Informatik II

Oliver Jack

Fachhochschule Jena
Fachbereich Elektrotechnik und Informationstechnik

Sommersemester 2010

Inhalt

Klassen

- Abstraktion
- Schnittstellen von Objekten
- Klassendefinition und -deklaration
- Die Klasse Schlange
- Spezielle Member-Funktionen
- Statische Members
- Initialisierung von Member-Variablen
- Selbstreferenz und Operator-Overloading
- Die Klasse Raumkoordinaten

Zusammenfassung

Vorlesung 5. Klassen, Teil 1

Klassen

- Abstraktion
- Schnittstellen von Objekten
- Klassendefinition und -deklaration
- Die Klasse Schlange
- Spezielle Member-Funktionen
- Statische Members
- Initialisierung von Member-Variablen
- Selbstreferenz und Operator-Overloading
- Die Klasse Raumkoordinaten

Zusammenfassung

5. Klassen, Teil 1

Vorige Vorlesung

- Funktionsdefinitionen und Default-Argumente
- Statische Variablen
- Overloading, Argumentübergabe und Gültigkeitsbereiche
- Rekursion
- Operatoren

Heutige Vorlesung

- Abstraktion und Objekte
- Klassendefinition und -deklaration
- Member-Funktionen
- Selbstreferenz und Operatoren
- Beispiele

Lernziele dieser Vorlesung

- Verständnis von Objekten und Klassen
- Umgang mit Objekten und deren Verhalten
- Verständnis der Konstruktion von Klassen und Objekten
- Kenntnis von Operatoren für Objekte

Problem und Modell

- Jede Programmiersprache beinhaltet Abstraktion.
- Z. B.: Assemler ist eine Abstraktion der Rechenmaschine, C ist eine Abstraktion von Assembler.
- Imperative Programmiersprachen, wie C erfordern ein Denken in Strukturen der Rechenmaschine.
- Der Programmierer muss eine Übersetzung der Strukturen des Problems in Strukturen der Rechenmaschine durchführen.
- Wünschenswert ist ein Denken in Strukturen des Problems, das zu lösen ist, ohne Notwendigkeit einer Übersetzung in Strukturen der Rechenmaschine.

Modell des Problems

- Alternative zur Modellierung der Maschine: Modellierung des Problems
- Frühe Programmiersprachen
  - LISP: jedes Problem lässt sich in Listen modellieren
  - APL: jedes Problem ist algorithmischer Natur
- Solche Ansätze sind gut für Probleme, die nah an diesen Konzepten sind.
- Probleme außerhalb der jeweiligen Domäne sind schwierig zu modellieren.

Objekte

- Objekt-Orientierung: Elemente der Problem-Domäne werden repräsentiert als Objekte im Lösungsraum
- Im Programm-Code wird das Problem (mehr oder weniger) direkt beschrieben.
- Verbindung zur Rechenmaschine: Jedes Objekt ist wie ein „kleiner Computer“.
- Objekte besitzen einen Status, Operationen und können Aktionen ausführen.
- Verbindung zur Realität: Objekte besitzen Eigenschaften und zeigen ein Verhalten.

Objektorientierte Programmierung ...

Jedes Ding ist ein Objekt

- Ein Objekt ist eine Variable, die Daten enthält.
- Ein Objekt kann auf Anforderung Aktionen ausführen.

Ein Programm ist eine Menge von Objekten

- Objekte senden einander Nachrichten.
- Nachrichten veranlassen Objekte zur Ausführung von Aktionen.

. . . Objektorientierte Programmierung

Jedes Objekt besitzt seinen eigenen Speicher

- Der Speicher besteht aus anderen Objekten.
- Komplexität des Programms wird auf (einfache) Objekte reduziert.

Jedes Objekt besitzt einen Typ

- Jedes Objekt ist eine Instanz einer Klasse (Klasse = Typ).
- Das entscheidende Unterscheidungskriterium einer Klasse ist die Menge der Nachrichten an Objekte einer Instanz der Klasse.

Klassen...

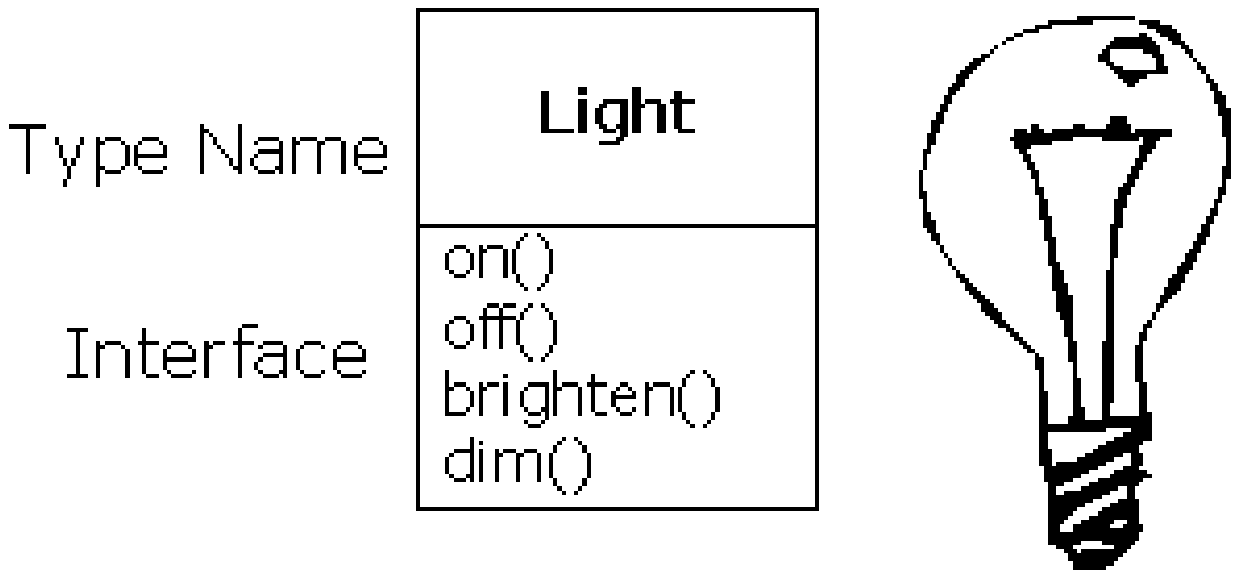
- Jedes Objekt ist eindeutig und einmalig.
- Verschiedene Objekte können gemeinsame Charakteristika besitzen.
- Objekte gehören zu einer Klasse.
- C++-Konstrukt und Schlüsselwort: class
- Beispiel: 0034582 gehört zur Klasse Konto.
- Von einer Klasse können Instanzen erzeugt werden.

