6 Objektorientierung und weitere Konzepte

Lernziele:

Klassenhierarchien durch Generalisierung/Spezialisierung
Verarbeiten verschiedener Objekte mit dyn. Bindung
virtuelle Methoden, abstrakte Klassen
Generizität und Templates

...

Vererbung zwischen Klassen

ADTs und Klassen

– Datenabstraktion
  - ADO simuliert durch Header- und Implementation-File
    - als “Modul”
  - ADT als Klasse in C++
    - Schnittstelle muß nicht simuliert werden
    - Rumpf wird “simuliert”

– Schnittstelle von Klassen enthält
  - public-Teil: für alle potentiellen Klienten
  - private-Teil: nur für die Klassenimplementation
  - protected-Teil: für spezialisierte Klassen (s.u.)

– kein Typbezeichner wird exportiert
  - Klasse = Typ
  - kein formaler Parameter für das gesamte Objekt
  - `this` im Rumpf kennzeichnet das nicht vorhandene Formalparameterobjekt

– Begriffe der objektorientierten Programmierung
  - Methode: (Realisierung der) Zugriffsoperation
  - Botschaft: Aufruf einer Zugriffsoperation/Methodenaufruf
  - Punktnotation (oder Pfeil) für Botschaft an Objekt

```
// angestellter.h
#ifndef __INC_ANGESTELLTER_H__
#define __INC_ANGESTELLTER_H__

class Angestellter {
public:
   Angestellter (char *name, int tarif);
   // ...
   char *gibName();
   float gehalt();
   // ...

private:
   char *name;
   int tarifgruppe;
};
#endif // __INC_ANGESTELLTER_H__

// Class Body Angestellter
#include ”angestellter.h”

Angestellter::Angestellter(char *name,
                           int tarif) {
   this–>name = name;
   tarifgruppe = tarif;
}

...

// End Body Angestellter
```

Oberklasse-Beispiel

```
//personal.h
#ifndef __INC_PERSONAL_H__
#define __INC_PERSONAL_H__

class Personal {
public:
   Personal(char *name);
   // ...
   char *gibNamen();
   virtual float gehalt();

private:
   char *name;
   // ...
};
#endif // __INC_PERSONAL_H__


// Class Body Personal
#include ”personal.h”

Personal::Personal(char *name) {
   this–>name = name;
}

char Personal::gibNamen() {
   return name;
}

float Personal::gehalt() {
   return 500.0; // Festbetrag
}

// End Body Personal
```

Unterklasse-Beispiel

```
#ifndef __INC_ANGESTELLTER_H__
#define __INC_ANGESTELLTER_H__
// Import Personal
#include ”personal.h”

class Angestellter:public Personal {
public:
   Angestellter(char *name, int tarif);
   // ...
   virtual float gehalt();
   // ...
private:
   // Anreicherungen gegenueber Personal
   int tarifgruppe;
};
#endif

// Class Body Angestellter
// Import Personal
#include ”personal.h”
// FROM Tarife IMPORT TarifFaktor, TarifTab
#include ”tarife.h”
#include “angestellter.h”

Angestellter::Angestellter (char *name,
                int tarif):Personal(name) {
   tarifgruppe = tarif;
}

// Gehaltsberechnung für Angestellter
float Angestellter::gehalt(void) {
   return (TarifFaktor*TarifTab[tarifgruppe]
      + Personal(*this).gehalt());
}
// End Body Angestellter
```

```
#ifndef __INC_LEITENDERANGESTELLTER_H__
#define __INC_LEITENDERANGESTELLTER_H__
//Import Angestellter
#include ”angestellter.h”

class LeitenderAngestellter:
          public Angestellter {
public:
   LeitenderAngestellter(char* name, int tarif
                         float zul);
   float gehalt();

private:
   // ...
   float zulage;
   // ...
};
#endif

// Class Body LeitenderAngestellter
// Import Angestellter
#include ”angestellter.h”
#include ”LeitenderAngestellter.h”

LeitenderAngestellter::LeitenderAngestellter
      (char *name, int tarif,
       float zul):Angestellter(name, tarif) {
   zulage = zul;
}

// Gehaltsabrechnung fuer leitende Angestellte
float LeitenderAngestellter::gehalt() {
  return zulage+Angestellter(*this).gehalt();
}
// End Body LeitenderAngestellter
```

Idee der Vererbung, Nutzung, Gefahren

– Vererben von Eigenschaften, Hinzufügen spezieller Eigenschaften: Datenaufbau und Methoden.
  Oberklasse, Unterklasse: Oberklasse vererbt, Unterklasse erbt; Unterklasse ist “abgeleitete” Klasse.

