2 Hilfsmittel zum Programmieren im Kleinen

Erstes Beispiel

C++ – Programm

```
#include <iostream.h>
int main()
{  int summe, a, b;
   // Lies die Zahlen a und b ein
   cout << ”a und b eingeben:”;
   cin >> a >> b;

   /* Berechne die Summe beider
      Zahlen */
   summe = a+b;

   // Zeige das Ergebnis auf dem
   // Bildschirm an
   cout << ”Summe=” << summe << endl;
   return 0;
}
```

Begriffe:

↓ für Info an das Hauptprogram

Hauptprogramm `main( ) { ... }`

Block { ... }

Kommentare `//` .... bis Zeilenende

`/*` ....

.... `*/`

Einbindung Ein– / Ausgabefunktionen: Datei `iostream.h`

Variablendeklaration von `summe, a, b`, Bezeichner, Typ

Programme können Werte an andere geben, sonst `void`

Vordefinierter Datentyp `int`

Standardeingabe, –ausgabe `cin >> a, cout << summe`

Zeichenkettenliteral `"Summe="`

Zuweisungszeichen = , Ausdruck, Wertzuweisung,
Anweisungsfolge

Wie ist ein Programm aufgebaut? (Syntax)

Welche Berechnung führt es aus? (Semantik)

Wie wird es aufgetragen; Ist es lesbar;
Wie schnell ist der Übersetzer? (Pragmatik)

folgender Abschnitt

Zusammenfassung: Semiotik

Welche Hilfsmittel zur Ausgestaltung von Programmen existieren für Deklarationen und Ablaufstrukturen

restliches Kapitel

Syntax und lexikalische Einheiten

Syntaxbeschreibung für Aufbausyntax durch EBNF

identifier ::= nondigit { nondigit | digit }
nondigit ::= a | b | ... | z | A | B | ... | Z | _
digit ::= 0 | 1 | 2 | ... | 9

```
AnzahlDerZeilen
Anzahl_der_Zeilen
```
} zulässig

```
1_Rang
A#B
```
} unzulässig

floating_literal ::= {digit}$^+$. {digit}$^+$[e[sign]{digit}$^+$] [floating_suffix]

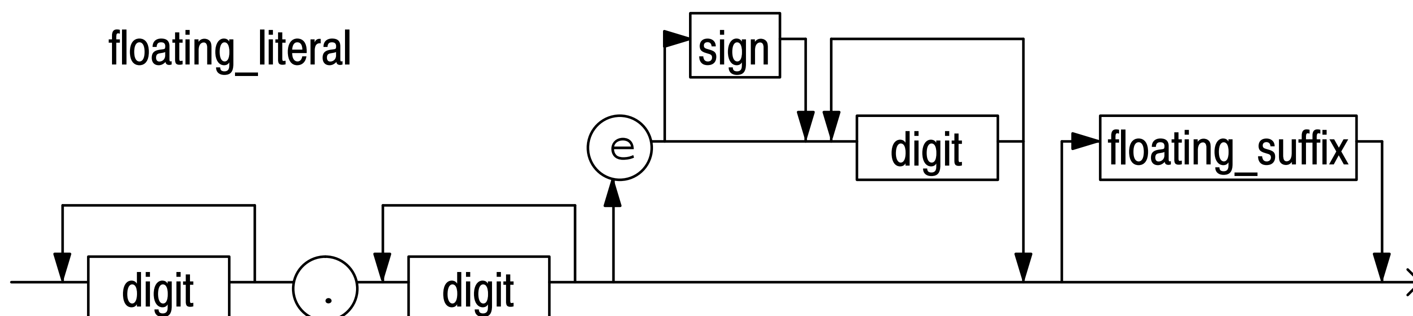
```
1.0e-5f
3.1415
```
} zulässig

```
1. e-10
.347_8
```
} unzulässig

Begriffe: nichtterminale Zeichen identifier
terminale Zeichen wie `.` oder `A`
::= "ist erklärt gemäß"
Metazeichen | , { , } , }$^+$, [,] , (,)

Gestaltungshilfsmittel bei Syntaxfestlegung
- Hintereinanderschreibung
- Option [...]
- Wiederholung { ... } bzw. { ... }$^+$
- Gruppierung (...)
- Fallunterscheidung ... | ...
- Rekursion

Syntaxdiagramme (äquivalent zu EBNF)



Lexikalische Einheiten im Eingangsbeispiel

```
#include <iostream.h>
int main()
{  int summe, a, b;
   // Lies die Zahlen a und b ein
   cout << ″a und b eingeben:″;
   cin >> a >> b;

   /* Berechne die Summe beider
       Zahlen */
   summe = a+b;

   // Zeige das Ergebnis auf dem
   // Bildschirm an
   cout << ″Summe=″ << summe << endl;
   return 0;
}
```

Struktur der EBNF von C++

- Schichten (lexikalische Syntax – kontextfreie Syntax)
- Komplexe: Ausdrücke, Deklarationen, Anweisungen, Programmstruktur

→ siehe Anhang

Syntax

- lexikalische
- kontextfreie (Aufbausyntax)
- kontextsensitive

} für C++, → Anhang I

kontextsensitive Syntax (umgangssprachlich,→ Anhang I)

Beispiele:

- Jeder Bezeichner muß deklariert sein
- Die Parameter bei Aufruf und Deklaration einer Prozedur müssen übereinstimmen
- Die Typen zweier Operanden zu einem Operator müssen typverträglich sein
- ...

Lexikalische Einheiten im Einzelnen

– Bezeichner

– Wortsymbole (Schlüsselwörter) [siehe Anhang I]

– Begrenzer

Abschluß	;	(Anweisung)
Anfang/Ende	{ .. }	(Block)
Abtrennung	,	(Aktualparameter)

– Kommentare

– Literale (s.u.)

- ganzzahlige
- reelle
- Zeichenliterale
- Zeichenkettenliterale

Programm–Layout als Pragmatik–Aspekt

- Programme lesbar für Entwickler (nicht Compiler)
  - bei Entwicklung
  - bei Veränderung
- Lesbarkeit durch Formatierungsregeln
  - durch Leerzeilen, Leerzeichen, Einrücken
  - durch entsprechende Bezeichner
  - durch Kommentare
- vergleiche Layout und Grammatik
- Lesbarkeit hängt nicht nur vom Layout ab:
  - Tricks
  - Mißachtung von Methodik
  - komplizierte Algorithmen

Einfache (skalare) Datentypen und ihre Operatoren

(zuerst vordefinierte, dann selbstdefinierte)

Ganzzahlige Datentypen:

Vordefinierte Datentypen

`short`	(üblicherweise 16 Bit)	–32768...32767
`int`	(16 oder 32 Bit)	s.o. oder –2147483648...2147483647
`long`	(32 oder 64 Bit)	entsprechend

`unsigned` (ohne Vorzeichen) – dadurch Wertebereich anders,
z.B.: `unsigned short` 0...65535

Über– oder Unterschreitungen werden zur Laufzeit in der Regel <u>nicht gemeldet!</u>

Ganzzahlige Literale

Oktalliterale	`0377`	`//` $(377)_8 = (255)_{10}$
Hexadezimalliterale	`0xAFFE`	`//` $(AFFE)_{16} = (45054)_{10}$
oder	`0XAFFE`	
Dezimalliterale	`123`	

Suffix bei Literalen: `U,u` für unsigned
`L,l` für long

Konstantendeklarationen (compilezeitbestimmt)

```
int const KILO = 1000;
ind const MEGA = KILO * KILO;
```

Ganzzahlige Operatoren

Arithmetische Operatoren:		
+	+i	unäres Plus (wird weggelassen)
–	–i	unäres Minus
++	++i	vorherige Inkrementierung
	i++	nachfolgende Inkrementierung
––	––i	vorherige Dekrementierung
	i––	nachfolgende Dekrementierung
+	i + 2	binäres Plus
–	i – 5	binäres Minus
*	5 * i	Multiplikation
/	i / 6	Division
%	i % 4	Modulo (= Rest mit Vorzeichen von i)
Relationale Operatoren:		
<	i < j	kleiner
>	i > j	größer
<=	i <= j	kleiner gleich
>=	i >= j	größer gleich
==	i == j	gleich
!=	i != j	ungleich
Bit–Operatoren		
<<	i << 2	Linksschieben (Multiplikation mit 2er–Potenzen)
>>	i >> 3	Rechtsschieben (Division durch 2er–Potenzen)
&	i & 7	bitweises UND
^	i ^ 7	bitweises XOR
|	i | 7	bitweises ODER
~	~i	bitweise Negation
&&	i && b	logisches UND
||	i || b	logisches ODER
!	!i	logische Negation
Kurzformen bei Zuweisung		
+=	i += 3	i = i + 3
–=	i –= 3	i = i – 3
*=	i *= 3	i = i * 3
/=	i /= 3	i = i / 3
%=	i %= 3	i = i % 3
<<=	i <<= 3	i = i << 3
>>=	i >>= 3	i = i >> 3
&=	i &= 3	i = i & 3
^=	i ^= 3	i = i ^ 3
|=	i |= 3	i = i | 3

Reelle Datentypen (Gleitpunkttypen)

vordefinierte Datentypen

`float`	(32 Bit)	7	Stellen hinter
`double`	(64 Bit)	15	Dezimalpunkt
`long double`	(80 Bit)	19	bei Normierung