. . .Klassen

- Objekte einer Klasse besitzen Klassenelemente (Members).
- Klassenelemente können Variablen und Funktionen (Methoden) sein.
- Beispiel: Jedes Konto besitzt einen Kontostand, akzeptiert Ein- und Auszahlungen.
- Objekte besitzen einen Zustand.

Schnittstelle

- Objekte besitzen eine Schnittstelle.
- Die Schnittstelle definiert, welche Methoden und Eigenschaften das Objekt besitzt.




```
Light lt;
lt.on();
```

Verborgene Implementierung

Anwendungsprogrammierung

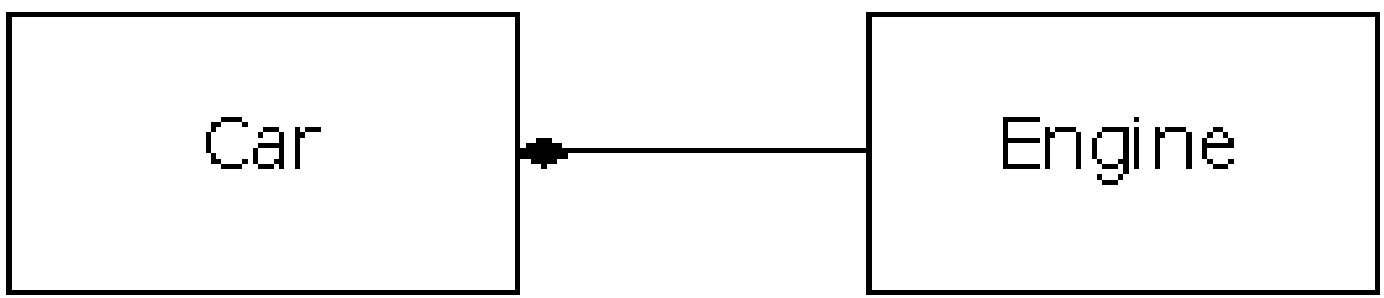
- Werkzeugkasten für die Benutzung einer Klasse
- Öffentliche Methoden und Variablen einer Klasse
- C++-Schlüsselworte: public, protected

Klassenprogrammierung

- Verborgene Klassenelemente, die interne Charakteristika einer Klasse definieren
- Private Methoden und Variablen einer Klasse
- C++-Schlüsselwort: private

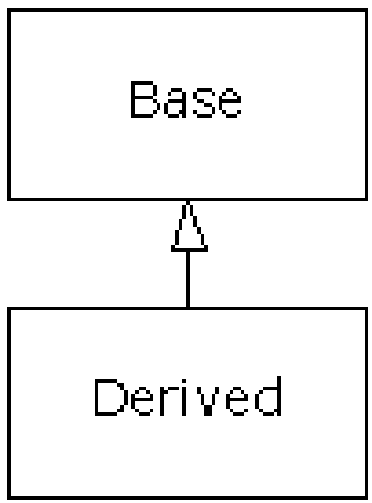
Wiederverwendung

- Objekte können aus anderen Objekten zusammengesetzt sein.
- Zusammensetzung heißt auch Komposition.
- Die Objektrelation heißt „has-a“
- Beispiel: „a car has an engine“
- UML-Notation (Unified Modelling Language)



