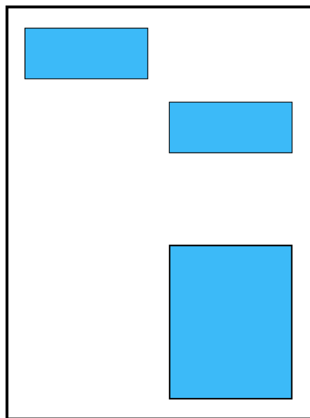


Heap & Stack

- Stack Arbeitsspeicher der zum Programm gehört
- Heap freier Arbeitsspeicher

```
void function()
{
 int a;
 a = 5;
 int b=a;
 int * p;
 p = new int;
 *p = 3;
 delete p;
}
```





Stack Heap

Heap & Stack

Stack

Vorteile:

- automatisch gelöscht

Nachteile:

- sehr begrenzter Speicherplatz

Heap & Stack

Stack

Vorteile:

- automatisch gelöscht

Nachteile:

- sehr begrenzter Speicherplatz

Heap

Vorteile:

- *unbegrenzter* Speicherplatz

Nachteile:

- Speichermanagement notwendig (Freigeben des Speichers)

Deklaration erfolgt über Pointer.

Nutzung bei großen Arrays und großen Klassen.

Variablen, Referenzen und Pointer

Erzeugung einer Variable (Erstellen einer Instanz)

```
1 int a;
2 int b = 5;
```

Variablen, Referenzen und Pointer

Erzeugung einer Variable (Erstellen einer Instanz)

```
int a;
int b = 5;
```

Referenzen

(Alias auf bestehende Objekte)

```
int a;
int & ref = a; // Referenzen müssen auf etwas zeigen
a = 5;         // ref = a = 5
```

Variablen, Referenzen und Pointer

Pointer

```
1 int * a;  // Pointer of Variable vom Type 'int',
2           // zeigt auf keine existente Variable!
3 a = 0;    // Speicherort als ungültig markieren
```

Variablen, Referenzen und Pointer

Pointer

```
int * a;  // Pointer of Variable vom Type 'int',
          // zeigt auf keine existente Variable!
a = 0;    // Speicherort als ungültig markieren

int * b = new int; // Erzeugung eines Speicherortes
                   // und Objektes
*b = 5;   // Inhalt von Pointer b = 5
```

Variablen, Referenzen und Pointer

Pointer

```
1  int * a;  // Pointer of Variable vom Type 'int',
2            // zeigt auf keine existente Variable!
3  a = 0;    // Speicherort als ungültig markieren
4
5  int * b = new int; // Erzeugung eines Speicherortes
6                     // und Objektes
7  *b = 5;   // Inhalt von Pointer b = 5
8
9  a = b;    // Speicherort von a identisch mit b
10 *a = 2;   // *a = *b = 2
```

Variablen, Referenzen und Pointer

Pointer

```
int * a;  // Pointer of Variable vom Type 'int',
          // zeigt auf keine existente Variable!
a = 0;    // Speicherort als ungültig markieren

int * b = new int; // Erzeugung eines Speicherortes
                   // und Objektes
*b = 5;   // Inhalt von Pointer b = 5

a = b;    // Speicherort von a identisch mit b
*a = 2;   // *a = *b = 2

delete b; // Alle mit new erzeugten Variablen müssen
          // mit delete gelöscht werden.
```

Variablen, Referenzen und Pointer

*, & Grammatik

```
int * a;
int b = 5;

a = &b;     // & ermittelt Speicherort
int c = *a; // * ermittelt Wert an Speicherort
```

Die Operatoren * und & wechselt zwischen Deklaration und Zuweisung ihre Bedeutung!

Variablen, Referenzen und Pointer

Beispiel

```
int a;
int& b = a;
int& c = b;
int *d = &c;

a = 6;
*b = 7;
*d = 8;
```

welcher Fehler ist hier eingebaut?

Variablen, Referenzen und Pointer

Beispiel

```
int a;
int& b = a;
int& c = b;
int *d = &c;

a = 6;
*b = 7; // Ungültige Dereferenzierung
*d = 8;
```

welcher Fehler ist hier eingebaut?

Variablen, Referenzen und Pointer

Beispiel

```
int a;
int& b = a;
int& c = b;
int *d = &c;

a = 6;

*d = 8;
```

welche Werte haben a,b,c,d am Ende ?

