Ein- und Ausgabe

- In C++ können die bekannten C-Funktionen `printf` und `scanf` verwendet werden
- Bevorzugt erfolgt die Ein- und Ausgabe aber über Datenströme
- Ein Strom (stream) ist eine Folge von Bytes, die sequentiell verarbeitet werden
- In der C++-Standardbibliothek sind z.B. die folgenden Ströme definiert

`cin`	Standardeingabe (Tastatur)
`cout`	Standardausgabe (Bildschirm)
`cerr`	Standardfehlerausgabe (Bildschirm)

- Für die Deklaration ist die Header-Datei `iostream` einzubinden

- Der Operator << formt die interne Darstellung in eine Textdarstellung um
- Der Operator liefert einen Verweis (*Referenz*) auf den Ausgabestrom (`ostream`) zurück. Damit können in der Folge weitere Operatoren auf den Strom angewendet werden

```
#include <iostream>

int main(){
    int i = 1;
    std::cout << i << ". C++";

    return 0;
}
```

- Der Strom kann über Manipulatoren geleitet werden
  - Ein Manipulator ist eine Operation, die den Inhalt des Datenstroms verändern kann (z.B. Formatierung der Ausgabe)
  - In der C++-Standardbibliothek stehen u.a. die folgenden Manipulatoren zur Verfügung

Manipulator	**Wirkung**
`endl`	fügt Zeilenende ein
`setw(n)`	Spalte für nächstes Ausgabeobjekt ist n Stellen breit
`left`	linksbündig ausrichten
`right`	rechtsbündig ausrichten
`setfill(c)`	c als Füllzeichen verwenden
`dec`	stellt Zahl dezimal dar
`oct`	stellt Zahl oktal dar
`hex`	stellt Zahl hexadezimal dar
`showbase`	0X bei hexadezimal und 0 bei oktal

Manipulator	Wirkung
`boolalpha`	stellt true und false textuell dar
`noboolalpha`	stellt true als 1 und false als 0 dar
`showpoint`	Darstellung mit Dezimalpunkt und Nachkommastelle
`noshowpoint`	nur die relevanten Stellen werden ausgegeben
`scientific`	wissenschaftliche Darstellung
`fixed`	festgelegte Nachkommastellen
`setprecision(n)`	Genauigkeit auf *n* Stellen anzeigen

```
#include <iostream>
#include <iomanip>

int main(){
    std::cout << std::setfill('0') << std::right
              << std::setw(4) << 12 << std::endl;
    return 0;
}
```

Welche Ausgabe erfolgt?

- Der Operator >> wandelt eine Folge von Zeichen in die interne Darstellung um
- Der Operator >> arbeitet nach folgenden Regeln
  - Führende Zwischenraumzeichen (*whitespace*) werden ignoriert
  - Zwischenraumzeichen werden als Endekennung genutzt
  - Andere Zeichen werden in den verlangten Datentyp umgewandelt

Zwischenraumzeichen { Leerzeichen, Tabulatorzeichen, Zeilenrücklauf
Zeilensprung, Seitenvorschub, Zeilenendekennung

- Erst nach Betätigung der Enter-Taste wird die Zeichenkette vom Betriebssystem an das Programm übergeben

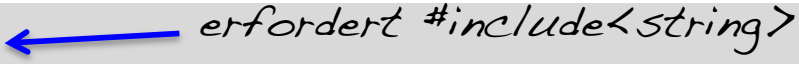
```
int i;
std::cout << "Bitte ganze Zahl eingeben:";

std::cin >> i;

std::cout << "Es wurde " << i << " eingegeben" << std::endl;
```

- Einzelne Zeichen können mit der Funktion `get` eingelesen werden

```
char c;
std::cin.get(c);
```

- Ganze Zeilen (inkl. Leerzeichen) können mit der Funktion `getline` eingelesen werden

```
std::string name;
std::cout << "Vor- und Nachname eingeben:";
std::getline(std::cin, name);
```