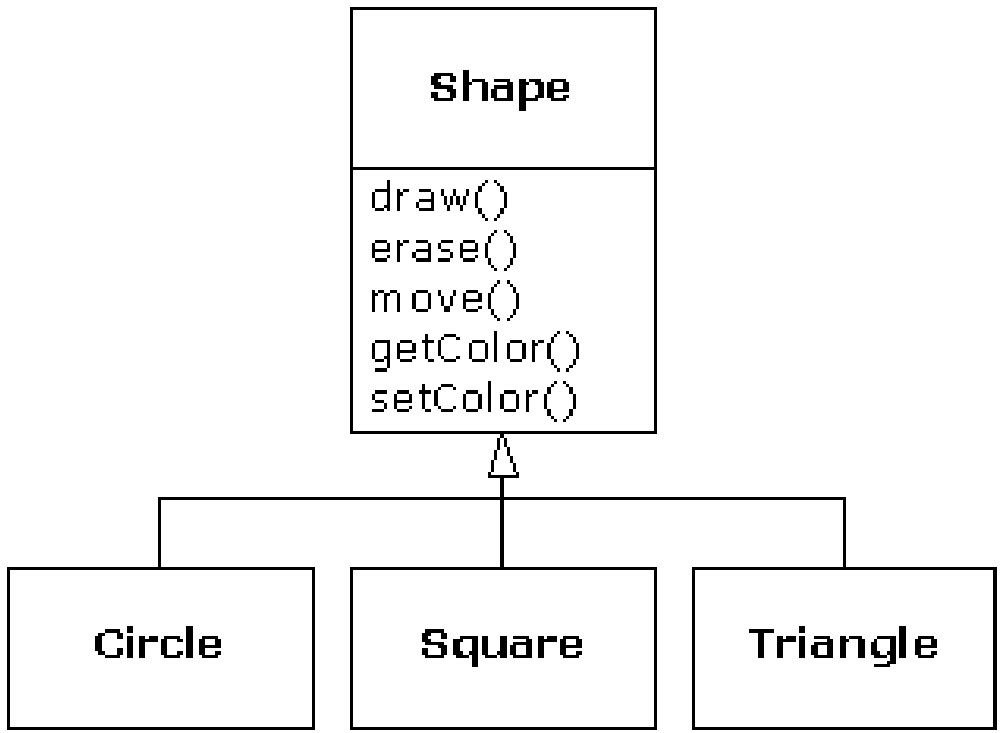
Vererbung...

- Wiederverwendung der Schnittstelle
- Eine neue Klasse kann aus einer vorhandenen Klasse mit ähnlichen Eigenschaften erzeugt werden.
- Erzeugung eines „Klons“ einer Klasse und anschließende Veränderung oder Ergänzung der Charakteristika.
- Vorhandene Klasse: Elternklasse, Basisklasse, Superklasse
- Neue Klasse: Kindklasse, abgeleitete Klasse, Subklasse



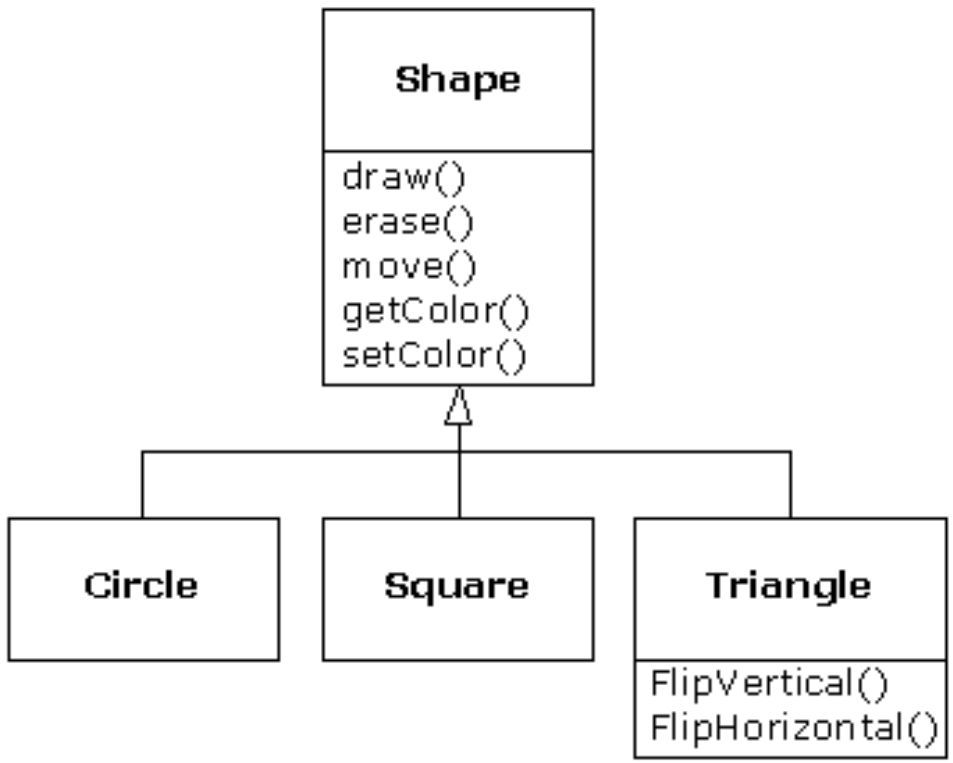
. . .Vererbung. . .

- Es kann mehrere Subklassen einer Superklasse geben.
- Subklassen duplizieren die Schnittstelle.



