LMU LUDWIG-MAXIMILIANS-UNIVERSITÄT MÜNCHEN

mobile and distributed systems group

Grundlagen C und C++

Einheit 05: Objektorientierung in C++

Lorenz Schauer

Lehrstuhl für Mobile und Verteilte Systeme

Heutige Agenda

Teil 1: Klassen und Objekte

- Header-Files
- Kompilieren mehrerer Dateien
- Konstruktoren & Destruktoren
- Referenzen auf Objekte

Teil 2: Vererbung

- Motivation und Syntax
- Konstruktoren der Oberklasse
- Regeln zur Vererbung in C++

Übungen

- Arbeiten mit Klassen & Objekten
- Vererbung

Lernziele

- Weitere Grundlagen zur Objektorientierung in C++ kennenlernen
- Arbeiten mit Klassen und Objekten
- Vererbung und weiterführende Aspekte…

Arbeiten mit Klassen

Wir können Klassendefinitionen und Programm in eine Datei schreiben:

```
#include <iostream>
#include <string>

class Person{

   public:
      std::string name, vorname;

   private:
      int alter;

   public:
      int getAlter(){
         return alter;
      }
      void setAlter(int alter){
         this->alter = alter;
      }
};

int main(){

   Person pers1;
   pers1.setAlter(35);

   std::cout << "Person 1 ist "<<pers1.getAlter()<< " Jahre alt.\n";

   return 0;
}
```

Das wird spätestens bei größeren Programmen sehr umständlich und fehleranfällig! Daher brauchen wir eine Struktur => Header-Files

Header-Files

Motivation:

- Steigerung der Effizienz (v.a. beim Debuggen & Kompilieren)
- Höhere Wartbarkeit und Erweiterbarkeit
- Trennung von Schnittstellen (Interfaces) und Implementierung

Grundlagen:

- In C++ wird jede Quelldatei separat kompiliert und später zu einem Programm verlinkt
- **Header-Dateien** helfen dabei Deklarationen und Definitionen zwischen verschiedenen Quelldateien auszutauschen
  - Enthalten v.a. Deklarationen, aber keine ausführbaren Anweisungen
  - Mögliche Endung: `.h` `.hpp` `.hxx`
  - Header-Dateien werden über `#inlcude` Direktive in den Quellcode eingebunden
  - Verhindern der Mehrfachausführung des Codes durch `#ifndef` Direktive

Beispiel einer Header Datei
Person.hpp


```
// Person.hpp

#ifndef PERSONDEFINITION
#define PERSONDEFINITION

#include <string>

class Person{

   public:
      std::string name, vorname;
   private:
      int alter;

   public:
      int getAlter();
      void setAlter(int alter);
};

#endif
```

Implementierung von Person.hpp

Die Implementierung der deklarierten Funktionen in Person.hpp erfolgt nun in der Implementierungsdatei Person.cpp

- Person.hpp & Person.cpp bilden zusammen einen gültigen C++ Code für die Klasse Person!

```
// Person.cpp

#include „Person.hpp"

int Person::getAlter(){
   return this->alter;
}

void Person::setAlter(int alter){
   this->alter = alter;
}
```

Anwendung der Klasse Person

Unsere Anwendung kann nun die Klasse Person ganz normal über ihre Header-Datei einbinden und benutzen

```
// Anwendung.cpp

#include <iostream>
#include <string>
#include „Person.hpp"

int main(){

   Person pers1;
   pers1.setAlter(35);

   std::cout << "Person 1 ist "<<pers1.getAlter()<< " Jahre alt.\n";

   return 0;
}
```

Kompilieren

Wiederholung: In C++ wird jede Quelldatei separat kompiliert und später zu einem Programm verlinkt. Daher gilt nun:

Es muss zunächst die Klasse Person kompiliert werden:

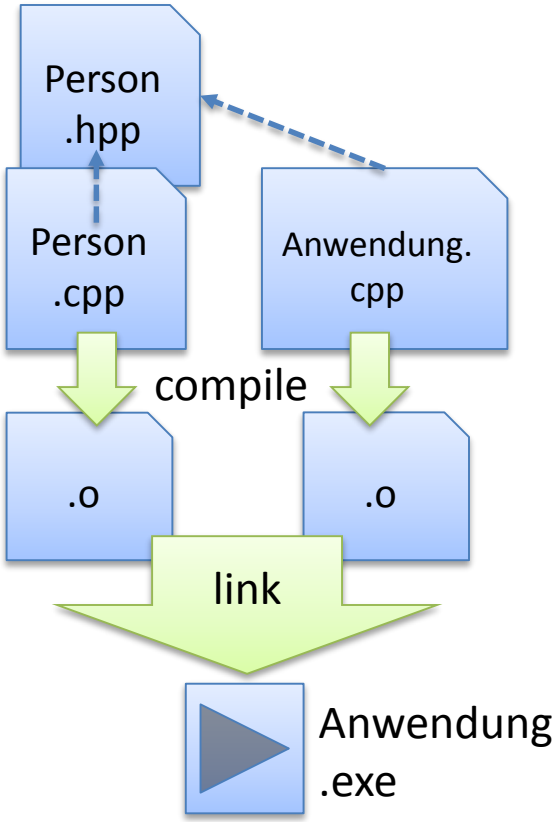
- `g++ -c Person.cpp`
- Erzeugt eine Objektdatei (i.d.R. person.o)

