Einführung in das objektorientierte Programmieren

Nils Schmeißer
Abteilung Kommunikation und Datenverarbeitung
Forschungszentrum Rossendorf e.V.

Inhalt

- Programmiersprachen
- Der objektorientierte Programmierstil
  - Begriffsbestimmung, Theorie, Anwendung
- OO Sprachen
  - C++ und Standardbibliotheken (STL, MFC)
  - Turbo-PASCAL, Java, Smalltalk, Eiffel, Oberon
- andere OO Systeme

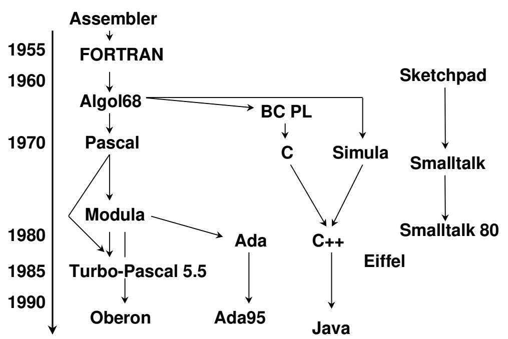
Literatur

- *„A Theory of Objects“*; M. Abadi, L. Cardelli
- *„Eiffel: The Language“*; B. Meyer
- *„Programming in Oberon: Steps beyond Pascal and Modula“*; M. Reiser, N. Wirth
- *“The C++ Programming Language”*; B. Stroustrup
- *„Java - Die Programmiersprache“*; K. Arnold, J. Gosling
- *„Objektorientierte Datenbanken“*; A. Heuer

Andere Quellen

- Unterlagen zum OOP/C++-Kurs:
  http://www.fz-rossendorf.de/FVTK/MITARB/schmei/doc/CPP-Kurs
- Unterlagen zum C-Kurs:
  http://www.fz-rossendorf.de/FVTK/MITARB/schmei/doc/C-Kurs

Geschichte der Programmierspr.



Programmiersprachen

- imperative Sprachen
  - Ada, ALGOL, BASIC, C, COBOL, FORTRAN, Modula-2, PL/1, Pascal, Simula, Snobol
- funktionale und applikative Sprachen
  - Lisp, Logo
- prädikative Sprachen
  - Prolog
- objektorientierte Sprachen
  - Smalltalk, Eiffel, Oberon, Java, Turbo-Pascal, C++, Fortran2000

imperative Sprachen 1

- ein Programm besteht aus einer Menge von *Befehlen*
  - BASIC:

```
10    let n=...
20    gosub 50
30    nf=nfak
40    end
50    let nfak=1
60    for i=1 to n
70      nfak=nfak*i
80    next i
90    return
```


imperative Sprachen 2

- prozedurale Sprachen gruppieren Befehle in Unterprogrammen
  - C:

```
int fac(int n) {
  if (n==0) return 1;
  return n*fac(n-1);
}
...
nf=fac(n);
```


funktionale und applikative Sp.

- ein Programm besteht aus einer Menge von Funktionen und deren Anwendung
  - Lisp:

```
(fac (n)
  (cond
    ((equal n 0) 1)
    (times n (fac
               (difference n 1)
             )
    )
  )
)
...
(fac n)
```


prädikative Sprachen

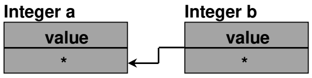
- in prädikativen Sprachen wird eine Menge von Fakten und Regeln vorgegeben
  - Prolog:

```
fac(0,1).
fac(n,f) -> fac(n-1,f/n).
```


objektorientierte Sprachen 1

- alle zum Lösen eines Problems nötigen Informationen werden als *Objekte* aufgefaßt
- Objekte empfangen und versenden Nachrichten

```
Integer a,b;     a  *  b
```



objektorientierte Sprachen 2

- Smalltalk:

```
Object subclass: #Integer
instanceVariableNames: 'value'
fac
  nfak:=Integer new.
  [0=value]
  ifTrue: [ nfak set: 1. ]
  ifFalse: [ nfak set:
     (((self minus: 1) fac: )
        times: value ). ]
  ^nfak.
!

n:=(Integer new) set: ...
n fac.
```

Der objektorientierte P.-Stil

- Basiskonzept (abstrakter Datentyp)
- Begriffsbestimmung
  - Objekt, Klasse, Instanz, Attribut, Methode
  - Vererbung
  - Polymorphie
- OOD (object oriented development)
- Kriterien für OO Sprachen

abstrakter Datentyp 1

- das Basiskonzept der objektorientierten Programmierung bildet der abstrakte Datentyp (ADT)
- ein ADT wird definiert durch:
  - seinen Wertebereich (Sorte)
  - die, über dem Wertebereich erklärten Operationen
  - eine Menge von Vorbedingungen
  - eine Menge von Axiomen

abstrakter Datentyp 2

- eine Sorte kann formal als Struktur dargestellt
- Beispiel:

```
koord=sort
   Integer x;
   Integer y;   <---- Komponenten
end;
```

bzw.

koord=x:Integer+y:Integer

abstrakter Datentyp 3

- einem ADT werden Operationen (die nicht notwendig nur Operanden vom Sortentyp enthalten) zugeordnet
- Beispiel:

```
vec=ADT(koord)
  SetX:(vec,Integer);
  +:(vec,vec)->vec;
  norm:(vec)->Integer;
end;
```


abstrakter Datentyp 4

- zur vollständigen Beschreibung eines ADT's gehören zusätzlich
  - die Vorbedingungen

```
pre: x=0; y=0;
```

  - eine Menge von Axiomen (Spezifikationen), die das Verhalten der Operationen festlegen

```
+ (vec a,vec b) =(x=a.x+b.x,y=a.y+b.y)
```


abstrakter Datentyp 5

- Beschreibung nach Meyer:

```
TYPE         vec[koord]
FUNCTIONS    SetX:   vec[koord] x Integer->vec[koord]
             +:      vec[koord] x vec[koord] -> vec[koord]
             norm:   vec[koord] -> Integer
CONDITIONS
AXIOMS               FOR ALL i:Integer,a,b,c:vec[koord]:
                     SetX(a,i) = a.x := i
                     +(a,b) = c:  ( c.x:=a.x+b.x,
                                    c.y:=a.y+b.y )
                     norm(a) = max(a.x , a.y)
```

abstrakter Datentyp 6

- abstrakte Datentypen werden als *Klassen* implementiert
- Beispiel:

```
class vec is
  Integer x,y;
  vec + (vec a,vec b) is
    return vec(a.x+b.x,a.y+b.y);
  end;
end;
```


Begriffsbestimmung 1

- *Objekt*: eigenständiges Individuum mit
  – charakteristischen Eigenschaften
  – einem inneren Zustand
  – Attributen
  – Methoden

Beispiel

- „Kreis“:
  – charakteristische Eigenschaften:
    - alle Punkte des Kreises habe den gleichen Abstand zum Mittelpunkt
  – innerer Zustand:
    - sichtbar/unsichtbar
  – Attribute:
    - Position, Radius, Farbe
  – Methoden
    - Darstellen (Zeichnen), Verschieben

Begriffsbestimmung 2

- *Klasse*:
  ist die Vereinigung und formale Beschreibung aller Objekte mit gleichen Eigenschaften
  – Klassen können als Implementation abstrakter Datentypen aufgefaßt werden
  – Klassen werden auch als “Objektfabriken” bezeichnet

Beispiel

```
class Kreisklasse is
  Punkt Mittelpunkt;
  Real Radius;
  Color Farbe;
  Kreis New(Punkt M0,Real R0,Color C0);
  Zeichnen();
  Verschieben(Punkt NeuerMittelpunkt);
end;
```


Begriffsbestimmung 3

- eine *Instanz* ist die konkrete Realisierung eines Objektes einer Klasse
  – eine Instanz ist genau ein Objekt
  – Instanzen haben einen Gültigkeitsbereich (Lebensbereich)
  – Instanzen werden durch einen *Konstruktor* erzeugt und einen *Destruktor* zerstört

Beispiel

```
program p1 is
  Kreis k3=Kreisklasse::New((4,5),3,Grün);
  subprogram ps1 is
    Kreis k1=Kreisklasse::New((1,2),3,Rot);
    Kreis k2=Kreisklasse::New((1,2),3,Rot);
    ...
  end;
...
end;
```

- obwohl k1 und k2 die gleichen Attributwerte haben, sind es voneinander verschiedene Objekte

Begriffsbestimmung 4

- *Attribute* sind die Variablen, die den inneren Zustand eines Objektes beschreiben
- Attribute korrespondieren in ihrer Bedeutung mit den Sorten des abstrakten Datentyps
- die zugehörige Klasse heißt *Komponentenklasse*

Beispiel

```
class Punktklasse is
  ...
end;
```

```
class Kreisklasse is
  Punkt Mittelpunkt;
  Real Radius;
  ...
end;
```