C++-Standardbibliothek

- Jeder C++-Compiler liefert eine Bibliothek von nützlichen Klassen und Funktionen mit
- Die Deklaration der entsprechenden Klassen und Funktionen wird über Header-Dateien eingebunden (ohne Endung)

Thema	Header
Ein-/Ausgabe	`<iostream>`
Zeichenkette	`<string>`
Fehlerbehandlung	`<exception>` `<stdexception>`
Hilfsfunktionen und –klassen	`<utility>` `<functional>`
Laufzeittyperkennung	`<typeinfo>`

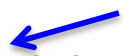
kleiner Ausschnitt

Klassen

- Klassen und Objekte sind Ihnen bereits bekannt
  - Objekte haben
    - Identität
    - Zustand
    - Verhalten
  - Eine Klasse ist ein „Bauplan“ für gleichartige Objekte

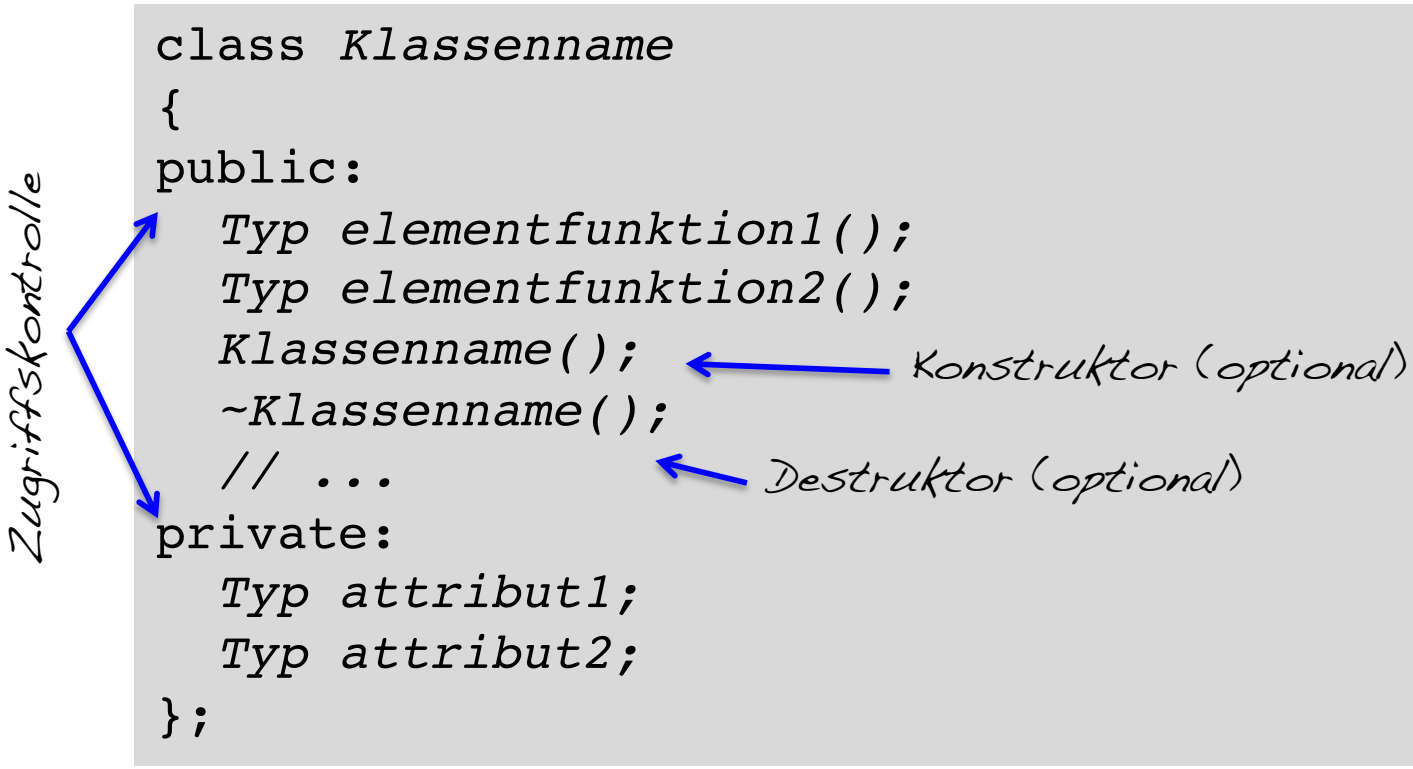
- Klassen ermöglichen benutzerdefinierte Datentypen