– Vererbungshierarchie wichtige Struktureigenschaft von Systemen:
  Einfachvererbung, Mehrfachvererbung,
  Spezialisierung, Generalisierung

– zur Wiederverwendung von Eigenschaften (Schnittstelle)
  zur Wiederverwendung von Code (Gehaltsberechnung)

– Gefahren der OOP
  Klassen für alles (nicht nur für abstrakte Datentypen)
  Wilde Beziehungen zwischen Klassen
  Umstrukturierung von Vererbungshierarchien schwierig
  dynamisches Binden erzeugt bei nichtdisziplinierter Programmierung unzuverlässige Programme

```
// FROM iostream IMPORT cout, <<
#include <iostream.h>
// Import Angestellter
#include ”angestellter.h”
// Import LeitenderAngestellter
#include ”leitenderangestellter.h”

void main () {
   Personal *a =
       new LeitenderAngestellter(”Meyer”,
                                 4, 2000);
   Personal *b =
       new Angestellter(”Mueller”, 2);
   Personal *c;

   cout << a->gehalt() << endl; // mit Zulage
      // fuehrt zu Aufruf der speziellen
      // Methode unter Nutzung der Methode der
      // Oberklasse, die wiederum die Methode
      // von Personal benutzt
   cout << b->gehalt() << endl; // ohne Zulage
   c = Angestellter(*a);
   cout << c->gehalt() << endl; // mit Zulage
}
```

Gemeinsame Verarbeitung und dyn. Bindung

Dispatching

– Jedes Objekt eines Typs aus einer Klassenhierarchie hat best. Typ während gesamter Existenz

– gemeinsame Verarbeitung ohne Fallunterscheidung
  s. obiges Beispiel
  Gehaltsberechnung je nach Typ angesprochen,
  macht jeweils etwas anderes

– Anspringen der spezifischen Gehaltsberechnung aufgrund einer Laufzeitkennung für Typ (”tag”)

Verarbeitung von Objekten aus einer Vererbungshierarchie

geg. Vererbungshierarchie

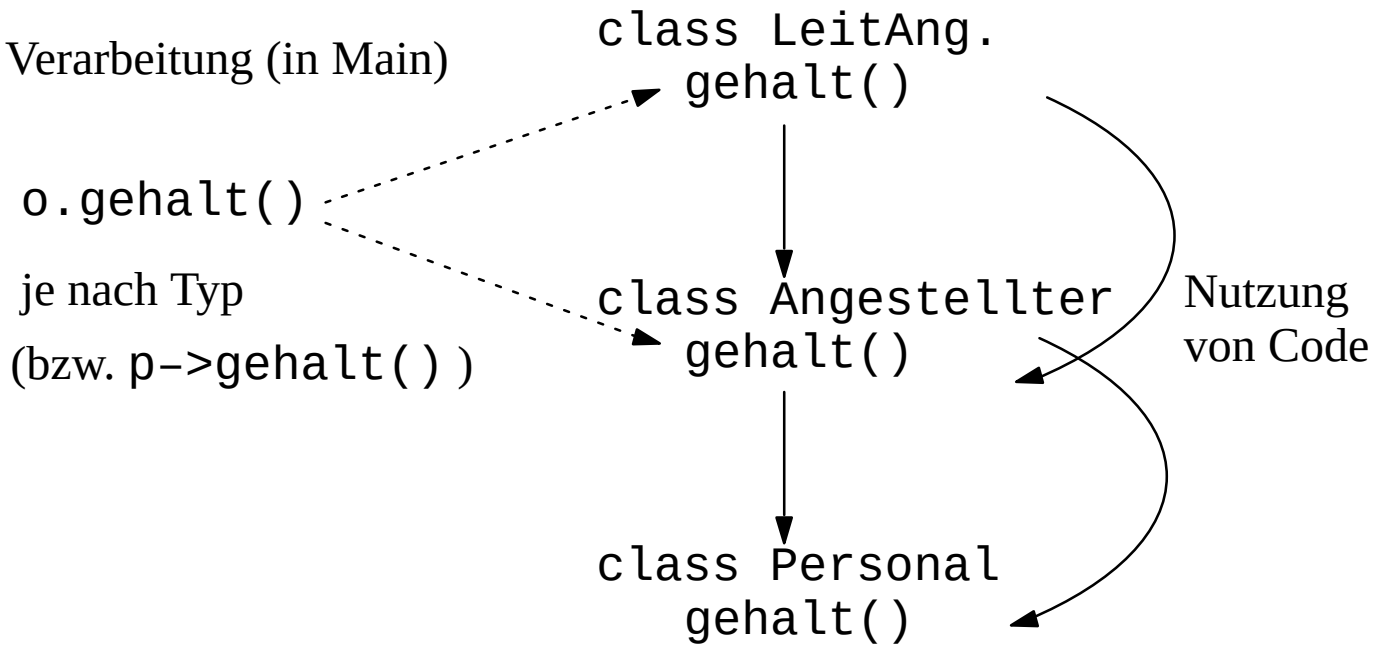
```
typedef Personal *PersZeiger;

void GehaltsBestimmung(Buffer *b) {
   while (b–>nonempty()) {
      PersZeiger p;
      b–>dequeue(p);
      cout << “Name: “ << p–>gibNamen();
      // ruft gibNamen() von Personal auf
      cout << “   Gehalt: “ << p–>gehalt();
      // Dispatching: gehalt() von Personal,
      // Angestellter oder Leitender–
      // Angestellter wird aufgerufen.
      cout << endl;
   }
}

class Buffer { // zur Ablage von Personen
public:        // aus Vererbungshierarchie
   Buffer();
   void enqueue(PersZeiger x);
   void dequeue(PersZeiger &x);
   bool nonempty();
   bool nonfull();
private:
   // ...
   Personal *bc[N]
};

// class body
   ...
   // z.B. zirkulaere Speicherung;
   // man beachte, dass das zirk. Feld jetzt
   // nur Zeiger auf Personen enthaelt
// end class body
```

