(5) Funktionen

Funktionsdeklaration / -definition
Funktionsprototypen
Funktionsaufruf
Wertrückgabe
Formal-/Aktualparameter
Parameterübergabe: Wert- / Referenzparameter

Funktionen

- **Funktionen stellen eine Form von Unterprogrammen dar**
  - gängige Sorten von, bzw. Bezeichnungen für Unterprogramme sind Funktionen, Prozeduren, Subroutines
- **Unterprogramme lösen Teilprobleme eines Algorithmus**
  - sie können deshalb in ihrer parametrierten Form als Algorithmenschemata für Teilprobleme betrachtet werden, die der Gesamtalgorithmus löst
- **Funktionen erlauben die Anwendung von Abstraktionsschritten in der Programentwicklung**
  - häufig benötigte Anweisungsfolgen, logisch eng zusammengehörige Anweisungen können unter einem Namen zusammengefaßt werden
  - Datenabhängigkeit der abstrakten Anweisungen kann durch Parametrierung gelöst werden

Aufbau einer Funktion

```
<Fkts-Rückgabetyp> Funktionsname ( <typ> Formaler Par.-Name [, ...] )
{
    // Funktionsrumpf
    // ggf. return Wert vom Typ <Fkts-Rückgabetyp>
}
```

- **Der <Funktionsrückgabetyp> gibt an, von welchem Typ die spätere Wertrückgabe sein wird**
- **Funktionsname benennt die Funktion**
  - In C wird die Funktion allein durch ihren Namen bezeichnet
  - In C++ gehören hingegen Anzahl+Typ der Parameter mit zur Funktionsspezifikation
- **Formale Parameternamen**
  - Platzhalter für Werte, die vom Aufrufer der Funktion übergeben werden
- **Funktionsrumpf**
  - Anweisungen zur Berechnung eines Ergebnisses der Funktion, welches dann über eine return-Anweisung an den Aufrufer zurückgeliefert wird

return - Anweisung

- **Syntax:**
  - return ausdruck ;
- **Bedeutung / Wirkung / Anwendung:**
  - Verlassen der aktuell geöffneten Funktionsschachtel mit der Möglichkeit zur Rückgabe eines Wertes.
  - Zwingend vorgeschrieben für Wertübergabe an aufrufende Funktion, sonst kann Angabe unterbleiben;
  - Evtl. sinnvoll zur Markierung des Funktionsendes.
- **Anmerkungen**
  - Typ von ausdruck muß gleich sein zu oder konvertierbar sein in Typ der Funktion
  - return ohne ausdruck nur erlaubt bei Funktionen ohne Rückgabewerte, d.h. vom Typ void.
  - Wichtig bei Wertrückgabe ist tatsächlicher Durchlauf durch return-Anweisung (Unterschied z.B. zu Pascal)

Beispiel: Funktion max zur Berechnung des Maximums zweier int-Werte

```
int max(int a,int b)
{
    int max;
    if (a > b)
        max = a;
    else
        max = b;
    return max;
}
```

```
int max(int a,int b)
{
    if (a > b)
        return a;
    else
        return b;
}
```

```
int max(int a,int b)
{
    return (a > b)? a : b;
}
```


Aufruf-(Aktual)-Parameter:

- **beim Aufruf einer Funktion mit Skalar (d.h. einfache Variable - kein Feld!) als Parameter wird Wert übergeben:**

```
int  max_2 (int a, int b)      // a,b - formale Parameter
{
   if (a >= b)
      return a;
   else
      return b;
}
```

Aufruf :

```
u = v + max_2(x, y) * w;
// x,y aktuelle Parameter
```

```
int  max_2 (int x, int y)
{
   if (x >= y)
      return x;
   else
      return y;
}
```

Jedes Vorkommen von a, b in der Definition wird bei Aufruf durch ***Wert von*** x, y ersetzt

Ausdruck als Aktualparameter

- **Call-by-value ermöglicht Ausdrücke als Aktualparameter zu verwenden**
  - Dieser Ausdruck darf selbst wieder Funktionsaufrufe enthalten
  - Mechanismus: Ausdruck auswerten - Wert bestimmen - Formalparameter ersetzen

```
void f(int  a)
{
   printf("a = %d\n", a);
   /* ... */
}
void main()
{
   int x = 1;

   f(2*x+1);
}
```

- **Ausgabe**
  - a = 3

Parameterübergabe

- **In C standardmäßig Wertparameter (call by value), Ausnahme: Felder C++: zusätzlich Adreßparameter (call by reference)**

```
void f(int  a)
{
   printf("a (Orginal)         = %d\n", a);
   a = 5;
   printf("a (nach Zuweisung)  = %d\n", a);
}
void main()
{
   int x = 1;

   f(x);
   printf("x (nach f())        = %d\n", x);
}
```

- **Ausgabe**
  - a (Orginal)   = 1
  - a (nach Zuweisung) = 5
  - x (nach f())   = 1

Der Wert einer Variablen kann durch einen Funktonsaufruf nicht (nach außen bleibend) verändert werden!!!

