

# Einführung in die Programmierung

---

MIT DER PROGRAMMIERSPRACHE C

PROF. DR. THOMAS GABEL

# Überblick über die Vorlesung

1. Algorithmen, Programme und Software
2. Einstieg in die Programmierung in C
3. Strukturiertes Programmieren in C
4. **Effizientes Programmieren in C**
5. Fortgeschrittene Aspekte der Programmierung in C



## 4. Effizientes Programmieren in C

1. **Felder und Zeichenketten**
2. Standardein- und -ausgabe
3. Zeiger
4. Felder und Zeiger
5. Benutzerdefinierte Typen
6. Anwendungsbeispiel "Verkettete Listen"
7. Kommandozeilenparameter
8. Dateiein- und -ausgabe
9. Programmerzeugung und Präprozessor
10. "Beliebte" Fehler



# Datenstrukturen

**Definition: Datenstrukturen** dienen dazu, logisch zusammenhängende Daten im Speicher abzulegen.

Beispiel:

- Die Verwaltung einer Firma speichert solche Daten wie Name, Nachname, Geburtsdatum, Adresse usw. eines jeden Angestellten.
- Frage: Wie kann man diese Daten übersichtlich aufbewahren?
- Für jeden Angestellten wird zum Beispiel eine Karteikarte angelegt.
- Die Datenstrukturen sind Abbildungen solcher Karteikarten auf den Speicher.

In C ist es möglich, Datenstrukturen in Form von **nutzerdefinierten, komplexen Datentypen** aufzubauen.

- **Felder** (engl. array)
- **Verbund** (auch benannt als Struktur, engl. struct)
- **Union** (auch bekannt als Vereinigungsstruktur, engl. union)
- **Aufzählungstyp** (auch bekannt als Enumeration, engl. enumeration)
- **Bitfelder** (ähnlich zu Verbünden, sehr ungebräuchlich)

# Felder (1)

**Definition:** Ein **Feld (Array)** in C ermöglicht es, eine geordnete Folge von Werten eines bestimmten Datentyps zusammenhängend abzuspeichern und zu verarbeiten.

- Im Feld kommen nur Objekte des gleichen Datentyps vor, z.B. ein Integer-Feld.

Der Operator für die Felddefinition und den Feldzugriff ist: **[]**

Syntax zur Definition eines Feldes:

**Datentyp Variablenname[Elementanzahl];**

- Unterschied zur (normalen) Variablendefinition: Angabe der eckigen Klammern und Größe
- Die Größe (Elementanzahl) muss immer positiv und **zum Zeitpunkt der Übersetzung des Programms bekannt** sein.

```
int a = 10;  
double feld[a];
```

- Wenn die Anzahl der benötigten Elemente im Feld noch unbekannt ist, ist diese Art der Felddefinition ungeeignet.
- Die Mischung von Variablen und Felddefinitionen in einer Zeile ist erlaubt.

```
int i, j, myArray[20];
```

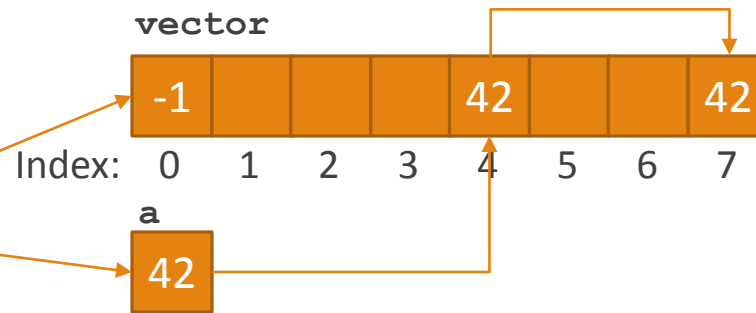
# Felder (2)

## Zugriff auf einzelne Feldelemente:

- Syntax: **variablenname[feldindex]** ;
- Durch die Definition eines Feldes werden so viele Variablen erzeugt, wie das Feld Elemente hat.
- Nebenbemerkung: Eindimensionale Felder werden oft auch als Vektoren bezeichnet.

## Beispiel mit Visualisierung

```
...  
int vector[8];  
...  
int a=42;  
vector[0] = -1;  
vector[4] = a;  
vector[7] = vector[4];
```



## Regel: Die Nummerierung der Feldindizes beginnt bei 0 (nicht bei 1)!

- Vorteil: `feldindex` kann ein beliebiger nichtnegativer Ausdruck sein, der zur Laufzeit ausgewertet wird → Anwendung z.B. in Schleifen
- im Beispiel: Elemente des Feldes sind mit `vector[0]` bis `vector[7]` zugreifbar.

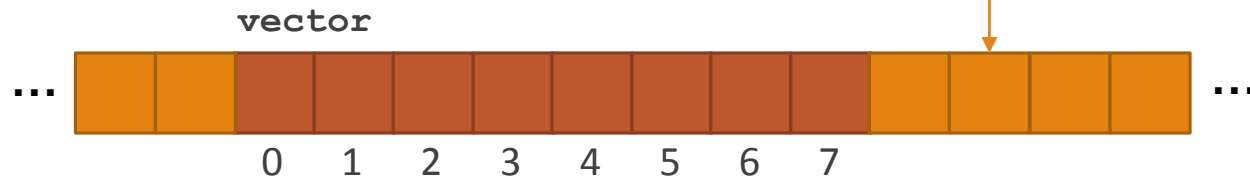
# Felder (3)

## Prüfung von Bereichsgrenzen

- Der **Compiler überprüft nicht**, ob der Feldindex in den zugelassenen Grenzen liegt.
- Es kann also auf „irgendwelche“ Speicherinhalte zugegriffen werden.
- Beispiel: **int vector[8];**

...

```
printf("Überraschung: %d", vector[9] );
```



## Probleme

- Es erfolgt keine Fehlermeldung oder Warnmeldung.
- bei lesendem Zugriff
  - Das Programm arbeitet mit falschen Werten weiter. → Wer weiß schon, was in diesen Speicherzellen steht?
  - Die Fehlerquelle lässt sich meist nur schwer lokalisieren.
- bei schreibendem Zugriff
  - unterschiedliches Verhalten; meist (insbesondere wenn der Zugriff weit außerhalb der Bereichsgrenzen liegt) kommt es zum Programmabbruch wegen Speicherzugriffsfehler (Segmentation Fault).

# Felder (4)

## Vergleich von Feldern

- Zwei Felder sind gleich, wenn alle Elemente in der Reihenfolge gleich sind.
- **Regel:** Dies kann man nicht mit dem Gleichheitsoperator (==) überprüfen.
- Lösungsvariante 1:
  - Schleife programmieren, die über alle Elemente des Feldes iteriert
  - bei jedem einzelnen den Vergleich durchführt
  - sobald ein Vergleich fehlschlägt, schlägt der Gesamtvergleich fehl
- Beispiel:

```
...  
int vec1[8], vec2[8];  
...  
    // put some value into the arrays  
...  
int i;  
int equal = 1;  
for ( i=0; i<8; i++ )  
{  
    if ( vec1[i] != vec2[i] ) equal = 0;  
}
```



# Felder (5)

## Vergleich von Feldern

- Zwei Felder sind gleich, wenn alle Elemente in der Reihenfolge gleich sind.
- **Regel: Dies kann man nicht mit dem Gleichheitsoperator (==) überprüfen.**
- **Lösungsvariante 2: Nutzung der Funktion `memcmp` aus der Bibliothek `string.h`**
  - überprüft ganze Speicherbereiche auf Gleichheit
  - liefert 1, wenn sich ein Unterschied in den Speicherbereichen findet, sonst 0
  - Beispiel:

```
...  
int vec1[8], vec2[8];  
...  
    // put some value into the arrays  
...  
int unequal = memcmp( vec1, vec2, sizeof(vec1) );
```

liefert Größe des gesamten Feldes



# Felder (6)

## Zuweisung von Feldern

- Regel: Eine Zuweisung eines Feldes zu einem anderen mit dem Zuweisungsoperator (=) ist nicht möglich.
- Lösungsvariante 1:
  - Schleife implementieren, die über alle Feldelemente iterieren und jedes einzeln zuweisen
- Lösungsvariante 2:
  - Nutzung der Funktion `memcpy` aus der Bibliothek `string.h`
- Beispiel:

```
...  
int vec1[8], vec2[8], vec3[8], i;  
...  
    // put some value into the the first array  
...  
for ( i=0; i<8; i++ )  
    vec2[i] = vec1[i]; // Lösung 1  
  
memcpy( vec3, vec1, sizeof(vec1) ); // Lösung 2
```

# Felder (7)

## Initialisierung von Feldern

- **automatische Initialisierung** von Feldern erfolgt analog zur Initialisierung von Variablen:
  - global definierte Felder (also globale Variablen definiert) werden mit 0 initialisiert
  - lokal definierte Felder werden i.a. nicht initialisiert
  - abhängig vom Compiler
- **manuelle Initialisierung** ist möglich durch Angabe einer Liste von Werten, die in geschweifte Klammern gesetzt werden
  - Beispiele:

```
int vector1[5] = { 6, -4, 6, 2, -1 };
int vector2[5] = { 6, -6, 2 };
int vector3[] = { 3, -5, 7, 2, -1 };
char hello1[5] = { 'h', 'e', 'l', 'l', 'o' };
char hello2[] = "hello";
```
- Frage: Was fällt Ihnen auf?

# Felder (8)

## Bemerkungen zur **Initialisierung** von Feldern

- Die Werte in der Liste müssen durch geschweifte Klammern eingeschlossen sein.
- Die Werte in der Liste dürfen nur Konstanten sein.
- Es dürfen auch weniger Elemente in der Liste stehen als die angegebene Elementanzahl.  
➔ siehe Beispiel 2
- Implizite Längebestimmung des Feldes ist möglich: Die Größe des Feldes wird dann anhand der Anzahl der Initialisierungswerte ermittelt. ➔ siehe Beispiel 3
- Spezialfall und häufigste Verwendung der Initialisierung von Feldern ist das Initialisieren von Zeichenketten. ➔ siehe Beispiel 5

```
int vector1[5] = { 6, -4, 6, 2, -1 };           // 1
int vector2[5] = { 6, -6, 2 };                   // 2
int vector3[]  = { 3, -5, 7, 2, -1 };           // 3
char hello1[5] = { 'h', 'e', 'l', 'l', 'o' };   // 4
char hello2[]  = "hello";                       // 5
```

# Felder (9)

Beispiel: Eindimensionales Feld von 4 Integer-Werten

```
void someFunction()  
{  
    int arr[4];  
    int sum;  
    arr[0] = 1;  
    arr[1] = 22;  
    arr[2] = -35;  
    sum = arr[0] + arr[1] + arr[2];  
    ...  
}
```

Achtung: Nicht initialisierte Arrays enthalten „Datenmüll“!

- Genauer: Enthalten das, was zufällig in der jeweiligen Hauptspeicherzelle zuvor bereits stand! (Und das kann „irgendwas“, also insbesondere auch nicht null, sein.)

# Felder (10)

Weitere Beispiele:

```
int my_array[10];           /* not initialized */
int my_array[5]  = { 1, 2, 3, 4, 5 }; /* initialized */
int my_array[]   = { 1, 2, 3, 4, 5 }; /* OK, initialized */
int my_array[4]  = { 1, 2, 3, 4, 5 }; /* warning */
int my_array[10] = { 1, 2, 3, 4, 5 }; /* OK, partially
                                         initialized */
```

Bemerkung zur teilweisen Initialisierung:

- bei teilweiser Initialisierung werden die restlichen Werte mit 0 initialisiert

**Explizite Initialisierung** von Feldern, d.h. Initialisierung nach der Definition

```
int i;
int my_array[10];
for (i = 0; i < 10; i++)
{
    my_array[i] = 2 * i;
}
```

- **vorzuziehende** Vorgehensweise

# Felder (11)

## Mehrdimensionale Felder

- benötigt man, um Tabellen, Matrizen u.ä. aufbauen zu können.

- Syntax zur Definition:

```
Datentyp Variablenname[Elementanzahl][Elementanzahl]...;
```

- Beispiel: Definition einer Matrix aus Ganzzahlen mit 6 Zeilen und 5 Spalten

```
int myMatrix[6][5];
```

- Syntax zum Zugriff auf Elemente des mehrdimensionalen Feldes:

```
variablenname[ i ][ j ]...;
```

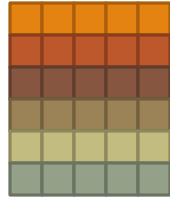
- Beispiel: Zugriff auf den Eintrag in Zeile 3 in Spalte 5

```
int value = myMatrix[2][4];
```

# Felder (12)

## Speicherstruktur

- intuitive Vorstellung



```
int myMatrix[6][5];
```

- im Hauptspeicher:



## Speicherbedarf:

- Der Speicherbedarf eines Feldes ergibt sich nach der Formel  
Größe des Datentyps x Anzahl der Elemente im Feld
- Frage: Welchen Speicherbedarf hat obiges Feld?
- Antwort: 4 Byte x (6x5) = 120 Byte = 960 Bit

## Manuelle Initialisierung mehrdimensionaler Felder

- erfolgt entsprechend der oben dargestellten Speicherstruktur
- Beispiel:

```
double _2dArray[2][3] = { {1.0, 2.1, 1.0e-5},  
                           {-1.3, 1024.5, 77.0} };
```



# Felder (13)

## Manuelle Initialisierung mehrdimensionaler Felder

- die erste Dimensionsangabe des mehrdimensionalen Feldes darf bei der Initialisierung weggelassen werden
- aber nur die erste

```
double _2dArray[][4]
    = { {1.0, 2.1, 1.0e-5, 3.0},
        {-1.3, 1024.5 } }; // unvollständige Initialisierung
```

## Beispiel zu zweidimensionalen Felder:

```
int arr[2][3]; //NOT arr[2,3]
int i, j;
int sum = 0;
arr[0][0] = 1;
arr[0][1] = 23;
arr[0][2] = -12;
arr[1][0] = 85;
arr[1][1] = 46;
```

```
arr[1][2] = 99;
for (i = 0; i < 2; i++)
{
    for (j = 0; j < 3; j++)
    {
        sum += arr[i][j];
    }
}
printf("sum = %d\n", sum);
```

# Felder (14)

**Frage:** Was passiert bei folgenden Code-Fetzen:

```
void foo(int i)
{
    i = 42;
}

/* later... */
int i = 10;
foo(i); /* What is i now? */
```

**Antwort:** Der Wert von `i` wird in die Funktion kopiert.

Übergabe eines Wertes an die Funktion ändert nicht den Wert (außerhalb der Funktion) – Dies ist **Call-By-Value**

- **Aber:** Bei Feldern ist dies nicht der Fall!

**Parameterübergabe** von eindimensionalen Feldern an Funktionen

- Bei Übergabe eines Feldes an eine Funktion wird **nicht** das ganze Feld kopiert.
- Übergeben wird nur die Anfangsadresse, also ein Zeiger auf den Beginn.

# Felder (15)

Übergebene Felder können daher modifiziert werden: **Call by Reference**

```
void foo(int arr[])
{
    arr[0] = 42; /* modifies array */
}

/* later... */
int my_array[5] = { 1, 2, 3, 4, 5 };
foo(my_array);
printf("%d\n", my_array[0]);
```

**Regel:** Bei der Parameterangabe: Die letzte Feld-Dimension kann ignoriert werden.

- Der Compiler erkennt dessen Größe selbst.

```
void foo2(int arr[5])    /* same as arr[] */
{
    arr[0] = 42;
}
```

# Zeichenketten (1)

*Definition:* Unter einer **Zeichenkette** (engl. *strings*) verstehen wir eine Folge von Zeichen vom Typ `char`.

- Zeichenketten werden in C als `char`-Felder realisiert.

