Themen

- Kontrollstrukturen
  - while
  - do ... while
  - continue
  - break
  - goto
- virtuelle Funktionen / virtuelle Destruktoren
- rein virtuelle Funktionen / abstrakte Klassen
- Laufzeit Typeninformation
- C++ Structures

while Schleife

- eine **`while`** Schleife führt einen Codeblock solange eine bestimmte Bedingung zutrifft.

```
int x=1;
while( x<=10 )
{
   std::cout << "x: " << x << "\n";
   ++x;
}
```

- **Increment nicht vergessen!!!**

do...while Schleife

- eine **do...while** Schleife führt einen Codeblock solange aus bis eine bestimmte Bedingung zutrifft

```
int x=1;
do
{
   std::cout << "x:" << x << "\n";
   ++x;
} while(x>=10);
```

do while Schleife

- eine **do...while** Schleife führt einen Codeblock solange bis eine bestimmte Bedingung zutrifft

```
int x=1;
do
{
   std::cout << "x:" << x << "\n";
   ++x;
} while(x>=10);
```

- **Increment nicht vergessen!!!**

Unterschiede zwischen den verschiedenen Schleifen

- **`for`** und **`while`** Schleifen sind identisch, d.h. die eine kann die andere ersetzen.
- **`while`** und **`do...while`** Schleifen sind sehr ähnlich. Der Hauptunterschied ist, dass die **`do...while`** Schleife mindestens einmal ausgeführt wird.

for <-> while

```
for(int x=1;x<=10;++x)
{
    ...
}

int x=1;
while(i<=10)
{
    ...
    ++x;
}
```

while <-> do...until

```
int x=20;
while(x<10)
{
  std::cout << x << std::endl;
  ++x;
}

int x=20;
do
{
  std::cout << x << std::endl;
  ++x;
} while(x<10);
```

continue

- **`continue`** springt zur nächsten Iteration der Schleife (**`for`**, **`while`**, **`do...while`**) in der es aufgerufen wird
- in **`do...while`** and **`while`** Schleifen kann **`continue`** leicht zu Endlosschleifen führen!!!

continue Beispiel

```
int x=1;
while( x<=10 )
{
   if(x==5)
   {
      continue;
   }
   std::cout << x << "\n";
   ++x;
}
```

continue Beispiel

```
int x=1;
while( x<=10 )
{
   if(x==5)
   {
      ++x;
      continue;
   }
   std::cout << x << "\n";
   ++x;
}
```

Ausgabe

```
1
2
3
4
6
7
8
9
10
```

break

- **`break`** wird benutzt um vorzeitig eine Schleife abzubrechen (**`for`**, **`while`**, **`do...until`**)
- **`break`** bricht nur die Schleife ab in der es aufgerufen wird

break Beispiel

```
int x=1;
while( x<=10 )
{
   if(x==5)
   {
      break;
   }
   std::cout << x << "\n";
   ++x;
}
```

break Beispiel

```
int x=1;
while( x<=10 )
{
   if(x==5)
   {
      break;
   }
   std::cout << x << "\n";
   ++x;
}
```