Wertrückgabe über Parameter

- **Über Wertparameter keine Veränderung einer Variablen der rufenden Funktion möglich (außer bei Feldern -- später!), da nur Wert übergeben und Adreßbezug verloren**

- **Wertrückgabe über Parameter ==> Notwendigkeit der Adreßübergabe**
  - Wohin soll denn ein Wert gespeichert werden
  - Speicheradressen von Formal- und Aktualparameter sind unterschiedlich

- **Da nur Wertparameter (in C) ==> Adreßübergabe der Zielvariable als Wertparameter vom Typ**

  Zeiger auf...
  (später)

Rekursion

- **Funktionen dürfen wieder weitere Funktionen aufrufen (Aufrufhierarchie)**
- **Funktionen dürfen auch sich selbst wieder aufrufen; Rekursion!**
- **Bekanntes Beispiel: Fakultätsfunktion**
  - Fakultät(1) := 1
  - Fakultät(n) := n * Fakultät(n-1)

- **in C / C++:**

```
int fakultaet(int n)
{
    if (n == 1)
        return 1;
    else
        return n * fakultaet(n-1);
}

main()
{
    printf("Fakultaet(5) = %d\n", fakultaet(5));
}
```

Übungsaufgabe: Rekursion / Labyrinthsuche

- **Gegeben Sei ein geschlossenes Labyrinth in Form eines zweidimensionalen Feldes. Hindernisse und freie Flächen seien geeignet kodiert dargestellt.**
- **Entwickeln Sie einen Algorithmus, der zwischen zwei anzugebenden Punkten des Labyrinths einen Weg bestimmt, bzw. meldet, daß kein Weg existiert.**

- **Im einzelnen:**
  - Das Labyrinth sei als Textdatei abgelegt, Ein 'X' soll ein Hindernis und ' ' einen Freiraum kennzeichnen. Die Größe des Labyrinths (x/y-Abmessungen) sei in den ersten beiden Zeilen der Datei angegeben. Diese Datei soll bei Programmstart eingelesen und angezeigt werden.
  - Anschließend sollen Start- und Zielpunkt für die Wegesuche abgefragt werden
  - Danach ist der Weg zwischen Start und Ziel zu bestimmen und geeignet anzuzeigen
  - Schließlich ist abzufragen, ob eine erneute Wegeplanung gewünscht wird. In diesem Fall ist das Verfahren ab Punkt 2) zu wiederholen.

Beispiellauf für Labyrinthsuche

```
Start, Ziel: 1 2 45 10

XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
X                             >>>>>>>>>>>>>>>>>>vX
XS>>>>>>>vX    X      X    XXX^<XXXX    XXXv<<<<<X
Xv<<<<<<<<X    X             X>^...XXXXXX  >>>>>vX
X>>>>>>>>vX    X    XXXXXX   X^<<<<<<<<<<<XXXXXv<X
Xv<<<<<<<<XXXXXXX      X >>>>>>>>>>>>>>>>^X    >vX
X>vXXXXX      X        X>^XXX  XXXXXXX....X   XXvX
X >>>>vX      XXXXX    X^<<<<<<<<<<<X.....Xv<<<<<X
Xv<<<<<XXXX     X      X........>>>^X.....X>>>>>vX
X>>>>>>>>>>>>>>vX      XXXXXXXXX^<<XXXXXXXXXXv<<<X
X  XXXXXXXXXv<<<X             >>>>>^X....X   Z   X
X        X  >>>vX        XXXXX^<XXXXXX...X   XXX X
X        X     >>>>>>>>>>>>>vX>^.........X       X
X        X     XXXXXXXXXv<<<<X^<<<<<<<<<<X       X
X        X          X   >>>>vX....XXXXX>^XXXXXXX X
X        X          Xv<<<<<<<X........X^<<<<<<<<<X
X                    >>>>>>>>>>>>>>>>>>>>>>>>>>>^X
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX

Weitere Suche?
```

Sichtbarkeit / Lokalität von Variablen

```
int f(int n)
{
    int x;
    n=n+1;
    x = 5;
    /* ... */
}

main()
{
    int  n=1, x=5;

    f(n);
}
```

```
int  x;              // x - global

int f()
{
  int    x, y;       // x-lokal zu f
             // globales x wird verdeckt!
  x = 1; y = 1;
  if (x == y) {
      int  x;        // x-lokal zu {}
      x = 2;
  } // 1.lokales x wird bis '}' verdeckt!
}
```

- **Frage: Wo ist welches n bzw. x gemeint???**
  - Jede neue Definition einer Variablen (in einer Funktion / in einem Block) erzeugt auch eine neue Speicheradresse, unter der ein Wert abgelegt werden kann
  - Im gleichen Sichtbarkeitsbereich darf eine Variable jedoch nur einmal definiert werden!
  - Dies gilt selbst in rekursiven Aufrufen (so sind z.B. alle 'n' bei der Fakultätsberechnung unterschiedlich!)