Bemerkungen:

- Zeichenketten treten auf bei der Programmierung von Benutzerschnittstellen (Ein-/Ausgabe).
- Zeichenketten treten auf in der Verarbeitung von Textdateien.
- Eine der wichtigsten Anwendungen von **Feldern** ist die Darstellung und Verarbeitung von **Zeichenketten**.
- In der Standardbibliothek von C finden sich viele Funktionen zur effizienten Verarbeitung von Zeichenketten.

# Zeichenketten (2)

In C sind Zeichenketten (Strings) immer **Felder (Arrays) vom Typ char**.

- Erkennungszeichen von Zeichenketten: Anführungsstriche "..."

**Frage:** Woher wissen Funktionen, die auf/mit Zeichenketten arbeiten, wann eine Zeichenkette zu Ende ist?

```
char string1[] = "Dies ist ein sehr sehr langer String";  
char string2[] = "kurzer String";  
printf("string1: %s\n", string1);  
printf("string2: %s\n", string2);
```

```
/* Kopieren von Zeichenketten, aus string.h */  
strcpy(string1, string2);  
printf("string1: %s\n", string1);
```

→ Beispiel: `string.c`

**Antwort:** Zeichenketten enden immer auf „\0“ (Null-Terminierung).

- Besser gesagt: Das letzte Zeichen einer C-Zeichenkette **muss** immer eine ' \0 ' sein!
- Somit kann das Ende der Zeichenkette ermittelt werden.
- Aber man muss ein „zusätzliches Zeichen“ spendieren.

# Zeichenketten (3)

## Zeichenkettenkonstanten

- Unterscheidung: einfache Anführungszeichen für **Zeichenkonstanten** → `'a'`
- Unterscheidung: doppelte Anführungszeichen für **Zeichenkettenkonstanten** → `"Hi!"`
- Eine **Zeichenkettenkonstante** wird intern als `char`-Feld dargestellt, dessen Länge um eins größer ist als die Anzahl der Zeichen in der Zeichenkette.
  - Dieses Zusatzfeld benötigt man, um das Null-Zeichen `\0` abzuspeichern, welches das Ende der Zeichenkette anzeigt.
  - Dieses abschließende `\0`-Zeichen ist bei Zeichenkettenkonstanten implizit.
  - Beispiel: Aus diesem Grund (vgl. Folie zu `scanf`) wird ein Zeichen für die Terminierung spendiert.

```
char s[100];  
scanf("%99s", s);
```

- Wenn man also eine Zeichenkette bearbeitet, so muss man immer abfragen, ob das aktuelle Zeichen das Null-Zeichen ist.

**Regel:** Ein Vergleich einer Zeichenkonstante mit Zeichenkettenkonstanten mit „gleichem Inhalt“ ist nicht möglich:

- `'a'` // 1 char und damit 1 Byte, nämlich das a
- `"a"` // 2 char und damit 2 Byte, nämlich das a und das `\0`

# Zeichenketten (4)

## Initialisierungsmöglichkeiten für Zeichenketten

- manuelle Initialisierung mittels eines Feldes von `char`:
  - explizites Null-Zeichen erforderlich
  - Beispiel: `char text1[4] = { 'o', 'l', 'd', '\0' };`
- Initialisierung mit einer Zeichenkettenkonstante **und** Größenangabe für das `char`-Feld:
  - explizites Null-Zeichen nicht erforderlich, aber Platz für dieses muss berücksichtigt werden (3 vs. 4)
  - Beispiel: `char text2[4] = "new";`
- Initialisierung mit einer Zeichenkettenkonstante **ohne** Größenangabe für das `char`-Feld:
  - explizites Null-Zeichen zur Terminierungsangabe nicht erforderlich
  - Größe wird (vom Compiler) automatisch ermittelt
  - Beispiel: `char science[] = "artificial intelligence";`

o	l	d	\0
---	---	---	----

## Fragen:

- Welchen Wert hat `text1[0]`?
- Welchen Wert hat `text2[17]`?
- Bei welchem Index `i` gilt `science[i] == '\0'`?
- Wie viele Zeichen umfasst der Speicherbereich für `science`?

# Zeichenketten (5)

Operationen für Zeichen in `ctype.h` → <http://www.cplusplus.com/reference/cctype/>

- Testfunktionen

```
int isalnum(char c);      int iscntrl(char c);
int isalpha(char c);      int isdigit(char c);
int islower(char c);      int isprint(char c);
isspace, isupper, isxdigit, ...
```

- Umwandlungsfunktionen

```
int toupper(char c);
int tolower(char c);
```

Beispiel:

```
/* isalnum example */
#include <stdio.h>
#include <ctype.h>
int main ()
{
    int i;
    char str[] = "c3po...";
    i=0;
    while (isalnum(str[i])) i++;
    printf ("The first %d characters are
    alphanumeric.\n",i);
    return 0;
}
```



# Zeichenketten (6)

## ASCII-Tabelle

- als Referenz
- Quelle: Screenshot des Wikipedia-Eintrags zum Stichwort „American Standard Code for Information Interchange“)

Dez	Hex	Okt	ASCII	Dez	Hex	Okt	ASCII	Dez	Hex	Okt	ASCII	Dez	Hex	Okt	ASCII
0	0x00	000	NUL	32	0x20	040	SP	64	0x40	100	@	96	0x60	140	`
1	0x01	001	SOH	33	0x21	041	!	65	0x41	101	A	97	0x61	141	a
2	0x02	002	STX	34	0x22	042	"	66	0x42	102	B	98	0x62	142	b
3	0x03	003	ETX	35	0x23	043	#	67	0x43	103	C	99	0x63	143	c
4	0x04	004	EOT	36	0x24	044	\$	68	0x44	104	D	100	0x64	144	d
5	0x05	005	ENQ	37	0x25	045	%	69	0x45	105	E	101	0x65	145	e
6	0x06	006	ACK	38	0x26	046	&	70	0x46	106	F	102	0x66	146	f
7	0x07	007	BEL	39	0x27	047	'	71	0x47	107	G	103	0x67	147	g
8	0x08	010	BS	40	0x28	050	(	72	0x48	110	H	104	0x68	150	h
9	0x09	011	HT	41	0x29	051	)	73	0x49	111	I	105	0x69	151	i
10	0x0A	012	LF	42	0x2A	052	*	74	0x4A	112	J	106	0x6A	152	j
11	0x0B	013	VT	43	0x2B	053	+	75	0x4B	113	K	107	0x6B	153	k
12	0x0C	014	FF	44	0x2C	054	,	76	0x4C	114	L	108	0x6C	154	l
13	0x0D	015	CR	45	0x2D	055	-	77	0x4D	115	M	109	0x6D	155	m
14	0x0E	016	SO	46	0x2E	056	.	78	0x4E	116	N	110	0x6E	156	n
15	0x0F	017	SI	47	0x2F	057	/	79	0x4F	117	O	111	0x6F	157	o
16	0x10	020	DLE	48	0x30	060	0	80	0x50	120	P	112	0x70	160	p
17	0x11	021	DC1	49	0x31	061	1	81	0x51	121	Q	113	0x71	161	q
18	0x12	022	DC2	50	0x32	062	2	82	0x52	122	R	114	0x72	162	r
19	0x13	023	DC3	51	0x33	063	3	83	0x53	123	S	115	0x73	163	s
20	0x14	024	DC4	52	0x34	064	4	84	0x54	124	T	116	0x74	164	t
21	0x15	025	NAK	53	0x35	065	5	85	0x55	125	U	117	0x75	165	u
22	0x16	026	SYN	54	0x36	066	6	86	0x56	126	V	118	0x76	166	v
23	0x17	027	ETB	55	0x37	067	7	87	0x57	127	W	119	0x77	167	w
24	0x18	030	CAN	56	0x38	070	8	88	0x58	130	X	120	0x78	170	x
25	0x19	031	EM	57	0x39	071	9	89	0x59	131	Y	121	0x79	171	y
26	0x1A	032	SUB	58	0x3A	072	:	90	0x5A	132	Z	122	0x7A	172	z
27	0x1B	033	ESC	59	0x3B	073	;	91	0x5B	133	[	123	0x7B	173	{
28	0x1C	034	FS	60	0x3C	074	<	92	0x5C	134	\	124	0x7C	174	
29	0x1D	035	GS	61	0x3D	075	=	93	0x5D	135	]	125	0x7D	175	}
30	0x1E	036	RS	62	0x3E	076	>	94	0x5E	136	^	126	0x7E	176	~
31	0x1F	037	US	63	0x3F	077	?	95	0x5F	137	_	127	0x7F	177	DEL

# Zeichenketten (7)

Operationen für Zeichen in `string.h` → <http://www.cplusplus.com/reference/cstring/>

- Kopierfunktionen

```
void * memcpy ( void * destination, const void * source, size_t num );  
void * memmove ( void * destination, const void * source, size_t num );  
char * strcpy ( char * destination, const char * source );  
char * strcat ( char * destination, const char * source );
```

- Vergleichsfunktionen

```
int memcmp ( const void * ptr1, const void * ptr2, size_t num );  
int strcmp ( const char * str1, const char * str2 );  
int strncmp ( const char * str1, const char * str2, size_t num );
```

- Suchfunktionen

```
const char * strchr ( const char * str, int character );  
const char * strpbrk ( const char * str1, const char * str2 );  
const char * strrchr ( const char * str, int character );  
const char * strstr ( const char * str1, const char * str2 );
```

- sonstige Funktionen

```
size_t strlen ( const char * str );  
void * memset ( void * ptr, int value, size_t num );
```

# Zeichenketten (8)

## Beispiele:

```
/* strncpy example */
#include <stdio.h>
#include <string.h>
int main ()
{
    char str1[] = "To be or not to be";
    char str2[40];
    char str3[40];

    /* copy to sized buffer (overflow safe): */
    strncpy( str2, str1, sizeof(str2) );

    /* partial copy (only 5 chars): */
    strncpy( str3, str2, 5 );
    str3[5] = '\0';    /* null character manually added */

    puts(str1);
    puts(str2);
    puts(str3);

    return 0;
}
```

```
/* memset example */
#include <stdio.h>
#include <string.h>

int main ()
{
    char str[] = "almost every
programmer should know memset!";
    memset(str, '-', 6);
    puts(str);
    return 0;
}
```

## 4. Effizientes Programmieren in C

1. Felder und Zeichenketten
2. **Standardein- und -ausgabe**
3. Zeiger
4. Felder und Zeiger
5. Benutzerdefinierte Typen
6. Anwendungsbeispiel "Verkettete Listen"
7. Kommandozeilenparameter
8. Dateiein- und -ausgabe
9. Programmerzeugung und Präprozessor
10. "Beliebte" Fehler



# Standardein- und -ausgabe (1)

**Ziel:** Kennenlernen aller Besonderheiten, im Kontext der Ausgabe auf den Bildschirm und der Eingabe von der Tastatur

## Vordefinierte (feste) Bezeichner

- EOF → end of file
- stdin → Standardeingabe (im Normalfall: Tastatur)
- stdout → Standardausgabe (im Normalfall: Bildschirm bzw. Terminal/Konsole)
- stderr → Standardfehlerausgabe (im Normalfall: identisch zu stdout)

## Warum diese **Bezeichner**?

- EOF wird gern genutzt, um Fehlerfälle kenntlich zu machen?
  - z.B. die Ausgabe hat - aus irgendwelchen Gründen - nicht funktioniert
- Später werden wir die Ein-/Ausgabe von/in Dateien kennenlernen; diese Bezeichner helfen der sauberen (gedanklichen) Trennung der Konzepte.
- Es handelt sich eigentlich um Zeiger auf Dateien, so dass mit diesen Bezeichnern genauso gearbeitet werden kann, wie bei der Ein-/Ausgabe in Dateien

# Standardein- und -ausgabe (2)

Die Funktionen für die Standardein- und -ausgabe sind in der Datei `stdio.h` deklariert.

- `#include <stdio.h>` ist also notwendig, wenn man mit den E/A-Funktionen arbeiten will.

Überblick (folgende Folien):

- **unformatierte Ein-/Ausgabe**  
→ Ein-/Ausgabe einzelner Zeichen oder von Zeichenketten
- **formatierte Ein-/Ausgabe**  
→ Daten werden in Übereinstimmung mit ihrem Datentyp ausgegeben/eingelese

# Unformatierte Ausgabe (1)

Prototyp: `int putchar(int c)`

Wirkung:

- schreibt Zeichen `c` als `unsigned char` nach `stdout`
- Rückgabewert ist entweder das geschriebene Zeichen oder aber EOF (im Fehlerfall)

Beispiel:

```
/* putchar example: printing the alphabet */
#include <stdio.h>

int main()
{
    char c;
    for (c = 'A' ; c <= 'Z' ; c++)
        putchar(c);
    return 0;
}
```

# Unformatierte Ausgabe (2)

Prototyp: `int puts(char* string)`

Wirkung:

- schreibt die Zeichenkette `string` nach `stdout` und ersetzt dabei das Endzeichen `'\0'` durch den Zeilentrenner `'\n'`
- Rückgabewert ist EOF im Fehlerfalle, ansonsten ein Integer  $>0$ 
  - Anzahl ausgegebener Zeichen inkl. Zeilenvorschub

Beispiel:

```
/* puts example : hello world! */
#include <stdio.h>

int main ()
{
    char string[] = "Hello world!";
    puts(string);
}
```



# Unformatierte Eingabe (1)

Prototyp: `int getchar(void)`

Wirkung:

- liest das nächste Zeichen von `stdin` als `unsigned char`
- im Fehlerfall erfolgt die Rückgabe EOF, ansonsten das gelesene Zeichen

Beispiel:

```
#include <stdio.h>          /* Schreibmaschine */
int main ()
{
    int c;
    puts("Enter text. Include a dot ('.') in a sentence to exit:");
    do
    {
        c = getchar();
        putchar(c);
    } while (c != '.');
    return 0;
}
```

# Unformatierte Eingabe (2)

Prototyp: `char* gets(char* string)`

Wirkung:

- liest eine Zeile von `stdin` in die Zeichenkette `string` hinein und ersetzt dabei den Zeilentrenner `'\n'` durch `'\0'`
- bei Fehlern ist die Rückgabe `NULL`
- am Zeiger `string` muss genügend (eigener) Speicher dranhängen, um die Zeichenkette aufzunehmen

Beispiel: 

```
/* gets example */
#include <stdio.h>
int main()
{
    char string [256];
    printf("Insert your full address: ");
    gets(string); // Warnung: Fehlerfall unberücksichtigt
    printf("Your address is: %s\n",string);
    return 0;
}
```

# Formatierte Ausgabe (1)

Prototyp: **int printf(<formatstring>, <par1>, <par2>, ...)**



Wirkung:

- schreibt die Werte <par1>, <par2>, ... nach stdout
- der <formatstring> steuert die Umwandlung der Parameter und deren Formatierung
- Rückgabewert ist die Anzahl der geschriebenen Zeichen oder aber EOF im Fehlerfall

Der **Formatstring** besteht aus

- dem Ausgabetext
- und ggf. den Formatanweisungen

Text im Formatstring wird **unverändert** widergegeben.

**Formatanweisungen** bestehen aus einem **Prozentzeichen** und einem Zeichen, das den Datentyp der auszugebenden Variable spezifiziert. Z.B. %d

- Beim Ausgeben wird jede Formatanweisung mit einem der Parameter der `printf()`-Anweisung besetzt.
- Eine Formatanweisung ist also nur ein Platzhalter.

