

C-Crashkurs

S. Hahn, D. Rybach, D. Stein

Betriebssysteme und Systemsoftware – WS 2007/08

**Human Language Technology and Pattern Recognition
Lehrstuhl für Informatik 6
Computer Science Department
RWTH Aachen University, Germany**

Weiterentwicklung von BCPL und B

- ▶ **BCPL: Ende der 60er Jahre von Martin Richards zum Bau von Betriebssystemen und Compilern entwickelt**
- ▶ **B: Ken Thompson erstellte 1970 mit B das erste UNIX System**

C

- ▶ **1972 von Dennis Ritchie in den Bell Laboratories entwickelt**
- ▶ **Wurde zur Entwicklung des UNIX-Betriebssystems verwendet**
- ▶ **Zunächst durch den Klassiker “The C Programming Language” von Brian Kernighan und Dennis Ritchie beschrieben und 1989 vom amerikanischen ANSI-Institut standardisiert**
- ▶ **Häufig nicht als Hochsprache angesehen, da maschinennahe Programmierung möglich**
- ▶ **Hohe Flexibilität, kleiner Sprachumfang (ANSI-C hat nur 32 Schlüsselwörter)**
- ▶ **keine Schutzmechanismen, kein strenges Typkonzept**
- ▶ **Programmiersprache für Programmierer**

Beispiel

```
1 #include <stdio.h>                /* Funktionsdeklarationen für I/O einbinden */
2
3 int maximum( int a, int b );       /* Funktionsdeklaration */
4
5 int global_max = 0;               /* Globale Variable global_max */
6
7 /*******/                          /* Kommentar */
8 /* Hauptprogramm
9 *****/
10 int main( int argc, char *argv[] ) /* Start des Hauptprogramms */
11 {
12     printf( "Hello World.\n" );    /* Ausgabe auf der Standardausgabe */
13     global_max = maximum( 1,2 );   /* Aufruf der Funktion maximum */
14 }
15
16 /******
17 * Funktion zur Maximumberechnung
18 *****/
19 int maximum( int a, int b )       /* Funktionsdefinition */
20 {
21     if (a>b)
22         return a;
23     else
24         return b;
25 }
```

Elementare Datentypen

Typ	(übliche) Größe [bit]	Wertebereich	Beispiele
char	8	$[-128 \dots 127]$	'a', '&', ' '
unsigned char		$[0 \dots 255]$	
short	16	$[-32768 \dots 32767]$	42, -13, 0x2a
unsigned short		$[0 \dots 2^{16} - 1]$	
int	32	$[-(2^{31}) \dots 2^{31} - 1]$	
unsigned int		$[0 \dots 2^{32} - 1]$	
float	32	6 Dezimalstellen	2.0, -3.142,
double	64	10 Dezimalstellen	3.3e-4

Variablendefinition:

▶ ohne Initialisierung:

```
int a;
float x, y, z;
```

▶ mit Initialisierung:

```
unsigned char c = 'a';
double q = 2.4, p = 1.3e-23;
```

Datentypen: Verbundtypen

```
struct [VerbundName] {
    Typ VariablenName;
    Typ VariablenName;
    ...
} [VariablenListe];
```

Beispiele:

```
1 struct point { /* Definiere Verbundtyp point */
2     int x;
3     int y;
4 };
5
6 struct point p1; /* Variable p1 mit Typ point */
7 p1.x = 5; /* Setze Element x */
8
9 struct rect { /* Definiere VerbundTyp rect */
10     struct point lo; /* mit Elementen vom Typ point */
11     struct point ru;
12 } rectA, rectB; /* Variablen mit Typ rect */
13
14 rectA.lo.y = 18;
```

Datentypen: Arrays

Typ VariablenName [AnzahlElemente];

- ▶ **Arrays in C sind immer 0-basiert:**
`int a[n]` definiert die Elemente `a[0]`, ..., `a[n-1]`
- ▶ **Elemente eines Arrays befinden sich in einem zusammenhängenden Speicherbereich**

Beispiele:

```
1 char s[26];
2
3 float grades[5] = { 1.0, 2.0, 3.0, 4.0, 5.0 };
4
5 int matrix[3][5] = { { 11, 12, 13, 14, 15 },
6                     { 21, 22, 23, 24, 25 },
7                     { 31, 32, 33, 34, 35 } };
8
9 s[0] = 'A';
10 s[25] = 'Z';
11
12 float myGrade = grades[2];
13
14 matrix[1][4] = -25;
```