Gleitpunktliterale

```
-123.789e6f        1.8E6  88.009 1e-03
  1.8L
```

`1.8E6  88.009 1e-03`: Datentyp `double`

Konstanten (compilezeitbestimmt)

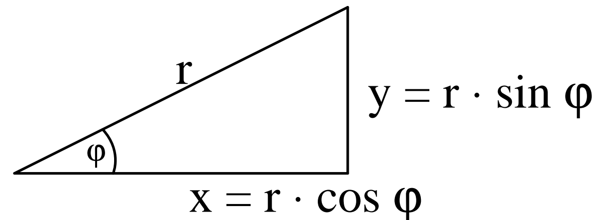
```
float const PI = 3.1415927;
float const PI_QUADRAT = PI * PI;
```

Operatoren

- arithmetische wie bei ganzzahligen bis auf `%`
- relationale wie bei ganzzahligen
- Kurzform bei Zuweisungen wie bei ganzzahligen

Beispiel für Ausdrücke mit vordefinierten Funktionen

Umrechnung von Polarkoordinaten in kartesische




```
#include <iostream.h>
#include <math.h>

double const PI = 3.141593;

void main() {
   double radius, winkel, x, y;

   /* Umrechnung von Polarkoordinaten
      in kartesische */

   cout << ”Geben sie Radius in cm und ”
        << ”Winkel in Grad ein: ”;
   cin >> radius >> winkel;

   x = radius * cos(PI*winkel/180.0);
   y = radius * sin(PI*winkel/180.0);

   cout << ”Die kartesischen Koordinaten ”
        << ”in cm sind:” << endl;
   cout << ”x=” << x << ” y=” << y << endl;
}
```

Der Zeichentyp `char`

In einem Byte mit Vorzeichen durch `signed char` oder ohne Vorzeichen durch `unsigned char` dargestellt:

–128 ... 127   0 ... 255

ASCII – Zeichen 0 ... 127, Rest 128 ... 255 nicht standardisiert.

```
char const STERN = '*';
char zeichen; int wert;
zeichen = '*';
wert = int(zeichen);
            // Typkonversion →int: Stellenzahl
zeichen = char(wert);
            // wieder zugehoeriger char-Wert
```

char – Operatoren

Operator	Beispiel	Bedeutung
`<`	`d < f`	kleiner
`>`	`d > f`	größer
`<=`	`d <= f`	kleiner gleich
`>=`	`d >= f`	größer gleich
`==`	`d == f`	gleich
`!=`	`d != f`	ungleich

Zeichenliterale: `'z'`   z ist darstellbares Zeichen

Darstellung von \ , ' , " durch \\ , \' , \"

Darstellung nicht druckbarer Zeichen: \o bzw. \xh
wobei o bzw. h die Stellenzahl im ASCII–Zeichenformat in oktaler bzw. dezimaler Darstellung ist.

Der Datentyp `bool` (spezieller ganzzahliger Datentyp)

Literale:

`true` ≙ wahr
`false` ≙ falsch

```
bool gr_Buchst, kl_Buchst, Buchst;
char c;
cin >> c;
gr_Buchst = (c >= 'A') && (c <= 'Z');
kl_Buchst = (c >= 'a') && (c <= 'z');
Buchst = gr_Buchst || kl_Buchst;
cout << Buchst;
```

`bool` – Operatoren

Operator	Beispiel	Bedeutung
`!`	`!i`	logische Negation
`&&`	`a && b`	logisches UND
`||`	`a || b`	logisches ODER

Aufzählungstypen (intern ganzzahlig)

```
// Typdeklarationen
enum Betr_Zust {an, aus};
enum Erg_Zust  {richtig, eing_f, ber_f1,
                ber_f2, ausg_f};
enum Wochentag {mo, di, mi, dn, fr, sa, so};

// Objektdeklarationen
Wochentag Werktag = mo; //mit Initialisierung
Erg_Zust z1, z2;        //mehrere Variablen
enum {Start, Betrieb, Ende} Status;
                        //anonyme Typdeklaration
```

`Werktag = mi;` später z.B. `Werktag = dn;`
`Status = Start;` später `Status = Ende;`
`z1 = ber_f1;`

intern zugeordnete Werte 0, ..., # Aufz.typ – 1

bei Erg_Zust: 0, 1, 2, 3, 4; braucht uns i. a. nicht zu interessieren

für EA, für Sonderfälle Wertebelegung angebbar.

Alternativ für `Erg_Zust`:

```
enum Erg_Zust {richtig=1, eing_f=2,
               ber_f1=4, ber_f2=8,
               ausg_f=16};
```

Bemerkungen:

1) intern auf ganzzahlige Werte abgebildet;
2) anonyme Typdeklaration nur, wenn nur eine Variable benötigt wird;
3) ganzzahlige Wertzuordnung nur für EA bzw. wenn für effiziente Berechnung nötig. Aufzählungswerte für Indexbereiche von Feldern nutzbar, wird durch Zuordnung erschwert.

Empfehlungen:

1) in C++ können mit `char`- oder `bool`-Werten und mit Aufzählungswerten als `int`-Werte gerechnet werden: nicht empfehlenswert!

2) Entscheiden Sie sich bei jeder Deklaration, welchen Basistyp sie verwenden:

   `short`, `int`, `long` und dabei `unsigned` oder normal für ganzzahlige Konstante oder Variable,
   `float`, `double`, `long double` für Gleitpunkt–Deklarationen.

3) Wenn Sie Ausdrücke bilden, achten Sie darauf, daß alle Operanden gleichen Typ haben; ebenso linke und rechte Seite einer Zuweisung.

4) In C++ gibt es viele implizite Typkonversionen innerhalb der ganzzahligen Typen, bzw. Gleitpunkttypen, aber auch zwischen beiden: nicht empfehlenswert!

Fallunterscheidungen

bisher Zuweisungen, ggf. in verkürzter Form wie `m+=1`, und Anweisungsfolgen

Betrachtung verschiedener Fälle

abhängig vom Wert eines Boole'schen Ausdrucks
(bedingte Anweisung, if-Anweisung)

abhängig von konkreten, explizit angegebenen Werten
(Auswahlanweisung, case-Anweisung, hier switch-Anw.)

Bedingte Anweisungen

```
if (schalter == an) {
   // Anweisungsfolge
};
```

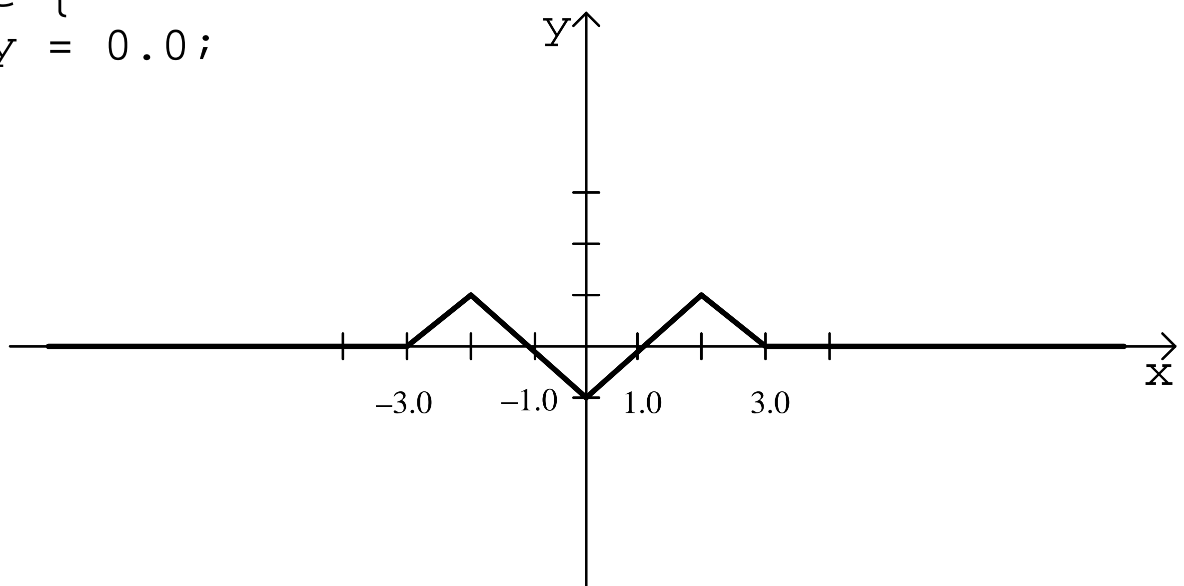
} einseitig bedingte Anweisung

```
// Vorzeichenbestimmung
if (zahl >= 0) {
   sig = 1; }
else {
   sig = -1;
};
```

} zweiseitig bedingte Anweisung

```
// Berechnung des Ordinatenwerts einer
// stückweise zusammengesetzten Funktion:

if (-3.0<=x && x<-2.0) {
   y = x+3.0;}
else if (-2.0<=x && x<0.0) {
   y = -1.0*x-1.0; }
else if (0.0<=x && x<2.0) {
   y = x-1.0; }
else if (2.0<=x && x<3.0) {
   y = -1.0*x+3.0;}
else {
   y = 0.0;
};
```

allg. bedingte Anweisung simuliert
→ Aufgaben



Fallstricke:

Dangling-else-Problem
falsches Semikolon
zusammengesetzte Bedingungen

Ratschlag:

stets Block für Anweisungsfolge: then-Teil, else-Teil, elsif-Teile
vermeidet dangling else und erleichtert Einfügen weiterer Anweisungen