- Eine Klasse (`class`) besteht aus
  - einem Klassennamen
  - Attributen (*data members*), die einen Typ und einen Variablennamen besitzen
  - Elementfunktionen (*member functions*), die Parameter und Rückgabewerte besitzen



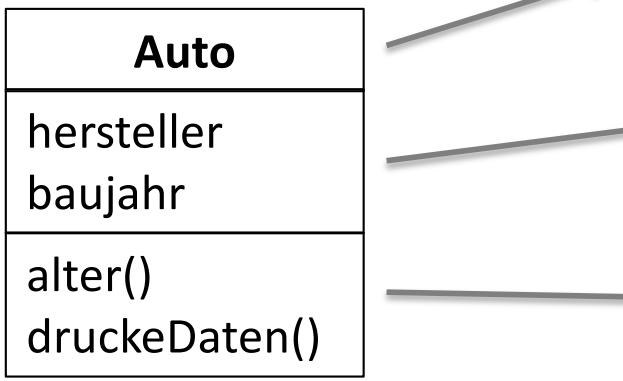
  - Konstruktoren, die als spezielle Elementfunktionen Initialisierungsaufgaben bei der Objekterzeugung übernehmen
  - Destruktoren, die als spezielle Elementfunktionen Aufräumarbeiten vor der „Zerstörung“ eines Objektes übernehmen

– Aufbau einer Klassendeklaration



– Trennung von Deklaration und Implementierung

Deklaration der Klasse erfolgt in einer Header-Datei

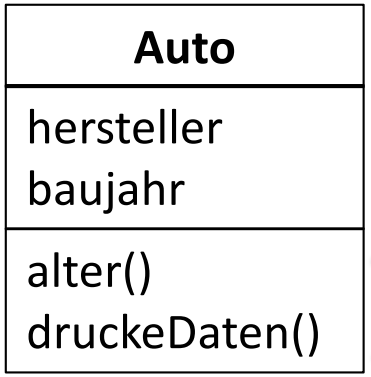



```
class Auto{
private:
    const char* hersteller;
    int baujahr;
public:
    Auto(const char* h, int b);
    int alter();
    void druckeDaten();
};
```

auto.h

Keine Implementierung der Elementfunktionen

Implementierung der Elementfunktionen erfolgt in einer weiteren Datei




```
#include "auto.h"
#include <iostream>

Auto::Auto(const char* h, int b) {
     hersteller = h;
     baujahr = b;
}

int Auto::alter(){
     // nicht optimal
     return 2019-baujahr;
}

void Auto::druckeDaten(){
    std::cout << hersteller << " von " << baujahr
              << endl;
}
```