Variablen, Referenzen und Pointer

Beispiel

```
int a;
int& b = a;
int& c = b;
int *d = &c;

a = 6;

*d = 8; // a=b=c=d=8
```

welche Werte haben a,b,c,d am Ende ?

Variablen, Referenzen und Pointer

Beispiel2

```
1 string s1("foo");
2 string s2("bar");
3
4 string & rs = s1;
5 string *ps  = & s1;
6
7 rs = s2;
8 ps = &s2;
```

welche Werte haben `s1`, `s2`, `*ps`, `rs` ?

Variablen, Referenzen und Pointer

Beispiel2

```
1  string s1("foo");
2  string s2("bar");
3
4  string & rs = s1;   // rs = s1 = "foo"
5  string *ps  = & s1; // *ps = s1 = "foo"
6
7  rs = s2;            // rs = s2 = "bar"
8  ps = &s2;           // *ps = s2 = "bar"
9
10                     // rs = *ps = s2 = "bar"
11                     // s1 = "foo"
```

welche Werte haben `s1`, `s2`, `*ps`, `rs` ?

Kompiliern, Linken, Dateiaufbau

Deklaration/Definition

Für jede Funktion/Klasse wird vor der *Definition* (was zu tun ist) eine *Deklaration* (wie aufzurufen) benötigt:

```
void doSomething(double input, int size); //Deklaration
...
void doSomething(double input, int size)  //Definition
{
...
}
```

Kompiliern, Linken, Dateiaufbau

Aufteilung in .h und .cpp Datei

somefile.h → *Interface*

```
void doSomething(double input, int size); //Deklaration
...
```

somefile.cpp → *Implementierung*

```
#include "somefile.h"
void doSomething(double input, int size)  //Definition
{
...
}
```

Kompiliern, Linken, Dateiaufbau

Beispiel:

mathfunctions.h

```
1 double sqr(double const x);
```

mathfunctions.cpp

```
1 double sqr(double const x) { return x*x; }
```

Kompiliern, Linken, Dateiaufbau

Beispiel:

mathfunctions.h

```
double sqr(double const x);
```

mathfunctions.cpp

```
double sqr(double const x) { return x*x; }
```

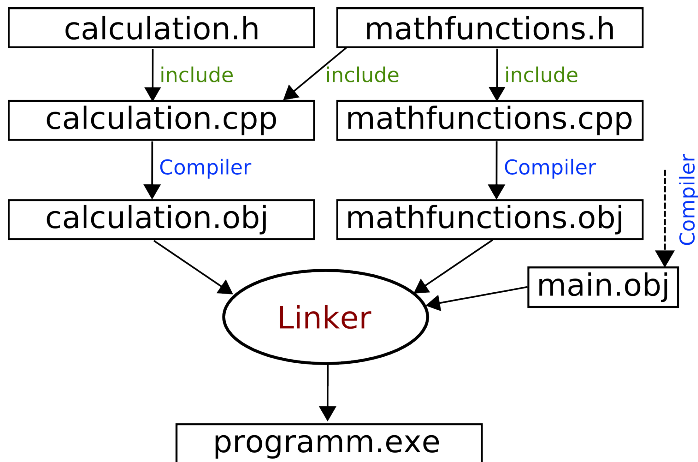
calculation.h

```
double calc(double const x);
```

calculation.cpp

```
#include "calculation.h"    //    Header erst da laden,
#include "mathfunctions.h"  // <- wo er gebraucht wird.
double calc(double const x) { return sqr(3.1414 * x); }
```

Kompiliern, Linken, Dateiaufbau



Kompiliern, Linken, Dateiaufbau

doppeltes Laden verhindern → Anpassen der .h Dateien

mathfunctions.h

```
#ifndef MATHFUNCTIONS_H
#define MATHFUNCTIONS_H

double sqr(double const x);

#endif
```

calculation.h

```
#ifndef CALCULATION_H
#define CALCULATION_H

#include "mathfunctions.h"
double calc(double const x);

#endif
```

Include Dateien

System-Dateien

```
#include <vector>  // C++ Datei
#include <stdio.h> // C Datei!
#include <cstdio>  // C++ Equivalent
```

Syntax: `<...>` - sucht erst in Systemverzeichnissen, dann im Projekt

Projekt-Dateien

```
#include "mathfunctions.h"
#include "DialogShowResults.h"
```