Datentypen: Aufzählung

```
enum [Name] {  
    Element_1 [=Wert], ..., ElementN [=Wert]  
} [VariablenListe];
```

- ▶ **Integer-Variablen, die nicht jeden beliebigen Wert annehmen dürfen, sondern auf eine begrenzte Anzahl von Werten beschränkt sind.**
- ▶ **Diese Werte werden über Namen referenziert.**

Beispiele:

```
1 enum Beer { Alt, Koelsch, Pils, Weizen };  
2  
3 enum Beer bottle = Pils;  
4 enum Beer glass = Weizen;  
5  
6  
7 enum { NORD, OST=90, SUED=180, WEST=270} direction;  
8  
9 direction = SUED;  
10  
11 int newDirection = direction + 45;  
12  
13 enum Month { January = 1, February, March, April, May, June,  
14             July, August, September, October, November, December };
```

Datentypen: typedef

```
typedef Typ Name;
```

- ▶ Ein bereits bestehender Datentyp wird über einen neuen Namen angesprochen.

Beispiele:

```
1 typedef char Character;  
2 Character c;  
3  
4 typedef unsigned int uint;  
5 uint i = 5;  
6  
7 typedef enum Month MonthType;  
8 MonthType m = October;  
9  
10 typedef struct {  
11     char name[80];  
12     uint matNr;  
13 } StudentType;  
14  
15 StudentType harald;  
16 harald.matNr = 12345;
```

Datentypen: Zeiger

Typ *VariablenName;

- ▶ Ein Zeiger ist eine Variable, die eine Speicheradresse beinhaltet.
- ▶ Die Größe und das Format einer Adresse ist System-abhängig.

Beispiele:

```

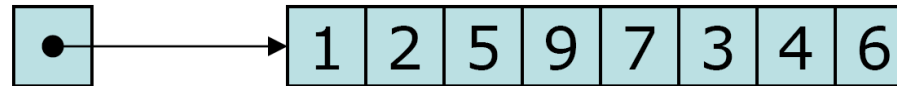
1  int i = 7;           /* Integer Variable      */
2  int *p_i;          /* Pointer auf Integer  */
3  int *p_j = NULL;   /* Initialisierter Pointer */
4
5  struct point *p_a; /* Pointer auf Verbundtyp */
6
7  p_i = &i;          /* Adresse von i        */
8  p_j = p_i;         /* Kopiere Adresse      */
9
10 *p_j = 23;         /* Dereferenziere p_j   */
11                          /* i = 23                */
12 *p_i = *p_j + 1;   /* i = 24                */
13
14 (*p_a).x = 7;      /* Zugriff auf Verbund- */
15 p_a->y = 9;         /* elemente              */
16 /* VORSICHT: p_a ist nicht initialisiert! */
17 /*                verweist auf unbekante Adresse */