(6) Programmierstil & Programmierrichtlinien

Programmierrichtlinien

- **Zweck: Gleichartige Gestaltung von Programmen**
  - Bessere Lesbarkeit (v.a. durch andere Teammitglieder)
  - Bessere Wartbarkeit
  - Aber auch Wahrung der Stabilität des Gesamtsystems
- **Tip: Wenige Regeln vereinbaren ...**
  ... und unter Strafe Einhaltung erzwingen
- **Wichtig: Genaue Beschreibung von ...**
  - Ein-/ Ausgangsparametern
  - Rückgabewerten
  - Seiteneffekte von Funktionen
- **Aufgabe: Kommentieren Sie den jetzigen Implementierungsstand der Random-Test-Aufgabe, versehen Sie ihn mit Headern, etc., drucken Sie ihn aus und bringen Sie ihn zur nächsten Vorlesung mit**

Beispiel für Programmierrichtlinie

- **Every file starts with the header specified below**
- **Every function starts with the header specified below**
- **extern-references for variables and functions must occur at the beginning of a file**
- **After an 'extern'-declarator a comment must follow that mentions the file where the corresponding variable/function is declared**
- **#define constants should be in capital-letters**
- **Variables and functions should be in small-letters**
- **Do not nest #include - directives wherever possible**
- **We use the following insert-rules and arrangements for brackets:**
  - only one statement per line
  - use 4 blanks for each insert-step
  - only 8 space tabs may be used (use 4 spaces for first indentation and
  - a tab for the next indentation level)
  - corresponding brackets **{**

    ...

- **}** **must be in the same column!**
  - Two functions should be separated by 5 blank lines

Beispiel für Programmierrichtlinie (2)

- **Some general comments:**
  - Keep functions as short as possible (no multi page functions)
  - Keep the file length reasonable (around 500-1000 lines)
  - make use of static variable and function declarations wherever possible. This easily indicates that a certain function only has some meaning for **this** file
  - don't use multiple variables for the same real world entity just to avoid unit conversions
  - reduce the number of global variables wherever possible. Use the static descriptor to declare a function local to a file.
  - Pass parameters to functions. Don't use global variables just for convenience
  - Be careful with the naming of global variables. Don't shadow global variables with local ones.
- **PLEASE RESPECT THE GUIDELINES. They should help to reduce the time we have to spend for maintaining our programs.**

Beispiel für Standard-Dateianfang

```
/*************************************************************************
**   FILENAME
*************************************************************************
**   AUTHOR:
**   DATE:
**   VERSION:
**************************************************************************
**   DESCRIPTION:
**
**
**************************************************************************
**   CHANGES:
**   DATE        AUTHOR      COMMENT
**
**************************************************************************/

/****************************  INCLUDE FILES  ****************************/

/*******************************  DEFINES   *****************************/

/********************** EXTERN FUNCTION DEFINITIONS *********************/

/********************* LOCAL FUNCTION DEFINITIONS **********************/

/*************************  EXTERN VARIABLES     ************************/

/**************************  LOCAL VARIABLES     ************************/
```

Beispiel für Standard-Funktionsheader

```
/*************************************************************************
**   FUNCTION_NAME min2 - Minimum of two integers
**************************************************************************
**   INPUT:    x,y - two integers whose min should be calculated
**   OUTPUT:   the min of x, y
**   REMARKS: no use of global variables
*************************************************************************/
int min2(int x, int y)
{
    return (x < y) ? x : Y;
}
```

```
/*************************************************************************
**   FUNCTION_NAME max2 - Maximum of two integers
**************************************************************************
**   INPUT:    x,y - two integers whose max should be calculated
**   OUTPUT:   the max of x, y
**   REMARKS: no use of global variables
*************************************************************************/
```


(7) Felder und Zeichenketten

Felder und Zeichenketten

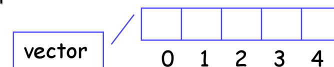
- **Beispiele aus der Mathematik**
  - Vektoren :
    a = (1, 2, 3, 4, 5, 6, 7)   u.ä.
  - Matrizen :

(	0	1	2	)	(Reihen- und Spaltenlauf!)
(	3	4	5	)	
(	6	7	8	)	

- **Sonstige Beispiele**
  - Ordner Nr. 10
  - Telefonkurzwahlnummer 5
  - Kontonummer #0815
  - Wartenummer 12
  - Personalstammdatensatz #4711
  - Student mit Matrikelnummer
- **Gemeinsamkeit: Basisdatentyp + Zugriff über (numerischen) Index**
  - Basisdatentyp, z.B. int, double, unsigned, etc., aber auch (*später*) Konto, Ordner, etc.