- die Klasse „Punktklasse“ ist eine Komponentenklasse von „Kreisklasse“

Begriffsbestimmung 5

- *Methoden* sind die über einer Klasse erklärten und ihr zugeordneten Operationen
- zwei spezielle Methoden
  - Konstruktor: generiert eine Instanz
  - Destruktor: zerstört eine Instanz
- ein Methodenaufruf ist Teil einer *Nachricht*, die an ein Objekt versandt wird

Beispiel

```
program p1 is
  Kreis k3=Kreisklasse::New((4,5),3,Grün);
  ...
  k3 . Verschieben((7,8)) ;
end;
```

- die gesamte, an k3 versendete Nachricht lautet:

  k3: „Verschiebe (dich) nach (7,8)“

Vererbung 1

- *Vererbung* ist die Weitergabe von Eigenschaften (Attributen und Methoden) einer *Basisklasse* an eine *abgeleitete Klasse*
- die Weitergabe von Werten ist ebenfalls möglich (*Klonen*)

Vererbung 2

- Vererbung gibt es in verschiedenen Ausprägungen:
  - *Spezialisierung*: Objektmenge der Unterklasse ist Teilmenge der Objektmenge der Oberklasse
  - *Is-a Hierarchie*: wie Spezialisierung
  - *Typhierarchie*: jedes Objekt des Untertyps verhält sich wie ein Objekt des Obertyps
  - *Klassenhierarchie*: Vererbung der Implementation

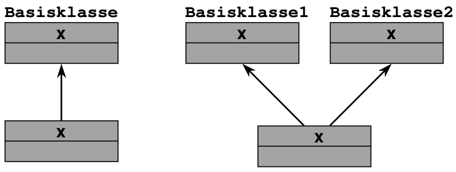
Beispiel

```
class Ellipsenklasse is
  Punkt Mittelpunkt;
  Real a,b;
  Ellipse New(Punkt M0,Real a0,Real b0);
  Verschieben(Punkt NeuerMittelpunkt);
end;

subclass Kreisklasse of Ellipsenklasse is
  Punkt Mittelpunkt;
  Real a,b;  // a=b
  Kreis New(Punkt M0,Real R0);
  Verschieben(Punkt NeuerMittelpunkt);
end;
```


Vererbung 3

- es wird ebenfalls zwischen einfacher und mehrfacher Vererbung unterschieden



Vererbung 4

- speziell auf die Weitergabe von Methoden bezogen ist
  - *Ersetzung*: das vollständige Überschreiben einer Methode
  - *Verfeinerung*: die ererbte Methode wird innerhalb der neuen Methode gerufen
  - *Delegation*: die Ausführung einer Methode wird an ein Objekt der Oberklasse (Prototyp) weitergereicht

Beispiel

- `Ellipsenklasse::Skalieren(Real fa,Real fb)is a=a*fa; b=b*fb; end;`
- *Ersetzung*:

```
Kreisklasse::Skalieren(Real f) is
  a=a*f; b=b*f; end;
```

- *Verfeinerung*:

```
Kreisklasse::Skalieren(Real f) is
  Ellipsenklasse::(f,1.0); b=b*f;
end;
```

- *Delegation*:

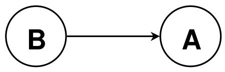
```
Kreisklasse::Skalieren(Real f) is
  Ellipsenklasse::(f,f);
end;
```


Vererbung 5

- manchmal kann es sinnvoll sein, die Erzeugung von Instanzen bestimmter Klassen zu verbieten (Festlegung eines Konzeptes); solche Klassen werden *abstrakte (Basis-)Klassen* genannt

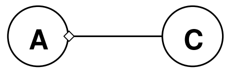
Beziehungen zwischen Klassen 1

- erbt eine Klasse B von einer Klasse A, so heißt A *Basisklasse* oder *Superklasse* von B, B heißt *Subklasse*
- B genügt der *is-a Relation* (B is-a A)



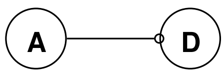
Beziehungen zwischen Klassen 2

- besitzt eine Klasse A ein Attribut vom Typ der Klasse C, so heißt C Komponentenklasse von A
- A genügt der *has-a Relation* (A has-a C)



Beziehungen zwischen Klassen 3

- benutzt eine Klasse A eine Instanz der Klasse D, so genügt A der *use-a Relation* (A use-a D)



- die Relationen sind vererbbar, also B has-a C und B use-a D

Polymorphie 1

```
class Gear is
  SwitchUp() is ... end;
end;
subclass ManualGear of Gear is
  SwitchUp() is ... end;
end;
subclass AutomaticGear of Gear is
  SwitchUp() is ... end;
end;
```

- die Methode **`SwitchUp()`** kann in vielen verschiedenen Formen auftreten, sie heißt *polymorph*

Polymorphie 2

- neben dem *universellen Polymorphismus* (vorhergehende Folie) gibt es den *ad-hoc Polymorphismus*

```
Integer Trunc(Integer x);
Integer Trunc(Real x);
```

- ad-hoc P. basiert auf dem Konzept der impliziten Typkonversion

Polymorphie 3

- das Konzept der Polymorphie erlaubt somit unterschiedliche Sichtweisen auf Objekte
- Beispiel:
  - eine Instanz der Klasse **`ManualGear`** ist ebenfalls mit den Eigenschaften der Klasse **`Gear`** ausgestattet, kann also wie eine solche agieren

Polymorphie 4

- eine Variable vom Typ einer Superklasse kann somit auch eine Instanz vom Typ einer Subklasse aufnehmen
- Beispiel:

```
Gear g;
g=new ManualGear;
```


Bindung 1

- der Bindungsmechanismus beschreibt, wie eine Methode einer Klasse aufgerufen wird
- verschiedene Bindungskonzepte erlauben eine effiziente Wiederverwendung bzw. gemeinsame Nutzung von Code auf sehr elegante Art und Weise

Bindung 2

- *statische Bindung*:
  - während der Übersetzung wird die Klassenzugehörigkeit der Instanz bestimmt und daraus die zu rufende Methode ermittelt
- Beispiel:

```
ManualGear g=new ManualGear;
g.SwitchUp();
```

ruft die Methode der Klasse **`ManualGear`**

```
Gear g=new ManualGear;
g.SwitchUp();
```

die der Klasse **`Gear`**

Bindung 3

- *dynamische (späte) Bindung*:
  - die Klassenzugehörigkeit der Instanz wird erst zur Laufzeit bestimmt
- Beispiel:

```
ManualGear g=new ManualGear;
g.SwitchUp();
```

ruft die Methode der Klasse **`ManualGear`**

```
Gear g=new ManualGear;
g.SwitchUp();
```

ruft ebenfalls die der Klasse Manual**`Gear`**

Bindung 4

- späte Bindung wird nicht von allen Sprachen unterstützt
- standardmäßig wird eine Methode statisch gebunden
- dynamisch zu bindende Methoden werden durch ein spezielles Schlüsselwort (meist “virtual”) gekennzeichnet
  → “virtuelle Methoden”

Bindung 5

```
class Gear is
  Init();
  virtual SwitchUp() is ... end;
end;
subclass ManualGear of Gear is
  Init();
  virtual SwitchUp() is ... end;
end;

Gear g = new ManualGear;

g.Init();      // Gear::Init
g.SwitchUp();  // ManualGear::SwitchUp
```

Beispiel 1

- ein Verfahren zum Lösen von Gleichungssystemen

  $$A\,x = b$$

  wird implementiert
  - ist A dünn besetzt, so werden die nicht-0 Elemente normalerweise als Liste abgelegt, um Speicherplatz zu sparen
    ➔ der Löser müßte neu implementiert werden

Beispiel 2

- die Neuimplementation ist nicht nötig, wenn der Löser über

```
class Matrix is
  int n,m;
  Datatype data[maxz][maxs];
  virtual real Get(int z,int s);
  virtual void Set(int z,int s,real v);
end;
```

erklärt wird

```
Matrix Solver(Matrix A,Matrix b);
```


Beispiel 3

```
Matrix Solver(Matrix A,Matrix b) is
  int i,j,k;
  real f;
  Matrix x = new Matrix;
  for k = 0 to A.n do begin
    if abs(A.Get(k,k))<eps then swap();
    for i = k+1 to A.n do begin
      f=A.Get(i,k)/A.Get(k,k);
      for j = k+1 to A.m do
        A.Set(i,j,A.Get(i,j)-f*A.Get(k,j));
      b.Set(i,b.Get(i)-f*b.Get(k));
    end;
  end;
  … return x;
end;
```


Beispiel 4

- eine dünn besetzte Matrix kann dann z.B. als

```
subclass SparseMatrix of Matrix is
  int n,m;
  Datalist data;
  virtual real Get(int z,int s);
  virtual void Set(int z,int s,real v);
end;
```

erklärt werden

Beispiel 5

- der Löser kann auch dann auf SparseMatrix Objekte angewandt werden, wenn sein Quelltext nicht verfügbar ist (Bibliothek)

```
Matrix A = new SparseMatrix;
Matrix b = new Matrix;
Matrix x;
...
x=Solver(A,b);
```


Zusammenfassung

- abstrakter Datentyp
  - Sorten, zugeordnete Operationen
- Klassenkonzept
  - Klasse, Instanz, Objekt
- Vererbung
  - einfach, mehrfach
- Polymorphie
- Bindung
  - statische und späte Bindung, virtuelle Methode

Und warum das Ganze?