```

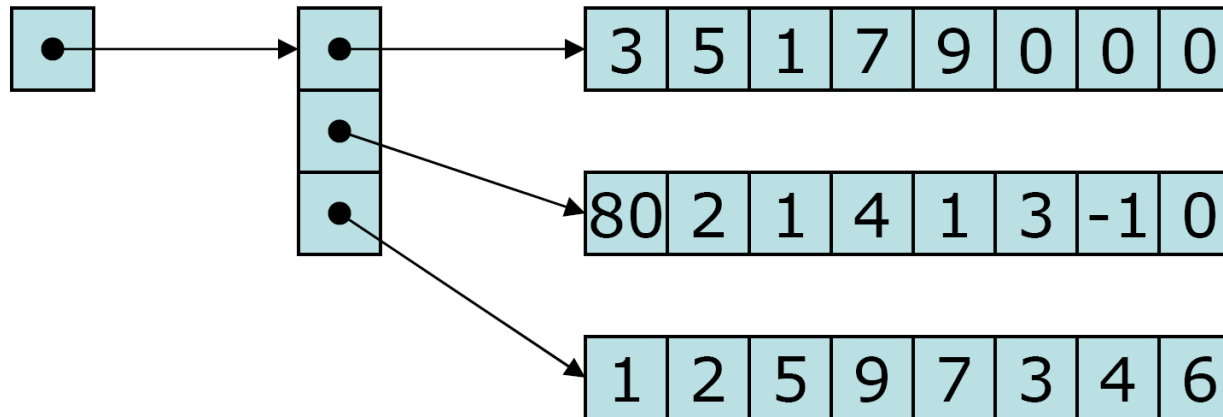
Datentypen: Zeiger und Arrays

► Array-Variablen sind Zeiger auf das erste Element

```
int numbers[8] = { 1, 2, 5, 9, 7, 3, 4, 6 };
```



```
int matrix[3][8];
```



```

1  int numbers[8] = { 1, 2, 5, 9, 7, 3, 4, 6 };
2  int matrix[3][8];
3
4  int *p = numbers;           /* *p == numbers[0];      */
5  int *q = &numbers[3];      /* *q == 9                */
6  int *r = numbers + 3;      /* *r == 9                */
7
8  char *lines[128];          /* pointer auf pointer    */
9  double **myMatrix;
```

Arithmetische Operatoren

- +** Addition
- Subtraktion / Negation
- *** Multiplikation
- /** Division
- %** Modulo

Bit-Operatoren

- &** Bitweises UND
- |** Bitweises ODER
- ^** Bitweises XOR
- ~** Invertieren

Shift-Operatoren

- <<** Links-Shift
- >>** Rechts-Shift

Zeiger-Operatoren

- &** Adressoperator
- *** Dereferenzierung
- >** Dereferenzierender Zugriff auf Element

Vergleichsoperatoren

- ==** Gleich
- !=** Ungleich
- <** Kleiner
- <=** Kleiner oder gleich
- >** Größer
- >=** Größer oder gleich

Logische Verknüpfungen

- &&** UND
- ||** ODER
- !** Negation

Typkonvertierung

Implizite Konvertierung Enthält ein Ausdruck Variablen oder Konstanten verschiedener Datentypen, wird der Typ einzelner Operanden automatisch umgewandelt. Dabei wird der Operand mit kleinerem Wertebereich an den größeren Datentyp angepasst.

Explizite Konvertierung Durch Voranstellen des Datentyps wird angegeben, in welchen Typ die nachfolgende Variable oder Konstante umgewandelt werden soll.

Beispiele:

- ▶ `float f = 1.0/3` → `0.333333`
implizite Konvertierung von 3 zu 3.0 und Durchführung einer Fließkommadivision
- ▶ `float f = 1/3` → `0.0`
ganzzahlige Division von 1 durch 3 ergibt 0, implizite Konvertierung von 0 zu 0.0
- ▶ `float f = (int)0.333333 * 3` → `0.0`
explizite Konvertierung von 0.333333 zu 0, anschließende Multiplikation mit 3 und impliziter Konvertierung zu 0.0
- ▶ `float f = (float)1/3` → `0.333333`
explizite Konvertierung von 1 zu 1.0, implizite Konvertierung von 3 zu 3.0 und Durchführung einer Fließkommadivision
- ▶ `float f = (float)(1/3)` → `0.0`
ganzzahlige Division von 1 durch 3 ergibt 0, explizite Konvertierung von 0 zu 0.0

Kontrollstrukturen: if

```
if (Bedingung)
    Anweisung;
[else
    Anweisung;]
```

Beispiele:

```
1 if (a > b)
2     max = a;
3 else
4     max = b;
5
6
7 if (arg[0] == '-')
8     if (arg[1] == 'v') {
9         verbose = 1;
10        printf( "more output\n" );
11    } else
12        printf( "Unknown option.\n" );
```

```
1 if (b == Pils)
2     glass = Tulpe;
3 else if (b == Weizen)
4     glass = Humpen;
5 else if (b == Koelsch)
6     glass = Stange;
7 else
8     glass = Standard;
```

Kontrollstrukturen: switch

```

switch (Ausdruck) {
    case KonstanterAusdruck1:
        Anweisung1a;
        Anweisung1b;
        ...
    case KonstanterAusdruck2:
        Anweisung2a;
        Anweisung2b;
        ...
    ...
    [default:
        AnweisungNa;
        AnweisungNb;
        ... ]
}

```

- ▶ Es wird zunächst der Ausdruck nach switch ausgewertet
- ▶ Das Ergebnis wird mit allen case-Konstanten verglichen
- ▶ Stimmt der Wert mit einer Konstanten überein, wird die Programmausführung dort bis zur nächsten break-Anweisung fortgeführt
- ▶ Gibt es keine Übereinstimmung, wird zur (optionalen) default-Marke gesprungen

