# C++

Objektorientierte Programmierung mit C++

2021, Tobias Joschko

# Inhalt

- **Vorstellung**
- Einleitung, Geschichte
- Grundlagen
- Klassen
- Vererbung
- Zusätzliche Themen
- Die Standardbibliothek

# Vorstellung

- Ablauf
  - Termine → 22 Stunden → Verteilt auf 8 Termine
  - Kombination Vorlesung + Labor
  - Übungsaufgaben
- Inhalt
  - UML
- Klausur (voraussichtlich 08.12.21)
- Übungsprojekte
  - Klasse Zahl
    - → Vererben an Klasse Bruch oder Komplex
  - ToDo Liste mit Klassen auf der command line
  - Spiel Schere-Stein-Papier oder TicTacToe

# Inhalt

- Vorstellung
- **Einleitung, Geschichte**
- Grundlagen
- Klassen
- Vererbung
- Zusätzliche Themen
- Die Standardbibliothek

# Einleitung, Geschichte

- Prozedural vs. Objektorientiert
- Einkapselung, Polymorphismus, Vererbung
- Statische und dynamische Bindung
- Geschichte von C++
- Literatur

# Einleitung, Geschichte
## Prozedural vs. Objektorientiert

- In traditionellen prozeduralen Programmiersprachen (z.B. C) besteht das Programm aus einer Einheit von Daten und Funktionen (lokal & global).
- Thematische Zusammenfassung von Daten und Funktionen zu Gruppen → Objekte
- Objekte sind für eigene Daten zuständig.
- Interaktion von unterschiedlichen Objekten bilden ein Programm.
- Dies ermöglicht eine verbesserte Strukturierung von (insbesondere komplexen) Programmen.

# Einleitung, Geschichte
## Einkapselung, Polymorphismus, Vererbung

- Einkapselung:
  - Objekt kennt eigene Daten und Operationen (Methoden).
  - Man kann Daten verbergen und nur über Funktionen modifiziert lassen.
  - Programmierer muss die Aufgabe in eine Interaktion von Objekten umwandeln.
- Polymorphismus:
  - Methoden und Operanden, welche dieselbe Aufgabe an unterschiedlichen Objekten ausführen dürfen den gleichen Namen haben. Die steigert die Lesbarkeit von Programmen.
  - → (Bruch) a + (Bruch) b --- (Vektor) a + (Vektor) b
- Vererbung:
  - Hierarchische Verknüpfung von Objekten.
  - Objekte der Unterklasse haben Zugriff auf Methoden der Oberklasse (z.B. Student kann Methoden von Person nutzen).

# Einleitung, Geschichte
## Statische und dynamische Bindung

- Im  Gegensatz zu traditionellen Programmiersprachen (z.B. C) erlaubt C++ die dynamische Bindung von Identifiern an ihre Typen während der Laufzeit und nicht schon vorher durch den Compiler (--> Polymorphismus).
- Diese Zuordnung während der Laufzeit erlaubt dem Programmierer mehr Freiheiten und ermöglicht polymorphe Methoden

```
string A = "aaa"
string B = "bbb"
string C = a + b; // (C = "aaabbb")
```

# Einleitung, Geschichte
## Geschichte von C++

- Bjarne Stroustrup startete mit der Erweiterung für C in den Bell Labs von AT&T in 1979. Zunächst wurde es “C mit Klassen“ genannt.
- Video von Bjarne Stroustrup: "Why I create C++"
- Ca. 1998: erster Iso Standard mit C++98
- 2011: C++11 als große Neuerung mit der Standardbibliothek
- C++14
- C++17
- C++20

# Einleitung, Geschichte
## Literatur

- Bücher:
  - Stroustrup → Siehe homepage
  - Allgemein C++ Bücher aus der Bibliothek
- Online Quellen:
  - http://www.cplusplus.com/doc/tutorial/
  - https://en.cppreference.com/w/
  - http://www.stroustrup.com
  - https://de.wikibooks.org/wiki/C%2B%2B-Programmierung
  - https://www.cprogramming.com/
  - Codecademy Cheatsheet
  - Project Euler
- Sonstiges:

# Inhalt

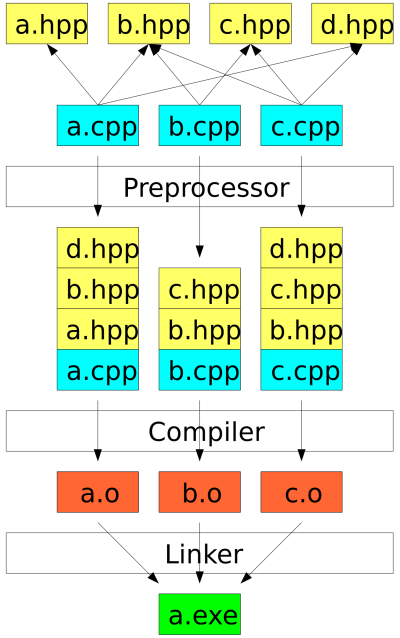
- Vorstellung
- Einleitung, Geschichte
- **Grundlagen**
- Klassen
- Vererbung
- Zusätzliche Themen
- Die Standardbibliothek