Bsp: jeder Wert ein Fall

```
int wert;
char eingabe;
cin >> eingabe;
switch (eingabe) {
   case 'I' : wert =    1; break;
   case 'V' : wert =    5; break;
   case 'X' : wert =   10; break;
   case 'L' : wert =   50; break;
   case 'C' : wert =  100; break;
   case 'D' : wert =  500; break;
   case 'M' : wert = 1000; break;
   default : cout <<
          "Ist keine roemische Ziffer!";
};
```

gleiche Behandlung für unterschiedliche Fälle:

```
switch Tag {
   case mo : Wochenanfangsbilanz(); break;
   case di : case mi: case dn :
   case fr : normale_Tagesbilanz(); break;
   case sa : Wochenabschlußbilanz(); break;
   default : Fehlerabbruch();
             // Fall wurde uebersehen
};
```

Bemerkungen:

- break-Anweisung verläßt die gesamte Anweisung
- es gibt keine Auswahllisten wie in anderen Sprachen
  di, mi, dn, fr oder di .. fr ; Simulation
- Werte in allen Fällen paarweise verschieden
- break-Anweisung nicht vergessen, sonst wird bei nächster
  Alternative weitergemacht
- Viele Fälle: besser bedingte Anweisungen anstelle von
  Auswahlanweisungen. Tritt in C nicht auf, da nicht
  bequem hinzuschreiben

Schleifen (Iterationen, Wiederholungen)

wiederholte Ausführung einer Anweisungsfolge

Unterscheidung: Anzahl der Fälle bekannt von 1 .. 100:
Zählschleife, for-Schleife

Anzahl der Fälle unbekannt :
while- oder until-Schleife, beides bedingte Schleifen

Keine Bedingung angegeben: Endlosschleife

Zählschleifen

$$fak(n) = \prod_{i=1}^{n} i$$

```
int nfak, n = ...;
// n - Fakultät für ein n, iterativ

// aufsteigend berechnet; n >= 1
nfak = 1;
for (int i=1; i<=n; i++) {
   nfak = nfak*i; // abg.: nfak *= i;
}

// ueblich absteigend; wieder n >= 1
nfak = 1;
for (int i=n; i>=1; i--)
   nfak = nfak*i; // abg.: nfak *= i;
}
```

```
// Fakultaetswerte von 1 bis 20
for (int n=1 ; n<=20; n++) {
   nfak = 1;
   for (int i=n; i>=1; i--) {
      nfak = nfak*i; // abg.: nfak *= i;
   }
   // Ausgabe der Fakultaetswerte
}
```

Bem.:

- for-Schleife in C++ auch anderweitig nutzbar: Semantik
  for (Initialisierung; Bedingung; Ausdruck) { Rumpf }:

  ```
  {Initialisierung
    while (Bedingung) {
      Schleifenrumpf;
      Ausdruck; // für Inkrem./Dekrem.
    }
  }
  ```

- Zählschleife kein Terminationsproblem; bei C++ aufpassen!
- for Schleife von C nur als Zählschleife verwenden
- insbes. keine Veränderung der Schleifenvariablen im Rumpf der Schleife!

Bedingte Schleifen

1. Problem: Summe von mehreren Werten ($\neq 0$), jetzt von außen eingegeben, “Länge” der Eingabe nicht vorbestimmt, 0 sei Ende der Eingabe.

while - Schleife:

```
...
int summe = 0, eingabe;
cin >> eingabe;

while (eingabe!=0) {  // Fortsetzungsbedingung
     // ist Negation einer Abbruchbedingung
   summe += eingabe;
   cin >> eingabe;
}
// Ausgabe
```

alternativ mit `do-while`-Schleife, wenn mindestens ein Wert $\neq 0$

```
do {  // keine until-Schleife; Pruefung
      // einer Fortsetzungbedingung am Ende
   cin >> eingabe;
   summe += eingabe; }
while (eingabe != 0);
// Ausgabe
```

2. Problem: Berechnung bis bestimmte Genauigkeit erreicht wird

```
float wurzel = 1.0, x;
float const eps = 1.0e-4;
cin >> x;
while (abs(wurzel*wurzel - x) > eps) {
   wurzel = 0.5*(wurzel + x/wurzel);
};
```

Bem.:

- üblicherweise while Schleifen, until-Schleifen
- until-Schleifen mit Abbruchbedingung simuliert durch:

  ```
  do { Rumpf }
  while (!Abbruchbedingung);
  ```

- Terminationsproblem!

Endlosschleifen

```
int summe = 0, eingabe;
while (true) {
   cin >> eingabe;
   if (eingabe==0) {
      break;
   }
   summe += eingabe;
}
```

Kontrollierte Sprünge

Sprunganweisungen:

```
break;// zum Verlassen der unmittelbar
      // umschließenden Programmeinheit,
      // z.B.Schleifen,Auswahlanweisungen
continue; // innerhalb Schleifen, sofort
      // nächster Durchlauf
return [expression]; // bei Unterprogrammen
      // sowie Hauptprogramm zur
      // Rückgabe eines Wertes (oder
      // Zustandsinformation)
goto label_identifier; // Sprung zu Sprungmarke
      // innerhalb eines Unter-
      // oder Hauptprogramms
```

Beispiele:

`break`	für Auswahlanweisung, für Verlassen von (Endlos)Schleifen
`continue`	entspricht exit zum Ende Schleifenrumpf; Beispiel später
`return`	(→ nächster Abschnitt); ferner für Rückgabe eines Return-Parameters
`goto`	(→ Aufgaben)

Bem.:

- strukturierte Programmierung: Sequenz, Iteration,
  Fallunterscheidung, keine Sprünge (goto-Kontroverse)
- saubere Sprünge: Sprungmarken sind Anfangs- oder
  Endpunkte von Kontrollstrukturen;
  bei `break`, `continue` und `return` der Fall

  strukturierte Prgogrammierung mit sauberen Sprüngen
- goto sorgfältig anwenden:
  (a) für Simulation von Kontrollstrukturen
  (b) für Effizienzverbesserung für Programme mit sauberen Sprüngen
  (c) für Programmstücke mit anderweitig klarer Struktur

Felder (Reihungen, Arrays)

Eingangsbeispiel Summenberechnung zweier Werte,
jetzt von 10 `float`-Werten: Deklaration eines Feldes,
Zusammenfassung von Werten des gleichen Typs

Feldobjektdeklaration, Feldtypdeklaration

```
unsigned const dimVal = 10;
float werte[dimVal]; // ein Feldobjekt
//Feldelemente werte[0],werte[1],...,werte[9]
//erstes Element hat Indexwert 0, letztes
//dimVal-1;
```

Oben implizite oder anonyme Felddeklaration. Nur falls genau ein Feldobjekt benötigt wird, oder diese Strukturen nicht anderweitig verwendet werden.
Besser mit expliziter Typdeklaration:

```
typedef float Feldtyp[dimVal];//Typdeklaration
```

Typdefinition Feldtyp

```
Feldtyp werte;  // Objekdeklaration des Typs
                // Feldtyp


typedef enum tage
            {mo, di, mi, dn, fr, sa, so};
typedef int StdProTag[so-mo+1];
```

Initialisierung von Feldern:

über Feldaggregate

```
Feldtyp werte = {1.0,2.0,3.0,4.0,5.0,6.0,
        7.0,8.0,9.0,10.0};
  // in Objektdeklaration,
  // falls Werte nicht veraendert werden:
Feldtyp const werte = ... (wie oben)
  // {} alle Komponenten von werte mit 0.0
  // initialisiert
```

oder über Zählschleifen

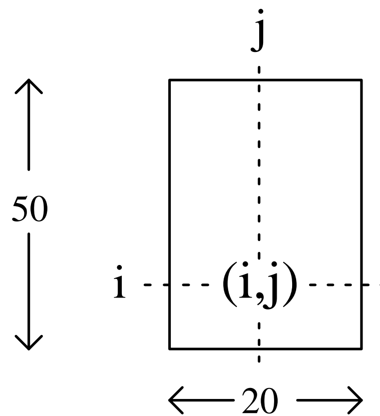
bei großer Komponentenzahl, komplexer Berechnung der Komponentenwerte:

```
for (int i=0; i<dimVal; i++) {
   werte[i] = float(i*i); // Indizierung,
                          // Komp. Zugriff
}
```

Felder und Zählschleifen für Berechnung: (Initialisierung (s.o.))

```
unsigned const dimVal=10;
typedef float f_A_T[dimVal];
f_A_T Werte;
float Summe=0, Mittelwert;
// Zaehlschleife z. Festlegen der Werte,
// evtl. über Eingabe
...
// Berechnung des Mittelwertes:
for (int i=0; i<dimVal; i++) {
   Summe += Werte[i];
};
Mittelwert = Summe/dimVal;
// Ausgabe ...
```

Mehrdimensionale Felder




```
double Mat[50][20]; // Feldobjekt,
  // 20x50-Matrix mit double-Komponenten;
  // ist eindim. Feld aus eindim. Feldkomp.
unsigned const dim1Val=50, dim2Val=20;
typedef double dZeilenT[dim2Val];
  // Zeilenvektortyp
typedef dZeilenT dMat_50_20_T[dim1Val];
  // Vektor aus Zeilen
dMat_50_20_T Mat;


// Bestimmung der minimalen Feldkomponente
// von Mat über 2 geschachtelte Schleifen:

double min = Mat[0][0];
for (int i=0; i<dim1Val; i++) {
   for (int j=0; j<dim2Val; j++) {
      double const Mij=Mat[i][j];
        // entspricht (Mat[i])[j]
      if (Mij<min) { min = Mij; };
   };
};
```