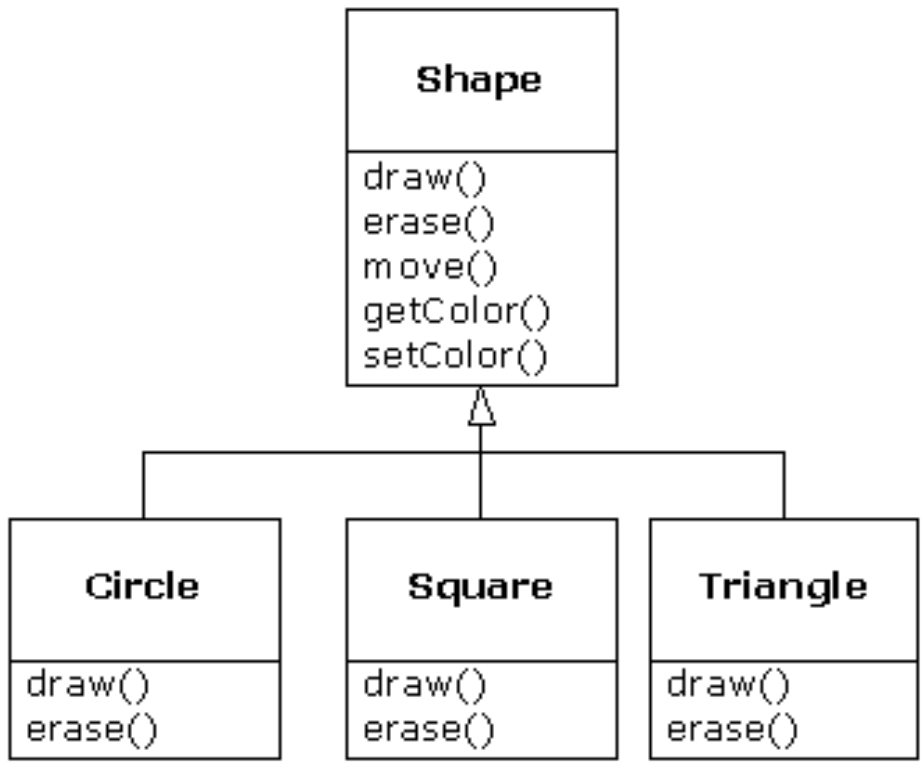
. . .Vererbung. . .

- Subklassen können um Methoden und Variablen ergänzt werden.
- Alle Methoden und Variablen der Superklasse bleiben unverändert erhalten.



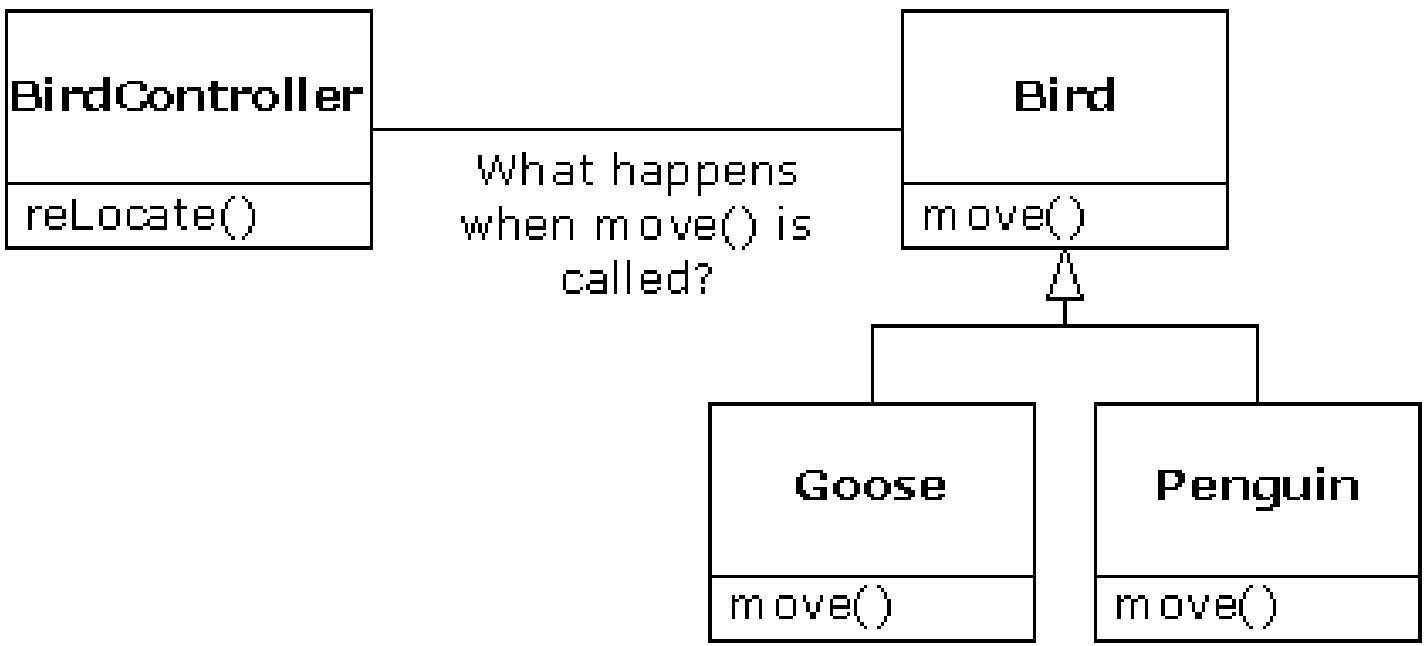
. . .Vererbung

- Subklassen können Methoden verändern (Overriding, Overloading).
- Die abgeleitete Klasse hat ein anderes Verhalten als die Basisklasse.