Syntax: `"..."` - sucht im Projektverzeichnis

Guter Programmierstil

- Lesbar - eindeutige Namen vergeben

vergleiche:

```
int w, h;

int N = T/dt;
int Nd = N/SI;
```

Guter Programmierstil

- Lesbar - eindeutige Namen vergeben

vergleiche:

```
int w, h;

int N = T/dt;
int Nd = N/SI;
```

mit

```
int height;
int width;

int maxIterationNumber = totalTime / timeIntervall;
int dataSize = maxIterationNumber / saveIntervall;
```

Guter Programmierstil

- Lesbar - eindeutige Namen vergeben
- Einheitlich - eine Struktur wählen und beibehalten

Programmierstil - *code style conventions*:

- Klassen starten mit Großbuchstaben
- Variablen und Funktionen starten mit Kleinbuchstaben
- neues Wort in einem Variablen- oder Funktionsnamen startet mit Großbuchstaben → *camelCase*
- Abkürzungen vermeiden,
- äußer bei Zählern und temporären Variablen

`i, j, k`	Integer Schleifen Zähler (for Schleifen)
`n, len, length`	Integer Zahlen von einer Auflistung
`x, y`	Kartesische Koordinaten.

Programmierstil

Funktionen:

```
bool isEmpty();
bool hasValues();

void setMaximum(double maximum); // set function
double maximum();                // get function
```

Klassen in C++

ohne Klassen

- vordefinierten Datentypen: int, double usw.

Klassen in C++

ohne Klassen

- vordefinierten Datentypen: int, double usw.

mit Klassen

- vector - speichert Datenarray
- complex - Complexe Zahlen
- matrix - Matrizen als Rechenobjekte
- Laser - Simulation . . .
- Crystal
- Spectrometer - Ansteuerung . . .

Klassen in C++

ohne Klassen

- vordefinierten Datentypen: int, double usw.

mit Klassen

- vector - speichert Datenarray
- complex - Complexe Zahlen
- matrix - Matrizen als Rechenobjekte
- Laser - Simulation . . .
- Crystal
- Spectrometer - Ansteuerung . . .

Klassen enthalten nicht nur **Daten**, sondern auch **Methoden**, um diese zu verarbeiten und weiterzugeben → *Objektorientierte Programmierung*.

Beispiel: vector - Klasse

klassische Datenarray

```
// erstellen
int size = 1000;
double* array = new double[size];
// füllen
for (size_t i=0; i < size; ++i) { array[i] = 2*i; }
// kopieren
memcpy(otherarray, array, size * sizeof(double));
// löschen
delete [] array;
```

Beispiel: vector - Klasse

klassische Datenarray

```
// erstellen
int size = 1000;
double* array = new double[size];
// füllen
for (size_t i=0; i < size; ++i) { array[i] = 2*i; }
// kopieren
memcpy(otherarray, array, size * sizeof(double));
// löschen
delete [] array;
```

Problem:

- Daten (array) und Informationen über Daten (size) sind getrennt.
- kopieren umständlich
- kein automatisches Löschen des Pointers

Beispiel: vector - Klasse

Datenarray als Klasse

```
// erstellen
vector<double> array;
array.resize(1000);

// füllen
for (size_t i=0;
     i < array.size(); ++i)
   { array[i] = 2*i; }

// kopieren
otherarray = array;

// löschen -> automatisch
```

klassische Datenarray

```
// erstellen
int size = 1000;
double* array = new double[1000];

// füllen
for (size_t i=0; i < size; ++i)
   { array[i] = 2*i; }


// kopieren
memcpy(otherarray, array,
   size * sizeof(double));

// löschen
delete [] array;
```

Aufbau einer Klasse

```cpp
// Interface
class Person
{

};
```

Aufbau einer Klasse

```
// Interface
class Person
{
public: // <- für alle
    zugänglich
   Person();  // Konstruktor
   ~Person(); // Destruktor
};
```

```
// Implementierung
Person::Person()  { }
Person::~Person() { }
```

Aufbau einer Klasse

```
// Interface
class Person
{
public:
   Person();  // Konstruktor
   ~Person(); // Destruktor

// Zugriff nur aus der Klasse
private:
   string m_name;
};
```

```
// Implementierung
Person::Person()
{
   m_name = "";
}
Person::~Person() { }
```