# Grundlagen

- Starten eines C++ Programms
- Grundstruktur eines Programms
- IDE
- Ein- / Ausgabe
- Statische Polymorphie
- Referenzen in C++
- Konstanten
- Datentypen
- Aufgaben

# Grundlagen
## Starten eines C++ Programms

- Zwei Arten Programme auf Computern auszuführen:
  - Interpreter Programmiersprachen werden zeilenweise ausgeführt (z.B. Perl, Python, Java).
  - Compiler die Code in Maschinencode Übersetzen und die als eigenständige Programme ausgeführt werden können (z.B. C, Pascal, C++).
- Compiler-Sprachen brauche keine zusätzliche Laufzeitumgebung und sind in der Regel performanter.
- Erstellen eines C++ Programms:
  - ***Sourcecode*** → ***Objektdatei*** → **Maschinencode**
  - g++ beispiel.cpp → a.out (Unix) oder a.exe (Windows)
  - g++ beispiel.cpp -o HelloWorld → HelloWorld oder Helloworld.exe



# Grundlagen
## Allgemeine Programmstruktur

- Ein C++ Programm kann aus unterschiedlichen Funktionen bestehen die auf verschiedene Dateien verteilt sein können, es gibt aber immer nur eine main() Funktionen von derer die anderen (in-)direkt aufgerufen werden.
- Aufbau eines C++ Moduls:

**[ Präprozessor Direktiven ]**
**[ Typ, Klassen - Deklarationen ]**
**[ Definition und Deklaration von Variablen (globale Variablen) ]**
**[ Deklaration von Funktionen (=Prototypen) ]**
**Definition von Funktionen (nur eine Funktion wird main genannt)**

# Grundlagen
## IDE

- Visual Studio Code
- (Eclipse)
- GitHub → Source Code Verwaltung
- Andere IDEs ?

# Grundlagen
## Namespace

- C++ bindet sehr viele Variablen-, Klassen- und Funktionsdefinitionen in include-Files. Dadurch besteht die Gefahr, dass gleiche Namen verwendet werden → Namespace
- Variablen, Klassen, etc. können in C++ in einem Namespace definiert werden.
- Zugriff mit Doppelpunkt ::

```
int x;       // voller Name ::x
Namespace A {
  int x;     // voller Name A::x
}
```

# Grundlagen
## Namespace

- Funktionen und Klassen der Standardbibliothek befinden sich im *Defaultnamespace* std

```
std::string name;
std::cout << "Hallo";
```

```
using namespace std;
string name;
cout << "Hallo";
```

- Nachteil von ‚using namespace std‘: Man holt sich alle Namen in seinen aktuellen Scope. Vielleicht auch welche von denen man nicht weiß → Namenskonflikt

- Andere Möglichkeit:

```
#include <iostream >
using std::cout; // std::cout hereinholen
```

# Grundlagen
## Ein-/Ausgabe

- C++ unterstützt gesamtes I/O-System von C plus eigene Objektorientierte Routinen, die für eigene Klassen erweitert Werden können.
- In C++ arbeitet man mit Streams:

| Stream | Bedeutung | Standardgerät | I/O-Operator |
|---|---|---|---|
| cin | standard input | Tastatur | > > |
| cout | standard output | Bildschirm | << |
| cerr | err | Bildschirm | > > |

- Namespace std: std::cin und std::cout
- Mit Unix Befehlen umlenken:
  - a.out < eingabe.txt
  - a.out > ausgabe.txt

# Grundlagen
## Statische Polymorphie (Funktionsüberladung)

- In C++ können mehrere unterschiedliche Funktionen den gleichen Namen habe → Bedingung: verschiedene Parameter (Anzahl, Typ)

```
void ausgabe (int x){
  cout << "Integer = " << x << endl;
}
void ausgabe (double x) {
  cout << "Double = " << x << endl;
}

int main () {
  ausgabe(2);
  ausgabe(3.141);
}
```

```
Integer = 2
Double = 3.141
```

# Grundlagen
## Referenzen in C++

- Eine Referenz ist ein Synonym (anderer Name, Alias) für eine bereits existierend Variable.

```
int x;      // Definition von x belegt Speicher
int &z = x; // x und z greifen auf die gleicher Speicherstelle
z = 8;      // x hat auch den Wert 8
```

- Unterschied zum Pointer in C:

```
void quadrat (int *x) {
  (*x) = (*x) * (*x);
}
int k = 2;
quadrat (&k);
```

```
void quadrat (int &x) {
  x = x * x;
}
int k = 2;
quadrat (k);
```

# Grundlagen
## Konstanten

- In C++ können Objekte definiert werden, denen ein fester Wert zugewiesen wird, der sich im gesamten Programm nicht ändern darf.
- Objektzugriff wird vom Compiler geprüft und eine Zuweisung ergibt eine Fehlermeldung

```
const int MAX = 10; // in C, #define MAX 10
const double PI = 3.1415927; // #define PI 3.14
Pi += 2; //Fehlermeldung beim Kompilieren
```