# Formatierte Ausgabe (2)

Beispiel: `printf("Die Summe von %d und %d ist %d", a, b, a+b);`

Liste der  
Format-  
anweisungen  
im Format-  
string:

Formatanweisung	Bedeutung
<code>%d</code>	int, short int oder char als Zahl in Dezimalnotation
<code>%o</code>	int, short int oder char als Zahl in Oktalnotation
<code>%x</code>	int, short int oder char als Zahl in Hexadezimalnotation
<code>%u</code>	unsigned int, unsigned char als Zahl in Dezimalnotation
<code>%c</code>	int, short int oder char als Zeichen
<code>%s</code>	Ausgabe von Zeichenketten (char* bzw. char[])
<code>%e, %E</code>	float in Exponentialschreibweise (kleines e, großes E)
<code>%f</code>	float in Fließkommenschreibweise
<code>%le, %lf</code>	long float (=double) in beiden Schreibweisen
<code>%g</code>	float ohne Angabe der nachfolgenden Nullen
<code>%p</code>	Zeigerwert (als Hexadezimalzahl)

# Formatierte Ausgabe (3)

Liste der speziellen Zeichenkonstanten (nur Auswahl) im Formatstring:

Spezielles Zeichen	Bedeutung
' \a '	bell, Klingel / Signalton
' \b '	backspace, Rückschritt
' \n '	newline, Zeilenvorschub
' \r '	return, Wagenrücklauf
' \t '	tab, Tabulator
' \\ '	backslash, Rückstrich
' \\" '	Anführungszeichen
' \0 '	Zeichen, dem die Zahl 0 zugeordnet ist, Ende einer Zeichenkette

# Formatierte Ausgabe (4)

## Erweiterte Formatanweisungen

- ermöglichen es, numerische Werte spaltenweise auszugeben
- ermöglichen es, die Anzahl der Dezimalstellen zu begrenzen

### Syntax: `% [flags] [width] [.prec] [l] type`

- Dabei sind `flags` nur für Zahlen von Bedeutung und können sein: `-`, `+`, `#`, `0`, Leerzeichen
  - `-` → linksbündige Ausgabe
  - `+` → Vorzeichen (`-` oder `+` wird stets angegeben)
  - `0` → linksseitige Auffüllung mit Nullen
  - `#` → Dezimalpunktangabe mit folgenden Nullen
  - Leerzeichen → wenn kein Vorzeichen geschrieben wird, wird ein Leerzeichen gesetzt
- `width` spezifiziert die minimale Breite des Ausgabefeldes (Spaltenbreite)
  - wenn Ausgabe größer (breiter) → Breitenangabe wird ignoriert
  - wenn Ausgabe kleiner (schmäler) → Auffüllung mit Leerzeichen (standardmäßige Linksausrichtung)
- `.prec` spezifiziert die Anzahl der auszugebenden Nachkommastellen bei Fließkommazahlen
- `l` steht für die long-Version des Datentyps
- `type` bezeichnet eine der „standardmäßigen“ Formatanweisungen (sh. vorige Folie)

# Formatierte Ausgabe (5)

## Beispiel

```
/* puts / printf example */
#include <stdio.h>
#include <stdlib.h>

int main()
{
    int a=3, b=5;
    double c=22.45;
    char d[] = "Testzeichenkette";
    puts(d);
    printf("Die Summe von %d und %d ist %d.\n", a, b, a+b);
    printf("%s\n", d);
    printf("%.5le\n%.6lf\n", c, c);
}
```

# Formatierte Eingabe (1)

Prototyp: `int scanf(<formatstring>, <par1>, <par2>, ...)`

Wirkung:

- liest von `stdin` unter der Kontrolle vom `formatstring` Werte ein
- über Zwischenraumzeichen und Zeilentrenner wird hinweggelesen („White Spaces“)
- `<par1>`, `<par2>`, ... müssen Zeiger sein
- Rückgabewert ist die Anzahl der eingelesenen und umgewandelten Eingaben oder EOF im Fall, dass ein Fehler aufgetreten ist
  - Rückgabewert kann zur Fehlerbehandlung benutzt werden

Beispiel: `int main()`

```
{  
    int a, num;  
    double c;  
    char d[80];  
    num = scanf("%d %lf", &a, &c );  
    scanf("%s", d);  
    printf("Ihre Eingaben waren: %d %lf %s\n", a, c, d);  
}
```

Adressen von a und c

79 Zeichen zuzüglich des terminierenden '\0'



# Formatierte Eingabe (2)

Beispiel: `scanf("%d %d %d", &a, &b, &c );`

Liste der  
Format-  
anweisungen  
im Format-  
string:

Formatanweisung	Bedeutung
<code>%d</code>	Zeiger auf int, Eingabe in Dezimalnotation
<code>%u</code>	Zeiger auf unsigned int, Eingabe in Dezimalnotation
<code>%c</code>	Zeiger auf char
<code>%x</code>	Zeiger auf int, Eingabe in Hexadezimalnotation
<code>%o</code>	Zeiger auf int, Eingabe in Oktalnotation
<code>%i</code>	Zeiger auf int, Eingabe dezimal/oktal/hexadezimal
<code>%e, %E</code>	Zeiger auf float, Eingabeformat wie bei printf()
<code>%f</code>	Zeiger auf float, Eingabe in Dezimalschreibweise
<code>%le, %lf</code>	Zeiger auf long-Version eines float (double)
<code>%s</code>	Zeiger auf char. Es werden alle Zeichen bis zum nächsten Whitespace gelesen und abgespeichert.

# Formatierte Eingabe (3)

## Erweiterte Formatanweisungen

- ermöglichen eine feinere Kontrolle der Eingabe

### Syntax: `%[*][width][lh]type`

- Dabei bedeutet `*`, dass die Zeichen eingelesen aber nicht gespeichert werden.
- `width` gibt die maximale Anzahl der zu lesenden Zeichen an
- `l` gibt an, dass der Parameter ein Zeiger auf eine long-Variable ist
- `h` gibt an, dass der Parameter ein Zeiger auf eine short-int-Variable ist
- `type` ist eine einfache Formatanweisung

### Beispiel:

```
char s[100];  
scanf("%99s", s);
```

- Bedeutung: Lese in die Zeichenkette `s` nicht mehr als 100 Zeichen.
- `scanf()` **ändert den Wert** der Variablen in der Liste der Argumente (in dem Fall: `s`).

# Formatierte Eingabe (4)

## Alternativer Weg zur „formatierten“ Eingabe

- Zeile mit `gets` als Zeichenkette einlesen
- die nun in der Zeichenkette gespeicherte Eingabe mit Umwandlungsfunktionen aus `<stdlib.h>` auslesen bzw. konvertieren

## Umwandlungsfunktionen

- `int atoi(char* string);` → „Ascii to Integer“
- `float atof(char* string);` → „Ascii to Float“

## Beispiel:

```
int main()
{
    int a;
    char help[80];
    gets(help);
    a = atoi(help);
    printf("Ihre Eingabe war: %d\n", a);
    return 0;
}
```

# Formatierte Eingabe (5)

Beispiel:

```
int val;
int result;
result = scanf("%d", &val);
if (result == EOF)
{
    /* print an error message */
}
```

Man beachte das `&val` in `scanf()` !

```
int val, result;
result = scanf("%d", &val);
```

- Es wird die Adresse der Variablen `val` an die Funktion `scanf` übergeben (also quasi ein Zeiger), so dass diese Funktion in die Speicherzellen von `val` etwas hineinschreiben kann.
- Dies ist **Call by Reference**!
- **Regel:** Das `&` ist notwendig, wenn `int`, `doubles`, etc. gelesen werden, aber nicht bei Zeichenketten (Strings), da diese bereits in einer Zeigervariablen gespeichert sind.

## 4. Effizientes Programmieren in C

1. Felder und Zeichenketten
2. Standardein- und -ausgabe
3. **Zeiger**
4. Felder und Zeiger
5. Benutzerdefinierte Typen
6. Anwendungsbeispiel "Verkettete Listen"
7. Kommandozeilenparameter
8. Dateiein- und -ausgabe
9. Programmerzeugung und Präprozessor
10. "Beliebte" Fehler



# Zeiger (1)

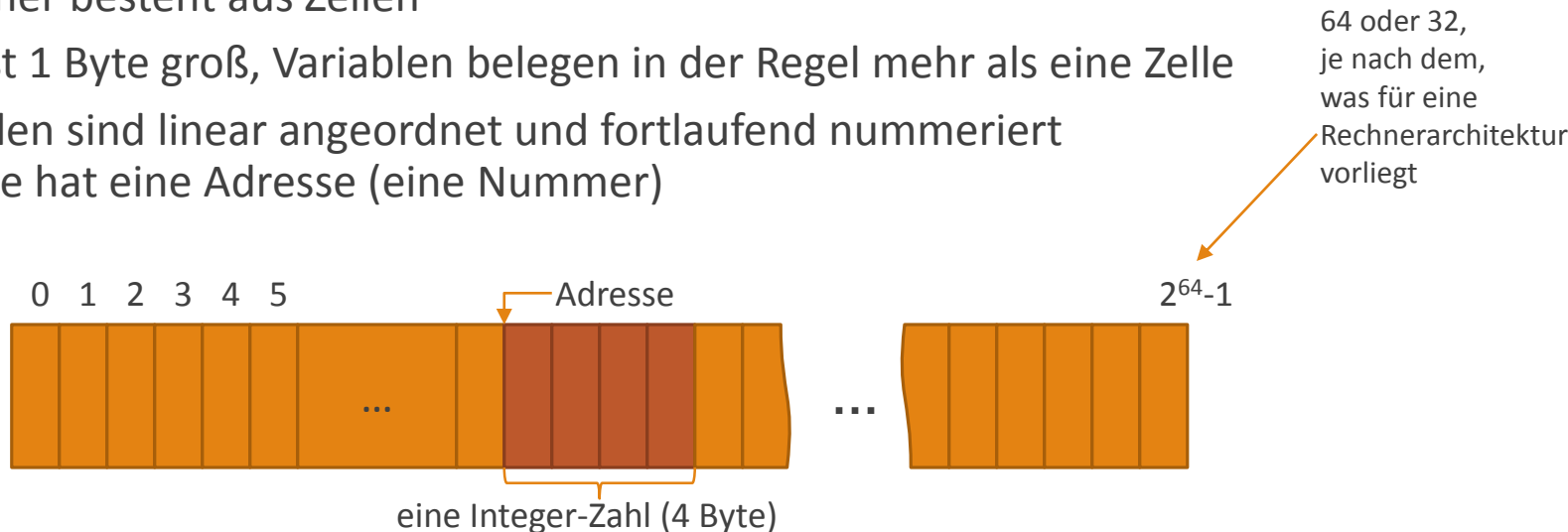
**Definition:** Ein **Zeiger** (engl. pointer) ist eine Variable, die die Adresse einer Variablen enthält.

Erläuterung:

- Begriff „**Adresse**“ bedeutet **Hauptspeicheradresse**
- Ein Zeiger speichert also eine Adresse im Hauptspeicher.

Erinnerung: Speicherorganisation

- Hauptspeicher besteht aus Zellen
- jede Zelle ist 1 Byte groß, Variablen belegen in der Regel mehr als eine Zelle
- Speicherzellen sind linear angeordnet und fortlaufend nummeriert  
→ jede Zelle hat eine Adresse (eine Nummer)



# Zeiger (2)

## Definition einer Zeigervariable

- Ein Zeiger ist eine Variable. → Frage: Hat diese Variable einen Typ?
- Antwort: **Ja, Zeiger sind typspezifisch!**
  - D.h. ein Zeiger ist zum Beispiel „**ein Zeiger auf ein Integer**“ oder ein „**Zeiger auf ein Double**“
- Eine Zeigervariable, die zum Zeigen auf einen bestimmten Typ definiert wurde, darf auch nur auf Variablen dieses Typs zeigen.
  - Der Datentyp eines Zeiger lässt sich nicht nachträglich ändern.

## Syntax: **datentyp\* name;**

- definiert eine Zeigervariable mit Namen `name`, die auf eine Variable vom Typ `datentyp` zeigt
- Hinweis: Wo das Leerzeichen in der Definition steht, ist „Geschmackssache“. Auch legitim:
  - **datentyp \* name;** // Leerzeichen vor und nach dem Stern
  - **datentyp \*name;** // Leerzeichen nur vor dem Stern

## Beispiele:

- **int\* ptr;** // ein Zeiger auf ein Integer
- **double\* zeiger;** // ein Zeiger auf ein Double

# Zeiger (3)

Frage: Wie weist man einer Zeigervariable nun einen sinnvollen Wert, also die Adresse einer anderen Variable, zu?

- Die Adresse einer Variablen  $x$  erhält man durch den Adressoperator  $\&$ :  $\&x$
- Dieser Adresswert lässt sich einer Zeigervariablen zuweisen.

Beispiel:

- $i$  sei eine ganz normale Integer-Variable,  $j$  sei ein Zeiger auf ein Integer.

```
int i = 10;  
int* j = &i; /* j "zeigt" auf i */
```

```
int i = 10;  
int* j; //alternativ  
j = &i; //alternativ
```

- Visualisierung:

Variablenname	Adresse	Inhalt
---------------	---------	--------

$i$

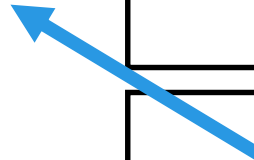
$0x123aa8$

$j$

$0x123aab$

10

$0x123aa8$





# Zeiger (4)

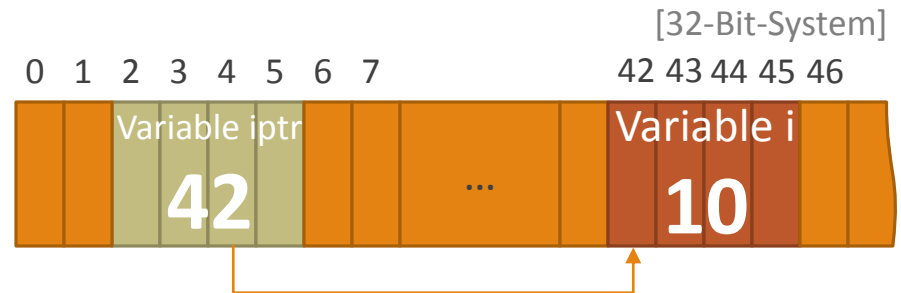
## Zeiger: Warum das alles?

1. Durch die Definition der Zeigervariablen wird nur so viel Speicherplatz reserviert, wie für die Darstellung einer Adresse notwendig ist.
  - auf 32-Bit-Rechnern: 32 Bit, also 4 Byte
  - auf 64-Bit-Rechnern: 64 Bit, also 8 Byte
  - Der sich daraus ergebende Vorteil wird erst deutlich, wenn benutzerdefinierte Typen besprochen worden sind.
2. Parameterübergabe an Funktionen

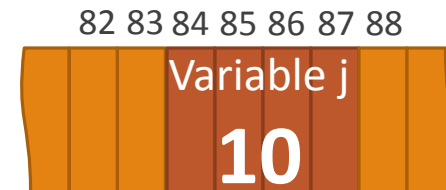
## Operatoren für Zeiger

- **Adressoperator: &**
  - bereits kennengelernt, siehe vorangegangene Folie
- **Inhaltsoperator (Dereferenz): \***
  - Wenn man diesen auf eine Zeigervariable anwendet, so liefert er den Wert / das Objekt, auf das der Zeiger zeigt.
- Beispiel:

```
int i = 10;
int* iptr = &i; // Adressoperator
int j = *iptr; // Dereferenzierung mit Inhaltsoperator
```



[Die Adresse 42 wird in der Regel als Hexadezimalwert angegeben, also: 0x0000002A]



# Zeiger (5)

Also: Das Datum, auf das ein Zeiger zeigt, ist mit dem **\*-Operator** zugreifbar.

```
int i = 10;
int* iptr = &i;
printf("i = %d\n", i );
printf("iptr = %x\n", iptr );
printf("iptr points to: %d\n", *iptr);
```

➔ Beispiel: `memory_example.c`

Regel:

- **&i** ist die Adresse von `i`
- **\*iptr** ist der Inhalt des Speichers an der Stelle, auf die `iptr` zeigt
- Der **\***-Operator führt eine Dereferenzierung durch.