Zeichenfelder

Zeichenketten werden in C als Zeichenfeld mit '\0' (Terminator) als Endkennung abgespeichert. Wir besprechen zunächst statische Zeichenfelder. (Dynamische Zeichenketten → Zeigerabschnitt)

```
// ZK, ZF sind Zeichenfelder konst. Laenge
char ZK[] = "Z_Kette"; // rechts ZK-Literal,
                   // obere Grenze aus Literal
char const ZF[8] = "Z_Kette";
  // fuer Literal auch gleichbedeutend
  // {'Z','_','K','e','t','t','e','\0'}
```

Bemerkungen:

- statische und dynamische Felder: statisch d.h. Anzahl der Komponenten zur Compilezeit bekannt; dynamisch in Verbindung mit Zeigern
- mehrdimensionale Felder mittels Komposition aus eindimensionalen (Matrix: Vektor von Zeilen); Ablage im Datenspeicher: Zeilen hintereinander.
- keine Überprüfung der Indexgrenzen bei Zugriff, keine Zuweisung ganzer Felder
- Begriffe: Feld, Feldkomponente, Feldtypdefinition, Feldtypdeklaration, Feldobjektdeklaration, Zugriff (Indizierung) auf Feldkomponente, Aggregate, Initialisierung eines Feldobjekts, statische/dynamische, eindimensionale/mehrdimensionale Felder, Zeichenfelder, ZK-Literal, Aggregat für Zeichenfeld

Verbunde

Name: Verbund, Record, Strukturen, etc.

Zusammenfassung unterschiedlicher Komponenten

Typdeklaration, Objektdeklaration einfacher Verbunde

```
enum FarbT {rot, gelb, blau};
struct PunktT{                    // Typ PunktT;
   float x, y; // koordinaten;   Typdefinition
   FarbT farbe;
   bool sichtbar;
   bool in_Ausschnitt;
} pkt1;                        // Objektdeklaration

// besser getrennt
struct PunktT {       // Typdeklaration
   s.o.
};
PunktT pkt1, pkt2;    // Objektdeklarationen
```

Zugriff auf Komponenten

```
pkt1.x = 12.0; pkt1.y = 15.0;
pkt1.farbe = rot; pkt1.sichtbar = true;
if((x1<=pkt1.x && pkt1.x<=x2)&&
   (y1<=pkt1.y && pkt1.y<=y1)) {
   pkt1.in_Ausschnitt = true;}
else {
   pkt1.in_Ausschnitt = false;
};
```

Anweisungsfolge für die Verarbeitung der (einiger) Komponenten

```
pkt2.y = pkt1.y; pkt2.x = pkt1.x;
             //gleiche Koordinaten für pkt2
```

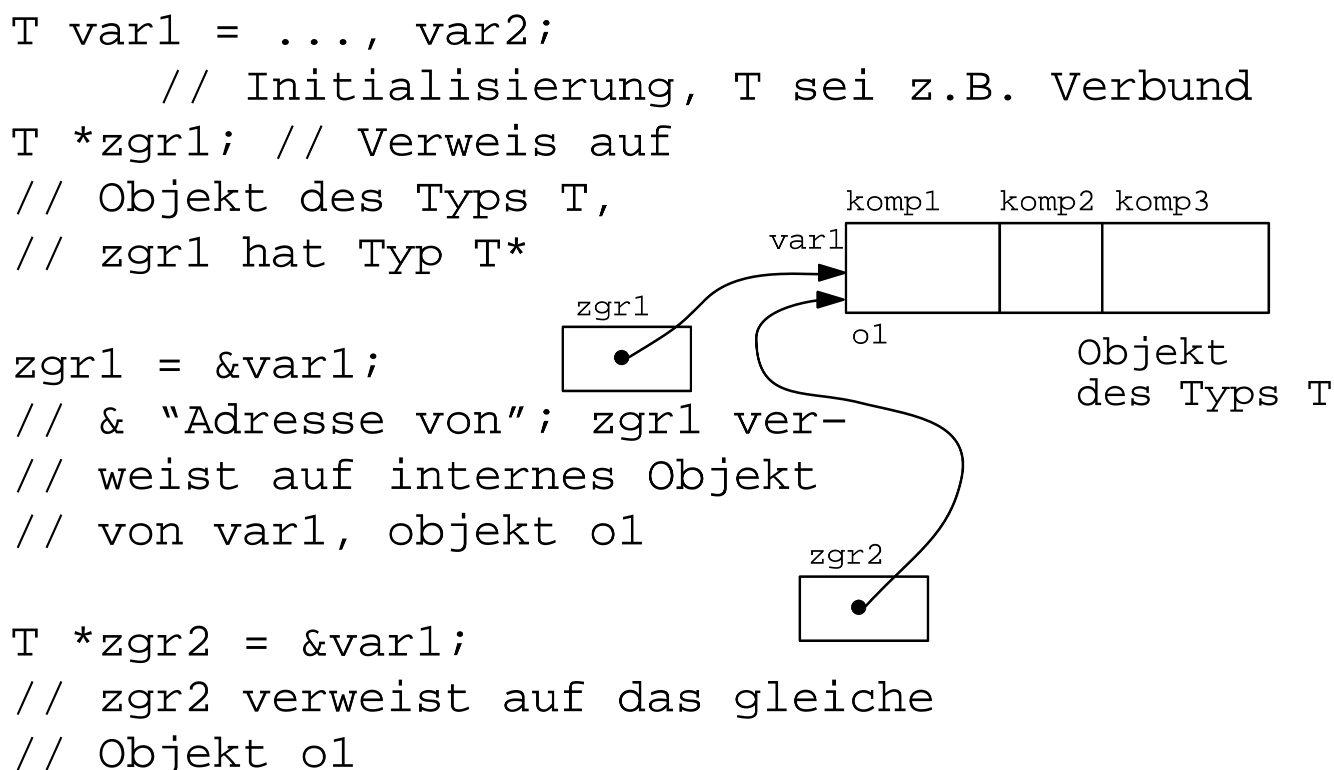
Zeiger und Referenzen

Bisher Objekte über Bezeichner eingeführt,
Objekte von bel. zusammengesetztem Typ,
unauflösliche Bindung Bezeichner, interner Wert (an best. Adresse).

Jetzt Zeiger (Verweis, Pointer, Zugriff) auf Objekte, die
nacheinander auf verschiedene Objekte des gleichen Typs
verweisen können.

Zeigerdeklarationen

```
T var1 = ..., var2;
     // Initialisierung, T sei z.B. Verbund
T *zgr1; // Verweis auf
// Objekt des Typs T,
// zgr1 hat Typ T*

zgr1 = &var1;
// & "Adresse von"; zgr1 ver-
// weist auf internes Objekt
// von var1, objekt o1

T *zgr2 = &var1;
// zgr2 verweist auf das gleiche
// Objekt o1
```



oder alternativ

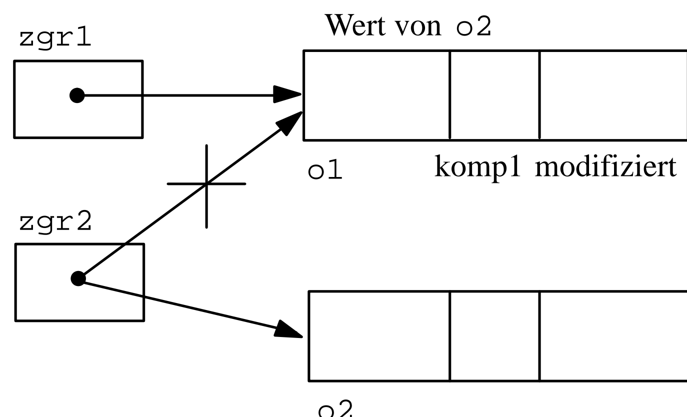
```
T *zgr2 = ptr1;
// zgr2 erhält Zeigerwert von zgr1, beide
// verweisen auf das gleiche Objekt o1
```

Zeiger sind typisiert (`T*`) und können nicht auf Objekte eines
anderen Typs verweisen

Zuweisung und Dereferenzierung

Anstatt der letzten initialisierten Deklaration hätten wir schreiben können:

```
T *zgr2;
zgr2 = zgr1;
// Zuweisung d. Zeigerwerts von zgr1 an zgr2


var2.komp1=...;var2.komp2=...;var2.komp3=...
// Objekt var2 (o2) erhaelt (neuen) Wert;
// bisher verweisen zgr1 und zgr2 auf das
// Objekt o1


*zgr1 = var2; // zgr1 verweist auf Objekt o1
// Dereferenzierung: Uebergang von Zeiger zu
// angezeigtem Objekt; dieses Objekt erhält
// neuen Wert (von o2)
*zgr1.komp1 = ...;
// komp1 von o1 erhaelt neuen Wert

// oder kuerzer fuer *zgr1.komp1 bei
// Veraenderung einzelner Komponenten:
zgr1->komp1 = ...;

zgr2 = &var2;
// zgr2 verweist
// jetzt auf o2
```



alternativ für letzteres, da `var1` und `var2` im Speicher hintereinander liegen:

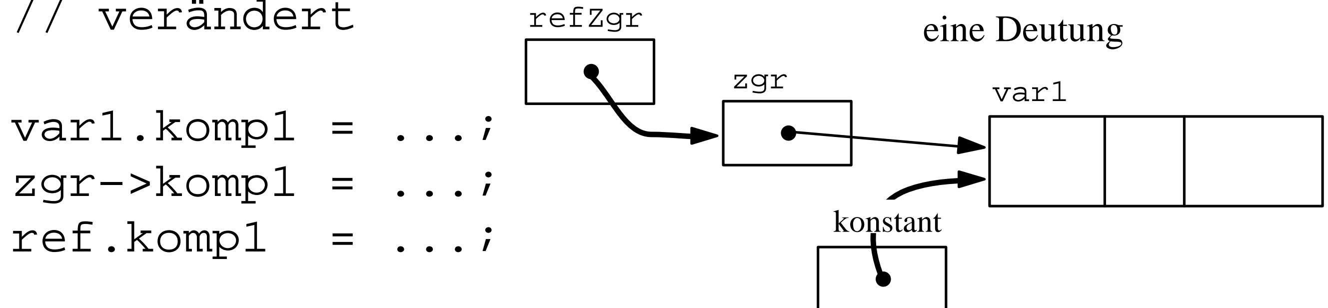
```
zgr2 = zgr1 + sizeof(var1);
       // gefaehrlich und funktioniert nicht
```

Bemerkungen:

- nicht `T* zgr1, zgr2;` schreiben; `zgr1` ist vom Typ `T*`, `zgr2` vom Typ `T`
- bisher Verweise auf Objekte im Laufzeitkeller, später auf Halde
- bei `T const *zgr1 = &var1;` ist `zgr1` ein Zeiger auf ein konstantes Objekt, der Zeiger ist nicht konstant; verschiedene Deklarationen verwenden

Referenzen

```
T var1;
T var2;
T *zgr = &var1; // Zeiger auf var1
T &ref = var1;  // ref ist Referenz auf var1
// ref ist neuer Name (Alias) für var1; wird
// bei Deklaration gesetzt und nicht mehr
// verändert
```




```
var1.komp1 = ...;
zgr->komp1 = ...;
ref.komp1  = ...;

ref = &var2; //nicht erlaubt
T *&refZgr = zgr; // Referenz auf Zeiger
ref = var2;  // Referenz bleibt gleich
             // Objekt wird verändert;
             // Wirkung von var1 = var2
```

Referenzen zum Abkürzen von Zugriffspfaden:

```
K11T &Komp = var1.komp1.komp11; // K11T, K2T
K2T &bestKomp = F[251]; // entspr. dekl. Typen
```

Bemerkungen:

- Zeiger auf Referenzen sowie Referenzen auf Referenzen unzulässig, Referenzen auf Zeiger wohl
- für alternativen Namen auf Variable, Konstante, Funktion (→ später)

Unterprogramme

Funktionen:	liefern Berechnungswert als Ergebnis; werden in Ausdruck aufgerufen
(Verfahrens-) Prozeduren:	liefern komplexere Berechnung haben Charakter einer komplexen Anweisung, Wert des Typs `void` (undef.)

Funktionen

Funktionsspezifikation

(Funktionskopf, in C Prototyp genannt)

Syntax:

`Rueckgabetyp Funktionsbez(`**Formalparameterliste**`);`

```
int strlen(char const *string);
                    // Funktionsprototyp, mit
                    // Formalparameterliste
void main() {
   char *str = ”Eine Zeichenkette”;

   cout << str << ” hat die Länge: ”
        << strlen(str) << endl; // Aufruf
}
```

Formen der Parameterliste einer Funktionsspezifikation

leere Liste:	`int func();`
gleichwertig ist:	`int func(void);`
Liste mit Parametertypen:	`int func(int, char);`
Liste mit Parametertypen und –bezeichner:	`int func(int x, char y);`

Empfehlung: mit Parameterbezeichnern

Funktionsrumpf

andere Namen: Funktionsdeklaration, –implementation

Wiederholung der Schnittstelle + Implementation (Berechnungsvorschrift)

Syntax:

`Rueckgabetyp Funktionsbez(`**Formalparameterliste**`)` **block**

Die Formalparameterliste muß für jeden Parameter einen Namen einführen

```
// Funktionsimplementation
int strlen(char const *x) {
   int i = 0; char const terminator = 0;
   while (x[i]!=terminator) {
      i++
   };
   return i;
}
```

Funktionsaufruf

Syntax:

`Funktionsbez(`**Aktualparameterliste**`)`

```
void main() {
   ... // s.o.
   cout << strlen(str);
            // Ausgabe eines Wertes
}
```

Vordefinierte Parameter

Vorgabe- / Vorbesetzungsparameter (engl. defaults) nach den anderen Parametern.

Beispiel:

`hex` konvertiere `long(int)`-Zahlen in Zeichenkette aus Hexadezimalzahlen. Im zweiten Parameter bedeutet 0: Zeichenkette gerade so lang, wie benötigt; sonst wie vom Aufrufer angegeben.

```
extern char* hex(long, int = 0);

cout << "**" << hex(31) << hex(32,3) << "**";
```

wird interpretiert als:

```
cout <<"**"<< hex(31,0) << hex(32,3) << "**";
```

und beide liefern das gleiche:

```
**1f 20**
```

Prozedur = komplexere Berechnung; Aufruf ist Anweisung

```
float erg;
...
void Hoch(float basis, unsigned int exp,
          float &resultat){    // Deklaration;
   // Die Prozedur berechnet basis^exp
   // basis, exp  Eingangsparameter,
   // resultat    Ausgabeparameter

   float tmp = 1.0;
   while (exp>0) {
      if ((exp%2)!=0) {
         // fuer exp ungerade
         tmp = tmp*basis;
      }
      basis = basis*basis; exp = exp/2;
   };
   resultat = tmp;
};
...
Hoch((a*y)/1.5, k, erg); // Aufruf
   // a,y als float dekl., k als unsigned


...
void exchange (float &x, float &y) {
   // Vertauschung zweier Werte;
   // beide Parameter sind Ein-/Ausgabeparam.
   float zwischen;
   zwischen = x; x = y; y = zwischen;
}
...
exchange(a,b); // Aufruf
```

Bedeutung eines Unterprogrammaufrufs und Parameterübergabemechanismen

Übergabemechanismen:

call-by-value: (Aufruf über den Wert)	Eingabeparameter wird in Formalparameter kopiert
call-by-reference (Aufruf über die Adresse)	Aktualparameter wird genommen und verändert

Erklärung Bedeutung Prozeduraufruf

```
float erg;
...
void Hoch(float basis, unsigned int exp,
          float &resultat){
// die Prozedur berechnet basis^exp

   float tmp = 1.0;
   while (exp>0) {
      if ((exp%2)!=0) {
         // fuer exp ungerade
         tmp = tmp*basis;
      };
      basis = basis*basis;
      exp = exp/2;
   };
   resultat = tmp;
}
...
Hoch((a*y)/1.5, k, erg);
```

```
basis=a*y/1.5;
exp=k;
```

Rumpf von Hoch als Inkarnation

```
{ tmp=1.0;
  while (...)
   ...
  resultat=tmp;
}
```

Aufruf mit Zeigern

- ist Aufruf über den Wert, d.h. der Zeiger wird nicht verändert (ggf. die lokale Kopie)
- Das angezeigte Objekt kann in der Prozedur natürlich geändert werden

Hauptprogramm

Funktion main

```
int main() { // int kann entfallen
   ...
   return 0;
}
```

Rückgabewert z.B. für gezielte Reaktion auf Fehler.

```
void main() {
   ...
} // braucht keine return-Anweisung
```

- Unterprogramm für verschiedene Komponentenzahl

```
...
unsigned const dimVal = 100;
typedef float f_FT[dimVal];
f_FT xv, yv;
...
float Skalarprodukt(f_FT avekt,f_FT bvekt){
   int og1, og2;
   float erg = 0.0;
   og1=int(sizeof(f_FT)/sizeof(float));
   og2=int(sizeof(f_FT)/sizeof(float));
   if(og1!=og2) { ... // Fehlerabbruch };
   for (int i=0; i<og1; i++) {
      erg = erg + avekt[i] * bvekt[i];
   }
   return erg;
}
```

- jetzige Formulierung noch vom Typ der Komponenten abhängig; vom Komponententyp nur verlangt, daß +, * verfügbar; später Nutzung von Generizität
- bei größeren Vektoren call-by-reference, obwohl Felder hier Eingabeparameter sind!