Aufbau einer Klasse

```cpp
// Interface
class Person
{
public:
   Person();  // Konstruktor
   ~Person(); // Destruktor
   void setName(string name);
   string name();
private:
   string m_name;
};
```

```cpp
// Implementierung
Person::Person()
{
   m_name = "";
}
Person::~Person() { }

// set Funktion
void Person::setName(string name)
{
   m_name = name;
}
// get Funktion
string Person::name()
{
    return m_name;
}
```

Aufbau einer Klasse

```
// Interface
class Person
{
public:
    Person();  // Konstruktor
    ~Person(); // Destruktor
    void setName(string name);
    string name();
    void setAge(int age);
    int age();
private:
    string m_name;
    int m_age;
};
```

```
// Implementierung
Person::Person()
{
    m_name = "";
    m_age = 0;
}
...
void Person::setAge(age)
{
    if ((age >= 0) && (age < 150))
    {
        m_age = age;
    }
}
int Person::age() {
    return m_age;
}
```

Nutzung von Klassen

Verwenden durch Erstellung einer Instanz.

```
class Person  // Klasse
{
   ...
};

Person child; // Erstellung einer Instanz
```

Eine Klassendeklaration (.h) enthält nur den *Bauplan*.

Nutzung von Klassen

Zugriff auf Klassenmember

```
1 // Erstellung einer Instanz
2 Person child;
3 
4 // Zugriff auf Member über 'Punkt'.
5 child.setName("Paul");
6 child.setAge(30);
7 
8 cout << "Name: "    << child.name()
9      << ", Alter: " << child.age() << endl;
```

Ausgabe

"Name: Paul, Alter: 30"

Nutzung von Klassen

Zugriff auf Klassenmember über Pointer

```
1  // Erstellung einer Instanz
2  Person* child = new Person;
3
4  // Zugriff auf Member über '->'.
5  child->setName("Paul");
6  child->setAge(30);
7
8  cout << "Name: "    << child->name()
9       << ", Alter: " << child->age() << endl;
10
11 delete child;
```

```
1  // Äquivalent zu
2  (*child).setAge(30);
3  ...
4  cout << "Name: " << (*child).name() << ...
```

Nutzung von Klassen

Schützen der Variablen durch ’private’

```
// Interface
class Person
{
public:
...
private:
   int m_age;
};
```

```
child.m_age = -3; // compiliert nicht, da private!
```

Nutzung von Klassen

Schützen der Variablen durch ’private’

```
1 child.m_age = -3; // compiliert nicht, da private!
```

keine public Variablen!

- Zugriff kontrollieren

Nutzung von Klassen

Schützen der Variablen durch ’private’

```
child.m_age = -3; // compiliert nicht, da private!
```

keine public Variablen!

- Zugriff kontrollieren
- Werte auf Gültigkeit überprüfen

```
void Person::setAge(age)
{
   if ((age >= 0) && (age < 150))
   {
      m_age = age;
   }
}
```

Konstruktor

Sicherstellen das Klasse nur gültige Werte enthält

```
class Person
{
public:
   // geänderter Konstruktor:
   Person(string name, int age);
   ~Person();
...
};

Person::Person(string name, int age)
{
   setName(name); // set-Funktionen
   setAge(age);   //   überprüfen Gültigkeit!
}
```

```
Person child("Paul", 30);
```

Initialisierungsliste

Initialisierung durch Zuweisung

```
Person::Person(string name, int age)
{
   m_name = name;
   m_age = age;
}
```

Initialisierungsliste in Konstruktor

```
Person::Person(string name, int age)
  : m_name(name), m_age(age)
  // direkte Zuweisung -> Initialisierungsliste
{
}
```

const Funktionen

```cpp
// Interface
class Person
{
public:
   Person();  // Konstruktor
   ~Person(); // Destruktor
   void setName(string name);
   string name() const;
   void setAge(int age);
   int age() const;
private:
   string m_name;
   int m_age;
};
```

Funktionen die die Daten der Klassen nicht verändern als const deklarieren.