Ausgabe

- 1
- 2
- 3
- 4
-

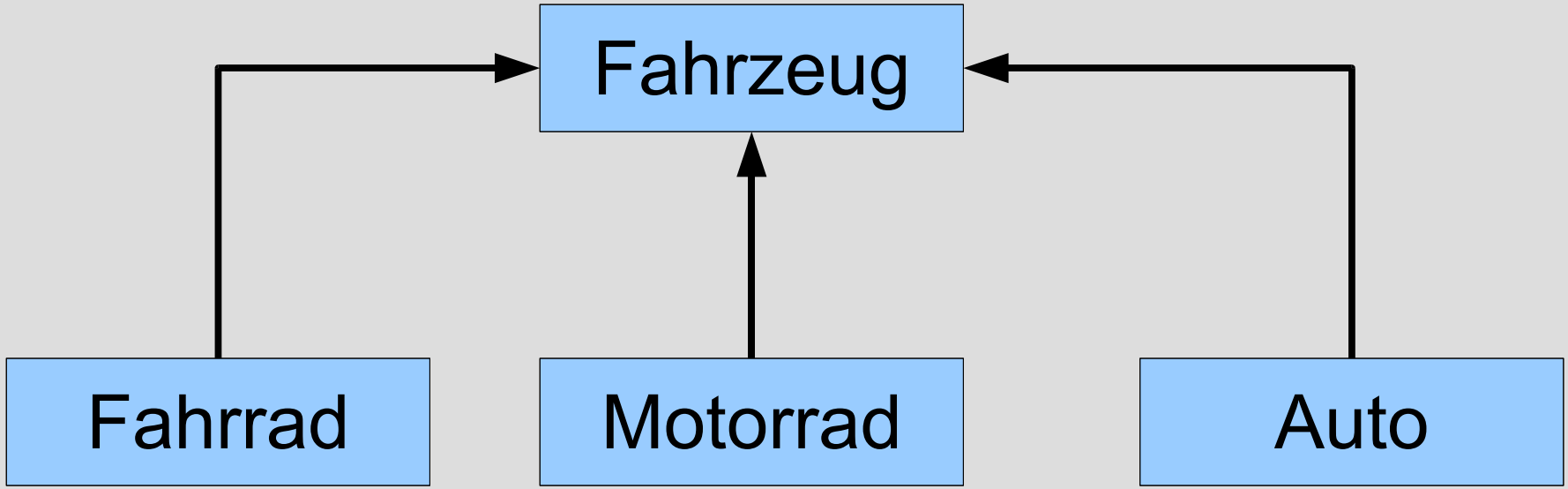
goto

- `goto` springt in einem Programm an eine bestimmte Stelle und führt den Code dort aus
- man kann nur innerhalb einer Funktion springen
- nützlich um aus verschachtelten Schleifen herauszuspringen
- Finger weg davon wenn es nicht unbedingt notwendig ist!!!

goto Beispiel

```
  ...
if(x==4)
{
   goto FOUND;
}
   ...

FOUND:
      std::cout << "Hurra\n";
```

Virtuelle Funktionen



Virtuelle Funktionen

```
class Fahrzeug{
   string getTyp() const{
      return “Fahrzeug”;
   }
};

class Auto : public Fahrzeug{
   string getTyp() const{
      return “Auto”;
   }
};
...
```

Virtuelle Funktionen

```
Auto* auto=new Auto();
Fahrrad* fahrrad=new Fahrrad();
Fahrzeug* fahrzeug=new Fahrzeug();
cout <<  auto->getTyp() << endl;
cout <<  fahrrad->getTyp() << endl;
cout <<  fahrzeug->getTyp() << endl;
```

```
Auto
Fahrrad
Fahrzeug
```

Virtuelle Funktionen

```
delete fahrzeug;
fahrzeug=fahrrad;
cout << fahrrad->getTyp() << endl;
cout << fahrzeug->getTyp() << endl;
```

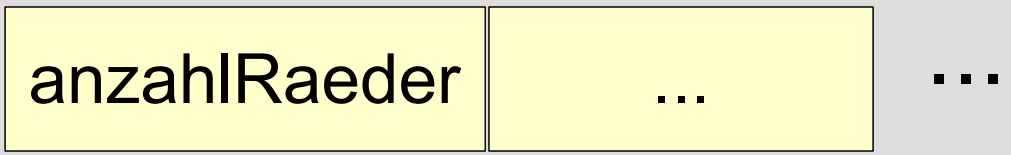
```
Fahrrad
Fahrzeug
```

Virtuelle Funktionen

```
class Fahrzeug{
   virtual string getTyp() const{
      return “Fahrzeug”;
   }
};

class Fahrrad : public Fahrzeug{
   virtual string getTyp() const{
      return “Fahrrad”;
   }
};
...
```

Virtuelle Funktionen

```
class Fahrzeug
{
    protected:
     unsigned int anzahlReader;
     ...
};
```



Virtuelle Funktionen

```
class Fahrzeug
{
    ...
    public:
       virtual string getTyp() const;
};
```