Dispatching (dyn. Bindung zur Laufzeit je nach Typ)



– Voraussetzung:
gleicher Name für Operationen zu Objekten verschiedenen Typs in verschiedenen Klassen

Name an der Wurzelklasse der Teilhierarchie
(hier `Personal`) mit `virtual` gekennzeichnet

Sichtbarkeit für Klassen

– Sichtbarkeit von Teilen der Exportschnittstelle
  private: nur im Rumpf der Klasse
  public: für alle Klienten der Klasse
  protected: nur für Spezialisierungen

– Sichtbarkeit der gesamten Schnittstelle
  Vererbung mit public: auch weitere Spezialisierungen können (nach Konvertierung) Operationen nutzen
  Vererbung ohne public ist private: Die Schnittstelle ist nur im Rumpf der Spezialisierung (nach Konvertierung) nutzbar

Virtuelle Methoden, abstrakte Klassen

– Vererbungshierarchien arbeitsteilig erstellt
  gemeinsames Protokoll bei Wurzelklasse festlegen
  abstrakte Klasse (hat keine Objekte)
  Achtung: DA $\neq$ abstr. Klasse

– gemeinsames Protokoll
  Zur Handhabung der Personen eines Unternehmens
  Methoden: keine Mischung abstr. und konkreter Methoden

Einführung einer abstrakten Klasse

– keine Komponenten
  nur pure virtual Methoden

```
class PersProtokol {
public:
   virtual char *gibNamen() = 0;
   virtual float gehalt() = 0;
   ...
};


class Personal : public PersonalProtokoll {
public:
   Personal(char *n);
   char *gibNamen();
   ...
   virtual float gehalt();

private:
   char *name;
};


// Class body

float Personal::gehalt() {
   return 500.0; // Festbetrag
}
...

// End Class body
```

Generizität und Templates

Parametrisierung für Wiederverwendung

– Parameter für
  verschiedene Typen
  Funktionen mit best. Profil
  Dimensionierung

– folgendes Beispiel generischer Puffer
  Eintragstyp variabel
  und Behältergröße einstellbar

```
template<class T, int size>
class Buffer {
public:
  Buffer();
  void enqueue(T x);
  void dequeue(T &x);
  bool nonempty();
  bool nonfull();

private:
  // ...
  T bc[size];
};

// Generic Class Body
template<class T, int size>
Buffer<T, size>::Buffer() {
  // ...
}

template<class T, int size>
void Buffer<T, size>::enqueue(T x) {
  // ...
} // End Body
```

```
int main()
{
  int i;
  float f;

  Buffer<int, 3> intbuffer;
  Buffer<float, 2> floatbuffer;

  intbuffer.enqueue(1);
  intbuffer.enqueue(2);
  intbuffer.enqueue(3);
  floatbuffer.enqueue(1.111);
  floatbuffer.enqueue(2.222);

  intbuffer.dequeue(i); cout << i << endl;
  intbuffer.dequeue(i); cout << i << endl;
  intbuffer.dequeue(i); cout << i << endl;
  floatbuffer.dequeue(f); cout << f << endl;
  floatbuffer.dequeue(f); cout << f << endl;
  return 0;
}


/* Ausgabe:
1
2
3
1.111
2.222
*/
```