Dann muss die Anwendung kompiliert werden

- `g++ -c Anwendung.cpp`
- Erzeugt Anwendung.o

Und schließlich wird ein lauffähiges Programm erzeugt mittels Verlinkung der beiden Klassen:

- `g++ Anwendung.o Person.o -o MeineAnwendung`



Die beiden letzten Schritte lassen sich auch zu einem Aufruf vereinigen:

- `g++ Anwendung.cpp Person.o -o MeineAnwendung`

Konstruktoren

Der Konstruktor wird implizit aufgerufen, wenn ein Objekt der Klasse erstellt wird

- Der **Default-Konstruktor** kann überschrieben werden
  - Heißt wie die Klasse
  - Hat keine Argumente und keinen Return Wert
  - Muss public definiert sein!
  - Ist zuständig für Initialbelegungen der Feldvariablen

```
// Beispiel in Person.hpp
#ifndef PERSON_H_
#define PERSON_H_

#include <string>

class Person {
   public:

      Person(); // Konstruktor
      std::string name, vorname;
      int getAlter();
      void setAlter(int);

   private:
      int alter;
};

#endif /* PERSON_H_ */
```

```
// Beispiel in Person.cpp

#include "Person.hpp"

Person::Person() {
   this->alter = 33;
   this->vorname="Max";
   this->name = "Mustermann";
}

void Person::setAlter(int alter){
   this->alter = alter;
}

int Person::getAlter(){
   return this->alter;
}
```

Konstruktoren mit Parameter

Es können auch mehrerer Konstruktoren mit unterschiedlichen Parametern definiert werden (Default-Konstruktor ist dann nicht mehr verfügbar!):

- Der „richtige“ Konstruktor wird dann über die verwendeten Parameter bestimmt.

```
// Konstruktoren in Person.cpp

Person::Person() {
   this->alter = 33;
   this->vorname="Max";
   this->name = "Mustermann";
}

Person::Person(std::string vorname,
std::string name) {
   this->vorname=vorname;
   this->name=name;
}

Person::Person(std::string vorname,
std::string name, int alter) {
   this->alter = alter;
   this->vorname=vorname;
   this->name = name;
}
```

```
// Beispiel in Anwendung.cpp

#include "Person.h"
#include <iostream>

int main(){

   Person p1;
   Person p2("Peter","Rupper");
   Person p3("Laura","Derling",22);

   p1.vorname="Hansi";
   p1.setAlter(23);

   std::cout << p1.vorname << " ist "
<<p1.getAlter() << " Jahre alt.\n";
   std::cout << p2.vorname << " ist "
<<p2.getAlter() << " Jahre alt.\n";
   std::cout << p3.vorname << " ist "
<<p3.getAlter() << " Jahre alt.\n";

   return 0;
}
```

```
Ausgabe:
Hansi ist 23 Jahre alt.
Peter ist 0 Jahre alt.
Laura ist 22 Jahre alt.
```

Copy-Konstruktoren

Neben dem Default-Konstruktor wird auch immer implizit ein copy-Konstruktor erzeugt:

- Verlangt als Argument ein Objekt der Klasse und erzeugt dann eine Kopie der Instanz
- Beispiel-Aufruf:
  - `Person p4(p2);` // Erzeugt eine Kopie der Person p2 und speichert sie als Person p4
- Auch dieser Implizite Konstruktor kann natürlich überschrieben werden!

```
// Konstruktoren in Person.hpp

public:
   Person();
   Person(std::string, std::string);
   Person(std::string, std::string,int);
   Person(const Person& p);
```

```
// Copy-Konstruktor in Person.cpp

//Würde in diesem Fall nur Name
und Vorname kopieren!
Person::Person(const Person& p){

   this->name = p.name;
   this->vorname = p.vorname;
}
```

Destruktoren

Destruktoren werden ebenfalls implizit zur Verfügung gestellt und können auch überschrieben werden!