Vektoren in C/C++

- **int vector [5] ;**
  - beschreibt homogene Datenstruktur mit 5 Komponenten
    vector [0], vector [1], vector [2], vector [3], vector [4]
  - Zählbeginn immer bei Komponenten-Nr. 0
  - Bezeichner der gesamten Struktur ist vector
  - Datentyp der Komponenten ist int
  - sequentielle Speicherung



- **mit Vorbesetzung**
  - int vector[5] = {1, 2, 3, 4, 5} ;
  - int vec2[5]   = {1, 2, 3};   // Fehlende Einträge werden mit 0 initialisiert!
- **mit impliziter Dimensionierung abhängig von Vorbesetzung**
  - int vector [ ] = {1, 2, 3, 4, 5} ;

Matrixdeklarationen / -definitionen

- **Matrix (3 X 4) , d.h. 3 Reihen (Zeilen), 4 Spalten**

```
double matrix [3][4];
double matrix [3][4] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12};
double matrix [ ][4] = {{1, 2, 3, 4},{5, 6, 7, 8},{9, 10, 11, 12}};
```

- **Gruppierung wesentlich bei automatischer Dimensionierung**
- **beliebig viele Dimensionen möglich (je nach Ressourcen)**
- **Zugriff auf Feldelemente durch vollständige Indizierung**
  - erstes Element : `matrix [0][0]`
  - letztes Element: `matrix [2][3]` // Achtung: Feldindizierung beginnt bei 0,0!
  - mit Hilfe von Zeigern . . . . (*etwas später*)

Übung: Suche nach einem Feldelement

- **Schreiben Sie ein Programm, das folgendes leistet:**
- **Hauptprogramm**
  - Definition eines 1-dim. Feldes [10] von double-Werten
  - Eingabe der Feldelemente
  - Eingabe einer Variable gleichen Typs (Vergleichswert)
- **Unterprogramm**
  - Vergleich des Vergleichswertes mit den Feldelementen
  - Rückgabe des Index des Feldelementes, das betragsmäßig den kleinsten Abstand zum Vergleichswert besitzt.
    (schreiben Sie für die Berechnung des Betrags des Abstands eine Fkt. abs)
  - Die aufrufende Funktion (Hauptprogramm) gibt das Feldelement, das zu dem Rückgabe-Index gehört, aus

Programm MinDist

Bestimmung des Feldelementes mit dem kleinsten Betragsabstand zu einem Vergleichswert

```
#include<stdio.h>
const int AnzElemente=10;   // Anzahl Feldelemente

double abs(double x)   // Bestimmung des Betrags
{
  return (x > 0) ? x : -x;
}

int mindist(double feld[], int AnzElemente, double v_wert)
{  //  bestimmt das Feldelement mit der minimalen Distanz zu einem Vergleichswert
  int i, MinInd;
  double Min, dist;
  MinInd = 0;
  Min = abs(feld[0]- v_wert);          // Vorbesetzung Distanzminimalwert
  for (i=1; i<AnzElemente; i++) {
       dist = abs (feld[i] - v_wert);
       if (dist < Min) {
            Min    = dist;             // neue Minimaldistanz
            MinInd = i;                // diesen Index merken
       }
  }
  return MinInd;
}
```


Programm MinDist (Hauptprogramm)

```
void main (void)
{
   double feld[AnzElemente], vergl_wert;
   int i,Index;

   printf("\nEingabe der Feldelemente\n");
   for (i=0;i<AnzElemente;i++)
   {
     printf("double-Element [%d]: ",i);
     scanf("%lf", &feld[i]);
   }

   printf("\nEingabe des Vergleichswerts: ");
   scanf("%lf", &vergl_wert);

   Index = mindist(feld, AnzElemente, vergl_wert);
   printf("\n\nDie minimale Distanz zwischen Vergleichswert %f und den
   Feldelementen\n", vergl_wert);
   printf("betraegt: %f. Das naechste Feldelement befindet sich unter
   Index %d\n", abs(feld[Index]-vergl_wert), Index);
   printf("und hat den Wert %f\n", feld[Index]);

}  // Programmende
```

Felder als Aktualparameter

- **Übergabe der Feldadresse - nicht Kopie des Feldes als Wertparameter**

```
void  incr_elems (int feld [10])
{
   int  i;
   for ( i = 0; i < 10; i++)
       feld[i] = feld[i] + 1;
}
```

- **Aufruf mit Feldvariable a_feld:**

```
int a_feld[10] = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
...
for (i = 0; i< 10; i++)
   printf("%d ", a_feld[i]);     ---> 0 1 2 3 4 5 6 7 8 9
incr_elems(a_feld);
for (i = 0; i< 10; i++)
   printf("%d ", a_feld[i]);     ---> 1 2 3 4 5 6 7 8 9 10
```

- **Implizite Behandlung als Referenz-Parameter:**
  Übergabe der Adresse, Dereferenzieren bei Zugriff auf Komponenten

Zeichenketten in C/C++

- **Zeichenketten sind Felder vom Typ char**
  - in C kein extra string-Datentyp
  - in C++ sehr flexible Klasse 'string', die vielfältige String-Objekte ermöglicht
- **Zeichenketten in C / C++ sind null-terminiert !**

```
#include <stdio.h>
void main ()
{
  char w [20];
  w [0] = ´s´ ;
  w [1] = ´t´ ;
  w [2] = ´r´ ;
  w [3] = ´i´ ;
  w [4] = ´n´ ;
  w [5] = ´g´ ;
  w [6] = ´\0´ ;   // Endekriterium für String-Funktionen
  printf(w);       // Ausgabe des Wortes 'string'
}
```

Alternativ:

```
char w[20] = "string";
```

Zeichenkettenfunktionen sind beispielsweise ...