Zeiger und Unterprogramme

Parameter und Zeiger

```
unsigned const dimVal=10;
float erg;
typedef float f_FT[dimVal];
f_FT *pF1, *pF2;
  // pF1 und pF2 sind Zeiger auf Felder
```

```
typedef f_FT* Z_f_FT;
Z_f_FT pF1, pF2;
```

```
float Skalarprodukt(f_FT *avekt,
                    f_FT *bvekt) {
  // Formalparameter jetzt Zeiger auf Felder
  // Ersetze in der letzten Formulierung
  // jeweils avekt bzw. bvekt durch
  // (*avekt), (*bvekt)
   ...
}

// Felder initialisieren über EA oder mit
// Schleifen
...

erg = Skalarprodukt(pF1, pF2); // Aufruf
```

übliche, unsaubere Lösung in C

```
unsigned dimVal=10; float erg;
typedef float *Z_f;
typedef float f_FT[dimVal];
f_FT F1, F2; // F1, F2 sind Zeiger auf die
             // erste Komponente
Z_f Feldanker1=F1; Feldanker2=F2;
  // Feldanker1, Feldanker2 zeigen je auf
  // erste Komponente und implizit auf
  // F1, F2

float Skalarprodukt(Z_f avekt, Z_f bvekt,
            unsigned len1, unsigned len2) {
  // Formalparameter jetzt Zeiger auf
  // Anfangskomponente; len1, len2 Laenge
  // der zu uebergebenden Felder

  if (len1 != len2) { ...
    // Fehlerabbruch; keine Garantie, dass
    // Felder gleiche Laenge haben
  }
  // ab hier wieder alte Formulierung

};
...

erg=Skalarprodukt(F1, F2, dimVal, dimVal);
       // Aufruf
```

Nutzung auch für dynamische Zeichenketten (zur Laufzeit variabel): wird hier nicht besprochen

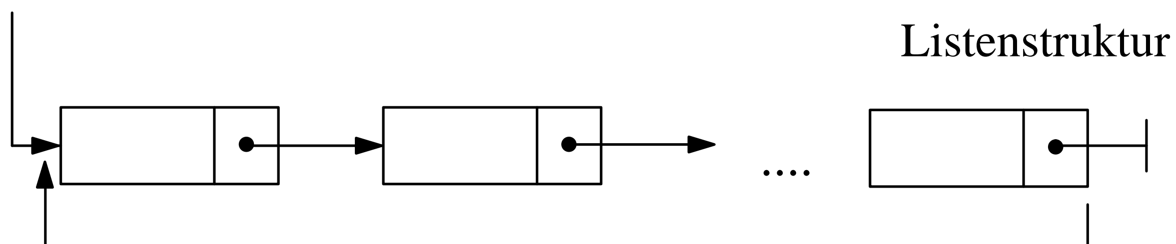
Zeiger auf Unterprogramme

```
#include <iostream.h>
int max(int x, int y) {
   if (x > y) { return x; }
   else       { return y; };
};
int min(int x, int y) {
   if (x < y) { return x; }
   else       { return y; };
}; //beide Funktionen haben gleiches Profil

int main() {
   int zahl1=10, zahl2=20; char eingabe;
   int (*fp)(int, int); // fp Zeiger auf
      // Funktion mit bestimmtem Profil
   while (true) {
      cout << "s=min, g=max, e=Ende:\n";
      cin >> eingabe;
      switch (eingabe) {
         case 's' : {fp = min; break;};
           // Zuweisung von min
         case 'g' : {fp = max; break;};
           // Zuweisung von max
         case 'e' : {goto Ende;};
         default: {cout<<"Falsche Eingabe";
                   continue;};
      };
  // Dereferenzierung des Funktionszeigers
  // und Aufruf der entsprechenden Funktion
      cout << (*fp)(zahl1,zahl2) << endl;
   };
   Ende: return 0;
}
```

Haldenobjekte, Listenverarbeitung

Zeiger und Haldenobjekte

- bisher Objekte beliebig komplizierter Struktur mit Typkonstruktor Feld, Verbund etc;
  in ihrer Struktur in Objektdeklaration festgelegt:
  “statische” Objekte
- Notwendigkeit dynamischer Objekte:



Anzahl nicht vorherbestimmbar zur Programmerstellungszeit, ändert sich zur Laufzeit

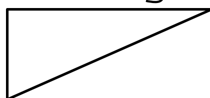
- bisher: Objektfestlegung durch Objektdeklaration
  unauflösliche Bindung externer Name – interner Wert
  Erzeugung des Objektes durch Abarbeitung einer Objektdeklaration

  jetzt: Objekterzeugung im Anweisungsteil
  Bereich auf den abgelegt wird: Halde (Heap)
  entsprechende Objekte: Haldenobjekte
- Haldenobjekte:
  durch Zeiger benannt (implizite, erzeugte Bezeichnung)
  durch Zeiger untereinander verbunden (verkettet, verzeigert)

Deklaration der Typen von Haldenobjekten, Zeigern

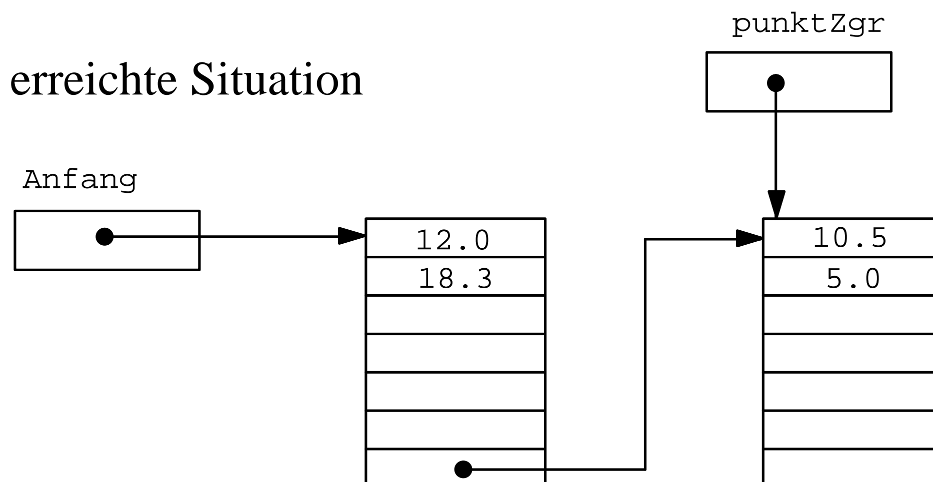
Wir beschränken uns auf Verbunde, da Objekte auf Halde verkettet werden sollen; Zeiger auf einzelne Felder auf der Halde ebenfalls möglich.

```
// Struktur der Haldenobjekte
void const *NULL=0;

struct PunktT;
typedef PunktT *zpunkt;
         // Zeiger auf Haldenobjekte
struct PunktT { // Struktur der Haldenobj.
   float x, y;
   farbT farbe;
   bool sichtbar;
   bool in_Ausschnitt;
   zpunkt n_koord; // Verw. auf n. Koordinate
   zpunkt next;  // Zeiger fuer Verkettung
};               // der Listenelemente

// Ankerzeiger
zpunkt Anfang=NULL; //zunaechst noch undef.
     // verweist spaeter auf Anfang der Liste
zpunkt punktZgr; // spaeter Zeiger auf akt.
                 // Element
```



Anlegen, Verändern, Verketten von Haldenobjekten

```
punktZgr = new PunktT;
           // 1. Element der Liste, Verweis
Anfang = punktZgr;
punktZgr->x = 12.0;
punktZgr->y = 18.3;
// andere Komponenten (farbe, etc.)
// noch undef.

punktZgr->next = new punkt;
 // erzeugt 2. Objekt und Verweis

punktZgr = punktZgr->next;

Anfang->next->x = 10.5;
 //Veraenderung ueber Zugriffspfad
punktZgr->y = 5.0 // Veraenderung ueber
                  // Laufzeiger
```