# Grundlagen
## Datentypen

| C | C++ | Inhalt |
|---|---|---|
| stdio.h | | printf … |
| | iostream | cout, cin … |
| math.h | cmath | sin, cos, sqrt .. |
| stdlib.h | cstblib | malloc, exit, .. |
| string.h | cstring | strlen, strcpy, … |
| | string | Klasse string |
| | vector | Klasse vector |

- u.v.m

# Grundlagen
## Datentypen

- std::string

```
string str1("Hallo"), str2 = "wie gehts", str3;
str3 = str1;
str3 += str2;
```

- std::vector

```
vector<string> vec1;
vec1.push_back(str1);
vec1.push_back(str2);
vector<string> vec2 (vec1);
cout << " Size von vec2 = " << vec2.size() << endl;
```

# Grundlagen
## Datentypen

- std::map

```
map<string, int> m { {"GPU", 30}, {"SSD", 128} };
m["CPU"] = 20;
```

- Etc → Siehe Standard Bibliothek

# Grundlagen
## Übungsaufgaben

- I/O Nutzen: Zahlen einlesen bis 0
  - Addieren → Summe
  - Als Vector zurück geben
- Funktion Überladen zum quadrieren unterschiedlicher Datentypen
- Telefonbuch mit std::map
  - Vordefinierte map abfragen über die Konsole
  - Map über Konsole definieren und abfragen …

# Inhalt

- Vorstellung
- Einleitung, Geschichte
- Grundlagen
- **Klassen**
- Vererbung
- Zusätzliche Themen
- Die Standardbibliothek

# Klassen

- Klassentypen
- Konstruktor / Destruktor
- Copy und Move Konstruktor
- Überladen von Operatoren
- Statische Variablen und Methoden
- Dynamische Speicherallozierung

# Klassen
## Klassentypen

- Sind Datentypen, die Attribute (Variablen) und Elementfunktion in Objekten zusammenfassen
  - → struct und class
- Zugriffsrechte:
  - public: freier Zugriff von außen erlaub
  - private: Zugriff nur in Methoden der Klasse
  - protected: Zugriff nur in Methoden von verwandten Klassen
- Struct ist standard public
- Class ist standard private
- Definition des structs Komplex:

```
struct Komplex {
   double _re;  // Membervariable
   double _im;
   void ausgabe();   // Methode
};
```

# Klassen
## Klassentypen

- Implementierung der Methode Komplex::ausgabe()
- this: Zeiger auf das Exemplar, dessen Methode ausgeführt wird, man diesen auch weg lassen, dann wird implizit auf die Membervariablen zugeriffen → **Vorsicht mit Namenskonflikten bei Funktionsparamtern!**

```
void Komplex::ausgabe() {
   cout << '(' << _re << ',' << _im << ')' << endl;
}
```

```
void Komplex::ausgabe() {
   cout << '(' << this->_re << ',' << this->_im << ')' << endl;
}
```

# Klassen
## Klassentypen

- Der Datentyp ”Komplex” kann auch als Klasse dargestellt werden. Um von außen darauf zuzugreifen müssen diese als public deklariert werden.
- Typischerweise sind Membervariablen private und deren Methoden public

```
class Komplex {
public:
   double _re;  // Membervariable
   double _im;
   void ausgabe();   // Methode
};
```

```
class Komplex {
   double _re;  // Membervariable
   double _im;
public:
   void ausgabe();   // Methode
};
```

# Klassen
## Konstruktor / Destruktor

- Konstruktor: Spezielle Methode die automatisch ausgeführt wird nachdem eine Instanz/Objekt der Klasse initiert wurde.
  - → Zweck: Initiliasieren der Member

```
cKomplex::cKomplex() {
   _re = _im = 0;
}
```

  - Initiliasierungsliste:

```
cKomplex::cKomplex(): _re(0), _im(0) { }
```

```
class cKomplex {
   double _re;  // Membervariable
   double _im;
public:
   cKomplex();
   cKomplex(int re, int im);
   ~cKomplex();
   void ausgabe();   // Methode
};
```

- Destruktor: Automatischer Aufruf kurz vor Zerstörung.
  - → Zweck: Zusätzlich belegte Resourcen freigeben.

```
cKomplex::~cKomplex() {
   cout << "Bye, bye!" << endl;
}
```

# Klassen
## Copy und Move Konstruktor

- Copy-Konstruktoren
  - sind Konstruktoren die eine Referenz (in der Regel const) auf eine Instanz einer Klasse erwarten

```
cKomplex::cKomplex(const ckomplex& obj);
```

  - kopieren die als Argumente übergebene Instanzen der Klasse in das instanziierte Objekt
  - werden nicht vom Compiler erzeugt, wenn die Klasse einen Move-Konstruktor oder Move- Zuweisungsoperator besitzt

```
cKomplex operator=(const cKomplex& obj);
```

- Der Compiler benötigt den Copy-Konstruktor, um automatisch Objekte zu kopieren oder neue Objekte aus bestehenden zu initialisieren → wird automatisch von Compiler generiert wenn nicht manuell erstellt