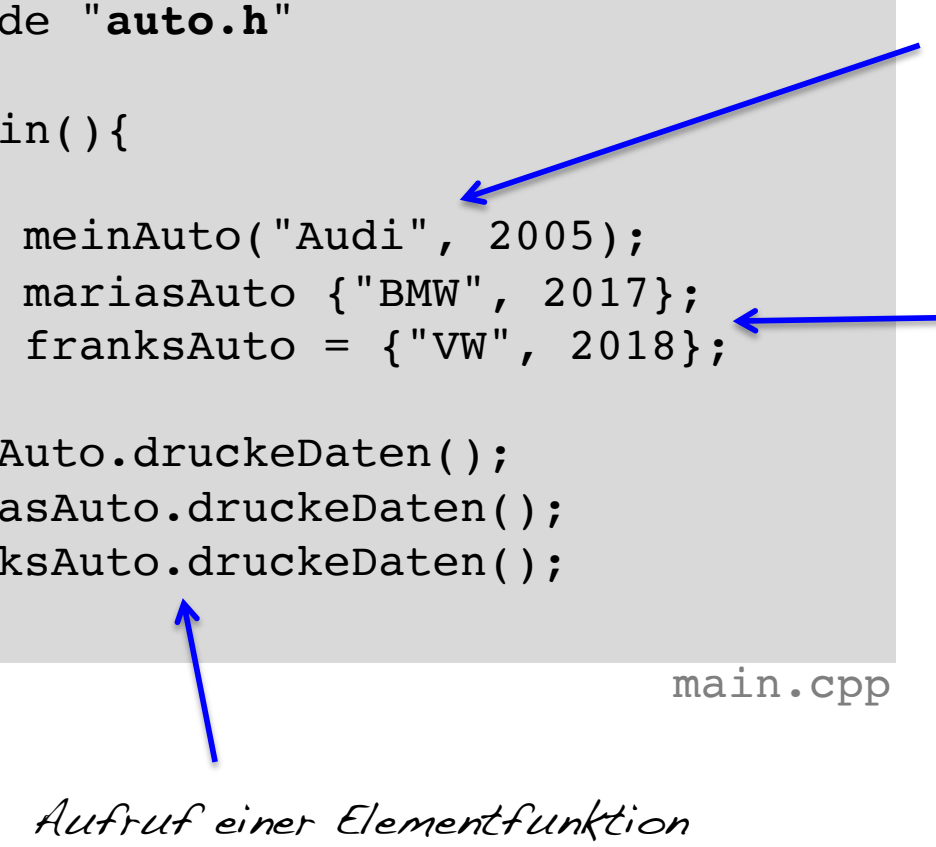
Konstruktor

auto.cpp

– Objekterzeugung

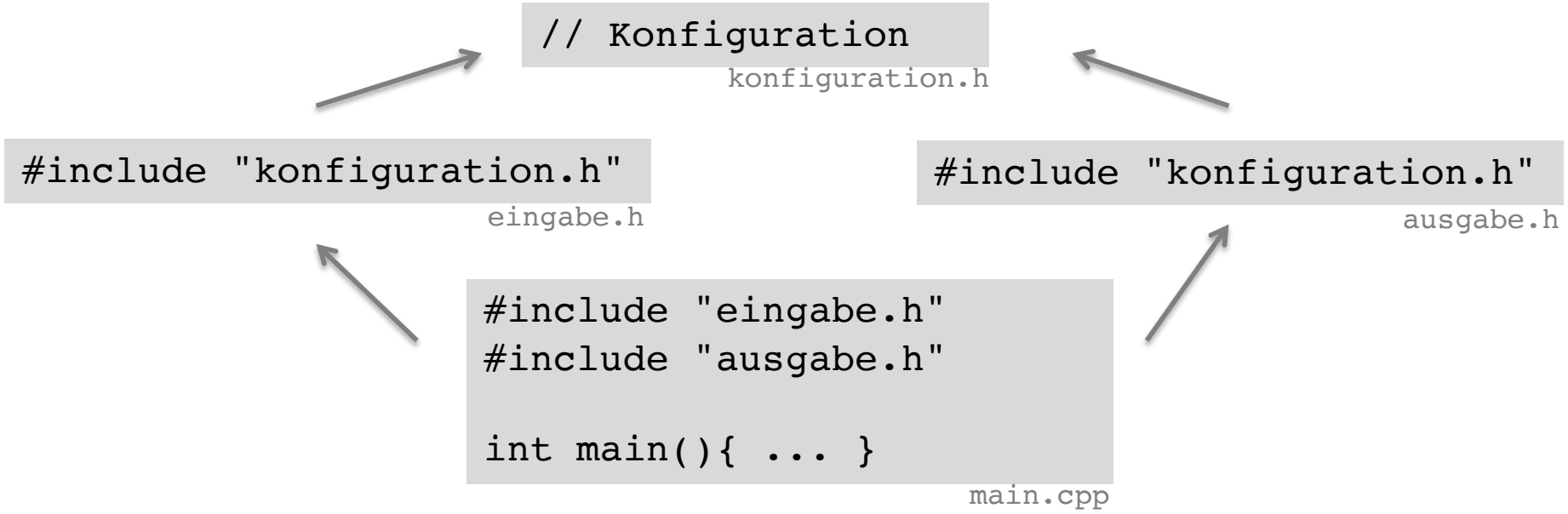
```
#include "auto.h"

int main(){

  Auto meinAuto("Audi", 2005);
  Auto mariasAuto {"BMW", 2017};
  Auto franksAuto = {"VW", 2018};

  meinAuto.druckeDaten();
  mariasAuto.druckeDaten();
  franksAuto.druckeDaten();
}
```