Polymorphie

- Methoden eines generischen Objekts (z. B. draw für shape) sind zur Compile-Zeit nicht bekannt.
- Späte Bindung (late binding)
- C++-Konzept und Schlüsselwort: virtual



Einleitung

- Eine Klasse fasst Datenstrukturen und zugehörige Funktionen zusammen
- Sie kann Datenelemente (Member-Variablen) und Funktionselemente (Member-Funktionen, Methoden) enthalten
- Member-Variablen repräsentieren einen Typ und Member-Funktionen repräsentieren Operationen, die auf dem Datentyp ausführbar sind

Einleitung

- Member-Variablen und Member-Funktionen stellen die Schnittstelle einer Klasse dar
- Teile der Klasse können entweder verborgen (gekapselt) oder explizit verfügbar gemacht werden; so kann eine Klasse ihre Datenstruktur vollständig von der Umgebung isolieren
- Eine Instanz einer Klasse heißt ein Objekt

Klassendeklaration...

class *klassenbezeichner* {
[private:]
    private daten und funktionen
[protected:]
    geschützte daten und funktionen
[public:]
    öffentliche daten und funktionen
};

. . .Klassendeklaration

- Eine Klasse definiert einen Datentyp, der zur Erzeugung von Objekten dient
- *private daten und funktionen* sind nur innerhalb der Klasse, d. h. für Funktionen innerhalb der Klasse, zugänglich
- *öffentliche daten und funktionen* sind auch von Funktionen außerhalb der Klasse zugänglich
- Wird keines der Schlüsselworte `private`, `protected` oder `public` angegeben, so werden alle Daten und Funktionen als privat angenommen

Definition und Deklaration

- Funktionen (Member-Funktionen) einer Klasse können bei der Klassendeklaration
  - vollständig implementiert sein
  - als Prototypen deklariert sein
- Bei Implementierung der als Prototypen angegebenen Funktionen ist als Qualifikator der Klassenbezeichner voranzustellen

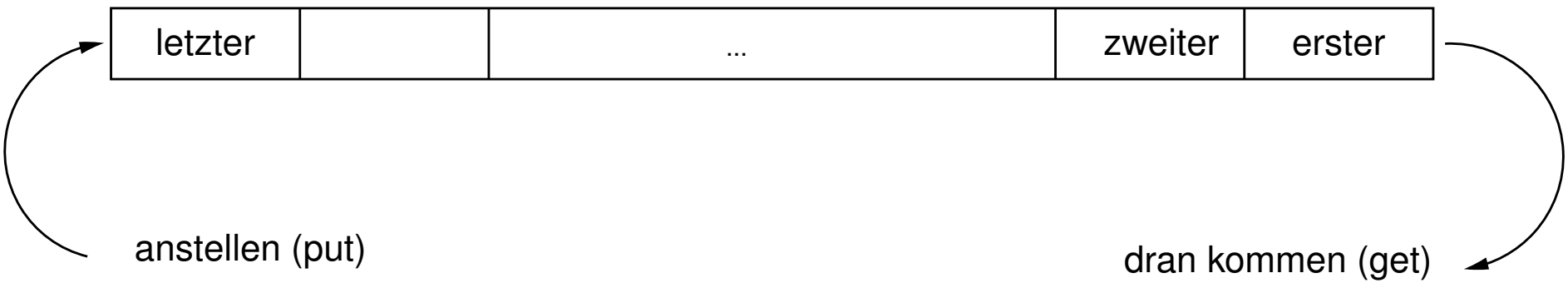
Klassendeklaration

```
class klasse {
    ...
    int func (...); // Prototyp
    ...
};
```

Implementierung

```
int klasse::func (...)
{
    ...
}
```

Datentyp Schlange



Beispiel Schlange q.h

```
/* FIFO Datenorganisation
 * Datentyp int
 * Dateneingabe: put (element)
 * Datenausgabe: get ()
 */
const unsigned int QLEN = 5; // maximale Laenge
class queue
{
private:
  unsigned int laenge; // aktuelle Laenge
  int data[QLEN]; // Elemente der Schlange
  int erster, letzter;
public:
  queue (); // Konstruktor, Initialisierung
  void put (int element);
  int get (void);
};
```

Beispiel Schlange q.cpp...

```
#include <iostream>
#include "q.h"
using namespace std;
queue::queue ()
{ erster = letzter = laenge = 0;
  cerr << "Schlange initialisiert" << endl;
}
void queue::put (int element)
{  if (laenge == QLEN) {
     cerr << "Schlange ist voll" << endl;
  } else
  { data[letzter] = element;
    letzter = (letzter + 1) % QLEN;
    ++laenge;
  }
  return;
}
```

...Beispiel Schlange q.cpp

```
int queue::get (void)
{ int element;
  if (! laenge)
  { cerr << "Schlange␣ist␣leer" << endl;
    return (0);
  } else
  { element = data[erster];
    erster = (erster + 1) % QLEN;
    --laenge;
    return(element);
  }
}
```

Beispiel Schlange qmain.cpp...