Member Funktionen

Welche Funktionen müssen zur Klasse gehören, welche nicht?

```
complex c(1,-3);    // c = 1 - 3i
// Member Funktion
double r = c.real() // r = 1

// Keine Member Funktion
complex c2 = sqr(c);  // nicht c2 = c.sqr();
```

Member Funktionen

Welche Funktionen müssen zur Klasse gehören, welche nicht?

```
complex c(1,-3);    // c = 1 - 3i
// Member Funktion
double r = c.real() // r = 1

// Keine Member Funktion
complex c2 = sqr(c);  // nicht c2 = c.sqr();
```

```
complex sqr(complex c)
{
   return (sqr(c.real()) + sqr(c.imag()));
}
```

➡ nur wenn direkter Zugriff auf member notwendig.

Beispiele

Welche Member werden benötigt?

Point Size Rect

Welche Methode?

Beispiele

Welche Member werden benötigt?

Point | Size | Rect

- x
- y

Welche Methode?

- `x()`, `setX(int)`
- `y()`, `setY(int)`

Beispiele

Welche Member werden benötigt?

Point

- x
- y

Size

- width
- height

Rect

Welche Methode?

- `x(), setX(int)`
- `y(), setY(int)`

- `width(), setWidth(int)`
- `height(), setHeight(int)`

Beispiele

Welche Member werden benötigt?

Point

- x
- y

Size

- width
- height

Rect

- left
- right
- top
- bottom

Welche Methode?

Point

- `x()`, `setX(int)`
- `y()`, `setY(int)`

Size

- `width()`, `setWidth(int)`
- `height()`, `setHeight(int)`

Rect

- `left()`, `right()`, `top()`, `bottom()` usw.
- `width()`, `height()`, `size()`
- `contains(Point)`

Beispiele

Welche Member werden benötigt?

Point

- x
- y

Size

- width
- height

Rect

- left
- right
- top
- bottom

Welche Methode?

- `x()`, `setX(int)`
- `y()`, `setY(int)`

- `width()`, `setWidth(int)`
- `height()`, `setHeight(int)`

- `left()`, `right()`, `top()`, `bottom()` usw.
- `width()`, `height()`, `size()`
- `contains(Point)`

Beispiel mit Klassen Point, Size und Rect

Ergebnis:

```
Point: x: 10, y: 20
Size: width: 30, height: 40
Rect: (x,y)-(x2,y2): (5, 10)-(15, 25)
Point inside Rect:yes
Center of Rect:(10, 17)
```

Klasse Point

```cpp
#ifndef POINT_H
#define POINT_H

class Point
{
public:
    Point();
    Point(int x, int y);

    int x() const;
    void setX(int x);
    int y() const;
    void setY(int y);

private:
    int m_x;
    int m_y;
};

#endif // POINT_H
```

```cpp
#include "point.h"

Point::Point() {
    setX(0);
    setY(0);
}

Point::Point(int x, int y) {
    setX(x);
    setY(y);
}

int Point::x() const {
    return m_x;
}

void Point::setX(int x) {
    m_x = x;
}

int Point::y() const {
    return m_y;
}

void Point::setY(int y) {
    m_y = y;
}
```

Klasse Size

```cpp
#ifndef SIZE_H
#define SIZE_H

class Size
{
public:
    Size();
    Size(int width, int height);

    void setHeight ( int height );
    void setWidth ( int width );

    int height() const;
    int width() const;

private:
    int m_width;
    int m_height;
};

#endif // SIZE_H
```

```cpp
#include "size.h"

Size::Size() {
    setHeight(0);
    setWidth(0);
}

Size::Size(int width, int height) {
    setHeight(height);
    setWidth(width);
}

void Size::setHeight ( int height ) {
    m_height = height;
}

void Size::setWidth ( int width ) {
    m_width = width;
}

int Size::height() const {
    return m_height;
}

int Size::width() const {
    return m_width;
}
```

Ableiten von Klassen

Beispiel

Angestellter

vorname
nachname

Ableiten von Klassen

Beispiel

Angestellter
vorname
nachname

Abteilungsleiter
vorname
nachname
abteilung
mitarbeiter

Ableiten von Klassen

Beispiel

Angestellter

Angestellter
vorname
nachname

Abteilungsleiter

Abteilungsleiter
Angestellter: vorname, nachname
abteilung
mitarbeiter

Ableiten von Klassen

Beispiel

Angestellter

vorname
nachname

Abteilungsleiter

Angestellter

vorname
nachname

abteilung

mitarbeiter

Beziehung zwischen Klassen:

- Abteilungsleiter ist-ein Angestellter

Ableiten von Klassen

```cpp
class Angestellter
{
public:
    string vorname
    string nachname;
};

class Abteilungsleiter : public Angestellter
{
public:
    string abteilung;
    vector<Angestellter> mitarbeiter;
};
```

Ableiten von Klassen

```
class Angestellter
{
public:
   string vorname
   string nachname;
};

class Abteilungsleiter : public Angestellter
{
public:
   string abteilung;
   vector<Angestellter> mitarbeiter;
};
```

→ ist-ein Beziehung

Ableiten von Klassen

```
class Angestellter
{
public:
    string vorname
    string nachname;
};

class Abteilungsleiter : public Angestellter
{
public:
    string abteilung;
    vector<Angestellter> mitarbeiter;
};
```

→ hat-ein Beziehung

Design von Klassen

ist-ein Beziehung muss vollständig stimmen:

```
class Bird
{
public:
   // birds can fly
   void fly();
};
```

Design von Klassen

ist-ein Beziehung muss vollständig stimmen:

```cpp
class Bird
{
public:
   // birds can fly
   void fly();
};
```

```cpp
class Pinguin: public Bird
{
};

Pinguin pinguin;
pinguin.fly(); // ups
```

Member Funktionen überschreiben

```cpp
class PulsedLaser {
public:
  void printPulseLength() { cout << "none" << endl;  }
};

class UltraShortLaser : public PulsedLaser {
public:
  void printPulseLength() { cout << "10 fs" << endl; }
};

class QSwitchedLaser : public PulsedLaser {
public:
  void printPulseLength() { cout << "5 ns" << endl; }
};
```

Member Funktionen überschreiben

```
class PulsedLaser {
public:
  void printPulseLength() { cout << "none" << endl;  }
};

class UltraShortLaser : public PulsedLaser {
public:
  void printPulseLength() { cout << "10 fs" << endl; }
};

class QSwitchedLaser : public PulsedLaser {
public:
  void printPulseLength() { cout << "5 ns" << endl; }
};

QSwitchedLaser qSwitchedLaser;
qSwitchedLaser.printPulseLength(); // output: 5 ns

UltraShortLaser ultraShortLaser
ultraShortLaser.printPulseLength(); // output: 10 fs
```

Member Funktionen überschreiben - virtual

```
void show(PulsedLaser & laser)
{
  laser.printPulseLength();
}

UltraShortLaser ultraShortLaser
// upcasting UltraShortLaser -> PulsedLaser
show(ultraShortLaser); // output: ??
```

Member Funktionen überschreiben - virtual

```
1 void show(PulsedLaser & laser)
2 {
3   laser.printPulseLength();
4 }
5
6 UltraShortLaser ultraShortLaser
7 // upcasting UltraShortLaser -> PulsedLaser
8 show(ultraShortLaser); // output: none
```

→ Die Funktion von PulsedLaser wird genutzt.

Member Funktionen überschreiben - virtual

```
class PulsedLaser
{
public:
  virtual void printPulseLength() { cout << "none" << endl;  }
};

class UltraShortLaser : public PulsedLaser
{
public:
  // ersetzen (wegen virtual)
  void printPulseLength() { cout << "10 fs" << endl; }
};
```

virtual : Funktion aus Basisklasse kann ersetzt werden.

Member Funktionen überschreiben - virtual

```
void show(PulsedLaser & laser)
{
  laser.printPulseLength();
}

UltraShortLaser ultraShortLaser
show(ultraShortLaser); // output: 10 fs
```

→ Die Funktion von UltraShortLaser wird genutzt.

Member Funktionen überschreiben - virtual

Gültigkeit von virtual

nur bei Pointern und Referenzen

→ Klasse darf nicht kopiert werden!

Member Funktionen überschreiben - virtual

Gültigkeit von virtual

nur bei Pointern und Referenzen

→ Klasse darf nicht kopiert werden!

```
void show(PulsedLaser laser) // keine Referenz oder Pointer
{
  laser.printPulseLength();
}

UltraShortLaser ultraShortLaser
show(ultraShortLaser); // output: none
```

→ Beim kopieren wird UltraShortLaser auf PulsedLaser reduziert!