# Klassen
## Copy und Move Konstruktor

- Move-Konstruktoren
  - sind Konstruktoren, die eine nicht konstante Rvalue-Referenz auf eine Instanz einer Klasse erwarten.

```
cKomplex(cKomplex&& obj);
```

  - besitzen ein ähnliches Einsatzgebiet wie Copy-Konstruktoren.
- verschieben das ursprüngliche Objekt in das zu instanziierende Objekt.
- Der Move-Konstruktor wird implizit vom Compiler angewandt, um Objekte zu verschieben, die nicht mehr benötigt werden. → Vermeiden von unnötigem Kopieren

# Klassen
## Copy und Move Konstruktor

- Strategie des Move-Konstruktors
  - 1. Setze die Attribute des neuen Objekts.
    2. Verschiebe den Inhalt des alten Objekts in das neue.
    3. Setze die Attribute des alten Objekts auf ihre Default-Werte.
- Der Move-Konstruktor wird
  - automatisch erzeugt, wenn alle Attribute einer Klasse und ihrer Basisklassen einen besitzen.
  - nicht erzeugt, wenn für die Klasse ein Copy-Konstruktor, Copy-Zuweisungsoperator, ein Move- Zuweisungsoperator oder ein Destruktor existiert.
- Besitzt eine Klasse einen Move-Konstruktor und einen Copy-Konstruktor, besitzt der Move- Konstruktor Vorrang.

# Klassen
## Copy und Move Konstruktor

- Besitzt eine Klasse einen Copy-Konstruktor, sollte sie auch einen Copy-Zuweisungsoperator anbieten. Entsprechendes gilt für den Move-Konstruktor und Move- Zuweisungsoperator.
- Eine Klasse unterstützt die Copy-Semantik, wenn sie einen Copy-Konstruktor und einen Copy-Zuweisungsoperator anbietet.
- Eine Klasse unterstützt die Move-Semantik, wenn sie einen Move-Konstruktor und einen Move-Zuweisungsoperator anbietet.
- Die Container der Standard Template Library und der std::string bieten Copy- und Move-Semantik an.
- Weiteres unter folgendem Link

# Klassen
## Überladen von Operatoren

- Überladen von  rationalen und logischen Operatoren
  - → Ergenis der Operation muss ein ganzzahliger *true*- oder *false*-Wert sein

```
int cKomplex::operator==(const cKomplex obj) {
   if (_re == obj._re && _im == obj._im) {return true;}
   else {return false;}
}
```

- Überladen von unären Operatoren

```
++obj;
// Hier wird die Operatorfunktion ohne Parameter definiert.
cKomplex operator++();
// ---
obj++;
// Zur Unterscheidung wird bei der Operatordefinition
// ein fiktives Argument angegeben.
cKomplex operator++(int notUsed);
```

# Klassen
## Überladen von Operatoren

- Überladen von zweistelligen Operatoren (z.B. a + b ) als Klassenmethode:

```
cKomplex cKomplex::operator+(const cKomplex& obj1) {
   cKomplex z;
   z._im = this->_im + obj1._im;
   z._re = this->_re + obj1._re;
   return z;
}
```

- Überladen von zweistelligen Operatoren als “non-member” Funktion:
  - 2 argument für jeden Operator
  - Funktion muss als friend des Klasse cKomplex definiert werden

```
cKomplex operator+(const cKomplex& obj1, const cKomplex& obj2) {
   cKomplex z;
   z._im = obj1._im + obj2._im;
   z._re = obj1._re + obj2._re;
   return z;
}
```

# Klassen
## Überladen von Operatoren

- Überladen vom ostream Operator (“ << ”):

```
ostream& operator<<(ostream& os, const cKomplex obj) {
   os << obj._re << ", " << obj._im;
   return os;
}
```

- Danach muss dieser als “friend” von cKomplex definiert werden um auf die privaten Daten zugreifen zu dürfen:

```
class cKomplex {
   double _re;  // Membervariable
   double _im;
   friend ostream& operator<<(ostream& os, cKomplex obj);
  ....
```

- Webseite mit Information zum Operator Überladen

# Klassen
## Statische Variablen und Methoden

- Statische Membervariablen existieren genau einmal (unabhängig von der Anzahl der Instanzen der Klasse)
- Sind in allen Instanzen der Klasse gleich
- Auch verfügbar, wenn keine Instanz der Klasse existiert
- Statische Variablen müssen initialisiert werden
- Statische Methoden können nur auf statische Variablen Ihrer Klasse zugreifen

```
class Zaehler {
  static int n;
public:
  Zaehler() {n++;}
  ~Zaehler() {n--;}
  static int get_n() {return n;}
};
```