- **#include <string.h>**
- **strlen - ermittelt die Länge einer Zeichenkette**
  - a = strlen (w) // Anzahl der Zeichen **vor** dem abschließenden Null-Zeichen, jedoch ohne dieses selbst!
- **strcpy - kopiert eine Zeichenkette**
  - strcpy (ziel_zk, quell_zk) // entspricht ziel_zk = quell_zk
- **strcat - hängt eine Zeichenkette an eine bestehende an (Konkatenation)**
  - strcat (ziel_zk, quell_zk) // entspricht ziel_zk = ziel_zk + quell_zk
- **strcmp - vergleicht zwei Zeichenketten**
  - if (strcmp (zk1, zk2) > 0) ...
  - liefert ...

    -1, wenn zk1 < zk2

    0, wenn zk1 == zk2

    +1, wenn zk1 > zk2

Beispielprogramm 'Zeichenkettenoperationen'

```
#include <string.h>
#include <stdio.h>

void main( void )
{
   char string[80];
   strcpy( string, "Hello world from " );
   strcat( string, "strcpy " );
   strcat( string, "and " );
   strcat( string, "strcat!" );
   printf( "String = %s\n", string );
}
```

Output:

```
String = Hello world from strcpy and strcat
```

Aufgaben

- **Deklarieren Sie einen Vektor mit 20 Gleitpunkt-Komponenten**
  - initialisieren Sie die ersten 5 Komponenten bei der Definition
  - weisen sie den nächsten 5 Komponenten im Programm-Rumpf einen Wert zu
- **Deklarieren sie ein Feld in Form einer Matrix mit zwei Dimensionen**
  - initialisieren sie es als obere Dreiecksmatrix
  - Weisen Sie drei Matrixkomponenten einen Wert zu
- **Definieren Sie eine String-/Zeichenkettenvariable, die 25 Zeichen aufnehmen kann**
  - Initialisieren Sie die Variable mit Ihrem Rufnamen
  - konkatenieren Sie Ihren Familiennamen
  - Löschen Sie den ganzen String auf adäquate Weise

(8) Ausdrücke

Ausdrücke in Programmiersprachen ~ Aufbau mathematischer Formeln

- **Beispiel :**

```
                 c
h =  a * b + -------- * ln (m)
                  m
             1 + ----
                  k*s
```

- **Komponenten:** **Operanden und Operatoren**

Operanden: gemischt
- Summenausdrücke
- Produktausdrücke
- Quotientenausdrücke
- einfache Variablen
- Konstanten

Aus der Mathematik bekannte Merkmale von Ausdrücken

- **Operatorvorrang:** "Punktrechnung vor Strichrechnung"

  Stelligkeit von Operatoren: Anzahl der zur Berechnung nötigen Operanden
  - Vorzeichen *einstellig*
  - Addition, Subtraktion, Multiplikation, Division *zweistellig*
  - Potenzierung *zweistellig* (a hoch b)

- **Assoziativität der Operatoren:** implizite Klammerung bei mehrgliedrigen Ausdrücken a*b*c --> (a*b)*c

- **nichttriviale Ausdrücke:**
  - funktionale Ausdrücke: ln (m)
  - Indizierte Ausdrücke in Matrix- oder Vektor-Operationen (ai,j)
  - Index-Ausdrücke $a_{i*k-1,j}$

in Programmiersprachen:

- **Ausdrücke sind Teile von Anweisungen, d.h. von ausführbaren programmiersprachlichen Gebilden**
- **Ausdrücke dienen**
  - der Berechnung von Werten und
  - der Steuerung des Programmablaufs

- **in C /C++ : Ausdrücke sind Folgen von Operanden und Operatoren, die**
  - einen Wert berechnen
  - ein Objekt bezeichnen
  - einen Objekt-Typ festlegen oder ändern
  - einen Seiteneffekt auslösen.