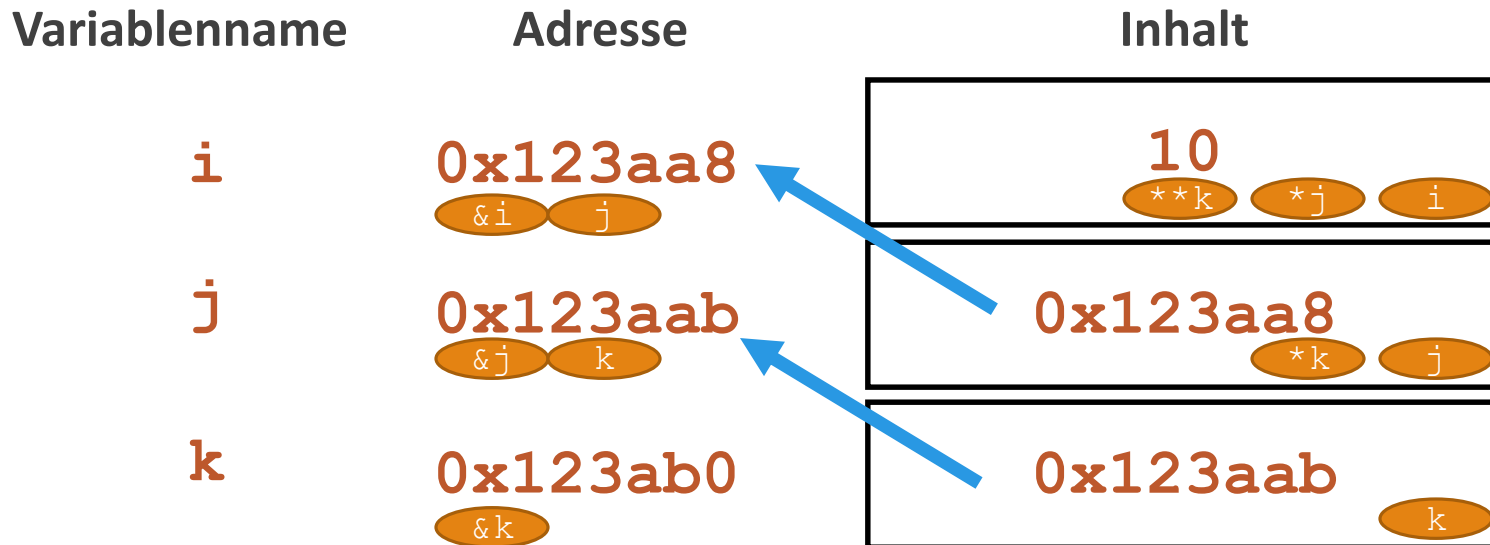
Achtung: Verschiedene Bedeutungen des Zeichens **\*** in C:

1. Multiplikationsoperator
2. Deklaration einer Zeigervariable
3. Dereferenzieren (Zugriff auf den Inhalt dessen, worauf ein Zeiger verweist)

# Zeiger (6)

Zeiger auf Zeiger sind auch möglich (usw.)!

```
int    i = 10;
int*   j = &i; // wie gehabt
int**  k = &j; // Zeiger auf die Zeigervariable j
printf("%x\t%d\n", &i, i); // i und seine Adresse
// Adresse von j, Wert von j und Wert dessen, worauf j zeigt:
printf("%x\t%x\t%d\n", &j, j, *j);
printf("%x\t%x\t%x\t%d\n", &k, k, *k, **k);
```



# Zeiger (7)

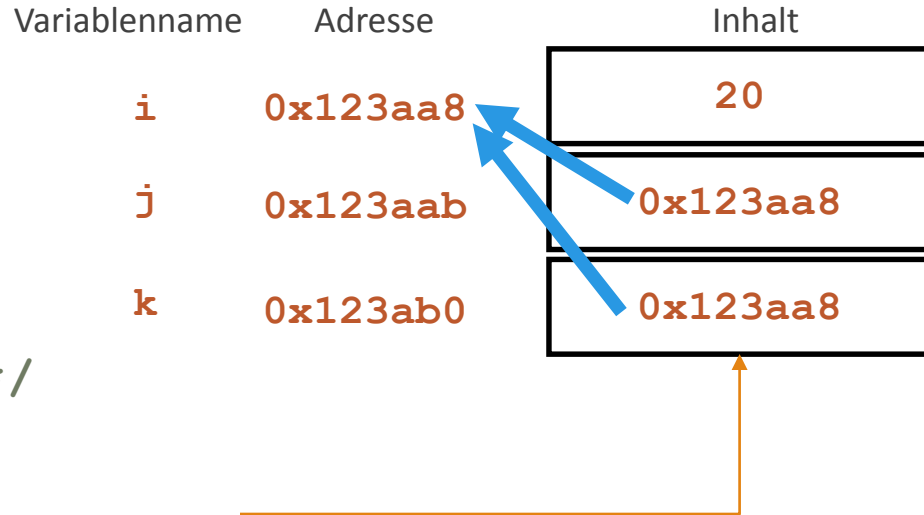
## Beispiele: Zuweisungen an Zeiger

```
int i = 10;  
int* j = &i;  
int* k;
```

```
// Assign to contents of j:  
*j = 20; /* Now i is 20. */
```

```
// Assign j to k:  
k = j; /* Now k points to i too.
```

```
// Assign to contents of j:  
*j = *k + i; // = 20 + 20 = 40
```



Wenn eine Zeigervariable auf der linken Seite einer Zuweisung steht, hängt das Programmverhalten davon ab, ob sie dereferenziert ist, oder nicht.

```
j = k;          *j = *k + 10;
```

# Zeiger (8)

Es existieren **zwei Möglichkeiten zur Konstruktion eines Zeigers** (also zur Erzeugung einer Zeigervariablen, welche eine Hauptspeicheradresse enthält).

## 1. Statische Vorgehensweise

- Variablen werden ganz normal (quasi „statisch“) angelegt.
- Anschließend wird eine Zeigervariable definiert, die auf die zuvor angelegte Variable zeigt.
- wie in den Beispielen auf den vorigen Folien

## Verwendung:

- Nutzung von Bibliotheksfunktionen, die Zeiger als Parameter erwarten
- Übergabe von großen Datenstrukturen (z.B. Verbünden) an Funktionen
- Damit: Realisierung des Parameterübergabemechanismus **Call-by-Reference** in C
- Beispiel: nächste Folie

## 2. Dynamische Vorgehensweise

- sh. in einigen Folien unter „Dynamische Speicherallokation“

# Zeiger (9)

Erinnerung: In C werden Variablen kopiert, bevor sie zu einer Funktion gesendet werden.

```
void incr(int i)
{
    i++;
}

/* ... later ... */
int j = 10;
incr(j); /* want to increment j */
/* What is j now? */
/* Still 10. So, incr() does nothing. */
```

- Dies ist **Call-By-Value** (vgl. Folien zum Thema „Parameterübergabemechanismen“).
- Der Wert kann nur lokal geändert werden.

Aber: Oft will man aber den Wert einer Variable verändern (im Rahmen des Funktionsaufrufes)! ➔ **Call-By-Reference**

# Zeiger (10)

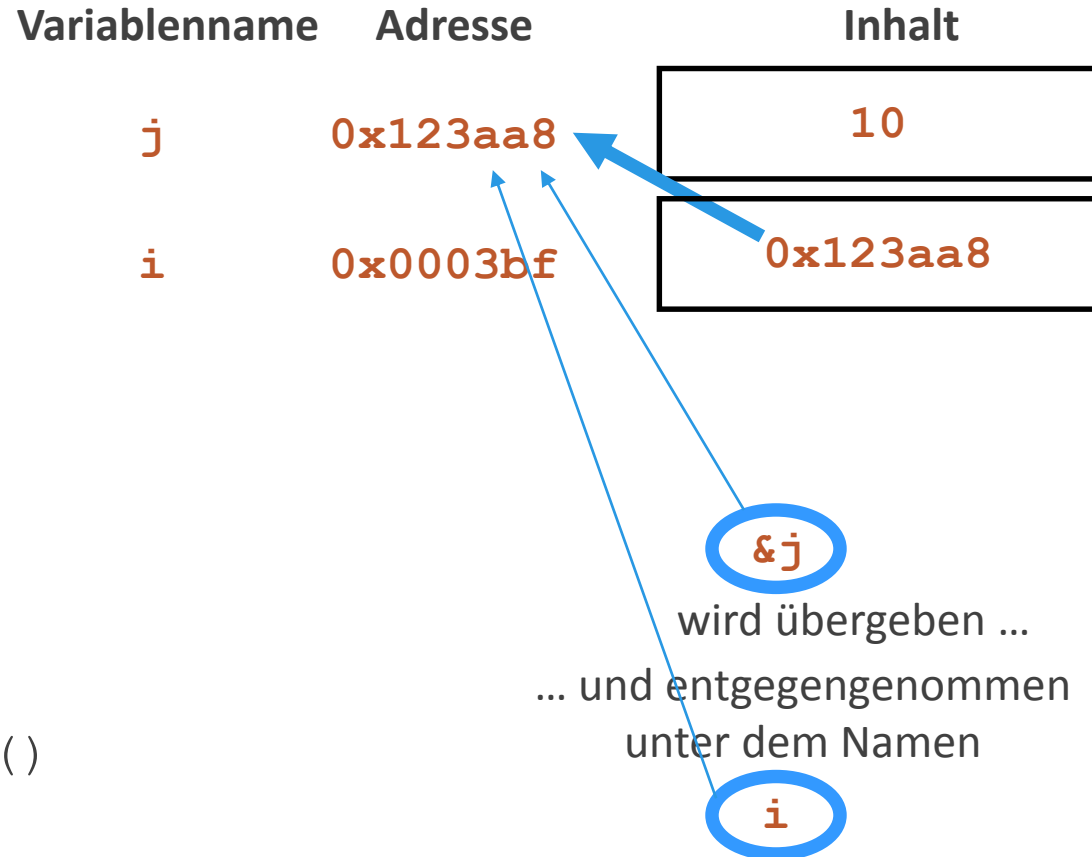
## Zeiger für Call-By-Reference

```
void incr(int* i)
{
    (*i)++;
}
/* ... later ... */
int j = 10;
incr(&j);
/* What is j now? */
/* Yep, it's 11. */
```

Alles klar?

Wir hatten dies zuvor: bei scanf()

```
int i;
scanf("%d", &i); /* read in i */
```



# Zeiger (11)

Achtung:

```
void incr(int* i)
{
    *i++;    /* Won't work as intended! */
            /* Parsed as: *(i++); */
}
```

- Korrekt wäre `(*i)++;`

Empfehlung:

- Stets Klammern benutzen, um Konfusion und Unübersichtlichkeit bzw. Mehrdeutigkeiten zu vermeiden!

Der **void**-Zeiger

- Bislang betont: Alle Zeiger sind **typspezifisch**, d.h. zeigen auf Variablen eines bestimmten Typs.
- Ausnahme: `void*` ist ein **typunspezifischer Zeiger**, der auf alle Datentypen zeigen kann.
- Achtung: Darf nicht zum Zugriff auf konkrete Daten verwendet werden.
- Beispiel: `void* a; // a ist void-Zeiger, Details dazu später`



# Dynamische Speicherallokation (1)

Erinnerung: Es existieren **zwei Möglichkeiten zur Konstruktion eines Zeigers**

Nun Betrachtung von:

## 2. Dynamische Vorgehensweise

- Zur Laufzeit wird vom Programm Hauptspeicher angefordert, um Daten abzulegen.
- Dafür stellt C spezielle Reservierungsfunktionen bereit.
- **Rückgabewert** dieser Reservierungsfunktionen sind **Zeiger auf den Anfang des reservierten Speicherbereiches**.

## Dynamische Speicherallokation

- Der Heap ist ein zusammenhängender Speicherbereich, der dem C-Programm zur Verfügung steht.
- Daten werden im Hauptspeicher so abgelegt:
  - Aufruf der Funktion **malloc(size)** (sh. nächste Folie) mit der genauen Angabe, wie viel Speicherplatz benötigt wird.
  - Wenn der Speicherplatz reserviert werden konnte, wird die Startadresse des Speicherblocks zurückgegeben.
  - Nun kann der Speicherblock mit Daten gefüllt werden.

# Dynamische Speicherallokation (2)

Die Funktionen `malloc()` und `free()`

- Syntax: `#include <stdlib.h>`  
`...`  
`void* malloc(unsigned int size);`  
`void free(void* ptr);`

Erläuterungen:

- Die Funktion `malloc()` reserviert während der Laufzeit des Programm einen Speicherblock und liefert die Startadresse des Blocks zurück.
- Weil dies während der Laufzeit geschieht, bezeichnet man diese Art der Speicheranforderung als „**dynamisch**“.
- Mit der Funktion `free()` kann dieser Speicher wieder freigegeben werden.
- Nach Aufruf von `free()` steht der ursprünglich dynamisch allokierte Speicherblock dem Programm nicht mehr zur Verfügung.
- Die Benutzung von `free()` sollte nicht vergessen werden, es sei denn, das Programm benötigt den angeforderten Speicher bis zu seiner Beendigung.
  - Bei Programmende „räumt“ das Betriebssystem ohnehin auf.

# Dynamische Speicherallokation (3)

**Frage:** Was passiert, wenn die dynamische Speicherallokation fehlschlägt?

- Z.B. wenn das Programm 10GB Speicher anfordert, aber der Rechner nur über 2GB verfügt

**Antwort:** Der Fehlerfall wird angezeigt durch Rückgabe des **NULL-Zeigers**.

**Definition:** Der **NULL-Zeiger** ist ein vordefinierter Zeiger, dessen Wert sich von allen regulären Zeigern unterscheidet.

- Er wird von Funktionen benutzt, um einen Fehler anzuzeigen.
- Bei jedem Aufruf einer Funktion, die einen Zeiger zurückliefert, sollte man den Zeiger mit dem NULL-Zeiger vergleichen und ggf. das Programm mit einer Fehlermeldung abbrechen.

**Regeln:**

- Zeiger, die momentan auf „nichts“ zeigen, sollten stets explizit gleich NULL gesetzt werden.
  - Beispiel: `int* meinNeuerZeiger = NULL;`
- Das Dereferenzieren eines NULL-Zeigers oder sogar der schreibende Zugriff über einen NULL-Zeiger führt zu undefiniertem Verhalten, in vielen Fällen zum Programmabsturz.
  - Beispiel: `int* meinZeiger = NULL;`  
`*meinZeiger = 42; // Undefiniert / Programmabsturz`

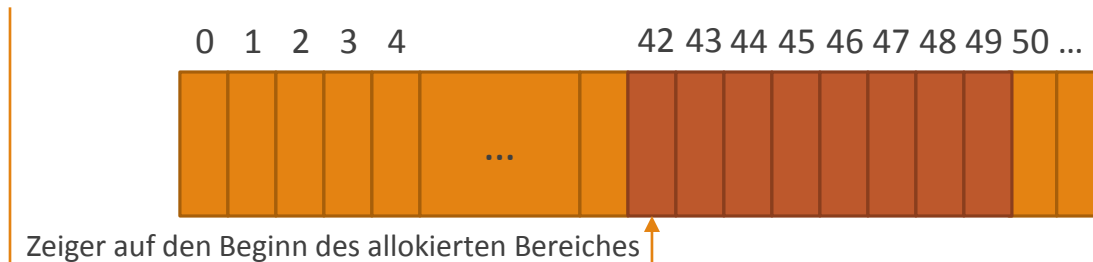
# Dynamische Speicherallokation (4)

## Größenangabe beim Aufruf von `malloc()`

- Bei jedem Aufruf von `malloc()` muss die Anzahl der zu reservierenden Bytes übergeben werden, d.h. die Größe des Speicherblockes, das durch den Zeiger referenziert wird.

- Beispiel: 

```
int* meinSpeicher;  
meinSpeicher = malloc( 8 );
```



**Problem:** Konstanten (wie 8) führen zu schlecht portierbaren Programmen.

**Regel:** Man sollte stets ein Vielfaches der Größe des referenzierten Datentyps anfordern. Dazu kann der `sizeof()`-Operator eingesetzt werden.