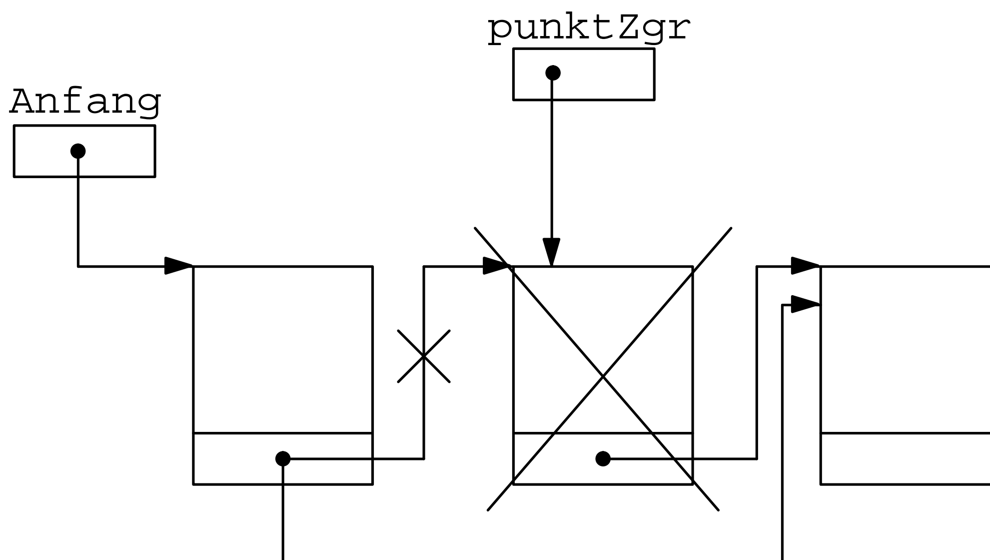


anonyme Haldenobjekte
Verweise bei Allokation jeweils mit erzeugt

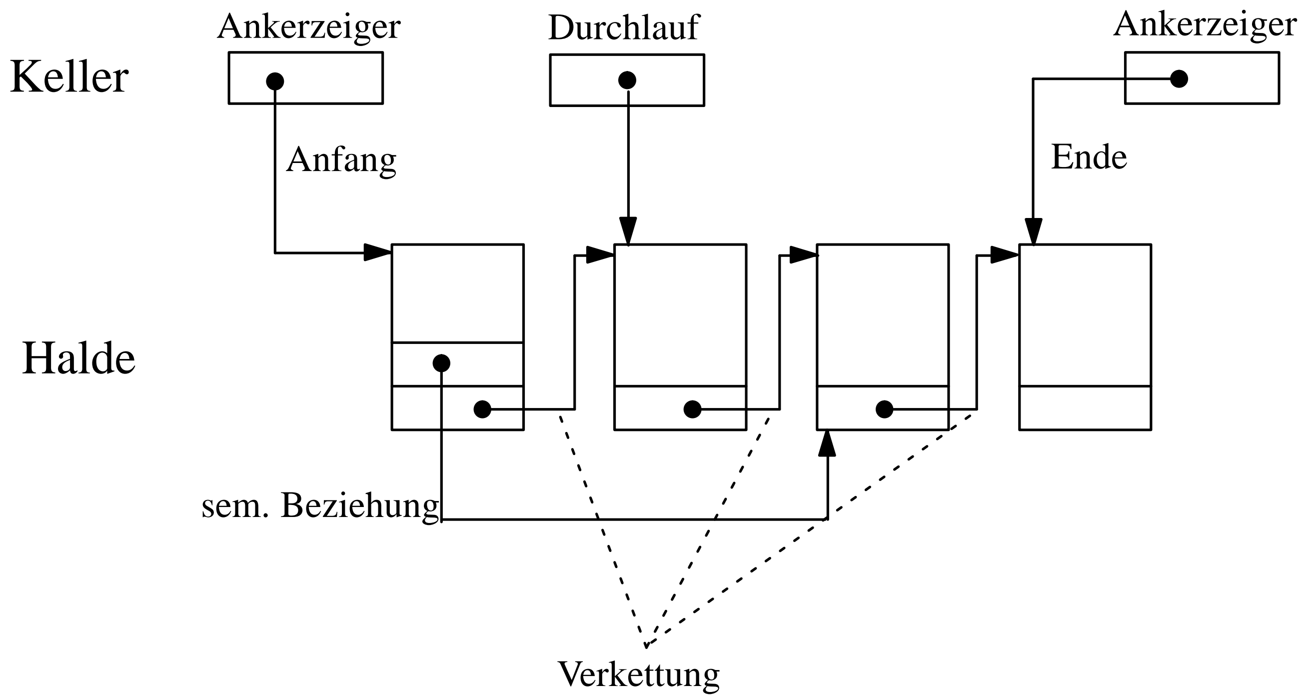
Erinnerung: auf Objekt (auf Halde oder Keller) können mehrere Zeiger verweisen.

Löschen von Elementen

```
// weiteres Listenelement

punktZgr->next = new PunktT;


// Loeschen des 2. Elements

punktZgr      = Anfang->next;
Anfang->next = punktZgr->next;
delete punktZgr;
```



Aufgaben von Zeigern



- Zeiger von außen:
  Einrichtung/Abkürzung eines Zugriffspfades
  Ankerzeiger
  Durchlaufzeiger
- Zeiger zwischen Objekten:
  Zusammenhang (Verkettung; z.B. für Durchlauf)
  inhaltliche (semantische) Beziehungen

Gefahren mit Zeigern

- unübersichtliche Strukturen
  (Zeiger sind gotos der Datenstrukturseite)
- Ansprechen/Verändern eines Objektes über verschiedene Namen (Aliasing):
  Veränderung eines Haldenobjekts über einen Zugriffspfad = Veränderung über anderen Zugriffspfad (Gefahr undurchsichtiger Programme)
- nicht mehr ansprechbares Haldenobjekt: ausgehängt (inaccessible object)
- hängender Zeiger nach `delete` (dangling references)

Effizienz und Zeiger

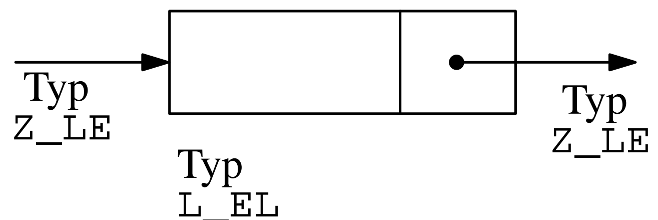
Speicherbereinigung (garbage collection)
wegen Zerstückelung der Halde:

- Kennzeichnung der freien Objekte
- Aufsammeln der freien Objekte
- Zusammenschieben des Haldenspeichers

Listenverarbeitung über Unterprogramme

einfache Deklarationen für lineare Liste

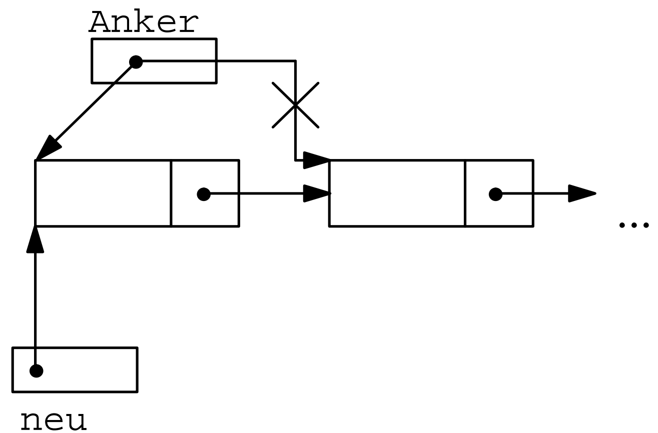
```
void const *NULL=0;
struct L_EL;
typedef L_EL *Z_LE;
struct L_EL {
   Info_T Info;
   //Info_T sei ein passend dekl. Typ
   Z_LE next;
};

Z_LE LinList;
```



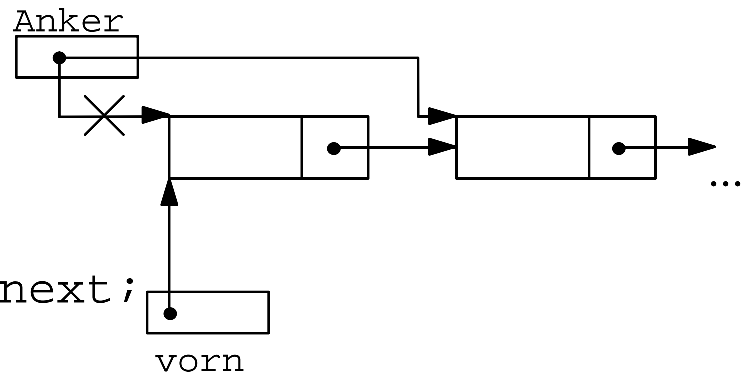
Strukturfestlegung

```
void LeereListe(Z_LE &Anker) {
   Anker = NULL;
};

unsigned LaengeDerListe(Z_LE ListenAnf) {
// Bestimme Laenge einer Teilliste
   unsigned Erg;

   if (ListenAnfang==NULL) {
      Erg=0; }
   else {
      Erg=1+LaengeDerListe(ListenAnf->next);
   };
   return Erg;
};
```

```
void AnfuegeVorn(Z_LE &Anker, Info_T i) {
// Fuegt vorne Listenelement ein;
// setzt Anker neu
   Z_LE neu;

   neu = new L_EL;
   neu->Info = i;
   neu->next = Anker;
   Anker = neu;
};
```




```
void LoescheVorn(Z_LE &Anker) {
// Loescht erstes Listenelement;
// setzt Anker neu

   Z_LE vorn;
   if (Anker!=NULL) {
      vorn = Anker;
      Anker = Anker->next;
      delete vorn; };
   else {
      ... // Fehlermeldung; Abbruch
   };
};
```




```
void Info_LE(Z_LE Anker, Info_T &i) {
   if (Anker!=NULL) {
      i=Anker->Info; };
   else {
      ... // Fehlermeldung; Abbruch
   };
};
```

```
// Verwendung der Liste

void main() {
   ...
   LeereListe(LinList);
   AnfuegeVorn(LinList, i1);
   AnfuegeVorn(LinList, i2);
   ...
   if (LaengeDerListe(LinList)!=0) {
      LoescheVorn(LinList);
   };
   ...
   if (LaengeDerListe(LinList)!=0) {
      Info_LE(LinList, i3);
   };
};
```

Weitere Operationen nötig: Navigieren, Einfügen innerhalb der Liste, Löschen innerhalb der Liste, etc.