- Werden implizit aufgerufen, wenn das Objekt zerstört wird
  - also wenn der durch das Objekt allokierter Speicher wieder freigegeben wird.
- Destruktoren haben die Form `~Klassenname();`
- Das Schlüsselwort `virtual` ermöglicht dynamisches Binden, so dass die „passende“ Methode erst zur Laufzeit bestimmt wird
  - Bei Destruktoren empfohlen, da somit der korrekte Speicherplatz wieder freigegeben werden kann

```
//Beispiel für Destruktor in
Person.hpp

class Person {
public:

   Person(); // Konstruktor
   virtual ~Person(); // Destruktor
   int getAlter();
   void setAlter(int);
}
```

```
//Beispiel für Destruktor in Person.cpp

#include <iostream>


Person::~Person() {
   std::cout << "Objekt kaputt!\n";
}
```

Pointers auf Objekte

Sehr häufig werden Objekte mittels Pointer referenziert und verwendet

- Analogie in Java: Referenzdatentypen
- Dynamisches Allokieren mittels `new`
- Speicherfreigabe mittels `delete`
  - Wichtig: Immer ein `new` mit einem `delete` schließen

```
// Beispiel für Pointers in Anwendung.cpp
#include "Person.h"
#include <iostream>

int main(){

   Person* p5 = new Person; // Leerer Konstruktor
   Person* p6 = new Person("Hans","Maier",25); // Eigener Konstruktor

   // Zugreifen auf Memberfunktionen und Variablen:
   (*p5).setAlter(22);
   p5->vorname = “Steffi”;
   p6->setAlter(33);

   std::cout << p5->vorname << " ist " <<p5->getAlter() << " Jahre alt.\n";
   std::cout << p6->vorname << " ist " <<p6->getAlter() << " Jahre alt.\n";

   delete p5;
   delete p6;
}
```

Vererbung

Vererbung ist eins der Schlüsselprinzipien der Objektorientierung

- Grund: Wiederverwendung und Erweiterbarkeit von Klassen
- Analog zu anderen objektorientierten Sprachen, wie bspw. Java

```
// Beispiel Student.h: Klasse Student erbt von Person:


#ifndef STUDENT_H_
#define STUDENT_H_

#include "Person.h"
#include <string>

class Student: public Person {

private:
   int matrikelnr;

public:
   Student();
   Student(int);
   Student(std::string, std::string,int,int);
   virtual ~Student();

   int getMatrikelnr();
};

#endif /* STUDENT_H_ */
```

Klasse Student erbt von Klasse Person

Konstruktoren

Destruktor

Getter-Methode

Konstruktoren der Oberklasse

Um einen Studenten zu erzeugen, sollten auch die Attribute der Oberklasse befüllt werden:

```
// Beispiel: Student.cpp
#include "Student.h"
Student::Student() : Person(){
}

Student::Student(int matnr) : Person() {
   this->matrikelnr = matnr;
}

Student::Student(std::string name, std::string vorname, int alter ,int matnr) :
Person(name,vorname,alter){

   this->matrikelnr=matnr;
}

Student::~Student() {
   // TODO Auto-generated destructor stub
}

int Student::getMatrikelnr(){
   return this->matrikelnr;
}
```

Leerer Basis-Konstruktor

Setzt Matrikelnummer und nutzt überschriebenen default Konstruktor der Oberklasse

Setzt Matrikelnummer und nutzt den 2. Parameterkonstruktor der Oberklasse

Destruktor

Getter-Methode

Regeln zur Vererbung

Mehrfachvererbung möglich! (Anders als in Java)

- Beispiel: Die Klasse C vereint die Funktionalitäten von A und B und fügt noch etwas hinzu:
  - `class C : public A, public B {…//Klassendefinition…}`

Zugriffskontrolle bei Ober- und Unterklassen:

Ist ein Element in A	public	protected	private
... wird es in B	public	protected	private
... wird es in C	protected	protected	private
... wird es in D	private	private	private

Subtyping und type casting ebenfalls möglich:

```
Person* p7 = new Student("Winnie","Tou",23,1111);
std::cout << p7->vorname<<"\n";

Student* s7 = dynamic_cast<Student*> (p7);
```

Programmieraufgabe

In einem Weihnachtschor beteiligen sich verschiedene Personen:

- Kinder bis 14 Jahren
- Jugendliche bis 22 Jahren

Eine Person hat:

- Einen Namen, ein alter und eine Angabe, ob er/sie beim Chor mitsingt
- Schreiben Sie entsprechende Getter-Methoden
- Eine Person kann also zum Chor kommen, dann singt sie mit
- Oder Eine Person kann wieder gehen, dann singt sie nicht mehr mit
- Zu Beginn muss die Person erstmal zum Chor kommen, damit sie mitsingt

Ein Kind ist eine Person und

- Singt, wenn es zum Chor gekommen ist „Stille Nacht“

Ein Jugendlicher ist auch eine Person und

- Singt, wenn er/sie zum Chor gekommen ist „Heilige Nacht“

Schreiben Sie entsprechende Klassen und lassen Sie in Ihrer Anwendung die Kinder und Jugendlichen abwechselnd singen, falls diese zum Chor gekommen sind.