# Klassen
## Dynamische Speicherallozierung

- Der ‚*new*‘ Operator
  - Reserviert Speicherplatz für eine Instanz auf dem Heap (=dynamischer Speicher)
  - Führt den Konstruktor aus und reserviert Speicher beim OS. Nicht mehr benötigter Speicher muss vom Programmierer wieder freigegeben werden. —> **Vorsicht**: **„Memory-Leak“**
  - Als „Memory-Leak“ oder auch Speicherleiche wird das unnötige Vorhandensein von nicht mehr benötigtem Speicher bezeichnet
  - Liefert den Zeiger auf die Instanz, dieser muss in einer Variable gespeichert werden

```
Zaehler *z = new Zaehler;
Zaehler *z2 = new Zaehler;

Zaehler *z_ar = new Zaehler[10];
```

# Klassen
## Dynamische Speicherallozierung

- Der ‚*delete*‘ Operator
  - Analog zu new ruft delete zuerst den Destruktor und gibt reservierten Speicher wieder frei
  - delete erwartet zum Löschen den von new zurückgelieferten Pointer
  - Eine Speicheradresse darf niemals zweimal mit delete gelöscht  werden —> Führt zum Programmabsturz. Zur Sicherheit kann man den Pointer auf 0 setzen.

```
delete z;
delete[] z_ar;
```

# Inhalt

- Vorstellung
- Einleitung, Geschichte
- Grundlagen
- Klassen
- **Vererbung**
- Zusätzliche Themen
- Die Standardbibliothek

# Vererbung

- Einfache Vererbung
- Zugriffskontrolle & Vererbungsart
- Vererbung verbieten
- Aufrufreihenfolge & Vererbung von Konstruktoren
- Mehrfache Vererbung
- Abstrakte Basisklassen

# Vererbung
## Einfache Vererbung

- Person ist die Basisklasse
- Student ist die abgeleitete Klasse
- Jede Instanz von Student kann als Person eingesetzt werden
  - ➔ Jeder Student ist eine Person
- Student erbt alle (public und protected) Eigenschaften (Membervariablen und Methoden) von Person

```
class Person
{
protected: // abgeleite Funktionen haben Zugriff
   string _name;
   int _geburtsjahr;

public:
   Person(string name, int geburtsjahr)
     : _name(name), _geburtsjahr(geburtsjahr) {}
   void ausgabe()
   {
      cout << "Person: " << _name << endl;
   }
};
```

```
class Student : public Person
{
   int _matrikel_nr;

public:
   Student(string n, int j, int m) : Person(n, j)
   {
      _matrikel_nr = m;
   }
   void ausgabe()
   {
      cout << "Student:\nName: " << _name <<
      "\tmatrikel_nr: " << _matrikel_nr << endl;
   }
};
```

# Vererbung
## Zugriffskontrolle & Vererbungsart

- Die Vererbungsart zeigt an, ob beim Vererben der Zugriff auf Elemente der Basisklasse eingeschränkt wird.
  - Sie wird vor dem Namen der Basisklasse angegeben.
  - Wie bei Memberdeklarationen gibt es die Schlüsselwörter public, protected und private (Standard-Vererbungsart).
- „friend“-Beziehungen und private Variablen oder Methode werden nicht vererbt.

```
class Person { /* ... */ };
class Student : public Person { /* ... */ };
class StudentPro : protected Person { /* ... */ };
class StudentPriv : private Person { /* ... */ }; // := class StudentPriv : Person { /*
... */ };
```

# Vererbung
## Zugriffskontrolle & Vererbungsart

| Ist ein Element in Person | public | protected | private |
|---|---|---|---|
| ... wird es in Student | public | protected | nicht übergeben |
| ... wird es in StudentPro | protected | protected | nicht übergeben |
| ... wird es in StudenPriv | private | private | nicht übergeben |

```
class Person { /* ... */ };
class Student : public Person { /* ... */ };
class StudentPro : protected Person { /* ... */ };
class StudentPriv : private Person { /* ... */ }; // := class StudentPriv : Person { /*
... */ };
```

# Vererbung
## Vererbung verbieten

- Das Schlüsselwort final sorgt dafür, dass …
  - von einer Basisklasse nicht geerbt werden kann

```
class NoInheritance final {};

class Derived: NoInheritance {};    // Error !
```

  - eine Klasse der Endpunkt einer Ableitungshierarchie ist

```
class Base {};
class LastClass final: Base {};
class LastLastClass: LastClass {}; // Error !
```

# Vererbung
## Aufrufreihenfolge Konstruktoren

- Immer, wenn ein Objekt einer abgeleiteten Klasse deklariert wird, wird eine Kette von Konstruktoren ausgeführt.
- Dadurch ist gewährleistet, dass jedes Attribut der Ableitungskette initialisiert wird.
- Die Abarbeitung einer Kette von Konstruktoren beginnt mit der Basisklasse und endet mit der Klasse, von der nicht mehr weiter abgeleitet wird.

```
struct A{};
struct B:A{};
struct C:B{};

C c;                    // A -> B -> C
```