Programmstruktur, Gültigkeit, Sichtbarkeit

Blockstruktur: lokale, globale Objekte

```
// Block 1:
{  unsigned i, j; // Dekl. 1,2

   // Block 1.1:
   {  unsigned i; // Dekl. 3
      float j;    // Dekl. 4

      j = ...; // Zuweisung an Objekt von
           // Dekl. 4: lok. Obj.;
           // Dekl. 2 verdeckt, aber gültig
   };
   ...
   i = ...; // Zuweisung an Obj. von Dekl.1
   j = ...; // Zuweisung an Obj. von Dekl.2
};
```

- globale Objekte, lokale Objekte
- Gültigkeitsbereich einer Deklaration:
  bis Ende Programmeinheit, hier Block
  außerhalb ungültig, Compiler prüft dies (Sicherheitsaspekt)
- Lebensdauer und Speicherverwaltung :
  außerhalb eines Blocks nicht gültig, nicht existent
  innerhalb eines Blocks ggfs. unsichtbar (verdeckt)
  nach Blockende Speicherbereich aufgegeben:
  Laufzeitkeller (effiziente Speicherverwaltung)

Unterprogramme: Gültigkeitsbereich, Existenzbereich

- Lokale Variablen einer Funktion/eines UPs außerhalb nicht gültig (wie bei Blöcken; keine Prozedurschachtelung in C) z.B. UP zum Vertauschen zweier Werte
- Formalparameter sind von Anzahl, Typ und Reihenfolge außerhalb sichtbar, die Formalparameternamen nicht
- Besonderheit:
  `static`-Variablen: Nur einmal auf statischem Speicherbereich angelegt, tauchen nicht in UP–Inkarnation auf

```
#include <iostream.h>

void func() {
     // zaehlt die Anzahl der Aufrufe
   static int anz = 0;

   anz++;
   cout << "Anzahl = " << anz << endl;
};

void main() {
   for (int i=0; i<6; i++) {
      func();
   };
}
```

Überladung von Funktionen

- Variablenbezeichner nicht überladbar
  gleiche Bezeichner = verschiedene Variablen, werden nicht durch Typ unterschieden
- Funktions– und Unterprogrammbezeichner überladbar:
  Verschiedene Deklarationen zum gleichen Funktions– oder Unterprogrammnamen:
  Überladung (overloading)
- Je nach Parametertypprofil (Anzahl, Reihenfolge, Typ der Parameter) wird vom Compiler das passende UP gewählt.

```
void exchange(double &x, double &y) {
   ....
};

void exchange(int &x, int &y) {
   ....
};


void main() {
   double dbl1 = 100.1, dbl2 = 33.3;
   int zahl1 = 5, zahl2 = 7;
   exchange(dbl1, dbl2);
   exchange(zahl1, zahl2);
}
```

Programmstruktur/Aufbau eines C-Programms

Compiler-Instruktionen (zum Beispiel `#include`)
Deklaration von globalen Variablen
Funktionsprototypen (Unterprogrammschnittstellen)

Implementierung, eigentliche Problemlösung:
`void main() {`
`  // Folge von Deklarationen und`
`  // Anweisungen };`

und Funktionsdefinitionen (Unterprogrammcode)

Beispiel:

```
#include <iostream.h>
float a=1.0, b=2.0;
  // globale Deklaration: Gefahr !
  // wie umschließender Block
void exchange(float &x, float &y);  // PT

void main() {
   // neuer Block
   float a = 10.0;
   //globales a gültig nicht sichtbar
   cout << ″globales a =″ << ::a << endl;
   // scope-Operator (::) erlaubt Zugriff
   // auf unsichtbares globales a
   {  int c=30; // neuer Block
   }
   cout << c << endl; // <- Compiler
   // liefert Fehler: c nicht gültig
   exchange(a,b);
};

void exchange(float &x, float &y) {...};
```

Resumee:

- lok. Deklarationen in Block, Unterprogramm:

  nahe an der Verwendung

  von außen nicht zugreifbar: Information Hiding
- globale Variable nicht (nur diszipliniert) anwenden
- Variablen werden verdeckt, Funktionen überladen
- static-Variable statisch
- scope-Operator erlaubt Zugriff auf globales evtl. verdecktes Objekt
- werden später andere und bessere Strukturierung eine C++–Programms kennenlernen!

Zusammenfassung

Anweisungen

- einfache Anweisungen
  - Zuweisungen
  - Prozeduraufrufe
  - Sprunganweisungen
- Zusammenges. Anweisungen
  - Anweisungsfolgen in Block
  - Fallunterscheidungen
    - bed. Anweisungen
    - Auswahlanweisungen
  - Schleifen
    - Zählschleifen
    - while-Schleifen
    - ”until-Schleifen”
    - Endlosschleifen

} Kontrollstrukturen (Vorschriften für Zusammenbau) für Ablaufkontrolle

Im Block oder Unterprogramm: Anweisungsteil
später auch in anderen Bausteinen

statischer Programmtext soll dyn. Durchlauf möglichst entsprechen

C ist wertorientierte Sprache; haben wir nicht genutzt; führt zu Programmierung mit Seiteneffekten

Datenstrukturierung

- einfache skalare Datenobjekte mittels
  - ganzzahliger Datentypen
  - reeller Datentypen
  - `bool`
  - `char`
  - Aufzählungstypen
- Zusammengesetzte Datenobjekte durch
  - Felder
  - Verbunde
  - Zeiger auf/in Datenobj.
  - Referenzen auf Datenobj.

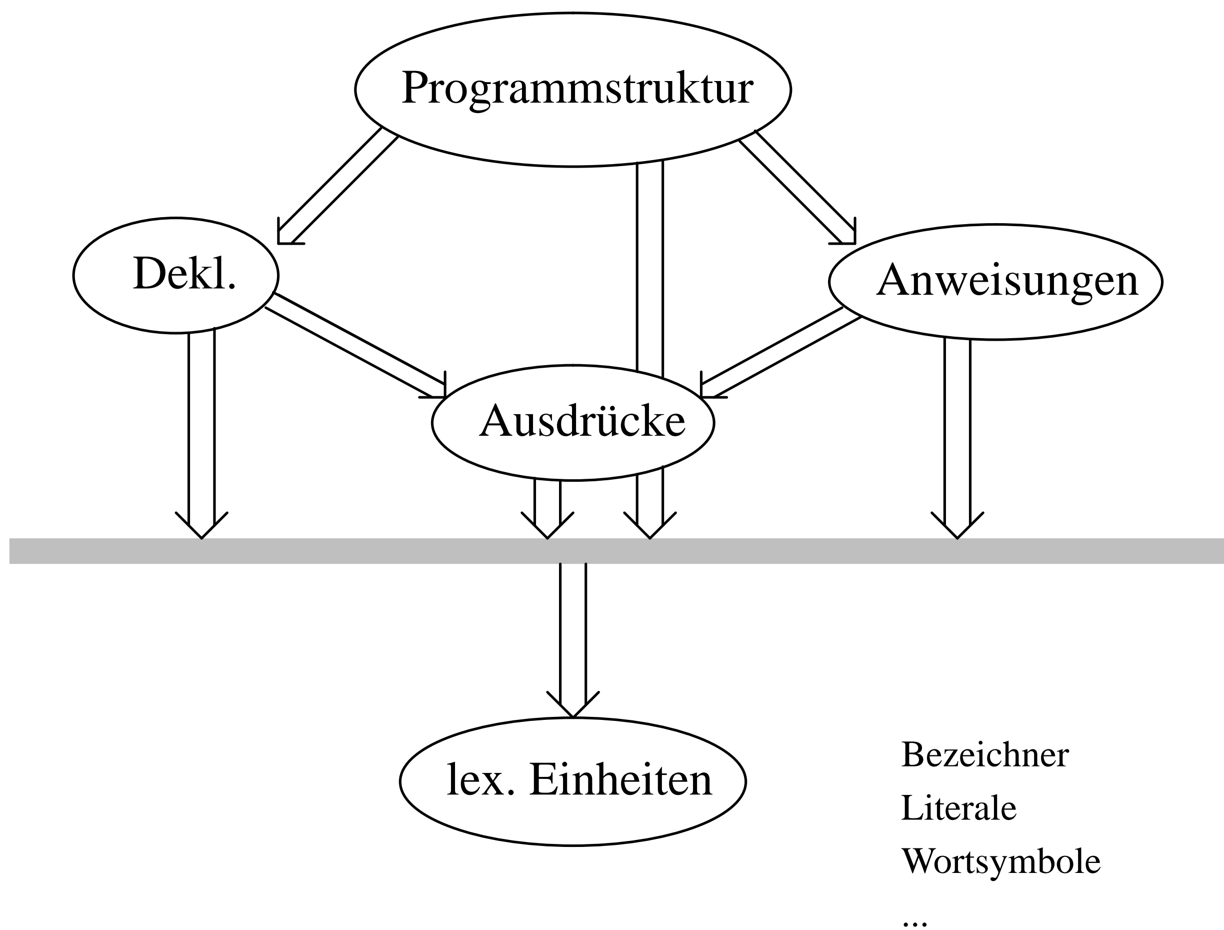
  } Datentypkonstruktoren: Vorschriften für Zusammenbau zusammeng. Datentypen

- Klassifikationen (orthogonal):

einfache DTen	⨝	vordef. DTen
zusammeng. DTen		selbstdef. DTen

- weitere Ergänzung durch Klassen

Gruppierung der Konstrukte in der k.f. Grammatik



Bezeichner
Literale
Wortsymbole
...

Entsprechung von Kontroll- und Datenstrukturen

Konstruktionsmuster	Anweisungsform	Datentyp
atomares Element	Zuweisung	skalarer Typ
Zusammenfassung	Anweisungsfolge (Block)	Verbundtyp
Fallunterscheidung	bed. Anweisung	`bool`
Auswahl	Auswahlanweisung	diskreter Typ (Aufz.)
Wiederholung Anzahl bekannt	Zählschleife	Feldtyp
Wiederholung Anzahl unbekannt	while- oder repeat–Anweisung	“Sequenztyp”
Rekursion	rek. Prozedur	rek. Datentyp
Allg. Graph	Sprunganweisung	Listenstruktur

Entsprechung eng
(Zählschleife, Felder)
oder lose
(rek. Prozedur, rek. Datentyp)