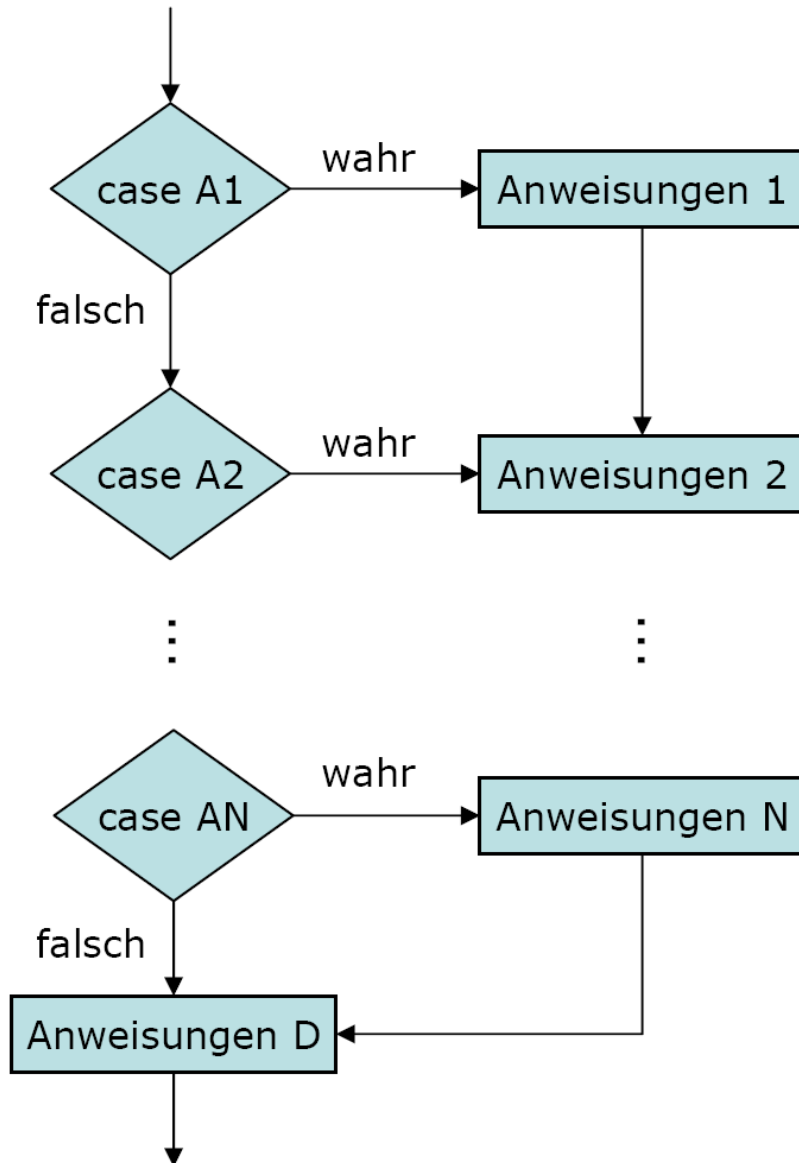
```

1 switch ( buchstabe ) {
2     case 'a': printf( "a\n" );
3     case 'b': printf( "b\n" ); break;