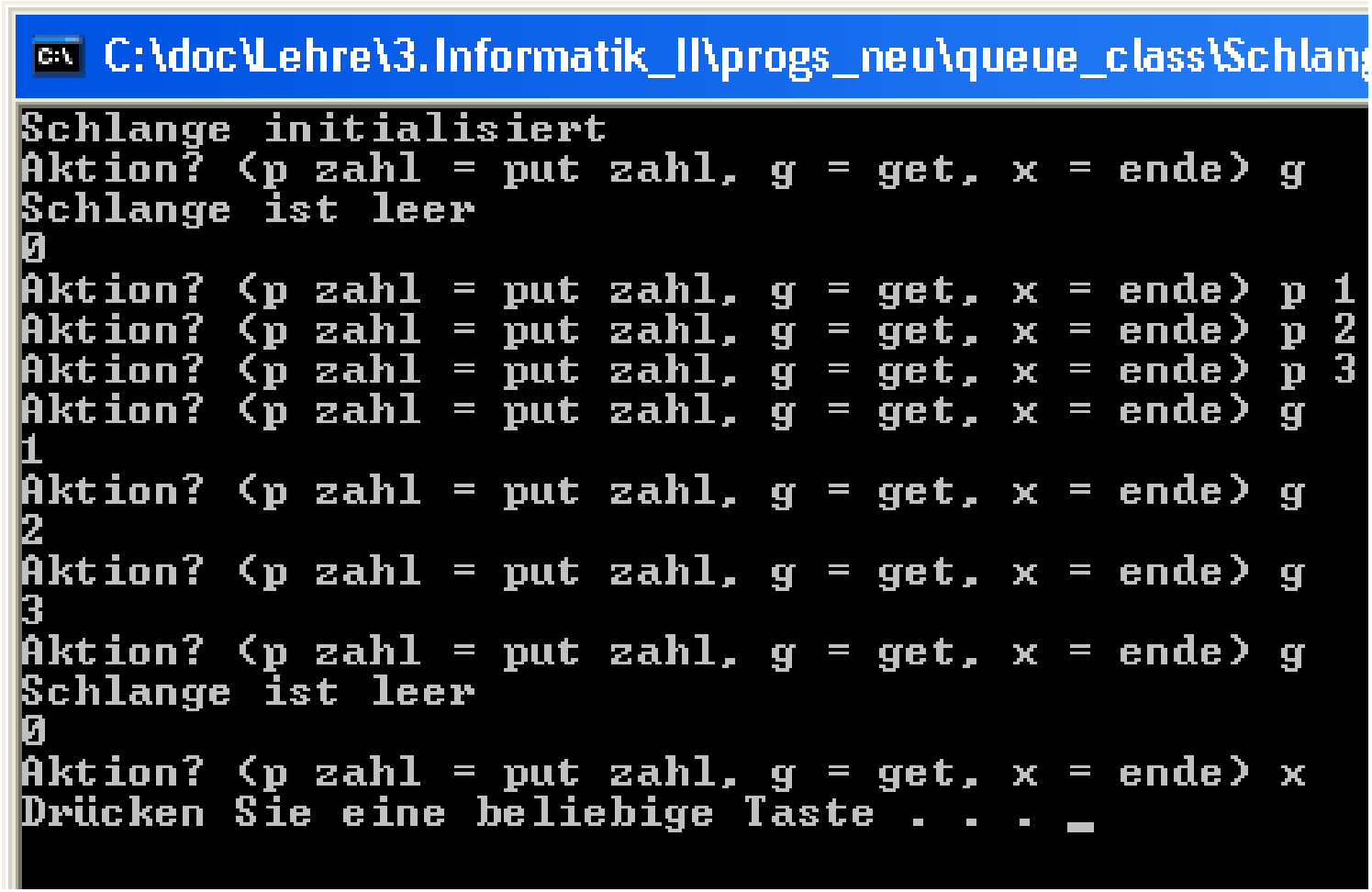
```
#include <iostream>
#include "q.h"
using namespace std;
int main ()
{
  queue a; // Erzeugung einer Instanz
  int element;
  char aktion;
  // p zahl = put zahl, g = get, x = ende

  do
  { cout << "Aktion? ";
    cout << "(p zahl = put zahl, g = get, x = ende) ";
    cin >> aktion;
```

...Beispiel Schlange qmain.cpp...

```
    switch (aktion)
    { case 'x':
        break;
      case 'p':
        cin >> element;
        a.put (element);
        break;
      case 'g':
        cout << a.get () << endl;
        break;
      default:
        continue; // Neue Eingabe
    } // end switch
  } while (aktion != 'x');
  return (0);
}
```

...Beispiel Schlange qmain.cpp

Programmausgabe



Spezielle Member-Funktionen

- Jede Klasse enthält mindestens vier Member-Funktionen
  1. Default constructor. Dient der Initialisierung
  2. Copy constructor. Dient der Duplizierung von Klasseninstanzen (Variablen)
  3. Destructor. Dient der Zerstörung von Klasseninstanzen
  4. Assignment operator. Dient der Zuweisung von Klasseninstanzen (Variablen)
- Falls diese Member-Funktionen nicht im Programm implementiert sind, werden sie vom Compiler automatisch generiert

Konstruktoren

- *klasse*::*klasse* ()
  Default constructor. Wird ausgeführt, wenn eine Instanz (Variable) von *klasse* deklariert wird; initialisiert alle Member-Variablen mit Zufallswerten
- *klasse*::*klasse* (const *klasse* &*org_klasse*)
  Copy constructor. Wird bei Call-by-value Parameterübergabe ausgeführt; kopiert alle Member-Variablen-Werte von *org_klasse* in die neue Instanz
  Kann auch zur Duplizierung von Instanzen verwendet werden:

```
klasse var1;
...
klasse var2 (var1); // var2 ist Kopie von var1
```

- Konstruktoren haben keine Typangabe (als Ergebniswert)

Destruktor und Zuweisung

- *klasse*::~*klasse* ()
  Destructor. Wird ausgeführt, wenn eine Instanz zerstört wird; erfolgt, wenn eine Instanz ihren Gültigkeitsbereich verläßt
- *klasse klasse*::operator = (const *klasse &org_klasse*)
  Zuweisungsoperator. Dient der Zuweisung von Instanzen von *klasse*; kopiert alle Member-Variablen von *org_klasse* in die neue Instanz

```
klasse var1;
klasse var2;

var1 = var2; // 'operator =' wird aufgerufen
```

- Destruktoren haben keine Typangabe (als Ergebniswert)

Konstruktor mit Parameter...