- Beispiel: 

```
int* myMem;  
myMem = malloc( 100 * sizeof(int) ); //speziell  
myMem = malloc( 100 * sizeof( *myMem ) ); //allgemeiner
```

# Dynamische Speicherallokation (5)

## Leerer Typ **void**

- Erinnerung: `void` hat keinen Typ und auf ihm sind keine Operationen definiert

## Verwendungsarten für `void`:

1. Kennzeichnung von Funktionen ohne Parameter und/oder ohne Rückgabewert mit `void`
2. `void*` als allgemeiner, d.h. typunspezifischer Zeiger, den man bei Bedarf in einen Zeiger eines ganz bestimmten Typs umwandeln kann

- Die `malloc()`-Funktion liefert einen `void`-Zeiger zurück.
- Um sauber zu arbeiten, muss der zurückgegebene Zeiger in den Zeigertyp umgewandelt werden, den man erwartet.
- Beispiel:

```
int* myMem;
```

```
myMem = malloc( 100 * sizeof(int) );
```



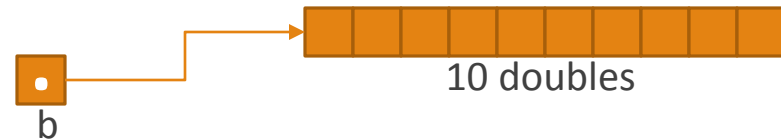
```
myMem = (int*)malloc( 100 * sizeof(int) );
```

- Durch die Typumwandlung (den Cast) vom zurückgelieferten `void*` in einen `int*` steht fest, dass es sich um einen Speicherbereich für Integer-Werte handelt. C bzw. die Zeigervariable `myMem` weiß dann auch, in was für Schritten „weitergesprungen“ werden muss, um zum nächsten Datum zu kommen (bei `int`: 4 Byte).

# Dynamische Speicherallokation (6)

Beispiel:

```
...  
int* a;  
double* b;  
  
...  
a = (int*) malloc( sizeof(int) );  
b = (double*) malloc( 10 * sizeof(double) );  
  
...  
free( a ); // aufräumen  
free( b );
```



- **Bemerkung:** `free(p)` gibt den Bereich frei, auf den `p` zeigt. Dieser Bereich muss aber vorher dynamisch, d.h. via `malloc()` –Aufruf, allokiert worden sein.

```
int a = 10;  
int* p = &a;  
free(p); // Kompiliert, aber Fehler zur Laufzeit
```

```
*** Error in `./main': free(): invalid pointer: 0x00007ffde5b15d64 ***  
Abgebrochen (Speicherabzug geschrieben)
```

# Dynamische Speicherallokation (7)

Frage: Was, wenn der dynamisch allokierte Speicherbereich nicht mehr ausreicht?

Antwort: Nachallokieren mittels **realloc()** !

- Syntax: **#include <stdlib.h>**

...

**void\* realloc(void\* ptr, unsigned int size);**

Regeln:

- Reallokation ändert den dynamisch allokierten Bereich, auf den `ptr` zeigt, auf eine Größe von `size` Bytes (Vergrößerungen **und** Verkleinerungen sind möglich).
- Inhalte im zuvor allokierten Bereich bleiben unverändert. Der neu hinzugekommene allokierte Bereich wird nicht initialisiert.
- Wenn als Parameter `ptr` der Wert `NULL` übergeben wird, so ist der Aufruf von `realloc` äquivalent zu einem `malloc`-Aufruf.
- Wenn `ptr` nicht `NULL` ist und als Größe 0 übergeben wird, ist der Aufruf von `realloc` äquivalent zu einem Aufruf von `free(ptr)` Aufruf.
- Wichtig: Wenn `ptr` nicht `NULL` ist, so muss `ptr` der Rückgabewert eines vorangegangenen `malloc`-Aufrufes gewesen sein.

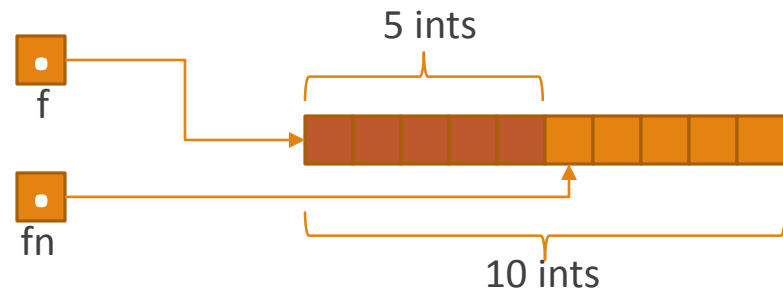
# Dynamische Speicherallokation (8)

## Rückgabewert von `realloc` ist

- ein Zeiger auf den neu (also zusätzlich) allokierten Bereich im Fall einer Vergrößerung
- ein Zeiger auf den Beginn des gesamten Bereiches (und damit identisch zu `ptr`) im Fall einer Verkleinerung des dynamisch allokierten Bereichs
- `NULL` oder ein Zeiger, der an `free` übergeben werden kann, wenn als Größe 0 angegeben wurde
- `NULL` im Fehlerfall
  - Im Fehlerfall bleibt der ursprünglich allokierte Bereich unangetastet.

Beispiel:

```
#include <stdio.h>
#include <stdlib.h>
int main(void)
{
    int* f = (int*)malloc( 5*sizeof(int) );
    printf("f allokiert bei Adresse %p\n", f);
    int* fn = (int*)realloc(f, 10*sizeof(int) );
    printf("fn reallokiert bei Adresse %p\n", fn);
}
```





# 4. Effizientes Programmieren in C

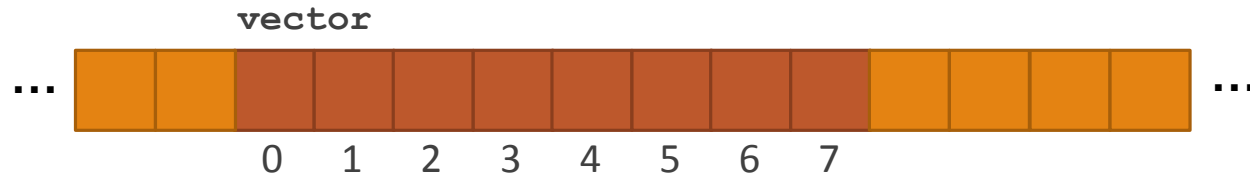
1. Felder und Zeichenketten
2. Standardein- und -ausgabe
3. Zeiger
4. **Felder und Zeiger**
5. Benutzerdefinierte Typen
6. Anwendungsbeispiel “Verkettete Listen”
7. Kommandozeilenparameter
8. Dateiein- und -ausgabe
9. Programmerzeugung und Präprozessor
10. “Beliebte” Fehler



# Felder und Zeiger (1)

## Dynamisches eindimensionales Feld

- Erinnerung: Speicherstruktur bei Feldern (für einen festen Wert von  $n=8$ )



- Problem:** Bei der Definition von

`int vector[n];`

muss  $n$  bereits zur Kompilierzeit seinen Wert haben (kann also nicht „live“ berechnet worden sein).

- Lösung:** Vereinbare einen Ersatztyp, der sich genauso verhält wie der gewünschte Typ!

```
int* vector;
```

```
int n;
```

```
...
```

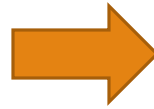
```
// berechne n irgendwie ...
```

```
...
```

```
vector = (int*)malloc( n * sizeof(int) );
```

```
...
```

```
free( vector );
```



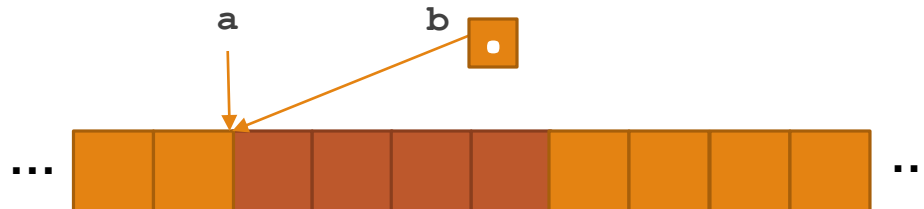
Erzeugt die gleiche Speicherstruktur wie im Fall der statischen Felderzeugung, aber nun „dynamisch“.

# Felder und Zeiger (2)

**Erkenntnis:** Die Variablendefinitionen `int a[4];` und `int* b;` vereinbaren vom Zugriffsverhalten her ähnliche Typen von Variablen: **Beide sind als Zeiger interpretierbar!**  
Insbesondere lässt sich `a` als Zeiger auf ein Integer interpretieren  
→ konkret: als Zeiger auf das erste Feldelement `a[0]`  
→ Daher: `a[0] == *b`

## Aber:

- `a` zeigt auf bereits reservierten Speicherplatz (4 int). `b` nicht.
    - Durch `int* b;` wird lediglich Speicher für einen Zeiger auf ein Integer reserviert.
  - `a` kann während des Programmablaufs nicht mehr „umgebogen“ werden. `b` schon.
  - `a` existiert nicht wirklich als Zeiger (nur gedanklich). `b` schon.
  - Eine Zuweisung an `a` der Form `a=...` ist nicht möglich.
  - Aber eine Zuweisung der Form `b=a;` ist erlaubt.
- bezeichnet man auch als **Zeigerkonstante**

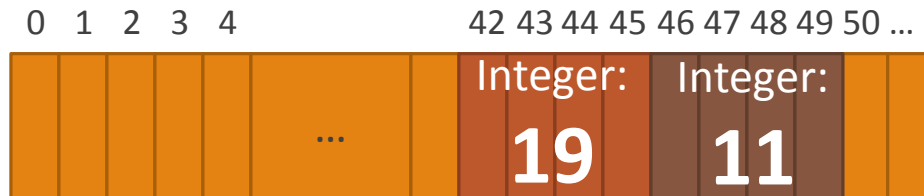


# Felder und Zeiger (3)

## Zeigerarithmetik:

- Wendet man arithmetische Operationen auf Zeigervariablen wie `int* i` an, so wird mit den Adresswerten gerechnet.
- Die Zeigervariable weiß, auf welchen Typ sie zeigt, und weiß auch um wie viele Byte sie „weitschalten“ muss, um auf den nächsten Wert dieses Typs zu zeigen.
- Beispiel:
  - `int* i` wurde definiert und `i` zeigt gerade auf die Adresse 42 (in hex 2a).
  - Dann: `i = i+1`; → Ergebnis: `i` zeigt nun auf 46 (bei 32/64-Bit-System), da auf das nächste `int` weitergeschaltet wird.

## Beispiel:



```
int* i;
...      // i==0x002a
printf("%x, %d", i, *i); //Ausgabe: 0x002a, 19
i++;     // i==0x002e
printf("%x, %d", i, *i); //Ausgabe: 0x002e, 11
```

# Felder und Zeiger (4)

## Operationen der Zeigerarithmetik:

1. Erlaubt ist das Addieren eines Zeigers mit einem Integer-Wert.
2. Erlaubt ist das Subtrahieren eines Integer-Werts von einem Zeiger.
3. Erlaubt ist das Subtrahieren zweier Zeiger voneinander.
4. Erlaubt ist die Nutzung von Inkrement- und Dekrementoperatoren

## 1./2. Addieren/Subtrahieren eines Integer-Wertes auf einen Zeiger

- Rückgabewert ist wieder ein Zeiger

- Beispiel: 

```
int a[10];  
int b = *(a+5); //liefert das 6-te Element des  
                //Feldes und entspricht dem Ausdruck  
                //a[5]
```

- Aufpassen: Klammern nicht vergessen!

```
int a[10];  
int b = *a+5; //liefert a[0]+5
```

- Durch Subtraktion kann man auf vorhergehende Elemente zugreifen.

# Felder und Zeiger (5)

## 3. Subtraktion zweier Zeiger voneinander

- Rückgabewert ist eine Ganzzahl (Integer)
- Anwendung i.d.R. nur sinnvoll, wenn beide Zeiger auf Elemente desselben Feldes zeigen
- Ergebnis ist dann Anzahl der Elemente zwischen beiden Zeigern

## 4. Inkrement-/Dekrementoperatoren auf Zeigern

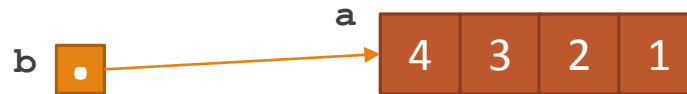
- liefern die Adresse des folgenden/vorhergehenden Elementes
- Beispiel: Was tut folgendes Programmfragment?

```
void mystrcpy( char* target, char* source)
{
    while ( *target++ = *source++ );
}
```

# Felder und Zeiger (6)

Beispiel: Zusammenhang zwischen Zeiger(arithmetik) und Feldern

```
int a[4];  
int* b = NULL;  
...  
a[0] = 4;  
a[1] = 3;  
a[2] = 2;  
a[3] = 1;  
...  
b = a;
```



- Es gilt:
  - `sizeof(a) = 16 Byte = 4 x 4 Byte` → Größe des gesamten Feldes
  - `sizeof(a[2]) = 4 Byte` → ein einzelner Integer-Wert
  - `sizeof(b) = 8 Byte` → Größe eines Zeiger auf einem 64-Bit-Rechner (4 Byte bei 32-Bit-System)
  - `sizeof(*b) = 4 Byte` → Dereferenzierung und damit ein einzelner Integerwert, identisch zu `a[0]`

# Felder und Zeiger (7)

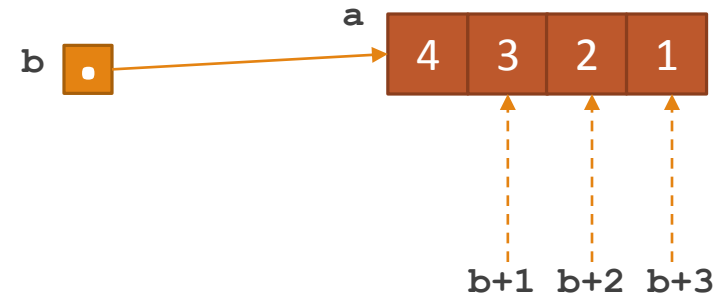
## Beispiel: Zusammenhang zwischen Zeiger(arithmetik) und Feldern (Fortsetzung)

- normale Zugriffe auf Feldelemente

- $a[0] \rightarrow 4$
- $a[1] \rightarrow 3$
- $a[2] \rightarrow 2$
- $a[3] \rightarrow 1$

- Zugriffe via Zeigerarithmetik

- $*b \rightarrow 4$  (Dereferenzierung)
- $*(b+1) \rightarrow 3$  (Zeiger um 1 int weitergeschaltet, danach dereferenziert)
- $*(b+2) \rightarrow 2$  (Zeiger um 2 int weitergeschaltet, danach dereferenziert)
- $*(b+3) \rightarrow 1$  (Zeiger um 3 int weitergeschaltet, danach dereferenziert)



**Regel:** Zugriffe über Feldindizes und über Zeigerdereferenzierung können äquivalent verwendet werden!

- Beispiel: Auch legitim sind ...

- $b[0] \rightarrow 4, b[1] \rightarrow 3, b[2] \rightarrow 2, b[3] \rightarrow 1$
- $*a \rightarrow 4, *(a+1) \rightarrow 3, *(a+2) \rightarrow 2, *(a+3) \rightarrow 1$
- Obwohl als Feld deklariert, kann ich `a` wie ein Zeiger ansprechen. Und: Obwohl als Zeiger deklariert, kann ich `b` wie ein Feld ansprechen.



# Felder und Zeiger (8)

Erinnerung: **Parameterübergabe** von eindimensionalen Feldern

- Bei Übergabe eines Feldes an eine Funktion wird **nicht** das ganze Feld kopiert.
- Übergeben wird nur die Anfangsadresse, also ein Zeiger auf den Beginn.
- **Äquivalenz** zur Übergabe eines Zeigers liegt auch hier vor!

```
...  
void test( double b[] ) // äquivalent: double b[6]  
{                       // äquivalent: double* b
```

b



→ Dies ist ein Verweis auf das „alte“ a aus dem Hauptprogramm; keine Kopie!

```
...  
}  
...  
int main(void)  
{  
    double a[6] = {1,2,3,4,5,6};  
    test(a);  
    ...  
}
```

a



# Felder und Zeiger (9)

## Zusammenfassung:

- **Zeigerarithmetik:** Man kann Integer-Werte von Zeigern subtrahieren und addieren.