4     case 'c': printf( "c\n" ); break;
5     default: printf( "no a,b,c\n" ); break;
6 }

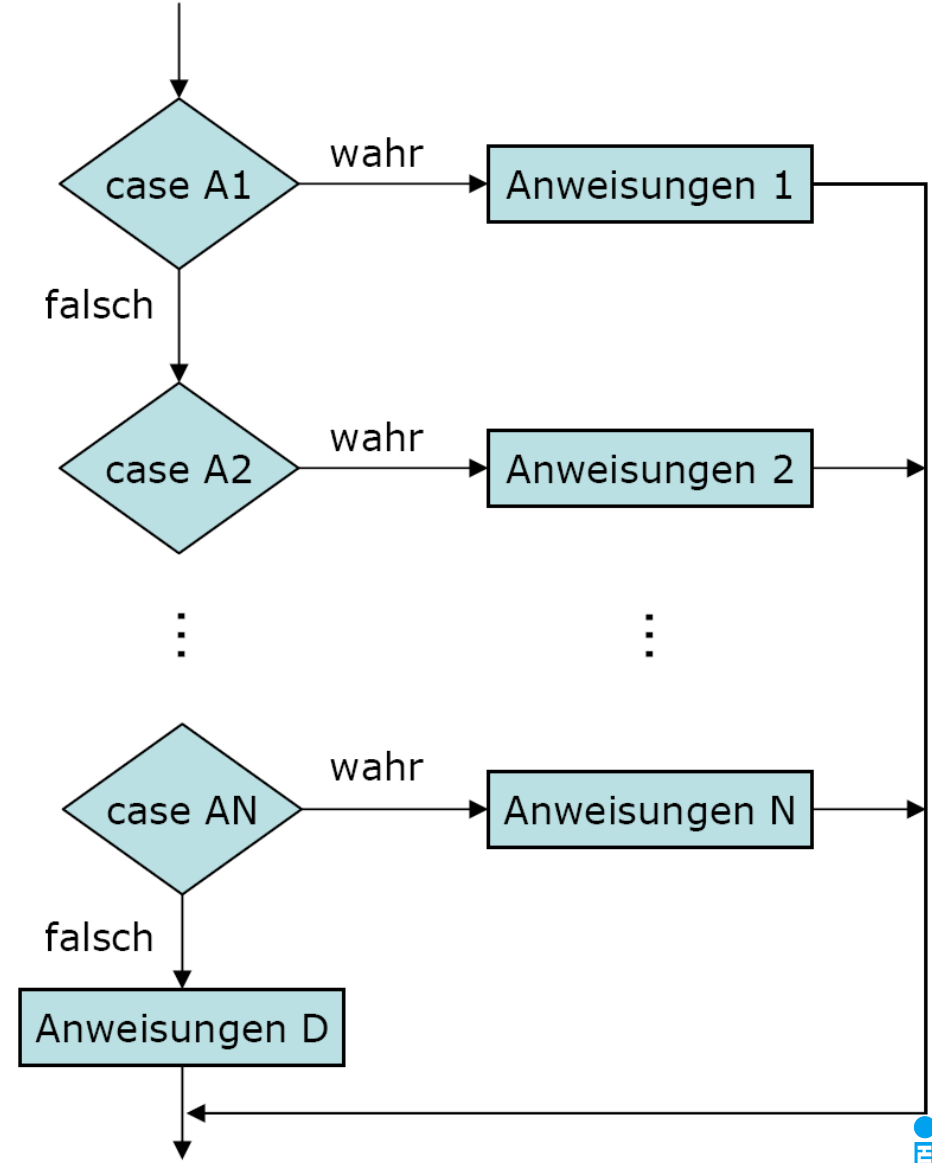
```