- Programme sollen:
  - sicher
  - sauber
  - wiederverwendbar
  - lesbar
  - schnell

  sein

Softwareentwurf 1

- Modularisierung
  - nach Funktionen:
    - » Funktionensammlungen, Bibliotheken
  - nach Daten bzw. Objekten:
    - » Klassenpool
- Top-down Entwurf
  - Anwendung → abstrakte Funktion
  - Zerlegung → festgelegter Systementwurf

Softwareentwurf 2

- Object Oriented Design
  - Identifiziere Objekte
  - Beschreibe Objekte
  - Beziehungen und Gemeinsamkeiten zwischen Objekten
  - Klassenbildung
  - Beziehungen zwischen Klassen
  - Klassenhierarchie
  - Methodenimplementation
  - Programm aus Objektbeschreibung

Kriterien für OO-Sprachen 1

- nach Meyer
  - Modularisierung nach Objekten
  - Abstrakte Datentypen
  - Freispeicherverwaltung (! C++)
  - Module, Klassen und Standardtypen
  - Vererbung
  - Polymorphismus und dynamische Bindung
  - Mehrfachvererbung (! Smalltalk)

Kriterien für OO-Sprachen 2

- nach Wegner
  - Objekte
  - Klassen
  - Vererbung

  - Strenge Typisierung
  - Abstrakte Datentypen
  - Nebenläufigkeit
  - Persistenz

OO-Sprachen

- “bedeutend”
  - Turbo-Pascal, Delphi, C++, Java, Smalltalk-80, Oberon-2, Eiffel, Ada 95
- weniger bedeutend
  - CLOS, Modula-3, Objective-C, Python, Sather, Simula, Tcl/Tk, VBScript, Visual Basic, Objective-Pascal

C++

Einführung in die Sprache und Nutzung der OO Eigenschaften

C++

- 1980-87 von B. Stroustrup bei AT&T entwickelt
- 1989 ANSI Standard
- C++ ist ein „besseres C"
- C++ implementiert den gesamten C Sprachumfang +
  - Klassen, Vererbung, späte Bindung

Sprachbeschreibung

- für die nächsten Folien soll folgende Notation gelten:
  - Schlüsselwörter werden gelb dargestellt
  - *Metavariable*, die beliebige andere syntaktische Strukturen verkörpern, werden kursiv gesetzt
  - in *[ ]* geklammerte Spracheinheiten sind optional zu verwenden
  - **`Code wird in Courier angegeben`**

C Grundlagen 1

- typisierte proz. imperative Sprache
- alle Variablen und Funktionen müssen deklariert und definiert werden
  - Deklarationen:

    vardecl ::= *[* aspec *] [* const *]* typename varname ;
    aspec ::= extern */* static */* register
- Standardtypen: int, float, double, char
  - Erweiterung: *[* signed */* unsigned *] [* long *]* stdtype
- Felder x[ ], Referenzen *x;

C Grundlagen 2

- Strukturen

  strukt ::= struct { komp1 ... kompn } varname ;
- Beispiel:

```
struct {
  int x,y;
  int color;
} point;

point.x=5;
```

Instanz

C Grundlagen 3

- Typdefinitionen

  typdef ::= typedef typedescr typename ;
- Beispiel:

```
typedef struct {
  int x,y;
  int color;
} Point ;

Point p;
```

```
struct Point {
  int x,y;
  int color;
};
```

C Grundlagen 4

- Steuerstrukturen
  - Verzweigung

    verzw ::= if (cond) anw *[* else anw *]*
  - Schleife

    forloop ::= for (zuw ; cond ; iter) anw
    awhile ::= do block while (cond) ;
    nwhile ::= while (cond) anw

C Grundlagen 5

```
if (x>=0) abs=x; else abs=-x;
```

```
for (i=0;i<n;i++) s=s+i;
```

```
do {
  xn=x-f(x)/fp(x);
  x=xn;
} while (abs(f(x))>0.2);
```

```
while (lock) ;
```


C Grundlagen 6

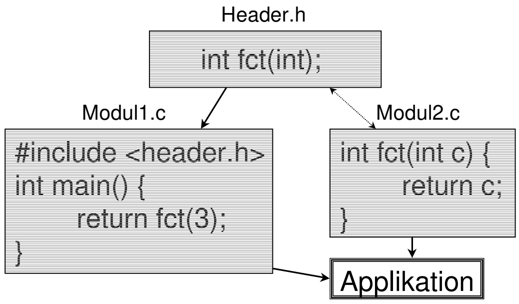
- Funktionsdefinitionen

  fdefdecl ::= aspec rettype fname (*[* arglist *]*) ;
  fdef ::= rettype fname (*[* arglist *]*) block
- Beispiel:

```
extern int fact(int x);
int fact(int x) {
  if (x==0) return 1;
  return x*fact(x-1);
}
```

  - eine nicht-*void* Funktion muß **immer** einen Wert zurückgeben (`return value;`)

C Grundlagen 7: Modulkonzept



C++ Spezifika

- neue Operatoren
  - new und delete
- ad-hoc Polymorphismus
  - Funktions- und Operatorüberladung
- Klassen
- Templates
- Streams

neue Operatoren in C++ 1

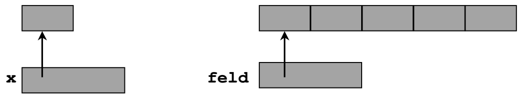
- new
  - allokiert die benötigte Menge Speicher für ein „Objekt“ vom angegebenen Typ
  - konstruiert die angegebene Anzahl von Instanzen
- delete
  - zerstört die referenzierten Instanzen
  - gibt einen reservierten Speicherbereich frei

neue Operatoren in C++ 2

■ Beispiel:

```
int *x;
x=new int;
...
delete x;
```

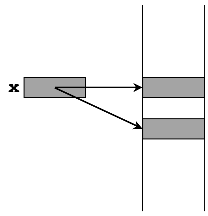
```
int n,*feld;
feld=new int [n];
...
delete feld;
```



neue Operatoren in C++ 3

■ Problem:

```
int *x=new int;
...
x=new int;
```



Beispiel

```
typedef struct {
  int n,m;
  double **data;
} Matrix;

Matrix NewMatrix(int n0,int m0) {
  Matrix M;
  M.n=n0; M.m=m0;
  M.data = new double* [M.n];
  for (int i=0;i<M.n;i++)
    M.data[i] = new double [M.m];
  return M;
}
```


ad-hoc Polymorphismus 1

■ Funktionen oder Operatoren, die sich nur durch ihre Semantik (Definitionsbereich), nicht aber durch ihre Syntax unterscheiden heißen *ad-hoc polymorph*

■ Beispiel: int abs(int)
double abs(double);

Funktionsüberladung

■ eine vorhandene Funktion

$$F{:}A_1xA_2x\ldots xA_n \rightarrow W_A$$

wird durch eine neue Definition

$$F{:}B_1xB_2x\ldots xB_m \rightarrow W_B$$

überladen ($A_1xA_2x\ldots xA_n \neq B_1xB_2x\ldots xB_m$)

■ z.B.:

```
int sum(int a,int b);
double sum(double a,double b);
```


ad-hoc Polymorphismus 2

■ Typkonversion:
- *integral promotion*: Aufzählungstypen können ohne Einschränkung ev. unter Informations-verlust ineinander konvertiert werden
- *integral conversion*: int ➔ unsigned type wird im 2er-Komplement das Bitmuster beibehalten
- *float-double*: ggf. unter Stellenverlust
- *floating-integral*: Streichung von Nachkommastellen (Keine Rundung!)

ad-hoc Polymorphismus 3

- existiert eine Funktion in mehr als einer Form, so wird aus den Typen der aktuellen Parameter der ➤ *best match* bestimmt

Niveau	Konversion
1	exact match, einfache Konversion
2	promotions, float-double
3	Standard Konversionen, Klassenzeiger Klassenreferenz
4	Nutzerdefinierte Konversion
5	ellipsis conversion

ad-hoc Polymorphismus 4

- Beispiel:

```
int round(int x);
int round(float x);
```

- `round(4);`
  - O.K.
- `round(1.2);`
  - Konflikt, da 1.2 double Konstante und *integral-promotion* und *float-double* auf selbem Niveau
- `round((float)1.2);`
  - O.K.

ad-hoc Polymorphismus & ADT

- definieren Sie Ihre Datenstrukturen als neue Typen
- erklären Sie Ihre Algorithmen als Funktionen über diesen Datentypen
- Beispiel: verwenden Sie

```
typedef double vec[3];
double norm(vec a);
```

anstatt

```
double norm(double a[3]);
```


Operatorüberladung 1

- ähnlich zur Funktionsüberladung können prinzipiell alle Operatoren überladen werden
- Einschränkungen:
  - es können keine neuen Operatoren erklärt werden
  - die Syntax eines Operators kann nicht geändert werden
  - die Semantik bestehender Definitionen kann nicht geändert werden
  - die Priorität bleibt erhalten