VTable

- ist ein Zeiger auf eine Tabelle mit Zeigern auf die virtuellen Methoden einer Klasse
- Wird eine virtuelle Methode einer Klasseninstanz aufgerufen, so holt das Programm sich zur Laufzeit den zeiger auf die korrekte Methode und führt diese aus

Virtuelle Funktionen

```
Auto* auto=new Auto();
Fahrrad* fahrrad=new Fahrrad();
cout <<  fahrrad->getTyp() << endl;
cout <<  auto->getTyp() << endl;
Fahrzeug* fahrzeug=auto;
cout <<  fahrzeug->getTyp() << endl;
fahrzeug=fahrrad;
cout << fahrzeug->getTyp() << endl;
Fahrrad
Auto
Auto
Fahrrad
```

Virtuelle Funktionen

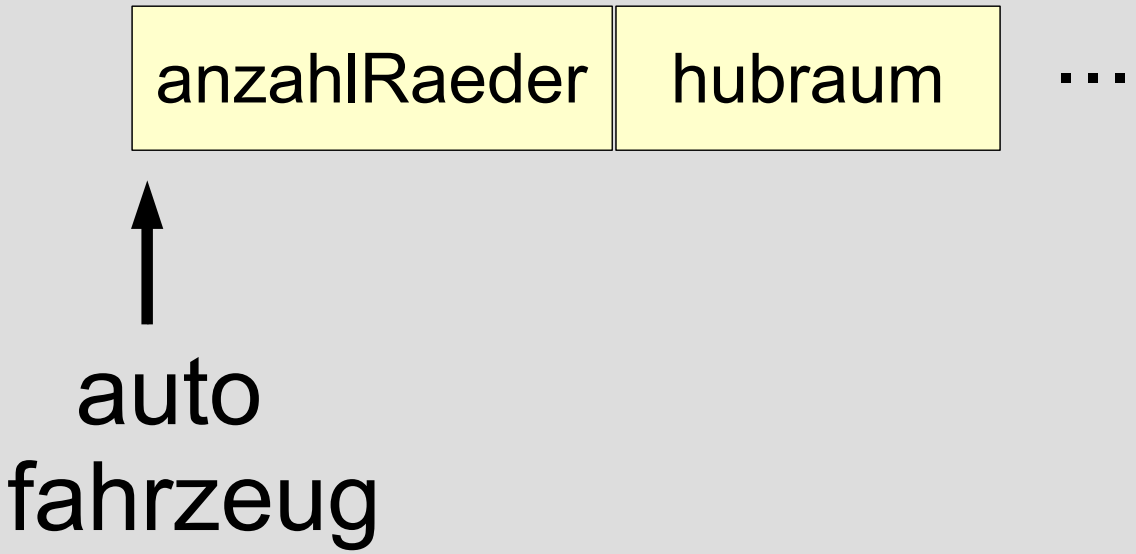
```
class Fahrzeug{
  unsigned int anzahlRaeder;
};

class Auto : public Fahrzeug{
  unsigned int hubraum;
};
...
```

Virtuelle Funktionen

```
Auto* auto=new Auto();
Fahrzeug* fahrzeug=auto;
delete fahrzeug;
```

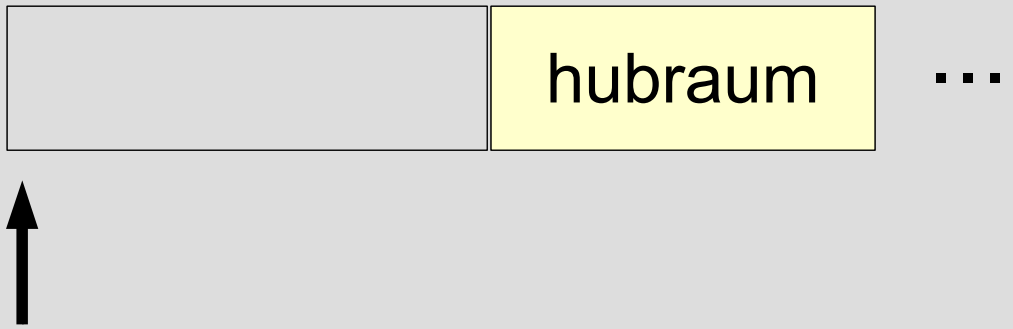
Virtuelle Funktionen

```
Auto* auto=new Auto();
Fahrzeug* fahrzeug=auto;
```