– heterogene Kollektion
  Kollektion für Objekte versch. Typs
  Typ aus Vererbungshierarchie
  Kollektion für Zeiger auf versch. Objekte

– Bsp. Puffer von vorne
  jetzt für versch. Vererbungshierarchien anwendbar

– Anwendbar für Vererbungsh. `PersProtokoll`
  oder etwa für graphische Objekte

```
template<class T, int size>
 void GehaltsBestimmung(Buffer<T, size>* b){
  while (b->nonempty()) {
    PersZeiger p;
    b->dequeue(p);
    cout << "Name: " << p->gibNamen();
    // ruft gibNamen() von Personal auf
    cout << "   Gehalt: " << p->gehalt();
    cout << endl;
    // Dispatching: gehalt() von Personal,
    // Angestellter oder Leitender-
    // Angestellter wird aufgerufen.
  }
}

int main()
{
  Personal *p=new Personal("Willi Wacker");
  Angestellter *a=
        new Angestellter("Ekel Alfred", 1);
            // Tarif 1
  LeitenderAngestellter *la=
        new LeitenderAngestellter
                ("Charlies Papa", 2, 100);
       // Tarif 2, Zulage 100

  Buffer<PersZeiger, 3>* personenbuffer =
        new Buffer<PersZeiger, 3>;

  personenbuffer->enqueue(p);
  personenbuffer->enqueue(a);
  personenbuffer->enqueue(la);
  GehaltsBestimmung( personenbuffer );
  return 0;
}
```

Ausnahmebehandlung (Exception handling)

Allgemeines

– Laufzeitfehler führen zu schwerwiegenden Störungen,
  oft nicht an der Stelle ihres Auftretens sondern später
  schwer zu erkennen

– Alle Fehlersituationen von Code abzufangen
  ist mühselig, oft gar nicht möglich

– Ausnahmebehandlung als Sprachkonzept
  Fehlerdefinition
  Fehlerbehandlung
  geregeltes Programmbeenden

– Trennung von
  normalem Ablauf
  und Fehlerbehandlung

– Fehler allgemeiner zu “Ausnahmen”
  für alle außergewöhnlichen Situationen
  außerhalb des normalen Ablaufs

– leider übl. Fehlersituationen
  Bereichsüberschreitungen bei Feldern
  Overflow, Underflow num. Datentypen
  nicht mit autom. Ausnahmeerweckung verbunden

– Für Fehler Hierarchie aus einfachen Klassen
haben entweder keine Struktur (meist)
oder int. Struktur dient zur Beschreibung des Fehlers

– Hierarchie vorab und global für Projekt festlegen:

```
class DataExc {
};

class Overflow  : public DataExc {
};
class Underflow : public DataExc {
};

...
```

– alternativ, weniger gut
Ausnahmen in Baustein festlegen und exportieren

– Fehler sind Objekte zu Fehlerklassen

```
// Fehlerklassen seien sichtbar

class Buffer {
  // Schnittstelle wie sonst, aber festl.,
  // welche Operationen welche Ausnahmen
  // auslösen können
  // ...
public:
   void enqueue(PersZeiger x);  // Overflow
   void dequeue(PersZeiger &x); // Underflow
  // ...
};

// Class Body
// ...

void Buffer::enqueue(PersZeiger x)
{
   if (n < N) {
      ...                  // Normalfall
   } else {
      throw Overflow(); // Ausnahme
   }
}

// dequeue() entsprechend mit throw Underflow

// End Body
```

```
try { // beliebige Anweisungen, die enqueue
      // und dequeue aufrufen
   ...
}
catch (Overflow) {
   // Ausnahmebehandlung durch Ausnahme–
   // behandler
   // irgendwelche Anweisungen zur
   // Begrenzung des Schadens
}
catch (Underflow) {
   // ...
   cout << “Fehler 37: Lesen einer leeren ”;
   cout << “Dateiliste.” << endl;
}
catch (...) { // irgendeine Ausnahme
   // ...
}
```

Weiterreichen von Ausnahmen

– gleiche Ausnahmen weitergereicht
  explizit durch `throw();`
  implizit, wenn kein passender Ausnahmebehandler vorhanden

– ist kein Ausnahmebehandler vorhanden, so wird das Gesamtprogramm abgebrochen

  ebenso, wenn “außerhalb” einer Ausnahmebehandlung keine mehr stattfindet und eine Ausnahme auftrat