main.cpp



Includeguards

- Auch bei der C++-Programmierung können die bekannten Präprozessordirektiven verwendet werden
- Bis auf `#include` wird die intensive Nutzung von Präprozessor-direktiven aber nicht empfohlen
- Eine Ausnahme bilden *Includeguards* zur Vermeidung der doppelten Einbindung von Header-Dateien



- Man schreibt daher in einer Header-Datei die Deklarationen zwischen eine Präprozessorabfrage
- Diese Abfrage prüft, ob eine symbolische Konstante bereits definiert wurde (`#ifndef` als Abkürzung für `#if !defined`)
- Die Definition der Konstanten wird ebenfalls durch die Abfrage kontrolliert

```
#ifndef KONFIGURATION_H_
#define KONFIGURATION_H_

// Konfiguration

#endif
```

- Sobald die symbolische Konstante einmal definiert wurde, werden alle weiteren Deklarationen beim wiederholten `#include` vom Präprozessor ausgeblendet

Konstruktoren

- Konstruktoren sind spezielle Elementfunktionen
  - gleicher Name wie die Klasse
  - kein Rückgabewert (auch nicht `void`)
  - werden aufgerufen, wenn das Objekt erzeugt wird
- Konstruktoren können überladen werden
- Falls kein Konstruktor deklariert ist, dann wird vom Compiler ein Standardkonstruktor erzeugt
- Ein Standardkonstruktor besitzt keine Parameter

erzwingt die geregelte Initialisierung eines Objekts

– Standardkonstruktor (hier vom Compiler erzeugt)

Kein Konstruktor deklariert

```
class Punkt {
    int x,y;
public:
    int getX();
    int getY();
    void setX(int);
    void setY(int);
};
```

punkt.h

```
#include "punkt.h";
#include <iostream>

int main(){
    Punkt p1;
    Punkt p2 {};
    // ...
}
```

main.cpp

```
#include "punkt.h"

int Punkt::getX(){return x;}

int Punkt::getY(){return y;}

void Punkt::setX(int x_koor){
    x = x_koor;
}

void Punkt::setY(int y_koor){
    y = y_koor;
}
```

punkt.cpp

Standardkonstruktor wird ohne Klammern aufgerufen

leere Initialisierungsliste ab C++11 erlaubt

– Überladen von Konstruktoren

```
class Punkt {
    int x,y;
public:
    Punkt();
    Punkt(int x_koor, int y_koor);
    int getX();
    int getY();
    void setX(int);
    void setY(int);
};
```

punkt.h

private ist die Standardeinstellung für class

Sobald ein Konstruktor deklariert ist, wird der Standardkonstruktor nicht mehr automatisch zur Verfügung gestellt

Übungsaufgabe

Geben Sie eine sinnvolle Implementierung der beiden Konstruktoren an. Erzeugen Sie dann ein Punkt-Objekt mit den Koordinaten (10,20).