- Es kann mehrere Konstruktoren für eine Klasse geben
- Typischer Einsatz bei parametrisierten Objekten

```
class queue {
private:
  unsigned int laenge; // aktuelle Laenge
  int data[QLEN]; // Elemente der Schlange
  int erster, letzter;
  int name; // Kennung der Schlange
public:
  queue (); // Schlange der Laenge QLEN
  queue (int id); // Nummerierung der Instanzen
  void put(int element);
  int get(void);
};
```

...Konstruktor mit Parameter

```
// Konstruktor mit Parameter
queue::queue(int id)
{
  erster = letzter = laenge = 0;
  name = id;
  cerr << "Schlange " << name << " initialisiert"
       << endl;
}
```

Statische Member-Variablen...

- Eine Member-Variable gilt für Instanzen einer Klasse, d. h., jede Instanz besitzt ihre eigene Variable
- Eine statische Member-Variable gilt für die Klasse, d. h., es gibt nur eine solche Variable für alle Instanzen
- statische Member-Variablen können z. B. zum Zählen von Instanzen einer Klasse verwendet werden

...Statische Member-Variablen

```
class queue {
private: /* ... */
public:
  static int anzahl; // Anzahl von Schlangen
  queue () // Implementierung in Def.
  { ++anzahl; // Neue Schlange
    // ...
  }
  ~queue () // Implementierung in Def.
  { // ...
    --anzahl; // Eine Schlange weniger
  }
};
int queue::anzahl = 0; // Initialisierung
queue q1, q2; // queue::anzahl nun 2
// Zugriff
cout << "Anzahl von Schlangen: " << queue::anzahl;
```

Statische Member-Funktionen...

- Sind statische Member-Variablen privat, so kann vom außen nicht auf sie zugegriffen werden
- Zum Zugriff auf private statische Member-Variablen von außerhalb einer Klasse gibt es statische Member-Funktionen

...Statische Member-Funktionen

```
class queue {
private:
  // Aktuelle Anzahl von Schlangen
  static int anzahl;
  // ...
public:
  static int schlangen_anzahl (void)
  {
    return(anzahl);
  }
  // ...
};

// Zugriff
cout << "Aktive␣Schlangen:␣"
     << queue::schlangen_anzahl ();
```

Bedeutung von `static`

`static`-Deklaration	
Benutzung	**Bedeutung**
Variable außerhalb Funktion	Gültigkeitsbereich der Variable ist die Datei, in der sie steht
Variable innerhalb einer Funktion	Variable ist permanent, sie wird nur einmal initialisiert und nur einmal erzeugt, auch wenn die Funktion rekursiv aufgerufen wird
Funktion	Gültigkeitsbereich der Funktion ist die Datei, in der sie steht
Member-Variable	Nur eine Variable pro Klasse (nicht eine pro Instanz)
Member-Funktion	Funktion kann nur auf statische Member-Variablen der Klasse zugreifen

Initialisierung von Member-Variablen...

- Member-Variablen können in Konstruktoren statt durch Zuweisungen direkt initialisiert werden
-

```
class drei_d {
private:
  int x, y, z; // Koordinaten
public:
  drei_d(int a, int b, int c) : x(a), y(b), z(c) {}
};
```

entspricht

```
class drei_d {
private:
  int x, y, z; // Koordinaten
public:
  drei_d(int a, int b, int c) {x=a; y=b; z=c;}
};
```

. . .Initialisierung von Member-Variablen. . .

- Bei konstanten Member-Variablen kann eine Zuweisung nicht erfolgen
- Die Initialisierung mit dem ":"-Konstrukt ist dann erforderlich
- Initialisierung geschieht in Reihenfolge der Deklaration!

...Initialisierung von Member-Variablen

Richtig:

```
class vektor {
private:
  const int streckfaktor;
  int x, y, z; // Koordinaten
public:
  vektor(int s) : streckfaktor(s),
    x(0), y(0), z(0) {}
  vektor(int s,
      int a, int b, int c) :
    streckfaktor(s),
    x(a*streckfaktor),
    y(b*streckfaktor),
    z(c*streckfaktor)
    {}
};
```

Falsch:

```
class vektor {
private:
  int x, y, z; // Koordinaten
  const int streckfaktor;
public:
  vektor(int s) : streckfaktor(s),
    x(0), y(0), z(0) {}
  vektor(int s,
      int a, int b, int c) :
    streckfaktor(s),
    x(a*streckfaktor),
    y(b*streckfaktor),
    z(c*streckfaktor)
    {}
};
```

Selbstreferenz...