Operatoren

Operator	Bedeutung	Prio	Stelligkeit	Assoz.
( )	Klammern in Ausdruck u. Funktion	1		l
[ ]	Klammernpaar für Feldindex-Ausdruck			
.	Punktop.: Auswahl einer Strukturkomp.		2	
->	Pfeilop.: Dereferenzieren u. Strukturkomp.		2	
!	logische Negation	2	1	**r**
	bitweise Negation			
++	Inkrement um 1			
--	Dekrement um 1			
-	neg. Vorzeichen			
(<*typ*>)	explizite Typumwandlung (casting)			
*****	Inhaltsoperator (Dereferenzieren)			
&	Adreßoperator			
sizeof	Speichergröße eines Objekts in Byte			
*****	Multiplikation	3	2	l
/	Division			
%	Modulo-Operator (Divisionsrest bei ganzzahl.Div.)			
+	Addition	4	2	l
-	Subtraktion			
<<	bit-weises Schieben nach links (left shift)	5	2	l
>>	bit-weises Schieben nach rechts (right shift)			

Operatoren (2)

Operator	Bedeutung	Prio	Stelligkeit	Assoz.
< **<=** **>** **>=**	Vergleichsoperator - kleiner als Vergleichsoperator - kleiner als oder gleich Vergleichsoperator - größer als Vergleichsoperator - größer als oder gleich	6	2	l
== **!=**	Vergleichsoperator - gleich Vergleichsoperator - ungleich	7	2	l
&	bit-weise UND (AND)	8	2	l
^	bit-weise XODER (eXklusives ODER, XOR)	9	2	l
|	bit-weise ODER (OR)	10	2	l
&&	logisches UND (AND)	11	2	l
||	logisches ODER (OR)	12	2	l
? :	Bedingter Ausdruck	13	3	**r**
= **+= -=** ***= /=** **%=** **>>= <<=** **&= |= ^=**	Zuweisungsoperatoren	14	2	**r**
,	Komma-Operator: Verkettung von Anweisungen	15	2	l

Reihenfolge der Operandenauswertung

- **festgelegt nur für: && || ? : ,**
  - e1 && e2 — Auswertung von e2 nur wenn e1 wahr
  - e1 || e2 — Auswertung von e2 nur wenn e1 falsch
  - e1 ? e2 : e3 — erster Operand e1 zuerst;
  - e1 , e2 — e1 zuerst, dann e2; Wert des Kommaausdrucks ist Wert von e2.
- **Alle anderen Operatoren: <u>nicht</u> festgelegt!!!**
- **Beispiele :**
  - (a + b) + (c + d) — unbekannt, aber keine Probleme
  - f (x, y) + g (y) — undefiniert und Problem der Ausführungsfolge der Funktionen, <u>wenn</u> y Rückgabeparameter
evtl. Wert von y in g(y) undefiniert !
  - x/x++ — implementierungsabhängig (x/x oder x/(x+1)?
  - printf("a[%3d]: %d\n", i, a[i++]); ???
- **Abhilfe: Vollständig klammern oder Ausdrücke aufspalten!!!**

Einfache Ausdrücke

- **zu den einfachen Ausdrücken gehören die**
  - Primärausdrücke
  - Funktionsaufrufe
- **und eine Teilmenge der Ausdrücke, die *mit* Operatoren gebildet werden**
  - Verweise auf Feld- und Strukturelemente
  - Adreßausdrücke

Primärausdrücke

Ausdruck	**Wert / Wirkung**
Konstante	Wert der Konstante (z.B. Zahlenwert)
konstante Zeichenfolge	Zeiger auf erstes Zeichen
Variablenname Zahlentyp, Zeigertyp, Aufzählungstyp, Strukturtyp, union-Typ	Wert der Variablen
Variablenname - Feldtyp	Zeiger auf erstes Feldelement, Adresse des Feldanfangs
Funktionsname	Zeiger auf Funktion, Adresse der Funktion
(<ausdruck>)	Wert und Typ des Ausdrucks in Klammern

- **Anfangspunkt einer rekursiven Definition von Ausdrücken:**
  - jeder Primärausdruck ist ein Ausdruck,
  - die Anwendung eines Operators auf Primärausdrücke erzeugt einen Ausdruck entsprechend der Operatorstelligkeit
  - die Anwendung eines Operators auf einen Ausdruck erzeugt einen Ausdruck

Auswahl von Feldelementen und Strukturkomponenten

- **Auswahlausdrücke mit Feld- und Strukturnamen und den Operatoren [ ], →, .**

Ausdruck	Wert / Wirkung
x [<ausdruck>]	x Feldvariable oder Zeigervariable <ausdruck> Integer-Ausdruck Wert/Auswahl eines Feldelements
x . name	Auswahl der Komponente name der Struktur x
x → name	Auswahl der Komponente name der Struktur, auf die x zeigt

Funktionsaufrufe

- **Funktionsname, Klammer, Argumentliste**
  - In C wird die Funktion allein durch ihren Namen bezeichnet
  - In C++ gehören hingegen Anzahl+Typ der Parameter mit zur Funktionsspezifikation
  - C++ bietet zudem die Möglichkeit, Default-Belegungen der Parameter vorzusehen (--> Programmieren II)

Ausdruck	Wert / Wirkung
f (<argumentliste>)	Aufruf der Funktion mit Namen f und Liste der Argumente in <argumentliste>

Adreßausdrücke

- **Adreßoperator &, Inhaltsoperator ***

- **Ausdruck** — **Wert / Wirkung**
  - & x — Adresse von x .   x L-Wert
  - * x — das Objekt / die Funktion, deren Adresse durch Ausdruck x bezeichnet wird
    x vom Typ Zeiger auf <typ> , * x vom Typ <typ>