Kontrollstrukturen: switch

switch ohne break



switch mit break



```
while (Bedingung)
    Anweisung;
```

- ▶ Solange die Auswertung der Bedingung einen Wert ungleich 0 liefert, wird der Anweisungsblock durchlaufen.

```
do
    Anweisung;
while (Bedingung);
```

- ▶ Die Anweisung wird immer mindestens ein mal ausgeführt
- ▶ Liefert beim Erreichen des `while`-Statements die Auswertung der Bedingung einen Wert ungleich 0, wird die Anweisung erneut ausgeführt.

```
for (Init; Bedingung; AnwS)
    Anweisung;
```

- ▶ Zuerst wird `Init` ausgeführt
- ▶ Ist die Schleifenbedingung erfüllt, wird der Schleifenkörper durchlaufen.
- ▶ Nach jedem Schleifendurchlauf wird `AnwS` ausgeführt
- ▶ Falls die Schleifenbedingung noch erfüllt ist, wird ein weiterer Durchlauf gestartet.

Beispiele:

```
1 int i, sum = 0;
2
3 for (i = 0; i <= 100; i = i+1)          /* Schleife wird 101 mal durchlaufen */
4     sum = sum + 1;
5
6
7 i = 0;
8 while (data[i] != -1) {                /* Schleife wird solange durchlaufen */
9     sum = sum - 1;                      /* bis data[i] == -1 */
10    i = i + data[i] + 1;
11 }
12
13
14 do {
15     i = i / 2;                          /* wird mindestens einmal ausgefuehrt */
16     sum = sum * 2;
17 } while (i > 0 && sum < 100);
```

Kontrollstrukturen: Schleifen

break

- ▶ Anwendbar bei switch, for-, while- und do while-Schleifen
- ▶ Verhindert die Ausführung weiterer Anweisungen innerhalb der Schleife
- ▶ Führt zum sofortigen Verlassen von for-, while- und do while-Schleifen

Beispiel:

```

1 while (1) {                /* Endlosschleife */
2     ...
3     if ( input==0 )
4         break;            /* Beenden der */
5                             /* Schleife */
6     ...
7 }

```

continue

- ▶ Anwendbar bei for-, while- und do while-Schleifen
- ▶ Bewirkt eine sofortige Rückkehr zur Schleifenanweisung

Beispiel:

```

1 for (i = 0; i < 7; i = i + 1) {
2     if ((i==2) || (i==4))
3         continue;
4     printf("%d ", i);
5 }
6
7 /* Ausgabe: 0 1 3 5 6 */

```

Dynamische Speicherverwaltung

- ▶ Häufig ist die Anzahl der zu verwaltenden Datenelemente zur Compile-Zeit unbekannt
- ▶ Die Verwendung von statischen Arrays würde zu einer ineffizienten Speichernutzung führen

```
void* malloc( size_t size )
```

- ▶ Reserviert einen zusammenhängenden Speicherblock der Größe `size`
- ▶ Gibt einen Zeiger auf den Anfang des Blocks zurück
- ▶ Der Inhalt des Blocks ist undefiniert

```
free( void *ptr )
```

- ▶ Gibt einen durch `malloc()` angelegten Speicherblock wieder frei

Beispiel:

```

1 char *str;
2 str = (char *)malloc( 256 * sizeof(char) ); /* legt einen Speicherblock für 256 char-Werte an */
3 if (str == NULL) {
4     printf( "Nicht genügend Speicher.\n" );
5     exit(1);
6 }
7 ...
8 free( str );                               /* Freigabe des nicht mehr benötigten Speichers */

```

Zeichenketten werden in C durch Zeiger auf nullterminierte char-Arrays dargestellt:

```
char *text = "Hallo";  
char text[6] = {'H', 'a', 'l', 'l', 'o', '\0'};
```

Standardfunktionen zum Manipulieren von Zeichenketten:

<code>strcpy, strncpy</code>	Kopieren von Zeichenketten
<code>strcat</code>	Aneinanderhängen von Zeichenketten
<code>strcmp</code>	Vergleichen von Zeichenketten
<code>strchr, strrchr</code>	Suche eines Zeichen innerhalb einer Zeichenketten
<code>strstr</code>	Suche eines Teilstrings innerhalb einer Zeichenkette
<code>strlen</code>	Länge einer Zeichenkette

Funktionen: Definition

```
Rückgabetyyp Funktionsname( Parameterliste )
{
    Anweisungen
}
```

return (Rückgabewert) : führt zum sofortigen Beenden der Funktion und der Rückgabewert wird als Ergebnis an die aufrufende Funktion zurückgegeben.

Beispiel:

```
1  int max;                /* global Variable */
2
3  int maximum (int a, int b)
4  {
5      int max = a;        /* lokale Variable, die nur innerhalb dieser Funktion gültig ist */
6      if (a < b)
7          max = b;
8      return max;        /* Verlassen der Funktion und Rückgabe von max */
9  }
10
11 void printHallo( void ) /* Funktion ohne Parameter und ohne Rückgabewert */
12 {
13     printf( "Hallo.\n" );
14 }
15
16 max = maximum( 1, 2 );  /* Aufruf von maxi mit Parametern 1 und 2 */
17 printHallo();          /* Aufruf von printHallo */
```

Funktionen: Parameter

Parameterübergabe ist in C immer *Call-by-Value*.

Call-by-Value

Änderungen der Parameter innerhalb der Funktion werden außerhalb der Funktion nicht sichtbar.

Beispiel:

```
1 void func ( int in )
2 {
3     in = in + 1;
4 }
5
6 int a = 7;
7 func( a );
8 /* Nach Aufruf: a == 7 */
```

Call-by-Reference

Wird durch einen “Trick” erreicht: Die Funktion hat einen Zeiger auf die Variable als Parameter.