- Initialisierung mit konstruktorinterner Liste
  - Ablauf bei einem normalen Konstruktoraufruf
    1. Vor Ausführung des Blocks wird Speicherplatz für die benötigten Datenelemente (z.B. `x` und `y`) beschafft
    2. Der Block wird ausgeführt und die Parameter werden zugewiesen
  - Beide Vorgänge können in einem Schritt zusammengefasst werden
  - Kann bei größeren oder vielen Datenelementen (Objekten) Laufzeitvorteile bringen

```
Punkt::Punkt(int x_koor, int y_koor) : x{x_koor}, y{y_koor}{
    // Block kann leer sein
}
```

  - Es wird erst die Liste abgearbeitet und dann der Block
  - Die Reihenfolge der Initialisierung richtet sich nach der Reihenfolge in der Klassendeklaration (nicht nach der Reihenfolge in der Liste)

- Delegierender Konstruktor (*delegating constructor*)
  - Um die Wartung von Programmen zu verbessern, ist doppelter Code zu vermeiden
  - Im Falle von Konstruktoren kann doppelter Code vermieden werden, indem ein Konstruktor einen anderen Konstruktor der Klasse aufruft

```
Auto::Auto(const char* h, int b) {
    hersteller = h;
    baujahr = b;
}

Auto::Auto(int b) : Auto("unbekannt", b){
}
```

- Initialisierung mit voreingestellten Werten (*default arguments*)
  - Um die Anzahl der Konstruktoren pro Klasse zu reduzieren, können Parameterwerte vorgegeben werden
  - Nur für die letzten Parameter einer Parameterliste können Werte vorgegeben werden
  - Bei überladenen Konstruktoren müssen eindeutige Aufrufe gewährleistet bleiben

```
class Auto{
private:
    const char* hersteller;
    int baujahr;

public:
    Auto(int b, const char* h = "unbekannt");
    int alter();
    void druckeDaten();
};
```

Reihenfolge der Parameter wurde hier vertauscht

Auswertung beim Aufruf

- In-Class Initialisierung
  - statt in einem Konstruktor

```
class A {
public:
   int a;
   int b;
   A() : a{1}, b{2} {}
};
```

kann die Initialisierung für nicht-statische Attribute auch bei der Deklaration erfolgen

```
class A {
public:
   int a {1};
   int b = 2;
};
```

Wird eingesetzt, um bei mehreren Konstruktoren pro Klasse eine Duplizierung identischer Zuweisungen zu vermeiden

- Kopierkonstruktor (*copy constructor*)
  - Initialisiert ein neues Objekt mit einem bestehenden Objekt
  - Die Elemente des Objekts werden bitweise kopiert
  - Wird aufgerufen, wenn
    1. ein neues Objekt erzeugt und mit einem bestehenden initialisiert wird
    2. ein Objekt per Wertübergabe (*by value*) an eine Funktion übergeben wird
    3. ein Objekt mit `return` als Wert (*by value*) zurückgegeben wird (hier kann der Compiler ggf. den Aufruf unterlassen, um im Rahmen einer Optimierung unnötige Kopieroperationen zu vermeiden)
  - Wird kein Kopierkonstruktor implementiert, dann erzeugt ihn der Compiler automatisch
  - Beispiele für den Aufruf

```
Punkt p1 {1,2}; // normaler Konstruktor

Punkt p2 {p1};  // Kopierkonstruktor
Punkt p3 = p2;  // Kopierkonstruktor

p3 = p1;        // Zuweisungsoperator
```

- Schematisch hat der generierte Kopierkonstruktor den folgenden Aufbau

```
Punkt::Punkt(const Punkt& obj) : x {obj.x}, y {obj.y}{
    // Ausgabe nur fuer eigene Experimente
    std::cout << "Copy-Konstruktor" << std::endl;
}
```

Reicht der automatisch generierte Kopierkonstruktor zum Kopieren einer verketteten Liste?