Virtuelle Funktionen

delete fahrzeug;



Virtueller Destruktor

```
class Fahrzeug{
  unsigned int anzahlRaeder;
  virtual ~Fahrzeug(){};
};

class Auto : public Fahrzeug{
  unsigned int hubraum;
  virtual ~Auto();
};
...
```

Rein Virtuelle Funktionen

- rein virtuelle Funktionen sind Funktionen die nur deklariert sind aber nicht definiert.
- die Deklaration einer rein virtuellen Funktion entspricht der einer virtuellen Funktion, es wird lediglich an das Ende =0 angehängt.
- Klassen mit rein virtuellen Funktionen sind abstrakte Klassen
- abstrakte Klassen können nicht instanziiert werden
- abgeleitete Klassen müssen die rein virtuellen Funktionen überschreiben und implementieren, sonst sind sie auch abstrakt

Rein Virtuelle Funktionen

```
class Fahrzeug{
  virtual string getTyp()=0;
};

class Auto : public Fahrzeug{
  virtual string getTyp(){
    return "Auto";
  }
};
...
```

Rein Virtuelle Funktionen

```
Fahrzeug* fahrzeug=new Fahrzeug();
Auto* auto=new Auto();
Fahrzeug* fahrzeug2=auto;
cout << fahrzeug->getTyp() << endl;
cout << auto->getTyp() << endl;
cout << fahrzeug2->getTyp() << endl;
```

Fehler. Es gibt keine Instanzen von abstrakten Klassen!

Rein Virtuelle Funktionen

```
// Fahrzeug* fahrzeug=new Fahrzeug();
Auto* auto=new Auto();
Fahrzeug* fahrzeug2=auto;
// cout << fahrzeug->getTyp() <<
 endl;
cout << auto->getTyp() << endl;
cout << fahrzeug2->getTyp() << endl;
```

```
Auto
Auto
```

Rein Virtuelle Funktionen

```
// Viereck.h
class Viereck{
   virtual void init()=0;

};

// Viereck.cpp
void Viereck::init()
{
    // initialisiert vier Punkte
        ...
}
```

Rein Virtuelle Funktionen

```
class Rechteck: public Viereck{
   Rechteck(){
     Viereck::init();
   }
};

class Trapez: public Viereck{
   Trapez(){
     Viereck::init();
    }
};
```

Laufzeit Typeninformation

- dient dazu zur Laufzeit des Programms die exakte Klasse eines Objektes zu bestimmen
- die Funktion **`typeid`** nimmt als Argument eine Instanz einer Klasse, oder einen Klassennamen und liefert ein Object vom Type ***`type_info`*** zurück

Laufzeit Typeninformation

```
Fahrzeug* fahrzeug=new Auto();
...
type_info info=typeid(fahrzeug);
type_info classInfo=typeid(Auto);
if(info == classInfo)
{
  cout << "Ich bin ein Auto" << endl;
}
```

Laufzeit Typeninformation

```
Fahrzeug* fahrzeug=new Auto();
...
type_info info=typeid(fahrzeug);
type_info classInfo=typeid(Fahrzeug);
if(info == classInfo)
{
  cout << "Ich bin ein Fahrzeug\n";
}
```

Structures

- mit dem C/C++ Schlüsselwort **`struct`** kann man sich eigene Datentypen definieren, die mehrere Daten zusammenfassen.

```
struct Point
{
    double x;
    double y;
};
```

Structures

```
struct Point
{
   double x;
   double y;
};

class Point
{
   public:
      double x;
      double y;
};
```

Structures

```
struct Point
{
    double x;
    double y;
};

Point p;
p.x=2.5;
p.y=4.7;
```

Structures

- **`struct`** Datentypen sind in C++ mit Klassen identisch
  - die Deklaration ist dieselbe, es wird lediglich das Schlüsselwort **`class`** durch **`struct`** ersetzt
  - der Zugriff auf die Attribute ist standardmäßig **`public`**