# Vererbung
## Vererbung von Konstruktoren

- Durch die using-Deklaration erbt eine Klasse alle Konstruktoren ihrer direkten Basisklasse
- Der Default-Konstruktor, der Copy- und Move-Konstruktor wird nicht vererbt
- Die abgeleitete Klasse erbt alle Charakteristiken des Konstruktors (public, private, etc.)
- Default-Argumente für Paramter eines Basisklassenkonstruktors werden nicht vererbt
- Konstruktoren mit denselben Parametern wie die abgeleitete Klasse, werden nicht vererbt.
- **Achtung**: Das Vererben von Konstruktoren birgt die Gefahr, dass ein Attribut in der erbenden Klasse nicht initialisiert wird

# Vererbung
## Mehrfache Vererbung

- Ähnlich zur einfachen Vererbung, werden die Namen der Basisklassen in einer kommaseparierten Liste angegeben
- Die Mehrfachvererbung folgt den Regeln der Einfachvererbung:
  - Jeder Basisklasse können ihre Zugriffsrechte vorangestellt werden.
  - Klassen besitzen per Default private-; Strukturen public-Zugriffsrecht
- Enthält eine Instanz einer Klasse mehr als eine Instanz einer Basisklasse, ist der Aufruf ihrer Mitglieder mehrdeutig und führt zu einem Fehler ➔ Diamond-Problem
- Mehrdeutige Aufrufe bei Mehrfachvererbung lassen sich lösen, indem dem mehrdeutigen Mitglied der Name der Basisklasse vorangestellt wird.

# Vererbung
## Mehrfache Vererbung: Virtuelle Basisklasse

- Mit einer virtuellen Basisklasse kann das Problem der Mehrfachvererbung behoben werden, bei der Objekte einer abgeleiteten Klasse mehre Instanzen einer Basisklasse besitzen ➔ Diamond-Problem
- Durch die Verwendung des Schlüsselwortes *virtual* bei der Vererbung von einer Basisklasse wird diese virtuell.
- Falls für virtuelle Basisklassen kein Default-Konstruktor verwendet wird, muss dieser explizit in der abgeleiteten Klasse aufgerufen werden.

# Vererbung
## Abstrakte Klassen

- Eine Klasse die über eine oder mehrere virtuelle Methoden verfügt
- Virtuelle Methode:
  - Wird mit virtuell deklariert
  - =0 an die Klassendeklaration gehangen

```
virtual string getGeschlecht() = 0;
```

- Abstrakte Basisklassen Klassen dienen oft als Schnittstelle (Interface) für Klassenhierarchien, da sie konkret vorschreiben was Klassen implementieren müssen

# Vererbung
## Abstrakte Klassen

- Regeln:
  - Eine Klasse, die rein virtuelle Methoden enthält, kann nicht instanziiert werden.
  - Eine abgeleitete Klasse muss Definitionen für alle rein virtuellen Methoden bereitstellen, um instanziiert werden zu können.
  - Eine rein virtuelle Methode kann in einer Klasse definiert werden, die als rein virtuell deklariert ist.
  - Wenn der Destruktor einer Klasse als rein virtuell deklariert ist, muss trotzdem eine Definition dieser Methode in derselben Klasse angegeben werden.
    - ➔ Beliebtes Verfahren in C++, um eine Klasse zur abstrakten Basisklassen zu erklären

# Vererbung
## Übungsprojekt

- Todo Liste als Gruppenarbeit
- GitHub Classroom link

# Inhalt

- Vorstellung
- Einleitung, Geschichte
- Grundlagen
- Klassen
- Vererbung
- **Zusätzliche Themen**
- Die Standardbibliothek

# Zusätzliche Themen

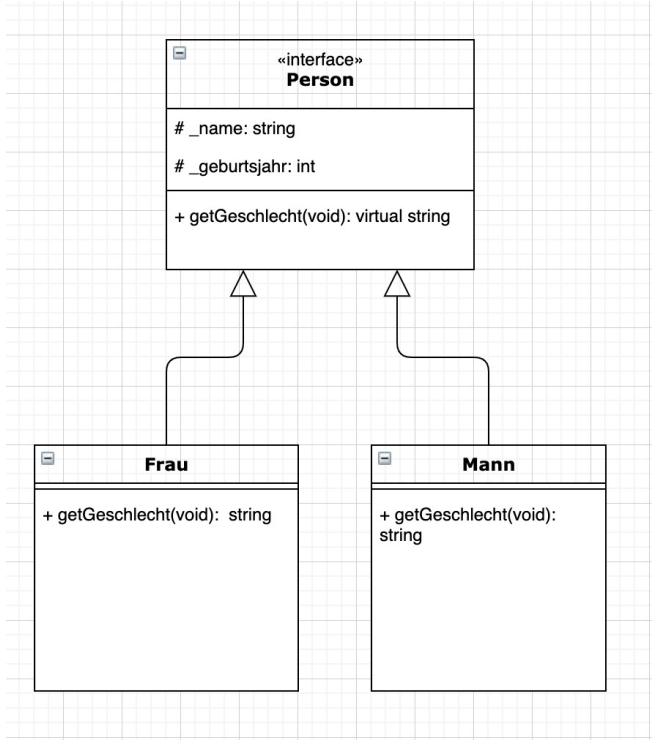
- UML
- Iteratoren – range based loops
- Templates
- Exception handling