Operatorüberladung 2

- Syntax der Deklaration:

rtype operator opname (atype1 *[* , atype2 *]*);

- Beispiel:

```
Vector operator - (Vector a,Vector b);
```

- Definition wie Funktionsdefinition

```
Vector operator - (Vector a,Vector b) {
  Vector c; ...
  return c;
}
```


Operatorüberladung 3

```
typedef struct {
  unsigned int dim;
  double x[10];
} Vector;

Vector operator - (Vector a,Vector b) {
  if (a.dim!=b.dim) error();
  for (int i=0;i<a.dim;i++)
    c.x[i]=a.x[i]-b.x[i];
  c.dim=a.dim;
  return c;
}

main() {
  Vector a={3,1,2,3},b={3,4,5,6};
  Vector c=a-b;
}
```

Operatorüberladung 4

- überladen werden kann z.B. auch [], allerding muß [] in diesem Fall direkt an den Datentyp gebunden werden

```
typedef struct {
  ...
  double &operator [] (unsigned int i);
  ...
} Vector;
```

- → Strukturen

Strukturen 1

- Strukturen lassen sich als “Vorstufe” zum abstrakten Datentyp interpretieren
- Strukturen können Komponenten beliebigen Typs und Funktionen enthalten
- Beispiel:

```
struct MyStruct {
  int x;
  int Value();
};
```


Strukturen 2

- eine, einer Struktur zugeordnete Funktion heißt → *Member-function*
- Member-functions können nur über eine Variable des Typs, dem sie zugeordnet sind, aufgerufen werden

Beispiel:

```
struct MyStruct m;
Value();        unzulässig
m.Value();      O.K.
```


Strukturen 3

- Member-functions müssen zusammen mit ihrer Struktur definiert werden

```
int MyStruct :: Value() { return x; }
```

- Komponenten derselben Struktur brauchen nicht qualifiziert zu werden
- Strukturen können nicht vererben

Array-Subscription

```
typedef struct {
  unsigned int dim;
  double x[10];
  double &operator [] (unsigned int i);
} Vector;

double &Vector::operator [] (unsigned int i) {
  if ((i<0) || (dim<=i)) error();
  return x[i];
}
...
Vector a;

a[0]=5.0;
printf("%lf\n",a[0]);
```


Datenkapselung 1

- die Komponenten einer Struktur sind generell von überall her zugänglich
- Beispiel:

```
main() {
  struct MyStruct m;

  ...
  printf("%d\n",m.x);
  printf("%d\n",m.Value());
}
```

Datenkapselung 2

- um Daten zu schützen, können sie verborgen werden und sind dann nur noch innerhalb von Member-functions zugreifbar
- Komponenten werden durch den Vorsatz des Schlüsselwortes private geschützt
- alle Komponenten nach private sind geschützt

Datenkapselung 3

- Beispiel:

```
typedef struct {
  private: int x;
           int y;
  public:  int Value();
           void Set(int v);
} MyStruct;

void MyStruct::Set(int v) { x=v; }

main() {
  MyStruct m;
  m.x=1;     // nicht zulässig
  m.y=1;     // nicht zulässig
  m.Set(1);  // O.K.
}
```


Klassen

- Klassendefinition und Instanzierung
- Vererbung
  - einfache und mehrfache
  - virtuelle Basisklassen
  - abstrakte Basisklasse
- universeller Polymorhpismus, Bindung
- Friends

Klassendefinition 1

- die Implementation abstrakter Datentypen erfolgt in C++ als Klasse

  classdef ::= class classname interface ;
  interface ::= { compdef }

- die Klassendefinition ist eine Anweisung und muß daher mit einem “;” abgeschlossen werden

Klassendefinition 2

- Beispiel: `class A { };`
- Klassendeklaration: `class A;`
- Attribute werden wie Komponenten einer Struktur erklärt
- Attribute sind nur innerhalb der Klasse gültig (private), können aber öffentlich werden

```
class B {
  int a1;
  A a2;
};
```


Klassendefinition 3

- Methoden werden ebenfalls als Komponenten erklärt (und damit dem zugrundeliegenden ADT zugeordnet)
- Beispiel:

```
class Point {
  int x,y;
  void SetX(int xn);
  int GetX();
};
```

Klassendefinition 4

- die Methoden werden in der Klassendefinition nur erklärt (Interfacedefinition)
- die Definition der Methoden erfolgt getrennt in der Form

  mdef ::= rtype cname :: mname (*[* args *]*) block
- Beispiel:

```
int Point::GetX() { return x; }
```


Klassendefinition 5

- Konstruktor:
  - ist eine spezielle Methode, die während der ➔ *Instanzierung* (automatisch) aufgerufen wird
  - der Konstruktor trägt immer den Namen der Klasse und hat keinen Rückgabewert (auch nicht void)

```
class Point {
  Point();
};
```


Klassendefinition 6

- der Konstruktor kann polymorph sein

```
class Point {
  Point();
  Point(int x0,int y0);
};
```

- ein Konstruktor ohne Argumente heißt *Standardkonstruktor* und sollte immer definiert werden

Klassendefinition 7

- Destruktor:
  - der Destruktor wird während der Zerstörung einer Instanz (automatisch) gerufen
  - er trägt den Klassennamen mit einer vorangestellten Tilde "~"

```
class Point {
  ~Point();
  ~Point(int x0,int y0);
};
```

  - auch der Destruktor kann polymorph sein

Instanzierung 1

- statische Instanzierung:
  - durch die Deklaration einer Instanzvariablen wird eine Instanz der Klasse generiert

```
Point p;
```

  - Lebensbereich der Instanz ist der Gültigkeitsbereich der Instanzvariablen
  - Beispiel:

```
{
  int x;
  Point p;
  p.SetX(2);
}
```

Gültigkeitsbereich

Instanzierung 2

- dynamische Instanzierung:
  - die Anwendung des Operators new auf eine Klasse generiert eine Instanz dieser Klasse

```
Point *p;  p = new Point;
```

  - die Instanz lebt bis zur expliziten Zerstörung durch Anwendung des delete-Operators auf **irgendeine** Referenz auf die Instanz

```
Point *r = p; delete r;
```

Instanzierung 3

- soll ein anderer als der Standard-konstruktor verwendet werden, so wird an die Instanzvariable ein Parameter-satz angehangen

```
Point p1(1,2);
Point *p2=new Point(3,y);
```

- die Auswahl des Konstruktors erfolgt nach den Regeln des ad-hoc Polymorphismus

Datenkapselung 1

- C++ kennt drei Zugriffklassen
  - public:
    Zugriff von überall her
  - protected:
    Zugriff nur in der Instanz und in Instanzen abgeleiteter Klassen
  - private:
    Zugriff nur in der Instanz

Datenkapselung 2

```
class A {
    int a;
  private:
    int b;
  protected:
    int c;
  public:
    int d;
    int e;
  private:
    int f;
    int M();
};
```

Code	
`int A::M() {`	
`f=1;`	innerhalb der Klasse
`c=2;`	innerhalb der Klasse
`e=3;`	innerhalb der Klasse
`}`	
`A m;`	
`m.a=-1;`	privat (default)
`m.b=-2;`	privat
`m.c=-3;`	protected
`m.d=-4;`	O.K.
`m.e=-5;`	O.K.
`m.f=-6;`	privat
`m.M();`	privat

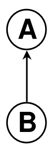
kurzer Rückblick

- Klassendefinition
  - **class A { t_i a_i; t_j m_j; ... };**
    t_j A::m_j(...) { ... }
- Instanzierung
  - statisch: **A a;**
  - dynamisch: **A *b; b=new A(...); delete b;**
- Datenkapselung
  - private, protected, public

Vererbung 1

- Klassen können voneinander abgeleitet werden

```
class A { … };
class B : A { … };
```



- eine abgeleitete Klasse übernimmt alle Attribute und Methoden den Basisklasse

Vererbung 2

- Beispiel:

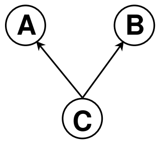
```
class Point {
  int x,y;
  Point(int x0int y0);
  void Draw();
  void MoveTo(int x1,int y1);
  ~Point();
};

class Circle : Point {
  int r;
  Circle(int x0,int y0,int r0);
  void Draw();
  ~Circle();
};
```

Vererbung 3

- C++ kennt mehrfache Vererbung

```
class A {
  int x;
  A();
};
class B {
  int z;
  B();
};
class C: A,B {
  C();
};
```



Vererbung 4

- Namenskonflikte werden durch vollständige Qualifizierung aufgelöst

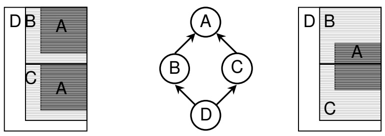
```
class A { int x; };
class B { int x; };
class C: A,B {
  int sum() { return A::x+B::x; }
};
```


Vererbung 5

- virtuelle Basisklassen

```
class A {};
class B:A {};
class C:A {};
class D:B,C { };
```

```
class A {};
class B:virtual A {};
class C:virtual A {};
class D:B,C { };
```