- Beispiele: 

```
int i[5] = { 1, 2, 3, 4, 5 };
int* j = i;  /* (*j) == ? */
j++;        /* (*j) == ? */
j += 2;     /* (*j) == ? */
j -= 3;     /* (*j) == ? */
```

- Frage: Wie viele Bytes hat ein Integer? Antwort: sizeof!

```
printf("size of integer: %d\n", sizeof(int));
printf("size of (int*): %d\n", sizeof(int*));
```

- Zeigerarithmetik addiert/subtrahiert Adressen nicht direkt, sondern in Vielfachen des Typs in Byte.

```
int i[] = { 1, 2, 3, 4, 5 };
int* j = i;
j++;      /* means: j = j + sizeof(int); */
```

# Felder und Zeiger (10)

## Zusammenfassung:

- Felder sind im Endeffekt („versteckte“) Zeiger.

```
int i[5] = {1, 2, 3, 4, 5};  
printf("i[3] = %d\n", i[3]);  
printf("i[3] = %d\n", *(i + 3));
```

- Die Ausdrücke `i[3]` und `*(i+3)` sind identisch!
- Der Ausdruck `i` ist identisch zu `&i[0]` !
- Zeigerarithmetik lässt sich anstatt von Feldoperationen einsetzen!

```
int array[1000];    // Beispiel mit Feld-Operationen  
for (i = 1; i < 998; i++) {  
    array[i] = (array[i-1] + array[i] + array[i+1]) / 3;  
}
```

```
int array[1000];    // Beispiel mit Zeiger-Arithmetik  
for (i = 1; i < 998; i++) {  
    *(array+i) = (*(array+i-1) + *(array+i) + *(array+i+1)) / 3;  
}
```

## 4. Effizientes Programmieren in C

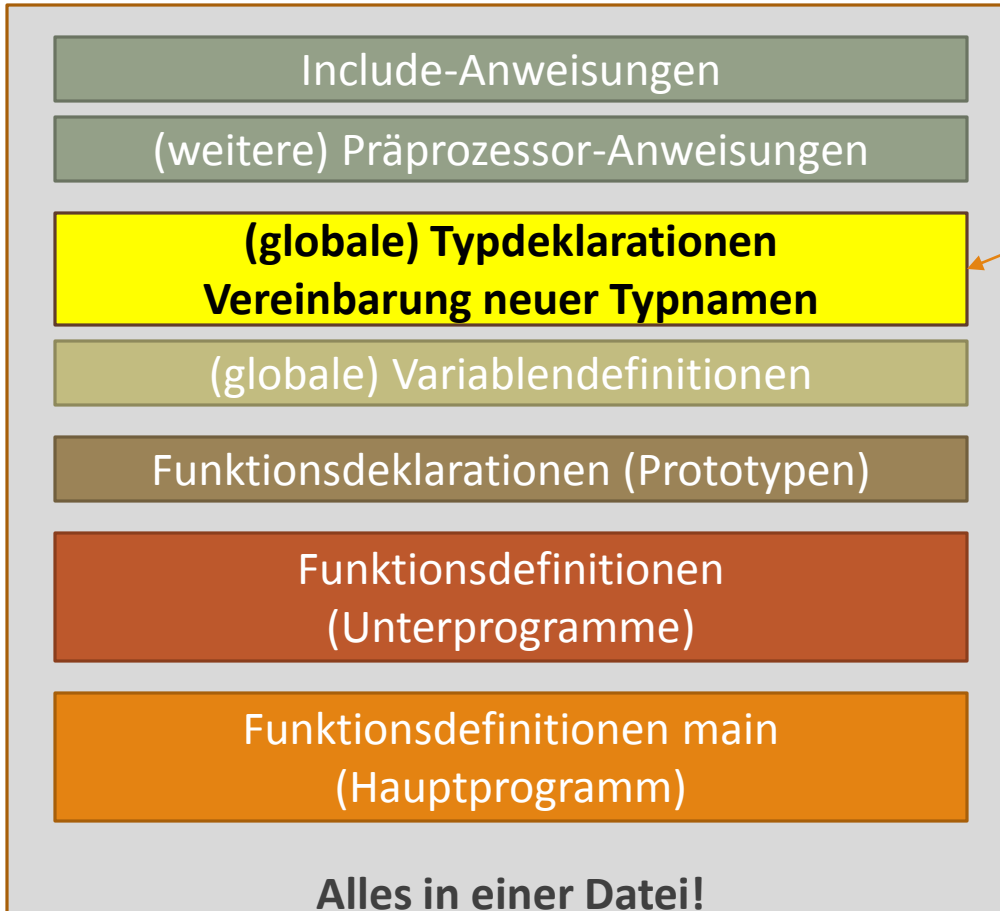
1. Felder und Zeichenketten
2. Standardein- und -ausgabe
3. Zeiger
4. Felder und Zeiger
5. **Benutzerdefinierte Typen**
6. Anwendungsbeispiel "Verkettete Listen"
7. Kommandozeilenparameter
8. Dateiein- und -ausgabe
9. Programmerzeugung und Präprozessor
10. "Beliebte" Fehler



# Benutzerdefinierte Datentypen

**Frage:** Wohin mit der Definition eigener Datentypen?

- Erinnerung: Abbildung zur Struktur eines einfachen C-Programms



**Antwort:**  
**Dorthin!**

# Verbünde (1)

**Definition:** Eine **Verbund** ist Datentyp, der eine **Ansammlung von Variablen** (auch unterschiedlichen Typs) unter einem einzigen Namen zusammenfasst.

- Diese Zusammenfassung wird in einem Speicherblock abgelegt.
- Insbesondere in den Programmiersprachen C und C++ werden Verbünde **auch als Strukturen** bezeichnet.
- Schlüsselwort in C (sowie auch in C++): **struct**

Syntax zur Typdeklaration: **struct** *<Typname>*  
{  
    *<Variablendeklarationen der Komponenten>*  
};

Beispiel:

```
// Deklaration
struct Point3D
{
    int id;
    double x, y, z;
};
```

```
// Variablendefinition
struct Point3D point1, pointm;
struct Point3D point2 = {1, 0.0, 0.0, 0.0};
struct Point3D pointlist[4];
struct Point3D* pointpointer;
```

# Verbünde (2)

Regeln (bei der Definition eines Verbunds):

- Der Verbund muss **mindestens eine Komponente** enthalten.
- Eine Komponente ist durch ihren **Datentyp und den Namen** festgelegt.
- Jede Komponente kann eine **einfache Variable**, ein **Zeiger**, ein **Feld** oder aber auch ein anderer **benutzerdefinierter Typ** wie z.B. ein anderer Verbund sein.
- Verbünde werden wie Variablen definiert und wie solche verwendet; können beispielsweise auch Rückgabewert einer Funktion sein.
- Der <Typname> ist der **Bezeichner** des Verbundes.
- `point1`, `point2` oder `pointm` (vorige Folie) bezeichnet man als **Verbundvariablen**.
- Alternativ hätten die Verbundvariablen auch direkt im Anschluss an die Verbunddeklaration definiert werden können:

```
// Deklaration mit gleichzeitiger Definition von Verbundvariablen
struct Point3D
{
    int id;
    double x, y, z;
} point1, point2, pointm;
```

# Verbünde (3)

Operatoren zum Zugriff auf einen Verbund bzw. auf dessen Komponenten:

- Zugriff auf den kompletten Verbund über die Verbundvariable: **varName**
- Zugriff auf eine Komponente des Verbunds über den **Punkt-Operator**:  
**varName.komponentenName**
- Wenn es sich bei der Verbundvariable um einen **Zeiger auf einen Verbund** handelt (im Beispiel: `pointpointer`), so erfolgt der Zugriff über den **Pfeil-Operator**:  
**varName-&gtkomponentenName**  
oder **alternativ mittels Dereferenzierung und Punktoperator**  
**(\*varName).komponentenName**

Zulässige Operatoren für Verbünde sind:

- **Übergabe** eines Verbunds an eine Funktion
- **Rückgabe** eines Verbunds aus einer Funktion
- **Zuweisung** zweier Verbünde gleichen Aufbaus → Kopieren erfolgt byteweise!
- **Anwendung** des `sizeof`-Operators → Frage: Ist diese Größe die Summe der Größen aller Komponenten?

**Vergleich zweier Verbünde: Nicht per Vergleichsoperatoren `==` oder `!=` möglich!**

- Stattdessen: **Vergleich muss komponentenweise erfolgen!**



# Verbünde (4)

Soweit bekannt: Verbünde stellen eine Möglichkeit dar, mehrere einfache Datentypen in einen zusammengesetzten Datentyp **zu vereinigen**.

Beispiel zur Deklaration und Verwendung:

- Deklaration eines `struct` erfolgt in der Regel außerhalb von Funktionen.

```
struct point
{
    int x;
    int y;
    double dist; /* distance from origin */
}; /* MUST have semicolon! */
```

- Erzeugung /  
Initialisierung:

```
struct point p;
p.x = 0; /* "dot syntax" */
p.y = 0;
p.dist = sqrt(p.x * p.x + p.y * p.y);
```

# Verbünde (5)

## Beispiel (Fortsetzung):

- Benutzung von structs

```
void foo()  
{  
    struct point p;  
    p.x = 10;  
    p.y = -3;  
    p.dist = sqrt(p.x * p.x + p.y * p.y);  
    /* do stuff with p */  
}
```

- dynamische Allokation von structs

```
struct point* make_point(void)  
{  
    struct point* p;  
    p = (struct point*) malloc( sizeof(struct point) );  
    return p;  
}  
/* free allocated memory for struct elsewhere */
```

# Verbünde (6)

## Beispiel (Fortsetzung):

- Zeiger auf structs:

```
void init_point( struct point* p )
{
    (*p).x = (*p).y = 0;
    (*p).dist = 0.0;
    // Alternative:
    // ("syntactic sugar")
    p->x = p->y = 0;
    p->dist = 0.0;
}
```

- Verbünde können Felder und andere structs enthalten:

```
struct foo
{
    int x[100];
    struct point p1;    /* Unusual */
    struct point* p2;   /* Typical */
};
```

# Verbünde (7)

## Felder von Verbünden

- lassen sich auf naheliegende Weise definieren
- Beispiel:

```
struct Person
{
    char vorname[30];
    char nachname[30];
} mieter[100];
struct Person studenten[10000];
```
- Es wird ein Feld mit 100 Elementen definiert, dessen einzelne Elemente Verbünde vom Typ `Person` sind.
- Anschließend wird ein weiteres Feld mit einer Größe von 10000 Elementen definiert.
- Zugriff auf einzelne die Komponenten (z.B. `vorname`) erfolgt durch eine Kombination von Feldzugriff und Verbundzugriff:

```
printf("Mieter 45 : %s\n", mieter[44].nachname );
printf("Student 88: %s, %s\n", studenten[87].nachname,
                                studenten[87].vorname );
```

# Vereinbarung neuer Typnamen (1)

**Idee:** Aus bestehenden Datentypen werden neue Datentypen erzeugt, mit einem eigenen (neuen) Namen versehen und später wie eingebaute Typen (Standarddatentypen in C) zur Definition von Variablen oder formalen Parametern verwendet.

Syntax: **typedef** *<alteTypbezeichnung>* *<neuerTypname>*;

Anwendungsbereiche:

- Verbesserung der **Lesbarkeit** großer Programme
- Verbesserung der **Portierbarkeit** eines Programms, denn man muss nur die abstrakten (neu definierten) Datentypen an die neue Umgebung anpassen  
➔ Aus diesem Grund befinden sich die selbstdefinierten Datentypen in der Regel in den Header-Dateien des Programms.

Beispiel:

```
typedef int ichNenneIntsAnders;  
struct _point3d {  
    int id;  
    double x, y, z;  
};  
typedef struct _point3d Point3D;
```

Diagramm zur Syntax:

- alter Typbezeichner (weist auf `struct _point3d` im letzten Zeilenabschnitt)
- neuer Typbezeichner (weist auf `Point3D` im letzten Zeilenabschnitt)

# Vereinbarung neuer Typnamen (2)

**Zusammenfassung:** Benutze **typedef**, um einen **Alias-Namen** zu definieren.

Originalname                      neuer Name

```
typedef struct _point3d Point3D;  
typedef int Length;
```

**Vorteile:**

- Die wiederkehrende Eingabe von `struct _point3d` wäre lästig.
- Rekursive `structs` sind möglich! Lesbarkeit des Codes wird bei Einführung neuer Namen erhöht.

- Beispiel:

```
// not so readable  
struct node  
{  
    int value;  
    struct node* next;  
};
```

```
// better readable  
struct _node  
{  
    int value;  
    struct _node* next;  
};  
typedef struct _node node;
```

# Vereinbarung neuer Typnamen (3)

Der alte Typ in einer typedef-Anweisung kann auch ein struct sein:

```
typedef struct /* Hier muss dem struct KEIN Name */  
{  
    int x;  
    int y;  
    double dist;  
} Point;  
Point p1, p2; /* no "struct" */
```

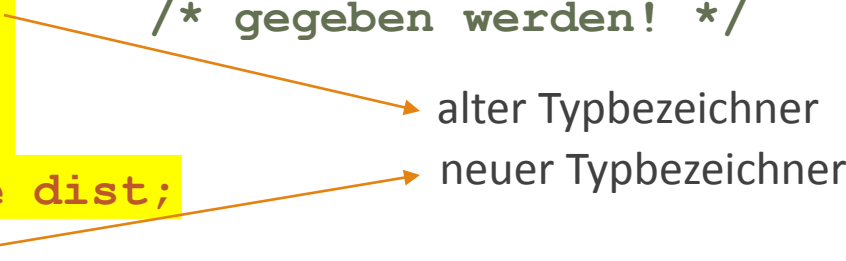


Diagram illustrating the mapping of the typedef statement:

- The word **struct** is identified as the **alter Typbezeichner** (old type designator).
- The word **Point** is identified as the **neuer Typbezeichner** (new type designator).

Anmerkung: Dies nennt man auch einen **anonymen Verbund** (anonyme Struktur).

Aber: Rekursive Definition erfordert eine explizite Namensangabe für den Verbund.

```
typedef struct _node  
{  
    int value;  
    struct _node* next;  
} node;
```




Diagram illustrating the recursive definition: The struct \_node\* next; line points to the \_node typedef, indicating a self-referencing pointer.

- Dieser Verbund ist nun nicht mehr anonym (sondern trägt den Namen `_node`)!