Speicherbelegung / Typumwandlungen

- **sizeof angewendet auf Objekt oder Typ**

- **Ausdruck** — **Wert / Wirkung**
  - sizeof <ausdruck> — Byte-Anzahl für Speicherung des Ausdrucks,
    bei Feldnamen: Speicherbedarf des Feldes
    Achtung: keine Auswertung von <ausdruck>
  - sizeof (<typname>) — Byte-Anzahl für ein Objekt des Typs <typname>
    bei char: 1 (ANSI C)

- **Explizite Typumwandlung**
- **Ausdruck** — **Wert / Wirkung**
  - (<typname>) x — Wert von x in Typ <typname> umgewandelt

Implizite Typumwandlung

von \ nach	void	Integer	Gleit-komma	Zeiger	Feld	Struktur	Funktion
void	#						
Integer	#	#	#	#			
Gleitkomma	#	#	#				
Zeiger	#	#		#	#		#
Feld	#						
Struktur	#					#	
Funktion	#						

- Integer: alle Ganzzahltypen
- Gleitkomma: alle Gleitkommatypen

Typumwandlungen

- **Umwandlung nach Integer**
  - von Integer: wenn Wert in Zieltyp darstellbar, dann umwandeln, sonst undefiniert
  - Kompromisse: vorzeichenloser Zieltyp - durch Abschneiden der MSB~Modulo-Bildung
  - von Gleitkomma: wenn ganzzahliger Anteil in Zieltyp darstellbar, dann umwandeln, sonst undefiniert
  - Zeiger (nur in C): Interpretation als gleichgroße Integerzahl
- **Umwandlung nach Gleitkomma**
  - float nach double möglich
  - double nach float durch Runden oder Abschneiden
  - von Integer entsprechende Näherung
- **Umwandlung nach Zeigertyp**
  - von Zeigertypen: jeder Zeigerwert kann zu jedem bel. Zeigertyp gewandelt werden
  - von Integer-Typen: nur NULL-Wert als NULL-Zeiger interpretierbar
  - von Feld-Typen: Ausdruck des Typs Feld-von-Typ T stets umgewandelt zu Zeiger auf T ; Feldname wird identifiziert mit Zeiger auf erstes Element (außer in sizeof)
  - von Funktions-Typen: Ausdruck des Typs Funktion mit Wert vom Typ T umgewandelt in Zeiger auf Funktion mit .... (außer im Funktionsaufruf)

Arithmetische Ausdrücke

- **Vorzeichenausdrücke mit unären Operatoren + , -**
  - + : liefert den Wert des Operanden +x ≡ x (impliziter Operator)
  - - : liefert den negativen Wert des Operanden : -x
- **Arithmetische Ausdrücke mit den binären Operatoren + , - , * , / , %**
  - + : Additionsoperator a + b
  - - : Subtraktionsoperator a - b
  - * : Multiplikationsoperator a * b
  - / : Divisionsoperator a / b (ganzahlige Division, wenn beide Operanden Integer)
  - % : Modulo -Operator a % b "a modulo b" (Divisionsrest bei ganzahliger Division)
- **Inkrement- und Dekrement mit den unären Operatoren ++ , -**
Ausdruck	Bezeichnung	Wert / Wirkung
++x	Präinkrement	Erst x um 1 erhöhen, dann im Ausdruck verwenden
x++	Postinkrement	Erst x im Ausdruck verwenden, dann um 1 erhöhen

    --x und x-- analog
  - bei Zeigern Erhöhen bzw. Erniedrigen jeweils um 1 x (Größe des Verweisobjektes) --> Weiterschalten auf nächstes Element

Beispiele

```
int i, j;

i = 1;
j = ++i + 1;   //  j = 3, i = 2
i = 1;
j = i++ + 1;   //  j = 2, i = 2
i = 1;
i = ++i + 1;   //         i = 3
```

- **Probleme aufgrund der Auswertungsreihenfolge von Ausdrücken bei der Programmierung mit Seiteneffekten:**
  - `i = 0;`
    `a [i] = i++;` // Auswertungsreihenfolge ungeklärt !
    // erst Indexausdruck a[0] oder erst Zuweisungsausdruck,
    // dann a[1] (Implementierungsabhängig !)
  - Zuweisungsausdruck in der Wertzuweisung an das Feldelement bewirkt Seiteneffekt