Vererbung 6

- abstrakte Basisklasse

```
class A {
  void m1()=0;       // abstrakte Methode
  void m2() { … };   // konkrete Methode
};
```

- A kann nicht instanziert werden

```
class B:A {
  void m1() { … };   // Konkretisierung m1
};
… B b;
```


Beispiel

```
class Gear {
    unsigned int gear;
  public:
    void SwitchUp()=0;
};

class ManualGear:Gear {
  public:
    void SwitchUp();
}

void ManualGear::SwitchUp() { ... }
```


Vererbung und Datenkapselung 1

- die Weitergabe von Eigenschaften kann beschränkt erfolgen

```
class Z : A, public B ,
             protected C ,
             private D {
  …
};
```

  - public: die Zugriffsbeschränkungen der Elternklasse werden übernommen
  - protected: public → protected
  - private: public, protected → private

Vererbung und Datenkapselung 2

```
class A {
  public: int x;
  A();
};
class B {
  protected: int z;
  public: B();
};
class C: protected A, public B {
  C();
};
C c;
c.x = 1;          // falsch, x ist protected
c.A();            // falsch, A() ist protected
c.z = 1;          // falsch, z ist protected
c.B();            // O.K.
```


Bindung 1

- Komponenten werden in C++ standardmäßig statisch gebunden
- soll eine Methode dynamisch gebunden werden, so muß dies durch das Schlüsselwort "virtual" kenntlich gemacht werden
- wird eine virtuelle Methode überschrieben, so bleibt sie virtuell

Bindung 2

```
class Gear {
  public:
    virtual void SwitchUp()=0;
    void SwitchDown();
    ~Gear();
};
class ManualGear : public Gear {
  public:
    virtual void SwitchUp();
    void SwitchDown();
    ~Gear();
};

Gear *g = new ManualGear();
g->SwitchUp();     // ManualGear::SwitchUp
g->SwitchDown();  // Gear::SwitchDown
```


Bindung 3

- Frage: Welcher Destruktor wird im vorhergegangenen Beispiel gerufen, wenn

  `delete g;`

  aufgerufen wird und warum?

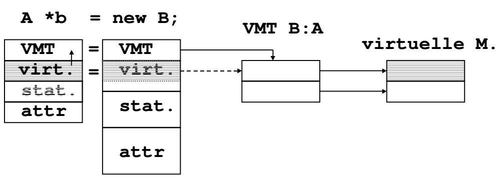
Bindung 4

- da der Destruktor zunächst nicht dynamisch gebunden wird, wird ~Gear() gerufen - das ist falsch!
  → virtueller Destruktor

```
class Gear { ...
  virtual ~Gear();
};
class ManualGear { ...
  virtual ~ManualGear();
};
```


Bindung 5 - VMT

- die dynamische Bindung wird über eine sogenannte VMT - Virtual Method Table organisiert



Beispiel 1

```
class Car {
    Gear *gear;
    Engine *engine;
  public:
    Car(Engine *e,Gear *g);
    virtual void Accel(double)=0;
};
Car::Car(Engine *e,Gear *g) {
  engine=e;
  gear=g;
}
```


Beispiel 2

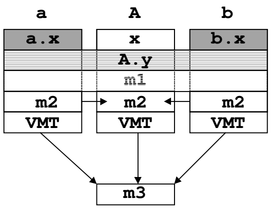
```
class Renault_R19:public Car {
  public:
    Renault_R19(Engine *e,Gear *g);
    virtual void Accel(double);
};
Renault_R19::Renault_R19((Engine *e,Gear *g)
             :Car(e,g) { }
void Renault_R19::Accel(double v) {
  ...
  if (engine->RPM()>3000) gear->SwitchUp();
  ...
}
Car *my_car=new Renault_R19(
                  new GasolineEngine(2.5),
                  new ManualGear(5));
```


static Members 1

- ein static Member existiert nur ein einziges mal für alle Instanzen

```
class A {
  int x;
  static int y;
  static void m1();
  void m2();
  virtual void m3();
};

A a,b;
```



- kein this

static Members 2

- ein static Attribut muß extra erklärt werden

```
class A {
  static int i;
};

int A::i;
```


Friends 1

- Beispiel

```
class Matrix {
  Matrix operator * (Matrix&);
};
class Vector {
  double operator * (Vector&);
};
```

- Wie sollte Matrix * Vector erklärt werden damit * effizient arbeitet?

Friends 2

- 1. Variante:

```
class Matrix {
  Matrix operator * (Vector&);
};
```

  – hängt von der Implementation von Matrix ab (muß also jedesmal neu erklärt werden)

- 2. Variante

```
class Matrix {
  friend Matrix operator*(Matrix&,Vector&);
};
```

Friends 3

- die Implementation als friend gilt als Hinweis darauf, daß die Klasse verändert werden kann
- ein friend ist quasi Member einer Klasse und hat Zugriff auf alle Komponenten der Klasse

```
class Matrix {
  friend Matrix operator * (Matrix&,Vector&);
};
class Vector {
  friend Matrix operator * (Matrix&,Vector&);
};
```


Zusammenfasung - Klassen

- Klassendefinition und Instanzierung
- Vererbung
  - einfache und mehrfache
  - Datenkapselung
  - virtuelle und abstrakte Basisklassen
- universeller Polymorphismus, Bindung
- Friends

Templates 1

- Templates stellen die Realisierung des parametrischen Typkonzeptes dar
- Motivation:
  - Verfahren können für einen abstrakten Typ T erklärt werden, ohne das der Typ bekannt ist
  - die Konkretisierung des Typs erfolgt erst bei der Nutzung des Verfahrens
- Beispiel: das Konzept des Vektors

Templates 2

– ein Vektor wird über einem Körper (T) erklärt

```
template<class T> class Vector {
    T *v;
    unsigned int dim;
  public:
    Vector(unsigned int s) {
      if (s==0) error("dimension must not be 0");
      v=new T[dim=s];
    }
    T &operator [] (unsigned int i) {
      return v[i];
    }
    unsigned int Dim() { return dim; }
};
```


Templates 3

- Nutzung:

```
void f() {
  Vector<int> v1(100); // 100 integer
                       // Elemente
  Vector<Gear*> g(10); // 10 Referenzen auf
                       // Getriebeinstanzen
  ...
  g[5]->SwitchUp();    // schalte G. 5
}
```


Templates 4

- Member-functions von Templates müssen nicht inline erklärt werden

```
template<class X> class Y {
    X x;
  public: void method(X x0);
};
```

- die externe Definition erfolgt mit voller Qualifikation:

```
template<class X> void Y<X>::method(X x0) {
  x=x0;
}
```

Templates 5

- der Quelltext der Implementation von nicht-inline Methoden von Templates muß nicht vorliegen, damit können quasi abstrakte Bibliotheken geschaffen werden

Templates und Vererbung

- Template Klassen können vererben:

```
template<class T> List { T *elements; };
template<class T> OrderedList:public List<T> {
}
class A;
template<class T> Based_on_A : A { };
```

- Templates können auf verschiedenen Typen basieren:

```
template<class A,class B,class List<A>> C {...};
```


Templates und Argumente

- Templates können Argumente erhalten:

```
template<class T,unsigned int s> class Array {
  T data[s];
};
```

- Templates können sogar parametrisch erben

```
template<class T,class A> class ParamContainer:
public Container<T>, public A {
  ...
};
```


Funktions Templates 1

- Grundidee:
  - ein Funktionstemplate definiert eine Familie von Funktionen
- Beispiel:

```
template<class T> void sort(Vector<T>&);
void f(Vector<int> &vi,Vector<float> &vf) {
  sort(vi);
  sort(vf);
}
```


Funktions Templates 2

- Implementation:

```
template<class T> void sort(Vector<T> &v) {
  unsigned int n=v.size();
  for (int i=0;i<n-1;i++)
    for (int j=n-1;i<j;j--)
      if (v[j]<v[j-1]) {
        T temp=v[j];
        v[j]=v[j-1];
        v[j-1]=temp;
      }
}
```


Funktionstemplates 3

– Co-Existenz von Templates und Spezialisierungen

```
template<class T> class Comp {
  public:
    static int comp(T &a,T &b) { return a<b; }
};
class Comp<char*> {
  public:
    static int comp(const char *a,const char *b)
      { return strcmp(a,b); }
};
template<class T> class SortableVector
  :public Vector<T>,public Comp<T> {
  ...
};
```

Funktionstemplates 4

■ Operationen als Template-Argumente

```
template<class T,class Comp> class Sort {
  public void sort(vector<T>&);
};
template<class T,class Comp>
void Sort<T,Comp>::sort(Vector<T> &v) {
  ...
  if (Comp::comp(v[j],v[j-1]))
  ...
}
template<class T> class LT:public Comp<T>;
template<class T> class GT:public Comp<T>;
Sort< int, LT<int> >::sort(vi);
Sort< int, GT<int> >::sort(vi);
```


Standard Template Library

■ ist eine Template Sammlung von SGI (http://www.sgi.com/Technology/STL)

■ die folgende Kategorien abdeckt
- Container
- Iteratoren
- Algorithmen
- Funktionsobjekte
- „Utilities“
- Speicherverwaltung

STL - Container

– Konzepte
- allgemeine, Folgen, assoziative Conatiner

– Containerklassen
- Folgen: vector, deque, list, slist, bit_vector
- assoziative Container: set, map, hash
- Stringpacket: char_traits, basic_string
- rope
- Adaptoren: stack, queue

STL - Iteratoren

– Konzepte
- einfache I., E/A I., Vorwärts-I., Bidirektionaler I., Random Access I.