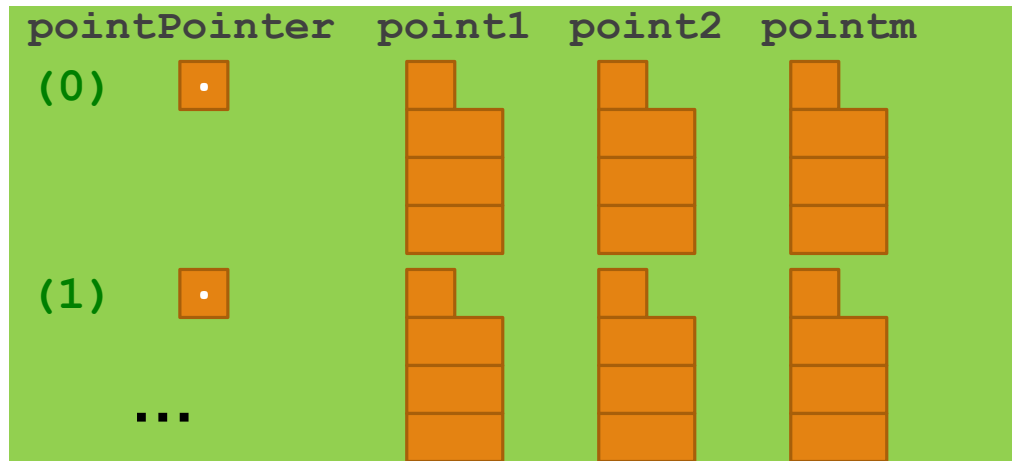
# Vereinbarung neuer Typnamen (4)

Paararbeit:

```
struct _point3d
{
    int id;
    double x, y, z;
};
typedef struct _point3d Point3D;
/* (0) */
Point3D point1, pointm,
        point2 = {1, 0.0, 0.0, 0.0};
/* (1) */
Point3D pointList[4];
Point3D* pointPointer;

point1.id = 2;
point1.x = 1.0;
point1.y = 2.0;
point1.z = 4.5;
/* (2) */
```

```
pointPointer = &pointm;
/* (3) */
(*pointPointer).id = 3;
// 3 Zugriffsarten:
(*pointPointer).x
    = (point1.x+point2.x) / 2.0;
pointPointer->y
    = (point1.y+point2.y) / 2.0;
pointm.z
    = (point1.z+point2.z) / 2.0;
/* (4) */
```



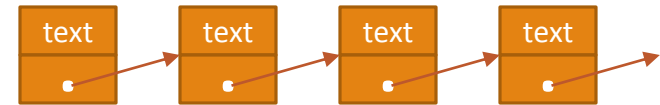


# Vereinbarung neuer Typnamen (5)

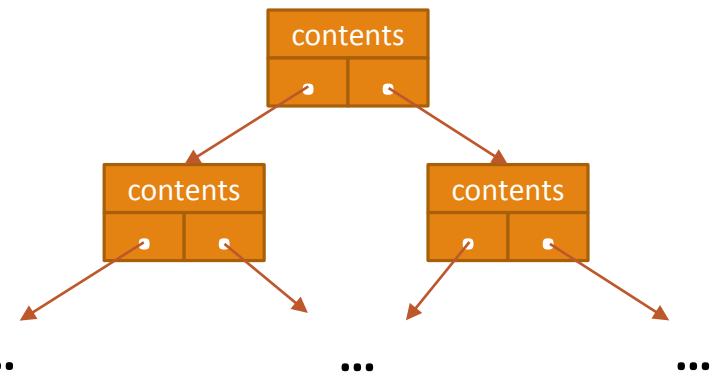
## Anwendung rekursiv definierter Verbünde

- zur Realisierung von „verketteten Listen“
- zur Realisierung von „Baumstrukturen“

```
struct _listElem
{
    char text[10];
    struct _listElem* next;
};
typedef struct _listElem ListElement;
```



```
struct _treeNode
{
    int contents;
    struct _treeNode* left;
    struct _treeNode* right;
};
typedef struct _treeNode TreeNode;
```



# Vereinbarung neuer Typnamen (6)

Verschachtelte Verbünde  
sind möglich

- Verbünde innerhalb von Verbünden
- mehrfaches Anwenden des  
Punkt-Operators zum Zugriff auf  
Komponenten tieferer Ebenen

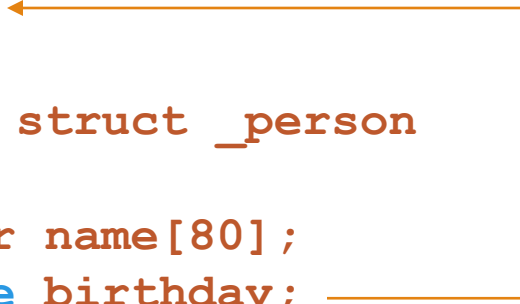
```
...
typedef struct _date
{
    unsigned int day, month, year;
} Date;

typedef struct _person
{
    char name[80];
    Date birthday;
} Person;

...
Person aFriendOfMine;

...
strcpy( aFriendOfMine.name, "Jan" );
aFriendOfMine.birthday.day = 1;
aFriendOfMine.birthday.month = 1;
aFriendOfMine.birthday.year = 1977;

...
```



# Aufzählungstypen (1)

**Definition: Aufzählungstypen** werden verwendet, wenn eine Variable nur wenige, ganz bestimmte Werte annehmen kann.

Syntax:

```
enum <TypName> { <bezeichner1>, <bezeichner2>, ..., <bezeichnerN> };
```

oder

```
enum <TypName> { <bez1>, ... , <bezN> } <var1>, ..., <varM>;
```

Vorteil:

- Wenn aussagekräftige Bezeichner verwendet werden, sind die Programme lesbarer und übersichtlicher.

Nachteil:

- Typfremde Zuweisungen an Aufzählungstypen durch den Compiler werden nicht erkannt.
- Der Wertebereich des neu definierten Aufzählungstyps muss relativ klein sein.

Beispiel:

```
enum days
{
    mon, tue, wed, thu, fri, sat, sun
} tag1;
enum days tag2;
```

# Aufzählungstypen (2)

## Regeln:

- Erstes Element der Aufzählung wird mit 0 gleichgesetzt.
- Für jedes folgende Element wird der assoziierte Zahlenwert um 1 erhöht.
- In der Definition kann einzelnen Element ein spezifischer Zahlenwert zugewiesen werden.

## Anmerkungen:

- Einsatz: sehr häufig, oftmals in Kombination mit **typedef** (vgl. Verbünde)
- Im Programm können nun die Abkürzungen verwendet werden.
- Der konkrete numerische Wert einer Variablen spielt meist keine Rolle.

## Beispiel:

- Diese Aufzählung ...

```
enum exampleEnumeration
{
    A, B, C, D=20, E, F, G=20, H
} var;
```



```
#define A 0
#define B 1
#define C 2
#define D 20
#define E 21
#define F 22
#define G 20
#define H 21
```

- ... ersetzt bzw. ist das Gleiche wie:

# Vereinigungsstruktur (Union)

**Definition:** Während in einem Verbund (`struct`) alle Elemente gleichzeitig gespeichert werden, kann eine **Vereinigungsstruktur** (`union`) nur eines ihrer Elemente halten.

Syntax: `union [Typname]`  
`{`  
    `<komponente1>;`  
    `...`  
    `<komponenteM>;`  
`} <varName1>, ..., <varNameN>;`

Vorteil:

- Einsparen von Speicherplatz bei der Verarbeitung großer Strukturen (z.B. bei Listen)
- maschinennahe Manipulation von Datentypen mit Operatoren

## 4. Effizientes Programmieren in C

1. Felder und Zeichenketten
2. Standardein- und -ausgabe
3. Zeiger
4. Felder und Zeiger
5. Benutzerdefinierte Typen
6. **Anwendungsbeispiel "Verkettete Listen"**
7. Kommandozeilenparameter
8. Dateiein- und -ausgabe
9. Programmerzeugung und Präprozessor
10. "Beliebte" Fehler

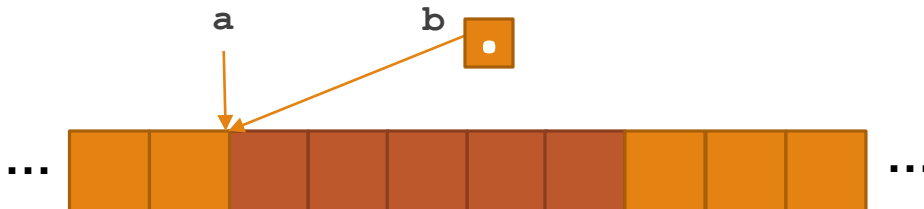


# Listen als Datenstruktur

## Grundsätzliche Unterscheidung in zwei Arten von Listen

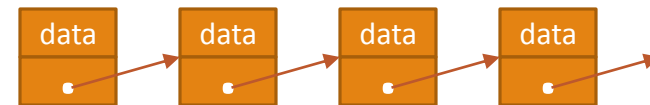
### Lineare Listen

- Realisierung als **statische oder dynamische eindimensionale Felder** (Vektoren)
- **Vorteile:**
  - einfache Struktur
  - einfacher Elementzugriff (wahlfreier Zugriff)
  - einfaches Löschen (komplett)
  - einfaches Sortieren
- **Nachteile:**
  - starre, nicht erweiterbare Struktur
  - Einfügen von Elementen ist teuer (höherer Aufwand)



### Verkettete Listen

- Verkettung der Listenelemente durch **zusätzliche Verkettungszeiger**
- **Nachteile:**
  - Struktur komplexer
  - Elementzugriff nur sequentiell
  - größerer Speicherbedarf durch Verkettungszeiger
  - Zugriffsfunktionen (Löschen, Suchen, etc.) sind komplizierter
- **Vorteile:**
  - erweiterbare Struktur (Relationen sind möglich)
  - sortiertes Einfügen ist billig (kein großer Aufwand)

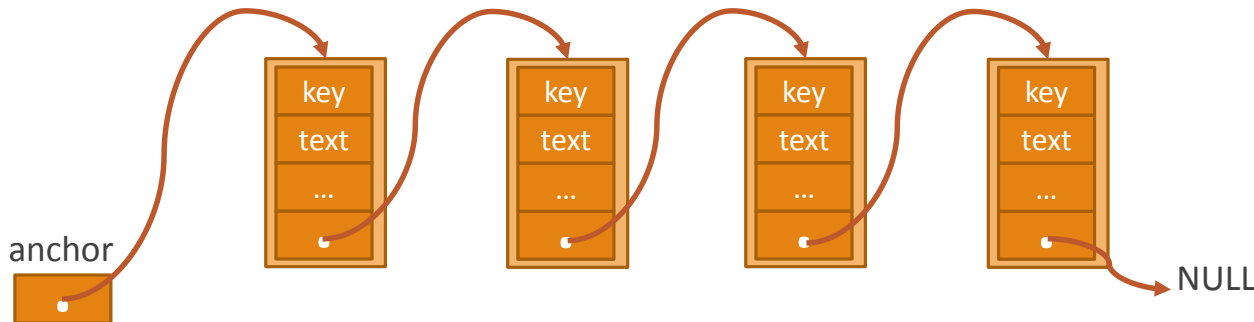


# Einfach verkettete Listen (1)

Datenstruktur (Beispiel):

```
struct _listElem
{
    unsigned int key; // Zugriffsschlüssel
    char text[10];    // beispielhafte Daten
    // ... weitere Daten
    struct _listElem* next; // Verkettungszeiger
};
typedef struct _listElem ListElement; // griffiger
                                      // Typbezeichner
ListElement* anchor; // „Anker“ der Liste
```

Speicherstruktur (Beispiel):





# Einfach verkettete Listen (2)

Welche **Zugriffsfunktionalitäten** benötigt man für das Arbeiten mit Listen?

- Initialisierung
- Aufbau
- Löschen
- Anzeigen (auf Elemente zugreifen)
- Element löschen/einfügen
- ...

Die Implementierung der entsprechenden Funktionalität ist **teilweise erheblich aufwändiger** als bei linearen Listen (eindimensionale Felder / Vektoren)!

- Daher: Am besten ist es, die zugehörigen Algorithmen in separate Funktionen zu kapseln.
- Dies erhöht die Modularität und Erweiterbarkeit des Programms!

# Einfach verkettete Listen (3)

## Szenario (Beispiel):

- Die verkettete Liste als Ganzes ist der Funktion zugeordnet, die den ursprünglichen Anker als Variable enthält (→ in unserem Beispiel: die `main()`-Funktion).
- Zugriffsfunktionen, die von dort aus aufgerufen werden, manipulieren die Liste.

```
struct _listElem
{
    // ... wie gehabt!
};
typedef struct _listElem ListElement; // griffiger
                                       // Typbezeichner

void main(void)
{
    ListElement* anchor = NULL; // hier ist die Liste "zu Hause"

    ... // hier: - direkte Manipulation der Liste ODER (besser)
           - Aufruf entsprechender Zugriffsfunktionen
}
```

# Einfach verkettete Listen (4)

## Listenaufbau:

- Ziel:

- Ansatz 1:

```
void main(void)
```

```
{
```

```
    ListElement* anchor = NULL;
```

```
    unsigned int i;
```

```
    for ( i=1; i<=4; i++ )
```

```
    {
```

```
        anchor = (ListElement*)malloc( sizeof(ListElement) );
```

```
        anchor->key = i;
```

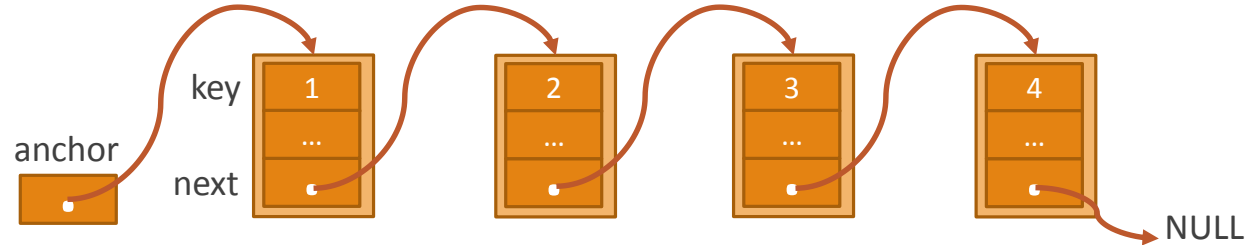
```
        anchor->next = NULL;
```

```
    }
```

```
}
```

- resultierende Speicherstruktur: siehe Tafel

- Fehlversuch



# Einfach verkettete Listen (5)

## Listenaufbau:

- Ziel: wie auf voriger Folie
- Ansatz 2: korrekter Aufbau, aber mit Sonderbehandlung des ersten Elements ☹

```
void main(void)
{
    unsigned int i;
    ListElement* anchor = (ListElement*)malloc( sizeof(ListElement) );
    anchor->key = 1;
    ListElement* help = anchor;
    ListElement* current = NULL;
    for ( i=2; i<=4; i++ )
    {
        current = (ListElement*)malloc( sizeof(ListElement) );
        current->key = i;
        help->next = current;
        help = current;
    }
    help->next = NULL;
}
```

# Einfach verkettete Listen (6)

## Listenaufbau:

- Ziel: wie zuvor
- Ansatz 3: korrekter Aufbau, elegantere Lösung, immer noch Sonderbehandlung

```
void main(void)
{
    unsigned int i;
    ListElement* anchor = (ListElement*)malloc(sizeof(ListElement));
    anchor->key = 1;
    ListElement* help = anchor;
    for ( i=2; i<=4; i++ )
    {
        help->next = (ListElement*)malloc( sizeof(ListElement) );
        help->next->key = i;
        help = help->next;
    }
    help->next = NULL;
}
```

# Einfach verkettete Listen (7)

## Listenaufbau:

- Ziel: wie zuvor
- Ansatz 4: rückwärtiger Aufbau, ohne Sonderbehandlung des ersten Elements

```
void main(void)
{
    ListElement* anchor = NULL;
    ListElement* help    = NULL;
    unsigned int i;
    for ( i=4; i>=1; i-- )
    {
        help = (ListElement*)malloc( sizeof(ListElement) );
        help->key = i;
        help->next = anchor;
        anchor = help;
    }
}
```

# Einfach verkettete Listen (8)

Liste anzeigen:

- ausgelagert in eine separate Funktion

```
void show( ListElement* listAnchor )
{
    // iterative Ausgabe der Liste, auf die anchor verweist
    while ( listAnchor != NULL )
    {
        printf("%u\n", listAnchor->key);
        listAnchor = listAnchor->next;
    }
}
```

- Aufruf aus `main()` heraus: `show( anchor );`

Weitere Funktionen:

- Liste löschen
- Element löschen
- Element einfügen

Weitere Aspekte:

- \* doppelt verkettete Listen → später (Alg&Dat)
- \* rekursive Funktionen

*Bemerkung:*

*Einfach verkettete Listen sind rekursive Datenstrukturen, denn jedes Listenelement enthält einen Zeiger auf eine Liste. → Der Einsatz rekursiver Algorithmen ist vorteilhaft.*

# 4. Effizientes Programmieren in C

1. Felder und Zeichenketten
2. Standardein- und -ausgabe
3. Zeiger
4. Felder und Zeiger
5. Benutzerdefinierte Typen
6. Anwendungsbeispiel “Verkettete Listen”
7. **Kommandozeilenparameter**
8. Dateiein- und -ausgabe
9. Programmerzeugung und Präprozessor
10. “Beliebte” Fehler





# Kommandozeilenparameter (1)

*Definition:* Unter **Kommandozeilenparametern** versteht man Argumente, die einem Programm beim Aufruf von der Systemebene mitgegeben werden können.