Beispiel:

```
1 void func ( int *in )
2 {
3     *in = *in + 1;
4 }
5
6 int a = 7;
7 func( &a );
8 /* Nach Aufruf: a == 8 */
```

Ein- / Ausgabe

`printf(string, a1, a2, ...)`

- ▶ Ausgabe auf der Standard-Ausgabe
- ▶ Die Funktion wertet den Kontrollstring aus und formatiert die Argumente entsprechend
- ▶ Einfacher Text im Kontrollstring wird unverändert ausgegeben
- ▶ Platzhalter im Kontrollstring werden durch die Werte der weiteren Argumente ersetzt:

```
1 int i = 42;
2 printf( "%d als Hex.-Zahl: %x\n", i, i );
3 char *text = "Hallo";
4 printf( "%s\n", text );
```

`scanf(string, a1, a2, ...)`

- ▶ Liest Zeichen von der Standard-Eingabe, interpretiert sie gemäß des Kontrollstrings
- ▶ Speichert die Wert in den Argumenten (Zeiger!)
- ▶ Kontrollstring enthält:
 - ▷ Whitespaces als Trennzeichen, werden ignoriert
 - ▷ Gewöhnliche Zeichen, die als nächstes Zeichen in der Eingabe vorkommen müssen

```
1 int i;
2 float x;
3 char str[80];
4 scanf("%d %f %s", &i, &x, str);
```

- ▶ Um Funktionen anderer Module verwenden zu können, müssen diese Funktionen zunächst deklariert werden (*Prototyp*)
- ▶ Dadurch wird dem Compiler mitgeteilt, welche Parameter eine Funktion erwartet und welchen Rückgabewert sie besitzt.
- ▶ Beispiel: `double sin (double x);`
- ▶ Der dazugehörige Programmcode ist in einer Objektdatei abgelegt, die zum Schluss zum Programm “hinzugelinkt” wird (siehe Compiler-Handbuch)

Header Dateien

Für ein Modul werden die Funktionsprototypen in der Regel in Header-Dateien zusammengefasst, die mit

```
#include "dateiname.h" bzw. #include <dateiname.h>
```

eingebunden werden.

Häufig verwendete Header Dateien:

- `stdlib.h` Standard Funktionen (`malloc`, `free`, `exit`, `qsort`, ...)
- `stdio.h` Funktionen zur Ein-/Ausgabe (`printf`, `scanf`, `fopen`, `fclose`, ...)
- `string.h` Funktionen zur Manipulation von Zeichenketten (`strlen`, `strcpy`, ...)
- `math.h` Mathematische Funktionen (`sin`, `cos`, `sqrt`, ...)

hello.h

```
1
2 /* Funktionsprototypen */
3 void read_name(char *name);
4
5 void hello(char *name);
```

hello.c

```
1
2 /* I/O-Funktionen (System-Header) */
3 #include <stdio.h>
4
5 /* Funktionsdefinitionen */
6 void read_name(char *name)
7 {
8     printf("your name: ");
9     scanf("%s", name);
10 }
11
12 void hello(char *name)
13 {
14     printf("Hello %s\n", name);
15 }
```

main.c

```
1 #include "hello.h"
2
3 /* Hauptfunktion */
4 int main()
5 {
6     char name[128];
7     read_name( name );
8     hello( name );
9     return 0;
10 }
```

Kompilieren und Linken mit gcc

```
# Erstelle Object-Datei hello.o aus hello.c
gcc -c hello.c

# Erstelle main.o
gcc -c main.c

# Linken der Object-Dateien
gcc -o hello main.o hello.o

# Programm ausführen
./hello
```

Operator ++ und --

- ▶ ++ inkrementiert, -- dekrementiert
- ▶ 2 Varianten: ++a und a++
(– analog)
- ▶ Unterschied: Zeitpunkt der Inkrementierung
- ▶ Beispiel:
 - ▷ `i = 0; a = 7; i = ++a;`
i und a haben den Wert 8
 - ▷ `i = 0; a = 7; i = a++;`
i hat den Wert 7, a den Wert 8

Operatoren +=, -=, *=, /=, &=, ...