– Iterator Tags
- iterator traits, iterator category, distance type, iterator tag Klassen, iterator Basisklassen

– Iteratorfunktionen
- Abstand

– Iteratorklassen
- istream, ostream, reverse, raw, sequence

STL - Algorithmen

– nicht mutierenede Verfahren
- for-each, Suche, Zählen, Vergleich

– mutierende Verfahren
- Kopieren, Vertauschen, Transformationen, Ersetzen, Füllen, Generieren, Löschen, Permutieren

– Sortieren
- Sortieren, Zusammenführen, Mengenoperationen, MinMax

– allgemeine Verfahren
- inneres Produkt, Partialsummen, Potenzieren

STL - Funktionsobjekte

– Konzepte
- Generatoren, unäre/binäre Funktionen, Monoid-Operationen, Zufallszahlengenerator

– vordefinierte Funktionen
- arithmetische Op., Relationen, logische Op.

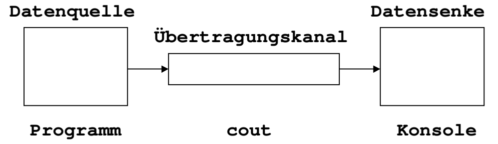
– function object adaptors

STL- Utilities/Speicherverwaltung

- Konzepte
- Funktionen
  - Relationsoperatoren
- Klassen
  - Paar
- Speicherklassen
  - Allokatoren, Iteratoren
- Funktionen
  - Erzeugen, Zerstören, temporärer Puffer

Streams 1

- Streams repräsentieren (gepufferte) Übertragungskanäle
- Streams gehören **nicht** zur Sprache C++



Streams 2

- prinzipiell könnte eine Streamklasse wie folgt aussehen

```
template<class T> class Stream {
    T *buffer;
  public:
    Stream(unsigned int maxsize);
    void Put(T t);
    T Get();
};
```


Streams 3

```
class A {
    Stream<int> *s;
  public:
    void Attach(Stream<int> *ss);
    void send(int i);
    int recv();
};

void A::Attach(Stream<int> *ss) { s=ss; }
void A::send(int i) { s->Put(i); }
int A::recv() { return s->Get(); }
```

```
Stream<int> s;
A a,b;  a.Attach(&s); b.Attach(&s);
a.send(10);  printf("%\n",b.recv());
```


Streams 4

- um die Nutzung handlicher zu gestalten wurden für die in C++ genutzten Streams die Operatoren << und >> überladen
- Philosophie:
  - in einen Datenstrom wird ein Element eingefügt, also

```
Stream<T> &operator << (Stream<T>&,T&);
...
void A::send(int i,int j) { s << i << j; }
```


Streams 5

- die C++ Standardbibliothek erklärt zwei Arten von Standard-Streams
  - bidirektionale Ströme
    - class ios;
  - unidirektionale Ströme
    - class ostream:public ios;
    - class istream:public ios;
    - Dateiströme: ifstream, ofstream
    - Stringströme: istrstream, ostrstream

Streams 6

- insbesondere werden zwei spezielle Instanzen bereitgestellt
  - istream cin;
    - unidirektionaler Eingabestrom, der standardmäßig mit der Konsole als Datenquelle verbunden ist
  - ostream cout;
    - unidirektionaler Ausgabestrom, der standardmäßig mit der Konsole als Datensenke verbunden ist

Streams 7

- cin und cout sind am jeweils anderen Ende mit dem Programm verbunden
- die Operatoren << und >> stehen für alle C++ Standardtypen bereit
- Beispiel:

```
#include <iostream.h>
main() {
  char name[1024];
  cout << "What's your name?";
  cin >> name;
  cout << "Hello " << name;
}
```


Ausnahmebehandlung 1

- Beispiel:
  - numerische Integration einer Funktion (Summierung mit Schrittweite)
  - Probleme:
    - numerischer Unterlauf
      - Ergebnis regulär aber falsch
    - numerischer Überlauf
      - Ergebnis kann regulär sein aber falsch
- Möglichkeit solche Zustände zu signalisieren und darauf zu reagieren

Ausnahmebehandlung 2

- Exceptions werden als Objekt repräsentiert

```
class Exceptionclass;
```

- Exceptions werden durch das throw Konstrukt "ausgelöst"

```
throw Exceptionclass();
```

- Handler:

```
catch (Exceptionclass &c) {
  Behandlungscode
}
```


Ausnahmebehandlung 3

```
class RangeCheck {
  public: RangeCheck(char*);
};

double integral(double eps) {
  double sum,psum;
  if (fabs(psum)<irgendwas)
    throw RangeCheck("Underflow");
  if (fabs(psum)>irgendwas)
    throw RangeCheck("Overflow");
}
...   try {
        i=integral(epsilon);
      }
      catch (RangeCheck &r) {
      }
```


Ausnahmebehandlung 4

- Exceptions können in einer Deklaration bekanntgegeben werden:

```
extern double integral(double) throw(RangeCheck);
```

- vordefinierte Ausnahmeklassen:
  - class exception;
    - class logic_error:exception;
      - » class domain_error:logic_error; invalid_argument, length_error, out_of_range
    - class runtime_error:exception;
      - » class range_error:runtime_error; overflow_error
    - class bad_exception;

Ausnahmebehandlung 5

- Regeln zur Verwendung von Exceptions
  - Ausnahmebehandlungen so festlegen, daß sie beschreiben, was eine Funktion **tut** und nicht, was sie **tun sollte**
  - Ausnahmen nur für wesentliche Funktionen
  - throw(); nur in trivialsten Fällen
  - bad_exception sollte immer in die Spezifikation aufgenommen werden
  - keine A.-Spezifikationen in F.-Templates
  - nur allgemeine A.-Spezifikationen für virtuelle M.

Ausnahmebehandlung 6

- keine A.-Spezifikationen in typedefs

```
typedef void(*FooType)(int,int) throw();
```

- terminate_handler nicht in Bibliotheken verändern
- unexpected_handler sollte entweder bad_exception oder eine andere gültige Ausnahme generieren

C++ und Performance

- inline-Expansion
- Zeigerarithmetik vs. Feldindizierung
- temporäre Instanzen

- HPC++

inline-Expansion

- normalerweise werden Funktionen bzw. Methoden über einen call-Befehl als Unterprogramm gerufen
- inline expandiert den Unterprogrammcode direkt an der Stelle des Aufrufes (damit entfällt der call-overhead)
- auch Methoden können inline expandiert werden

Beispiel

```
inline double fabs(double x) {
  if (x<0) return -x;
  return x;
}
double dec(double x) {
  return x-1;
}

...
double y;

y=dec(4.0);
y=fabs(-3.4);
```

```
    ld fp0,4.0
    push fp0
    call M_dec
    pop fp0
    ld (Y),fp0
```

```
    ld fp0,-3.4
    ld fp1,fp0
    sub fp0,0
    jsgnp M1
    neg fp1
M1: ld (y),fp1
```


Zeigerarithmetik vs. Indizierung

- in älteren C Compilern waren Feldzugriffe über den Index des Feldelementes relativ ineffizient implementiert (das ist heute nicht mehr der Fall!)
- prinzipiell ist für

```
T F[dim],*p;
```

der Zugriff *(p+i) gleichwertig zu F[i]

```
*(p+i)==F[i];   // p+i ~ p+sizeof(T)*i
```

temporäre Instanzen 1

- problematisch sind temporäre Instanzen, diese entstehen z.B. bei folgendem Konstrukt

```
class Integer {
  int value;
  Integer operator + (Integer &b) {
    Integer s;
    s.value=value+b.value;
    return Integer(s);
  }
};
```


temporäre Instanzen 2

- für „größere“ Objekte ist es oft besser nur auf +=, -=, *=, /=, ... Operatoren zurückzugreifen, da hier die Konstruktion einer temporären Instanz entfallen kann
  - der linke Operand erhält hier gleichzeitig das Ergebnis der Operation

HPC++

- High Performance C++
- wird derzeit als ANSI Standard vorbereitet
- besondere Eigenschaften
  - Loop level Parallelism: Compiler-Direktive
  - Parallele Standard Template Library
  - Feld-Container
  - Interface für externe Objekte (CORBA)
- Open/MP konformer C++ Compiler ?