Abstrakte Klassen - pure virtual

```
class PulsedLaser
{
public:
  // pure virtual function: muss von ableitender
  // Klasse implementiert werden
  virtual void printPulseLength() = 0;
};

UltraShortLaser ultraShortLaser
show(ultraShortLaser); // output: 10 fs
```

Abstrakte Klassen - pure virtual

```
1 class PulsedLaser
2 {
3 public:
4   // pure virtual function: muss von ableitender
5   // Klasse implementiert werden
6   virtual void printPulseLength() = 0;
7 };
8
9 PulsedLaser laser; compiliert nicht, da abstrakte Klasse
```

Initialisierungsliste

```cpp
class Angestellter {
public:
   Angestellter(string Abteilung);
   ...
   string vorname;
   string nachname;
   string abteilung;
};

class Abteilungsleiter : public Angestellter {
public:
   // Konstruktor mit Initialisierungsliste
   Abteilungsleiter(string Abteilung)
     : Angestellter(Abteilung)
   {}
   vector<Angestellter> mitarbeiter;
};
```

→ Konstruktor von abgeleiteten Klassen immer in Initialisierungsliste aufrufen.

Beispiel: ABCD Matrizen

Welches Klassendesign ist sinnvoll zur Berechnung einer Gaußpropagation?

Beispiel: ABCD Matrizen

Welches Klassendesign ist sinnvoll zur Berechnung einer Gaußpropagation?

- Lens → ABCD
- Space → ABCD
- GaussianBeam → LaserBeam
- BeamPropagation (zur Berechnung)
  - GaussianBeam
  - vector<ABCD*>

Beispiel: ABCD Matrizen

```
1  ...
2  #include <Core>
3  USING_PART_OF_NAMESPACE_EIGEN
4  typedef Matrix<double,2,2> MatrixABCD;
5
6  class ABCD
7  {
8  public:
9      // Constructors
10     ABCD();
11     ABCD(double A, double B, double C, double D);
12     // Destructor
13     ~ABCD();
14 ...
15     void setMatrix(const MatrixABCD & matrix);
16     const MatrixABCD matrix() const;
17     const double A() const;
18     const double B() const;
19     const double C() const;
20     const double D() const;
21
22     void print();
23     string toStdString();
24
25     virtual double length();
26
27 };
```

Beispiel: ABCD Matrizen

```
1  #include "abcd.h"
2  ...
3  class ABCDPrivate
4  {
5  public:
6      MatrixABCD matrix;
7  };
8
9  ABCD::ABCD()
10     : d(new ABCDPrivate)
11 {
12     d->matrix.setIdentity();
13 }
14
15 void ABCD::setMatrix(const MatrixABCD & matrix)
16 {
17     d->matrix = matrix;
18 }
19 const MatrixABCD ABCD::matrix() const { return d->matrix;}
20 const double ABCD::A() const { return d->matrix(0,0); }
21 const double ABCD::B() const { return d->matrix(0,1); }
22 const double ABCD::C() const { return d->matrix(1,0); }
23 const double ABCD::D() const { return d->matrix(1,1); }
24
25 double ABCD::length() { return 0; }
```

Beispiel: ABCD Matrizen - Linse

```
1  #include "abcd.h"
2
3  class Lens : public ABCD
4  {
5  public:
6      Lens() { m_FocalLength = 0; }
7      Lens(double focallength) {
8          setFocalLength(focallength);
9      }
10
11     void Lens::setFocalLength(double focallength)
12     {
13         m_FocalLength = focallength;
14
15         MatrixABCD matrix;
16         matrix << 1.0 , 0.0, -1.0/m_FocalLength   , 1.0;
17
18         setMatrix(matrix);
19     }
20     double focalLength(){ return m_FocalLength; }
21
22 private:
23     double m_FocalLength;
24 };
```

Beispiel: ABCD Matrizen - Space?

```
1 #include "abcd.h"
2
3 class Space : public ABCD
4 {
5 ...
6 };
```

Beispiel: ABCD Matrizen - Space?

```
1  #include "abcd.h"
2
3  class Space : public ABCD
4  {
5  public:
6      Space(){}
7      Space(double distance){
8          setDistance(distance);
9      }
10
11     void Space::setDistance(double distance){
12         m_distance = distance;
13
14         MatrixABCD matrix;
15         matrix <<  1.0 , m_distance, 0.0, 1.0;
16
17         setMatrix(matrix);
18     }
19
20     double Space::distance() { return m_distance; }
21     double Space::length()   { return distance(); }
22
23 private:
24     double m_distance;
25 };
```