`a += 7;`

Abkürzung für `a = a + 7;`

?-Operator

`x = (Bed ? Wert1 : Wert2);`

Abkürzung für

```
if (Bed )
    x = Wert1;
else
    x = Wert2;
```

Beispiel:

```
1 int maximum ( int a, int b )
2 {
3     return (a > b ? a : b);
4 }
```

Mehrfachzuweisungen

Falls einer Reihe von Variablen derselbe Wert zugewiesen werden soll, kann dies durch eine Anweisung geschehen:

```
a = b = c = d = e = 7;
```

statt

```
a=7; b=7; c=7; d=7; e=7;
```

Rechnen mit Zeigern

- ▶ Auf Zeigern können arithmetische Operatoren angewendet werden
- ▶ Ist `ptr` ein Zeiger, setzt bspw. `ptr++` den Zeiger auf das nächste Element
- ▶ Der Compiler berücksichtigt dabei die Größe der Daten, auf die der Zeiger zeigt

```
1 char *str = "Betriebssysteme";
2 printf("%c %c %c\n", *str, *(str + 2), *(str + 5));
3 /* Ausgabe: B t e */
4
5 int data[5] = { 8, 2, 9, 3, 5 };
6 int *ptr = data; /* 8 */
7 ptr++;          /* 2 */
8 ptr += 3;      /* 5 */
```

Beispiel: strcpy – Vom Anfänger zum Profi

```
1 void strcpy1( char *dest, char *src )
2 {
3     int i;
4     for (i=0; src[i] != '\0'; i=i+1)
5         dest[i] = src[i];
6     dest[i] = '\0';
7 }
```

```
1 void strcpy3( char *dest, char *src )
2 {
3     while ( *src != '\0' ) {
4         *dest = *src;
5         src++;
6         dest++;
7     }
8     *dest = '\0';
9 }
```

```
1 void strcpy2( char *dest, char *src )
2 {
3     int i = 0;
4     while( (dest[i] = src[i]) != '\0' )
5         i = i+1;
6 }
```

```
1 void strcpy4( char *dest, char *src )
2 {
3     while ( *dest++ = *src++ );
4 }
```

Beispiel: Verkettete Liste

```

1 #define NAME_LEN 80                /* Konstante */
2
3 typedef struct _person {          /* Listenelement */
4     char *name;                  /* Zeichenkette beliebiger Länge */
5     char matrikelNr[7];          /* Zeichenkette mit fester Länge */
6     enum {Mi,Do,Fr} gruppe;
7     struct _person *next;        /* Zeiger auf nächstes Element */
8 } tPerson;
9
10 tPerson *liste = NULL;
11 tPerson *person = NULL;
12
13 liste = person = (tPerson *)malloc( sizeof( tPerson ) ); /* Speicher für Listenelement */
14 person->name = (char *)malloc( NAME_LEN * sizeof( char ) ); /* Speicher für Zeichenkette */
15 strcpy( person->name, "Hartmut" ); /* Zeichenkette in Element kopieren */
16 strcpy( person->matrikelNr, "123456" );
17 person->gruppe = Do;
18 person->next = NULL; /* Kein Nachfolger = Ende der Liste */
19
20
21 liste->next = person = (tPerson *)malloc( sizeof( tPerson ) ); /* Neues Element hinten anfügen */
22 person->name = (char *)malloc( NAME_LEN * sizeof( char ) );
23 strcpy( person->name, "Werner" );
24 strcpy( person->matrikelNr, "234567" );
25 person->gruppe = Fr;
26 person->next = NULL;

```

Beispiel: Kommandozeilenparameter

```
1 #include <stdio.h>
2
3 typedef char * STRING;
4 void func1( int var1, STRING var2 );
5
6 int main (char argc, char *argv[] )           /* argc: Anzahl der Parameter      */
7 {                                             /* argv: Parameter als Array von Strings */
8     int i;
9
10    printf( "Dateiname: %s\n", argv[0] );     /* argv[0]: Names des Executables   */
11
12    for (i = 1; i < argc; i++)               /* Alle Parameter ausgeben         */
13        func1( i, argv[i] );
14
15    return 0;
16 }
17
18 void func1( int var1, STRING var2)
19 {
20    printf( "%i. Parameter: %s\n", var1, var2 );
21 }
```

```
> ./param
Dateiname: ./param

> ./param Eins Zwei
Dateiname: ./param
1. Parameter: Eins
2. Parameter: Zwei
```

- ▶ **B. Kernighan, D. Ritchie: *The C-programming language*, Second Edition, Prentice-Hall, 1988**
- ▶ **UNIX man Pages**
 - `man malloc`
 - `man 3 printf`
 - ...