C++ und C

- C++ verwendet einen anderen Bindungs-/Aufrufmechanismus für Funktionen als C

```
extern "C" double fabs(double);
#ifdef __cplusplus
 extern "C" {
#endif
   int abs(int);
   #include <string.h>
#ifdef __cplusplus
 }
#endif
```


Turbo-Pascal, Delphi

Entwicklung

- Pascal wurde 1970 von N. Wirth entwickelt (zunächst als reine Lehrsprache)
- 1983 ISO, DIN
- Dialekte: UCSD, MS, MT+, Turbo
  - Unit Konzept (Module) durch UCSD und Borland

Eigenschaften

- erste OO-Sprache unter MS-DOS
- OO seit Version 5.5
- Klasse wird als strukturierter Datentyp erklärt (muß im Typdefinitionsteil eines Programmes erklärt werden)
- nur einfache Vererbung
- nur public und private
- kein Inlining
- kein ad-hoc Polymorphismus

Klassendefinition

```
type Vector=object
              constructor Init(d0:Integer);
              function Get(i:Integer):Real;
              procedure Set(v:Real,i:Integer);
            private
              var dim:Integer;
              x:^Real;
            end;
     Point=object(Vector)
              ...
           end;

constructor Vector.Init(d0:Integer);
begin
  ...  self.Set(0.0,0);
end;
```


Instanzierung

- der Konstruktor muß immer explizit gerufen werden
  - statisch:

```
var x:ObjectType;
x.Init();
```

  - dynamisch:

```
type PObjectType=^ObjectType;
var x:PObjectType;
x:=new(PObjectType,Init(...));
```


Semantik von new und dispose

- neue Syntax und Semantik für new:
  - function new(Pointertype [, Konstruktor]) : Pointer;
  - new allokiert Platz auf dem Heap und ruft zusätzlich den angegeben Konstruktor auf
  - new gibt eine Referenz auf den allokierten Bereich zurück
- new Syntax und Semantik für dispose:
  - procedure dispose(Pointer [, Destruktor]);
  - dispose ruft den angegebenen Destruktor und gibt anschließend den Speicher frei

Destruktion

- eine Instanz wird zerstört wenn
  - der Gültigkeitsbereich der statischen Instanzvariablen verlassen wird
  - der dynamisch allokierte Platz freigegeben wird
- in beiden Fällen muß der Destruktor explizit gerufen werden, also

```
var a:ObjectType; ...  a.Init; ... a.Done;
```

```
var p:PObjectType; ...
    p:=new(PObjectType,Init);
... dispose(p,Done);
```


Polymorphe Methoden

- Methoden werden standardmäßig statisch gebunden
- dynamisch zu bindende Methoden werden durch „virtual“ gekennzeichnet
- Beispiel:

```
type ObjectType=object
       procedure S;
       procedure V; virtual;
     end;
```

Bibliotheken

- Turbo-Pascal wurde durch das Konzept der Turbo-Vision Bibliothek sehr erfolgreich
- die Turbo-Vision Bibliothek wurde nach Turbo-C++ übernommen
- unter MS-Windows wurde Object-Windows in Anlehnung an die Turbo-Vision eingeführt

Abteilung Kommunikation und Datenverarbeitung

Turbo-Vision 1

- ist eine Klassenbibliothek zur Erstellung SAA konformer Nutzerschnittstellen
- es gibt
  - Widget-Klassen
    - Dialogboxen, Fenster, Eingabezeilen, Menüs, ..
  - eine Applikationsklasse
  - eine Ereignisklasse

Abteilung Kommunikation und Datenverarbeitung

Turbo-Vision 2

- Beispiel:

```
program TVGUID01;

uses App;

type TMyApp = object(TApplication)
     end;

var
  MyApp: TMyApp;

begin
  MyApp.Init;
  MyApp.Run;
  MyApp.Done;
end.
```


Abteilung Kommunikation und Datenverarbeitung

Turbo-Vision 3

Abteilung Kommunikation und Datenverarbeitung

Oberon

Abteilung Kommunikation und Datenverarbeitung

Eigenschaften

- N. Wirth, J. Gutknecht
- System + Oberfläche + Sprache
  - Nachfolger von Pascal und Modula
  - Blockstruktur
  - Modulkonzept
  - strenge Typisierung
  - Typerweiterung
  - ad-hoc Polymorphismus (Oberon-2)
  - Trait-basiert

Module

```
MODULE Calc;
IMPORT In, Out;

VAR a:Integer;

PROCEDURE SetA*();
BEGIN
  In.Open; In.Int(a);
END SetA;

PROCEDURE Privat : Typ ;
BEGIN
  ...
END Privat;

BEGIN
  ...
END Calc.
```

```
System.Compile Calc.Mod;

Calc.SetA 100

Calc ~System.Free
```

- der Hauptblock eines Moduls wird beim Laden des Moduls gerufen
- ein Modul ist persistent

Klassen 1

- Klassen werden als Erweiterung von Records betrachtet

```
TYPE      AClassPtr = POINTER TO AClass;
          AClass = RECORD
                      priv_m    : PROCEDURE (...);
                      public_m*: PROCEDURE (...);
                      priv_a    : Integer;
                      public_a*: Typ;
                   END;
```

- öffentliche Komponenten werden durch einen * gekennzeichnet

Klassen 2

- die Bindung erfolgt durch explizite Übergabe der Instanz in einem formalen Parameter

```
PROCEDURE ( self : AClass ) public_m* (...) ;
BEGIN
  In.Open;
  In.Integer(self.priv_a);
END public_m;
```


Vererbung

- ursprünglich objektbasiert, also von Objektprototyp abgeleitet
  - mögliche Inkonsistenz in Methoden
  - Speicherplatz
- Klassenbasiert
  -
```
TYPE BClassPtr = POINTER TO BClass;
     BClass = RECORD(AClass)
                  new_m : PROCEDURE(...);
              END;
```

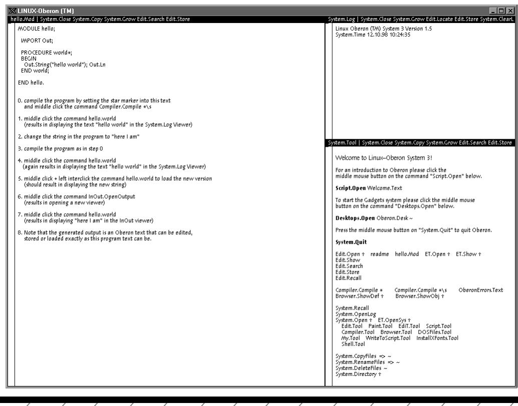

Instanzierung

- Oberon verfügt über eine garbage-collection, d.h. Instanzen müssen nur erzeugt werden

```
VAR b : BClassPtr;

NEW(b);
```

- genauer: es gibt keine Möglichkeit eine Instanz explizit zu zerstören



Java

Entwicklung

- 1990 ursprünglich unter dem Namen „Oak" als Sprache für die Programmierung „hybrider Systeme im Consumer Bereich" (! Ada) von J. Gosling bei Sun entworfen, hatte aber nie Erfolg
- 1993 in Java umbenannt und als Sprache für Internetanwendungen verwendet (! portabel)

Eigenschaften 1

- C++ ähnliche Syntax
- Garbage-collection
- multi-threading
- soll sicher und robust sein
- Interpretersprache (basiert auf virtueller Maschine, die Java-Bytecode ausführt), daher ineffizient

Eigenschaften 2

- Unterschiede zu C++
  - keine Typdefinitionen, kein Pre-Prozessor
  - keine Strukturen oder Unions
  - keine nicht-Member-Funktionen
  - keine Mehrfachvererbung
  - keine automatisch Typumwandlung
  - keine manuelle Speicherverwaltung (!Garbage-collection) und Zeigermanipulation

Klassendefinition & Instanzierung

- Klassen:

```
class Point {
  private double x,y;
  public Point() { x=0; y=0; }
  public double GetX() { return x; }
  public void SetX(double v) { x=v; }
}
```

- Instazierung:

```
Point p; ... p=new Point();
```


Vererbung 1

- Vererbung in Java bedeutet Erweiterung

```
class Pixel extends Point {
  private int color;
  public Pixel() {
    super.SetX(0);
    color=0;
  }
}
```

- alle Klassen sind von Object abgeleitet

Vererbung 2

- die Vererbung kann durch Festlegung von “final” Klassen beschränkt werden

```
final class KeineVererbung {
  ...
}
```

- auch die Methodenvererbung kann eingeengt werden

```
class Klasse {
  final public NichtVererbbar() { ... }
}
```


Interfacevererbung

- statt Mehrfachvererbung unterstützt Java Interfacevererbung

```
interface MyInt1 {
  public void M1();
}
interface MyInt2 {
  public void M2();
}
class Klasse extends Basisklasse
            implements MyInt1,MyInt2 {
...
}
```


abstrakte Methoden

- wie auch in C++ gibt es die Möglichkeit abstrakte Methoden (abstrakte Klassen) zu erklären

```
abstract class ABaseclass {
  abstract public void M();
}
```

- eine Klasse muß abstrakt sein, sobald eine abstrakte Methode existiert

Implementierung

- Definition und Implementation werden in einem Modul gehalten
- Klassen können bzgl. eines Moduls lokal oder global sein
- Module/Klassen werden durch die Anweisung

  import Kategorie.Modul.Klasse;

  bereitgestellt

Programme

- die “Hauptklasse” eines Java-Programms muß die statisch Methode main enthalten

```
class MyProg {
  public static void main(String args[]) {
  }
  ...
}
```