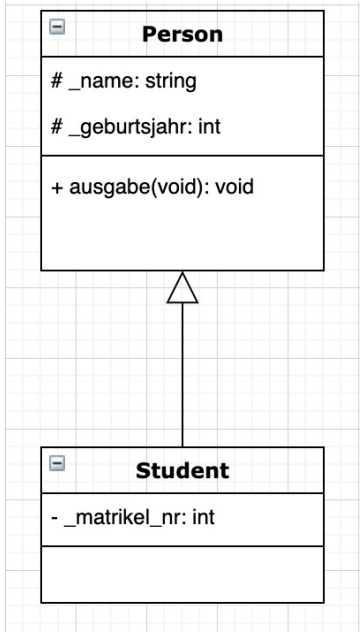
# Zusätzliche Themen
## UML

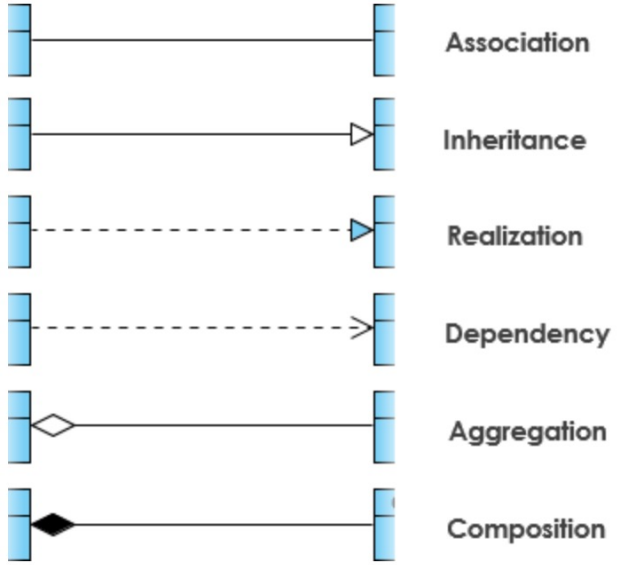
- Die **Unified Modeling Language (UML)** (dt. vereinheitlichte Modellierungssprache) eignet sich dazu Software Strukturen aber auch andere Systeme zu visualisieren, dokumentieren und designen.
- Die Entwicklung von UML begann in den 1990ern und wurde 2005 von ISO standardisiert.
- UML hat 14 Diagrammarten, welche sich grob in zwei Kategorien, Strukturdiagramme (z.B. die Beschreibung SW Klassen) und Verhaltensdiagramme (z.B. Beschreibung von Prozessen oder Programmabläufen), aufteilen lassen.
- Wir werden uns im Rahmen der Vorlesung auf Klassendiagramme zur Beschreibung der Klassenstruktur eines Programms konzentieren.

# Zusätzliche Themen
## UML

- Wiki Klassendiagramm
- Weitere Erklärungen zur Anwendung von Klassendiagrammen



# Zusätzliche Themen
## Funktions-Templates

- Templates dienen dem Compiler als sogenannte "*Kopiervorlagen*".
- Funktions-Templates werden durch die Verwendung der Schlüsselwortes *template* definiert. Darauf folgen Typ- oder Nichttyp-Paramter.
- Die Parameter werden durch die Schlüsselwörter *typename* oder *class* definiert.
- Für den ersten Typ-Parameter ist es üblich den Namen *T* zu verwenden.
- Die Parameter können wie gewohnt in der Funktion verwendet werden:

```
template <typename T>
void tausch(T &a, T &b)
{
   T temp = a;
   a = b;
   b = temp;
}
```

```
template <int N>
int nTimes(int n)
{
   return N * n;
}
```

# Zusätzliche Themen
## Funktions-Templates instanziieren

- Funktions-Templates werden instanziiert indem man die Template-Parameter durch konkrete Werte ersetzt.
- Der Compiler
  - Erzeugt automatisch eine Instanz des Funktions-Templates aufgrund der Argumente
  - Muss die Template-Argumente ableiten können um ein Funktions-Template zu erzeugen.
  - Wenn er die Template-Argumente nicht ableiten kann müssen diese konkret angegeben werden.

```
template <typename T>
void tausch(T &a, T &b)
{ ....

int x = 1, y = 2;
tausch(x,y);
```

```
template <int N>
int nTimes(int n)
{ ...

nTimes<10>(5)
```

# Zusätzliche Themen
## Funktions-Templates Überladen

- Funktions-Templates können überladen werden.
- Es gelten dabei die folgenden Regeln:
  - Templates unterstützen keine automatische Typkonvertierung.
  - Ist eine freie Funktion eine genauso gute oder bessere Wahl wie ein Funktions-Template, wird die freie Funktion vorgezogen.
  - Durch einen Aufruf der Form ***func<type>(...)*** mit einem Template-Argument type wird explizit ein Funktions-Template aufgerufen.
  - Durch einen Aufruf mit leerer Template-Argumentliste ***func<>(...)*** zieht der Compiler nur Funktions-Templates in Betracht.