Vergleiche und logische Ausdrücke

- **Vergleichsoperatoren: > >= < <= == !=**
- **Vergleiche mit arithmetischen Operanden**
  - gewohnte math. Interpretation
- **Vergleiche mit Zeigern:**
  - Vergleichbarkeit nur innerhalb derselben Struktur bzw. desselben Feldes,
  - Relation in Bezug auf Indexordnung und Abspeicherreihenfolge der Elemente
  - Vergleichsausdrücke erzeugen einen der Wahrheitswerte wahr (=1) oder falsch (=0) und sind damit vom Typ int
  - Vergleichsausdrücke sind also logische Ausdrücke
- **Achtung: keine Kettenbildung gemäß der math. Abkürzung: a < b < c**
  - weil a < b < c --> (a < b) < c, Jedoch (a < b) liefert 0 oder 1
  - somit --> (0/1) < c
  - Lösung: Auflösen der Kette durch Benutzung des UND Operators :
    (a < b) && (b < c)

logische Ausdrücke

- **alle int - Ausdrücke**
- **alle Vergleichsausdrücke**
- **alle durch Anwendung der Operatoren && (UND), | | (ODER), ! (NICHT) auf logische Ausdrücke entstehenden Ausdrücke**
- **Zeiger (implizite Typwandlung nach Integer)**
  - Überprüfung auf NULL-Zeiger
- **Auswertung von links --- Abbruch, wenn Ergebnis vorliegt**
  - Bsp.: a && b, b nicht ausgewertet, falls a falsch
  - Bsp.: a || b, b nicht ausgewertet, falls a wahr
- **Vergleichsoperatoren binden stärker als logische Operatoren**

Bedingte Ausdrücke u. Klammerausdrücke

- **Bedingte Ausdrücke mit dem 3-stelligen Operator ___ ? ___ : ___**
- **Komma-Ausdrücke mit dem 2-stelligen Operator ___ , ___**

Ausdruck	**Wert / Wirkung**
e0 ? e1 : e2	e0 auswerten, falls wahr → e1 auswerten , falls falsch → e2 auswerten
e1 , e2	1. e1 auswerten 2. e2 auswerten Typ und Wert des Gesamtausdrucks werden durch Typ und Wert von e2 bestimmt!

- **Anwendung des Kommaausdrucks:**
  - Verwendung zweier Ausdrücke, wo nur ein Ausdruck erlaubt ist
    for (i=0, j=0; i<100; i++, j++)
  - **Achtung: Bei Verwendung bei einem Funktionsaufruf** muß geklammert werden, um eine Verwechslung mit der Parametertrennung durch Kommata zu vermeiden!

Konstante Ausdrücke

- **Konstante Ausdrücke sind Ausdrücke, die der Compiler zur Übersetzungszeit auswerten kann**
  - Beispiel: `int i = 2*3+4/2;  --> int i = 8;`
    (von konstanten Ausdrücken wird nur das Ergebnis verwendet, jedoch kein Auswertungscode erzeugt)
- **Anwendungsbeispiele**
  - Argument der #if-Präprozessordirektive
  - Dimensionierung von Feldern
  - case-Ausdrücke in switch-Anweisungen
  - explizite Definition von Aufzählungswerten
  - Initialisierungswerte von static- und extern-Variablen
- **konstante Ausdrücke können aus Integer- und char-Konstanten durch Anwendung der folgenden Operatoren gebildet werden:**
  - unär : + - ~ . !
  - binär: + - * / % << >> == != < <= > >= & ^ | && ||
  - ternär: _ ? _ : _

Zuweisungen

- **Abkürzungsoperatoren**
  ursprünglicher Zuweisungs-Operator ist =
- **zusätzliche Zuweisungsoperatoren zur Abkürzung der**
  - binären arithmetischen und der
  - binären Bit-Ausdrücke
  - Schema:
    Operand_1 = Operand_1 ⊗ Operand_2 ⇒ Operand_1 ⊗= Operand_2
- **Vorteile**
  - manchmal bessere Lesbarkeit des Ausdrucks, insbesondere bei langen Var.-Namen
  - Evtl. Geschwindigkeit, wenn der linke Operand aufwendig berechnet werden muß;
    Beispiel: `a[3*fkt(x)][x*z*fkt2(x,y,z)] = a[3*fkt(x)][x*z*fkt2(x,y,z)] + 5;`
  - `--> a[3*fkt(x)][x*z*fkt2(x,y,z)] += 5;`
    Bei dieser += Lösung muß die aufwendige Berechnung der Indices nur einmal erfolgen. Hierdurch kann sich evtl. ein Geschwindigkeitsvorteil ergeben

Zuweisung als Ausdruck

- **Unterschied in C/C++ zu allen anderen geläufigen Programmier-sprachen:**

  Zuweisungen sind Ausdrücke
- **Folgen:**
  - Zuweisungen dürfen überall da verwendet werden, wo Ausdrücke des betreffenden Datentyps erlaubt sind !
    Beispiel: if (fp = fopen("Datei xxx", "r")) // Hier ist wirklich die Zuweisung gemeint!
- **Aber auch**
  - jeder Ausdruck wird durch Anhängen von ';' zu einer Anweisung:

    ausdruck ;
  - Also z.B. auch `2*3*x;` // Hier sinnlos, da der Wert nicht verwendet wird
    aber auch: `printf("Text");` // Printf ist hier als Funktion Teil eines Ausdrucks.
    (printf liefert als Funktion die Anzahl der gedruckten Zeichen zurück. Dies wird jedoch häufig nicht benötigt!)