- Bei jedem Aufruf einer nicht statischen Member-Funktion wird automatisch ein Zeiger, der auf das aktuelle Objekt zeigt, übergeben; der Zeiger heißt `this`
- Der `this`-Zeiger ist ein impliziter Parameter jeder nicht statischen Member-Funktion
- Der `this`-Zeiger wird als Selbstreferenz bezeichnet
- Der Ausdruck `*this` bezieht sich auf das Objekt, für das die Member-Funktion aufgerufen wurde.

. . .Selbstreferenz

- Bei einer nicht konstanten Member-Funktion der Klasse `C` hat `this` den Typ `X*`.
- Bei einer konstanten Member-Funktion der Klasse `C` hat `this` den Typ `const X*`.
- Die Adresse von `this` ist nicht ermittelbar und `this` kann nichts zugewiesen werden.

Operator-Overloading...

- Für jede Klasse können Operatoren umdefiniert werden. Die Umdefinition heißt Überladen (Overloading)
- Das Überladen geschieht analog zu der Weise, wie bei normalen Datentypen
- Die Semantik von Operatoren kann nutzerdefiniert überladen werden, nicht dagegen deren Signatur, Priorität und Assoziativität

. . .Operator-Overloading

- Es ist nicht möglich, neue Operatoren einzuführen (z. B. `** %$@#`)
- Überladbar sind die folgenden Operatoren:

```
[]  ()  ->  ++  --  &  *  +
-  ~  !  /  %  <<  >>  <
>  <=  >=  ==  !=  ^  |  &&
||  =  *=  /=  %=  +=  -=  <<=
>>=  &=  ^=  |=  ,  new  delete
```

- Nicht überladbar sind `.  .*  ::  ?:`

Beispiel Raumkoordinaten `dreid.h...`

- Ganzzahlige Raumkoordinaten als Klasse
- Neben Konstruktor und Ausgabefunktion sind Zuweisungsoperator (=) sind die Inkrementierungsoperatoren (++) als Overloading realisiert
- Inkrementierungsoperatoren sowohl als präfix als auch suffix
- Inkrementierungsoperatoren benutzen den impliziten Parameter `this`

...Beispiel Raumkoordinaten dreid.h

```
class drei_d {
private:
  int x, y, z; // Koordinaten
public:
  drei_d () // Konstruktor
  {
    x = y = z = 0;
  };
  drei_d (int i, int j, int k) // Konstruktor
  {
    x = i; y = j; z = k;
  };
  drei_d operator = (const drei_d op2);
  drei_d operator ++ (void); // praefix
  drei_d operator ++ (int nil); // suffix
  void show(void);
};
```

Beispiel Raumkoordinaten dreid.cpp...

```
#include <iostream>
#include "dreid.h"
using namespace std;
// Overload =
drei_d drei_d::operator = (const drei_d op2)
{
    x=op2.x; y=op2.y; z=op2.z;
    return *this;
}
// Overload praefix ++
drei_d drei_d::operator ++ (void)
{
    ++x; ++y; ++z; // ++ fuer int
    return *this;
}
```

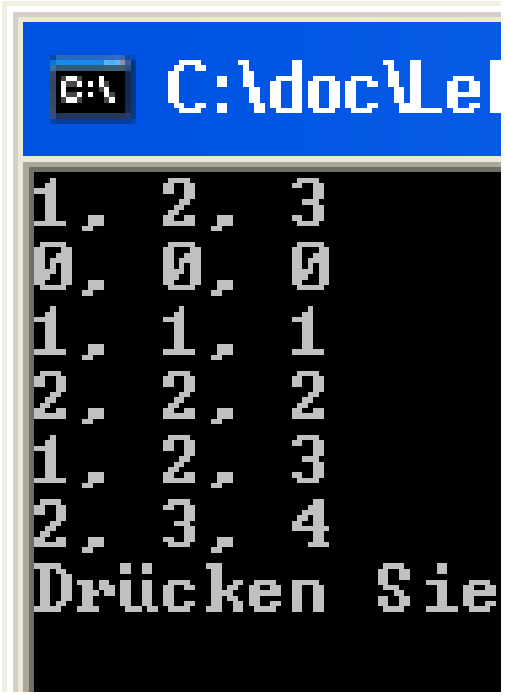
...Beispiel Raumkoordinaten dreid.cpp

```
// Overload suffix ++
// Parameter int nil wird nicht benutzt! Bewirkt suffix
drei_d drei_d::operator ++(int nil)
{
    drei_d erg=*this;
    ++x; // ++ fuer int
    ++y; // ++ fuer int
    ++z; // ++ fuer int
    return erg;
}
// Ausgabe einer 3D-Koordinate
void drei_d::show (void)
{
    cout << x << ", " << y << ", " << z << endl;
}
```

Beispiel Raumkoordinaten dreidmain.cpp

```
#include "dreid.h"
int main ()
{
  drei_d x (1,2,3), y;

  x.show ();
  y.show ();
  ++y;
  y.show ();
  y++;
  y.show ();
  y=x++;
  y.show ();
  x.show ();
  return (0);
}
```

Ausgabe



Klassen, Teil 1

- Objektorientierung modelliert das Problem.
- Objekte besitzen einen Zustand.
- Objekte können Nachrichten senden und empfangen.
- Objekte können Aktionen ausführen.
- Objekte können hierarchisch strukturiert sein.
- Objekte werden durch Klassen beschrieben und definiert.
- Es gibt spezielle Member-Funktionen für die Konstruktion und Destruktion von Objekten.
- Objekte können auf sich selbst Bezug nehmen (`this`).