# Zusätzliche Themen
## Klassen-Templates

- Klassen-Templates werden durch die Verwendung der Schlüsselwortes *template* definiert. Darauf folgen Typ- oder Nichttyp-Paramter.
- Die Parameter werden durch die Schlüsselwörter *typename* oder *class* definiert.
- Die Parameter können wie gewohnt in der Funktion verwendet werden

```
template <typename T, int N>
class Array{
  T el[N];
```

# Zusätzliche Themen
## Klassen-Templates instanziieren

- Klassen-Templates werden instanziiert indem man die Template-Parameter durch konkrete Werte ersetzt.
- Ein Klassen-Template kann im Gegensatz zu einem Funktions-Template seine Argumente nicht automatisch ableiten. ➔ Jedes Template-Argument muss explizit in spitzen Klammern angegeben werden

```
template <typename T>
void tausch(T &a, T &b)
{ ....

int x = 1, y = 2;
tausch(x,y);
```

```
template <typename T, int N>
class Array{
  .....

Array<double, 10> doubleArr;
Array<cKomplex, 10> cKomplexArr;
```

# Zusätzliche Themen
## Klassen-Templates – Methoden-Templates

- Methoden-Templates sind Funktions-Templates, die in Klassen oder Klassen-Templates verwendet werden.
- Methoden-Templates können innerhalb oder außerhalb der Klasse definiert werden.

# Zusätzliche Themen
## Iteratoren

- Werden verwendet, um die Elemente eines Containers der Reihe nach zu durchlaufen.
- Werden als Template-Klasse implementiert.
- Verhalten sich ähnlich zu Zeigern:
  - Dateninhalt ausgeben: *p
  - Zeiger auf nächstes Element verschieben: ++p
- Algorithmus zur Ausgabe des Inhalts eines Container auf die Konsole:

```
template <class P>
void ausgabe(P a_beginn, P a_end) {
   for (; a_beginn != a_end; ++a_beginn)
   {
      std::cout << *a_beginn << std::endl;
   }
}
```

# Zusätzliche Themen
## Iteratoren

- Die Container-Klassen der C++ Standardbibliothek (z.B. vector, map, list, etc) verfügen schon über Iteratoren.
- Die Fähigkeiten des Iterators hängen dabei von der Struktur des  Container ab.
- Die Container-Klassen unterstützen normalerweise folgende Methoden:
  - .beginn():  Iterator auf das erste Element des Containers
  - .end(): Iterator der Hinter das letzte Element zeigt
- Weitere Infos

# Zusätzliche Themen
## Exception Handling

- Werden aus try- und catch-Blöcken zusammengesetzt

```
try {
// Bad File Name
// or missing file handles
}
catch(const BadFileName& e) {
   // handle exception
}
catch(const MissingFileHandle& e) {
   // handle exception
}
```

# Zusätzliche Themen
## Exception Handling: try

- **try**:
  - Grenzt den Bereich ab in dem eine Ausnahme geworfen werden kann
- **Vorgehen nach einer geworfenen Ausnahme:**
  - Die Programmausführung springt zum passenden catch-Block, der unmittelbar dem try-Block folgt.
  - Wird kein passender catch-Block gefunden, wird der Aufruf Stack gegebenenfalls bis zur main- Funktion zurückverfolgt.
  - Die Funktion terminate ruft den Default-Terminatehandler abort auf.
  - Die Funktion abort bricht den aktuellen Prozess ab.

# Zusätzliche Themen
## Exception Handling: throw

- **throw**
  - Löst eine Ausnahme aus (throw e)
  - Der Typ der Ausnahme entscheidet, welcher catch-Block ausgeführt wird.
  - Die Ausnahme e wird als Argument an den catch-Block übergeben, um sie bei der Ausnahmebehandlung zu verwenden.
  - Die Methode e.what() der Ausnahme e gibt Informationen zu dieser zurück.
  - In einem catch-Block kann die Ausnahme durch throw wieder ausgelöst werden.
- **Ausnahmen**
  - Sind im Header exception definiert.
  - C++ bietet bereits eigene Ausnahmen an (z.B. std::out_of_range) → Weitere Infos
  - Eigene Ausnahmen sollten vom Typ std::exception abgeleitet werden.

# Zusätzliche Themen
## Exception Handling: catch

- **catch**
  - Auf einen try-Block folgen eine oder mehrere catch-Blöcke.
  - Die catch-Blöcke geben an, wie bestimmte Typen von Ausnahmen behandelt werden.
  - Die catch-Blöcke werden in der Reihenfolge ihres Auftreten geprüft.
  - Der erste passende catch-Block wird ausgeführt.
  - Eine Ellipse ( ... ) fängt alle Ausnahmen ab ( catch( ... ) { )
- **catch-Blöcke**
  - Sollen vom Speziellen zum Allgemeinen geordnet sein.
  - Sollen dir Argumente als konstante Referenz annehmen (const Exception& e).

# Inhalt

- Vorstellung
- Einleitung, Geschichte
- Grundlagen
- Klassen
- Vererbung
- Zusätzliche Themen
- **Die Standardbibliothek**

# Die Standardbibliothek

- ..