- der Modulname muß mit dem Haupklassennamen übereinstimmen

Internetanwendungen 1

- für “Applets”, die als Internetanwendungen geschrieben werden gibt es ein anderes Verfahren

```
import java.applet.*;
class MyApplet extends Applet {
  public void start() {
    // beim Aufruf des HTML-Dokumentes
  }
  public void stop() {
    // beim Verlassen des HTML-Dokumentes
  }
}
```

Internetanwendungen 2

- die Einbindung in das HTML-Protokoll erfolgt durch

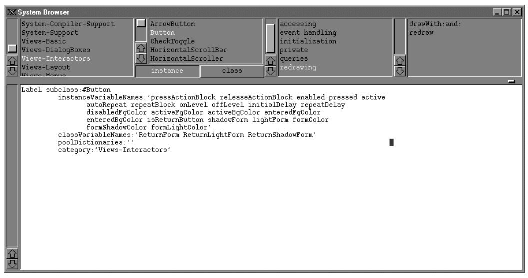
```
<html>
  <body>
    <applet code="MyApplet.class"
            width=640 height=480>
    </applet>
  </body>
</html>
```


Smalltalk-80

Eigenschaften

- reine OO-Sprache
- untypisiert
- Interpretersprache (virtuelle Maschine)
- voll implementiertes Nachrichtenkonzept
- garbage-collection
- interaktive Programmierumgebung

interaktive Programmierungebung



Klassendefinition

- jede Klasse hat eine Superklasse, im einfachsten Fall “Object”
- Klassen werden in Kategorien geordnet
- Klasseneigenschaften:
  - Instanzvariablen: private Attribute
  - Klassenvariablen: gemeinsame Attribute a la static bei C++
  - Kategorie

Klassendefinition und Vererbung

```
Superklasse subclass:#AClass
  instanceVariableNames:'priv1 priv2 ...'
  classVariableNames:'shared1 shared2 ...'
  poolDictionaries:''
  category:'Kategorie'
!
```

Methodendefinition

- Methoden werden ebenfalls kategorisiert

```
!AClass methodsFor:'methodcategory'!

m1 arg1 arg2 ...
...
!
```


Invokation

- Smalltalk basiert auf dem Nachrichtenkonzept, d.h. alle Methodenaufraufe sind Bestandteile einer Nachricht an eine Instanz

  Empfängerobjekt Sel1 Arg1 Sel2 Arg2 .

Beispiel

```
Object subClass:#Complex
  instanceVariableNames:'imag real'
  classVariableNames:''
  poolDictionaries:''
!

!Complex methodsFor:'arithmetic'

Real: rv Imag: iv
  real := rv .
  imag := iv .
!
```

Selektor1

Eiffel

Eigenschaften

- von B. Meyer entwickelt
- implementiert alle, von ihm aufgezählten Kriterien für OO-Sprachen
- läßt Definition von Vorbedingungen zu
- keine statische Instanzierung

Klassendefinition

```
class MyClass
creation
  Konstruktor
feature {ALL}
  - oeffentliche Komponenenten
  Konstruktor(i:integer) is
  do
    ... !!value.make(1,i)
  end
feature {NONE}
  - private Komponenten
  value : array [ integer ]
end
```

Vererbung

- Einfach- und Mehrfachvererbung

```
class DClass
inherits
  AClass
  BClass rename
    b1 as BClassb1
  end
end
feature {ALL}
...
end
```


Instanzierung

- Eiffel kennt keine statisch Instanzierung
- es werden Instanzvariablen erklärt, die Instanzierung erfolgt explizit durch den Aufruf eines Konstruktors

```
ivar : DClass

!!ivar.Konstruktor
```


Operatoren

- in Eiffel können Operatoren überladen und es können neue O. definiert werden

  infix “ opsymbol ” [, altname]
- Beispiel:

```
class Complex
feature {ALL}
  infix "+",Plus (b:Complex) is
  do
    ...
  end
end
```


Vorbedingungen

```
class Vector
feature {ALL}
  Dim : integer is
  do
    return dim;
  end
  infix "+" (b:Vector):Vector is
  reqiure
    Dim=b.Dim
  do
    ...
  end
feature {NONE}
  dim : integer
end
```


FORTRAN 90/95/2000

- Fortran 90
  - Operatorüberladung (ad-hoc P.)
  - generische Funktionen
- Fortran 95
  - FORALL
  - pure und Elementfunktionen
- Fortran 2000
  - OO Eigenschaften:
    - Konstr. und Destr.
    - Vererbung
    - Polymosphismus
    - Prozedurreferenzen

OO-Systeme

- OO-GUI‘s
- OO-Betriebssysteme
- verteiltes OOP (CORBA, DCOM)
- OO-Datenbanken

OO GUI’s

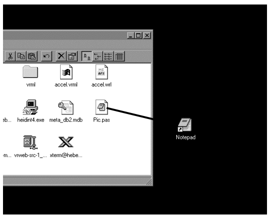
- graphische Nutzerschnittstellen
- Philosophie:
  - Schreibtischphilosophie
  - generisch Kommandos (drag and drop)
  - direktes Manipultieren (OLE)
  - Icons
  - Persistenz von Objekten

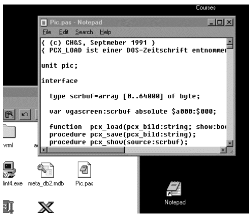
OO-GUI‘s

- Beispiele:
  - Smalltalk
  - Oberon
  - GeoWorks Ensemble
  - NextStep
  - MS-Windows 3.x, 95 (OLE)
  - MS-WindowsNT
  - IBM OS/2 Presentation Manager

Drag and Drop

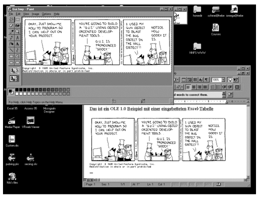
- Erzeugen einer Instanz von B bei gleichzeitigem Öffnen der Instanz a





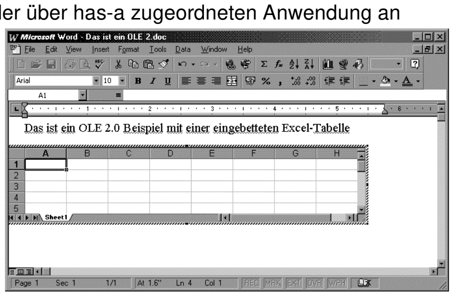
OLE 1.0

- OLE - Object Linking and Embedding
  - ein Objekt enthält einen Verweis auf ein anderes Objekt
  - bei „anklicken“ des enthaltenen Objektes wird das zugehörige Programm gestartet



OLE 2.0

- wie OLE 1.0, aber die Applikation nimmt die Gestalt der über has-a zugeordneten Anwendung an



OO-Betriebssystem

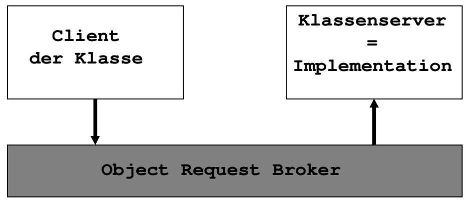
- Mach-Kernel
  – objektorientierter Betriebssystemkern
- Microsoft Windows mit OLE
- Oberon
- Smalltalk

Verteiltes OOP

- CORBA
  - von der Object Management Group (OMG) festgelegter Standard
  - Common Object Request Broker Architecure
- DCOM
  - Konkurrenzprodukt von Microsoft

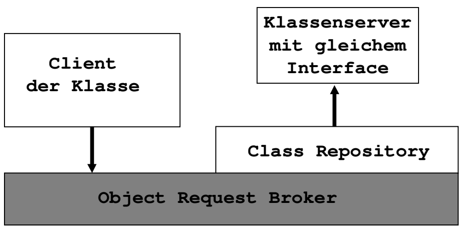
Prinzip von der ORB Technik

- Client-Server-Architektur



Interfacevererbung

- CORBA legt nur Interfacevererbung fest



Datenbanken

- relationale Datenbanken (RDBM)
  - Menge von Relationen
  - R, SQL/DS, DB2, INGRES, ORACLE, DBASE
- semantisches Datenbankmodell (SMDB)
  - Entity-Relationship Modell, prädikatenlogische und mengentheoretische Modelle
  - RM/T, TAXIS, SAM, IFO
- objektorientiertes Datenbankmodell (OODBM), i.A. Abbildung von OO Konzepten auf RDBMS oder SMDBS

OODBMS

- Strukturkonzepte
  - » Objektidentität, Klassen, Typen, Beziehungen, zwischen Klassen, Strukturvererbung
- Operationen
  - » Relationen, Generatoren, objekterhaltende Operationen
- höhere Konzepte
  - » Metaklassen, ADT, OO-Konzepte
- GemStone, ONTOS/VBASE, O_2, POET, POSTGRES