- Diese Parameter liefern zusätzliche Informationen über die Tätigkeit, die vom Programm ausgeführt werden soll.
- Beispiel: **cp test.txt alttest.txt**  
➔ Neben dem Programmnamen `cp` hat man noch zwei Parameter: `test.txt` und `alttest.txt`
- Anwendungszwecke (Beispiele):
  - Parameterübergabe bei Systemprogrammen
  - Auswählen von verschiedenen Optionen mit Hilfe von Schaltern
  - Datentransfer von der Aufrufebene in ein Programm hinein

Abgrenzung: Die Nutzung von Kommandozeilenparameter ist kein Ersatz oder keine Alternative zur GNU/Linux-spezifischen Umlenkung von Ein-/Ausgabedatenströmen.

- **myprog < inputfile > outputfile**
- `scanf` und `printf` können dann zur Ein- und Ausgabe genutzt werden

# Kommandozeilenparameter (2)

Dank der sehr engen Kopplung zwischen C und UNIX bzw. GNU/Linux ist die Definition und die Auswertung der Kommandozeilenparameter **sehr einfach**.

## Regeln:

- Ein Parameter ist eine **Zeichenkette**, die beim Programmaufruf hinter dem Programmnamen vorkommt.
- Mehrere Parameter werden **immer** durch ein **Leerzeichen** getrennt.
- Diese Parameter werden immer **an das aufgerufene Programm übergeben**.
- **Entgegengenommen** werden diese Parameter immer in der **Hauptfunktion `main()`**, deren Funktionskopf dafür aber die folgende Form annehmen muss:

```
int main( int argc, char* argv[] )
```

- Bedeutung der Parameter:
  - **int argc** steht für die Anzahl der Parameter, die dem Programm übergeben werden
  - **char\* argv[]** ist ein Feld von Zeichenketten, welches die einzelnen Parameter enthält

# Kommandozeilenparameter (3)

Beispiel: `#include <stdio.h>`  
`#include <stdlib.h>`

```
int main( int argc, char* argv[] )
{
    int first;
    double second;

    if (argc != 3) // Genau 2 Parameter sind erwünscht!
    {
        printf("Error!\n");
        exit(-1);    // Programmabbruch
    }                // nützliche Fkt. aus stdlib:
    first = atoi( argv[1] ); // Umwandlung ASCII->int
    second = atof( argv[2] ); // Umwandlung ASCII->float
    return 0;
}
```

# Kommandozeilenparameter (4)

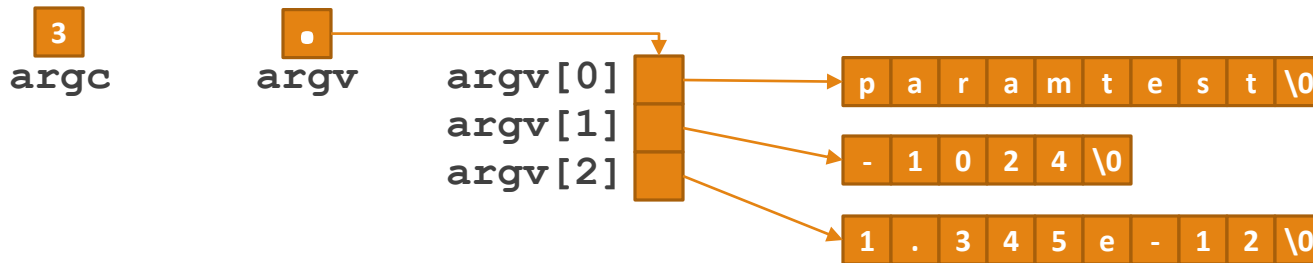
## Weitere Regeln:

- Wenn ein Programm aufgerufen wird, wird zunächst eine Initialisierungsroutine aufgerufen, welche die Kommandozeilenparameter für die Übergabe an `main()` aufbereitet.
- **`argv[0]`** enthält stets den **Programmnamen**
  - Im Beispiel der Aufrufes `cp test.txt alttest.txt` steht in `argv[0]` also "cp".
- **`argc==1`** bedeutet demzufolge, dass dem Programm **keine Parameter** mitgegeben wurden.
- Alle Parameter werden in Form von Zeichenketten an das Programm übergeben.
- Bei der Benutzung von Zahlen als Kommandozeilenparameter muss also eine geeignete **Konvertierung** vorgenommen werden.
  - mit Hilfe von Funktionen aus `stdlib.h` möglich.
  - „ASCII to Float“ = „`atof`“ → Umwandlung einer Zeichenkette in eine **Fließkommazahl**  
**`float atof( const char* s );`**
  - „ASCII to Integer“ = „`atoi`“ → Umwandlung einer Zeichenkette in eine **Ganzzahl**  
**`int atoi( const char* s );`**

# Kommandozeilenparameter (5)

Beispiel:

- der Programmname sei **paramtest**
- der Programmaufruf: **paramtest -1024 1.345e-12**
  - also zwei übergebene Parameter
- resultierende Speicherstruktur nach Programmaufruf



# Kommandozeilenparameter (6)

Beispiel:

```
#include <string.h>
int main(int argc, char* argv[])
{
    int i;
    /* process command-line arguments */
    for (i = 0, i < argc; i++)
    {
        if (strcmp(argv[i], "-b") == 0)
        {
            /* whatever... */
        }
    }
    /* ... rest of program ... */
}
```

Kommandozeilenparameter sind `argv[0]`, `argv[1]`, ...

- `argv[0]` ist der Programmname → Beispiel: `command_line_options.c`
- `strcmp()`: Funktion vergleicht Zeichenketten (bei Gleichheit ist der Rückgabewert 0)
- in `<string.h>` definiert

# 4. Effizientes Programmieren in C

1. Felder und Zeichenketten
2. Standardein- und -ausgabe
3. Zeiger
4. Felder und Zeiger
5. Benutzerdefinierte Typen
6. Anwendungsbeispiel “Verkettete Listen”
7. Kommandozeilenparameter
8. **Datei- und -ausgabe**
9. Programmerzeugung und Präprozessor
10. “Beliebte” Fehler



**Regel:** Der Zugriff auf die Dateien ist **zeichenorientiert**, d.h. es wird immer eine Folge von Einzelzeichen eingelesen.

- Es wird nicht zwischen Dateitypen unterschieden.
- Alle Hardware-Komponenten werden ebenfalls als Dateien aufgefasst.
- Wir werden auch spezialisierte Funktionen zum Dateizugriff kennenlernen, die einer Datei bestimmte strukturierte Voraussetzungen unterstellen.

Unterscheidung des Abstraktionsgrads der Dateiein- und -ausgabe:

- **A. High-Level-I/O**
  - Daten in der Datei werden als zeilenweise strukturierte Textdatei angesehen.
  - Unter Angabe des Standarddatentyps kann man auf die Datei zugreifen.
- **B. Typisierte Dateien**
  - Daten (in der Datei) sind eine Folge identisch strukturierter Datensätze
  - Zugriff erfolgt unter Angabe der Strukturgröße und der Anzahl der Strukturen
- **C. Low-Level-I/O**
  - Daten als vollkommen unformatierter Bytestrom
  - Zugriff unter Angabe der zu speichernden/lesenden Bytes



# High-Level-I/O (1)

- A. High-Level-I/O
- B. Typisierte Dateien
- C. Low-Level-I/O

Zugriff auf Dateien erfolgt nach dem folgenden grundsätzlichen **Schema**:

## 1. Öffnen der Datei

- Es erfolgt eine Verbindung zwischen der externen („realen“) Datei (auf dem Speichermedium) und dem Programm.

## 2. Lesender und/oder schreibender Zugriff auf die geöffnete Datei

## 3. Schließen der Datei

- Datei wird dadurch wieder in einen definierten Anfangszustand gebracht.

## Öffnen von Dateien mit `fopen`

- Syntax: **`FILE* fopen( char* fname, char* mode)`**
- Mit diesem Befehl wird eine Datei, die durch den Namen `fname` bezeichnet wird, zur Bearbeitung geöffnet.
- Für weitere Operationen auf der Datei wird ein `FILE`-Zeiger auf die Datei zurückgeliefert.
- Wenn beim Öffnen ein Fehler auftrat, wird `NULL` zurückgeliefert.
- Die Zeichenkette `mode` umfasst bis zu 2 Zeichen & bestimmt, wie die Datei geöffnet wird:
  - **"r"** → zum Lesen (read)
  - **"w"** → zum Schreiben (write)
  - **"a"** → zum Anhängen (append)
  - 1. Zeichen
  - 2. Zeichen
  - **"b"** → Binärdatei (binary)
  - **"t"** → Textdatei (text)

# High-Level-I/O (2)

- A. High-Level-I/O
- B. Typisierte Dateien
- C. Low-Level-I/O

## Schließen von Dateien mit `fclose`

- Nachdem eine Datei bearbeitet wurde, sollte sie mit der Funktion `fclose()` geschlossen werden.
- Syntax: `int fclose( FILE* fp )`
- schließt den Datenstrom für die Datei, die mit dem `FILE`-Zeiger `fp` bezeichnet wird
- Für Dateien mit schreibendem Zugriff wird vorher noch der **Puffer** geschrieben.
- Bei einem Fehler ist der Rückgabewert `EOF`, ansonsten `0`.
- Bei Programmende werden automatisch alle Dateien geschlossen.
- Die maximale Anzahl (gleichzeitig) offener Dateien wird durch das Betriebssystem vorgegeben.

# Zwischenschub: Erinnerung

- A. High-Level-I/O
- B. Typisierte Dateien
- C. Low-Level-I/O

## Erinnerung: Zeichenorientierte Ein- und Ausgabe auf den Bildschirm

- Syntax:

```
#include <stdio.h>
int putchar( int c );
int getchar();
```

- Die Funktion `putchar()` dient zur unformatierten Ausgabe eines einzelnen Zeichens auf dem Bildschirm.
- Sie arbeitet gepuffert, wobei die Pufferung zeilenorientiert erfolgt.
- Die Funktion `getchar()` dient zum Einlesen einzelner Zeichen über die Tastatur.
- Die Funktion benutzt ebenfalls eine zeilenweise Pufferung.
- Man kann also nicht ein einzelnes Zeichen (ohne Betätigung von Enter) einlesen.
  - Abhilfe: Funktionen wie `getkey`, `getch` o.ä. nutzen

## Zwei Arten des Schreibens in und Lesens aus Dateien

1. zeichenorientiertes (unformatiertes) Schreiben in / Lesen aus Dateien
2. formatiertes Schreiben in / Lesen aus Dateien

### 1. Zeichenorientiertes Schreiben in Dateien

- Idee: Einzelne Zeichen werden in die Datei **geschrieben**.
- Syntax: **`int fputc( int c, FILE* fp )`**
  - funktioniert wie `putchar`
  - Anstatt auf `stdout` wird aber in die Datei, der `fp` zugeordnet ist, geschrieben.
  - Also: `putchar` ist `fputc` auf `stdout`!
- Die Funktion `fputc()` schreibt das Zeichen `c` in die Datei `fp`.
- Rückgabewert ist das geschriebene Zeichen selbst oder `EOF`, wenn ein Fehler auftrat.
- Syntax: **`int fputs( char* string, FILE* fp )`**
  - funktioniert ähnlich wie `puts`
  - Anstatt auf `stdout` wird in die Datei, der `fp` zugeordnet ist, geschrieben.
  - Ein Endzeichen `'\0'` wird nicht geschrieben, aber auch das Zeilenendzeichen `'\n'` nicht.
  - Also: `puts` ist `fputs` auf `stdout` plus Zeilenendzeichen!

## 1. Zeichenorientiertes Lesen aus Dateien (Fortsetzung)

- Idee: Einzelne Zeichen werden aus der Datei **gelesen**.
- Syntax: **`int fgetc( FILE* fp )`**
  - funktioniert wie `getchar`
  - Anstatt von `stdin` wird aus der Datei, der `fp` zugeordnet ist, gelesen.
  - Also: `getchar` ist `fgetc` auf `stdin`.
- Die Funktion `fgetc()` liest ein Zeichen aus der Datei `fp`.
- Der Rückgabewert ist das gelesene Zeichen oder `EOF`, wenn das Dateiende erreicht wurde oder aber ein Fehler auftrat.
- Syntax: **`char* fgets( char* string, int n, FILE* fp )`**
  - funktioniert ähnlich wie `gets`
  - Anstatt von `stdin` wird aus der Datei, der `fp` zugeordnet ist, gelesen.
  - liest höchstens `n-1` Zeichen
  - Das Zeilenendezeichen `'\n'` wird mit in die Zeichenkette `string` aufgenommen; das Endezeichen `'\0'` wird angehängt.
  - Also: `gets` ist ähnlich wie `fgets` auf `stdin`.
  - **Vorteil:** Im Gegensatz zu `scanf()` lassen sich mit `fgets` (sowie auch `gets`) Zeichenketten einlesen, die auch Leerzeichen (oder andere Whitespaces) enthalten! An das Ende der Eingabe wird `'\0'` angehängt.

# High-Level-I/O (5)

- A. High-Level-I/O
- B. Typisierte Dateien
- C. Low-Level-I/O

Beispiel:

```
#include <stdio.h>
int main(void)
{
    char text[256];
    printf("Die zu durchsuchende Zeichenk. eing. (max. 255 Zeichen):\n");
    fgets( text, 256, stdin);
    printf("Eingelesen wurde: %s\n", text);
    char suchtext[32];
    printf("Die zu suchende Zeichenkette eingeben (max. 31 Zeichen):\n");
    fgets( suchtext, 32, stdin);

    // Etwas Sinnvolles tun ...
    // ... z.B. überprüfen, ob der Suchtext im Text enthalten ist.

    return(0);
}
```

## 2. Formatiertes Schreiben in und Lesen aus Dateien

- Idee: Die hierfür zur Verfügung stehenden Funktionen ähneln stark den bekannten Bildschirm-ein-/ausgabefunktionen `scanf` und `printf` mit dem Unterschied, dass ein zusätzlicher Parameter `FILE* fp` vorhanden ist.
- Rückgabewert sind die gelesenen bzw. geschriebenen Zeichen oder -1, wenn ein Fehler aufgetreten ist.
- Syntax: **`int fprintf( FILE* fp, char* format [, arg] ... )`**
  - funktioniert wie `printf`
  - Anstatt auf `stdout` wird in die Datei, der `fp` zugeordnet ist, geschrieben.
  - Also: `printf` ist `fprintf` auf `stdout`!
- Syntax: **`int fscanf( FILE* fp, char* format [, arg] ... )`**
  - funktioniert wie `scanf`
  - Anstatt von `stdin` wird aus der Datei, der `fp` zugeordnet ist, gelesen.
  - Also: `scanf` ist sozusagen `fscanf` auf `stdin`.

# High-Level-I/O (7)

- A. High-Level-I/O
- B. Typisierte Dateien
- C. Low-Level-I/O

Beispiel: Einlesen und Anzeigen einer Datei ganzer Zahlen

```
#include <stdio.h>
#include <stdlib.h>
int main( void )
{
    FILE* fp;
    int help; //ganze Zahlen
    fp = fopen("test.dat", "r"); //Datei öffnen
    if ( fp == NULL ) //Überprüfung, ob das Öffnen geklappt hat
    {
        printf( "Error opening file %s\n", "test.dat");
        exit(-1);
    }
    while ( fscanf(fp, "%d", &help) != EOF ) //Dateiende erreicht
        printf( "%d\n", help );
    fclose( fp ); //Datei schließen
    return 0;
}
```

test.dat

-3 7 1024  
4 0  
1 -4

Frage: Welche Ausgabe erfolgt auf dem Bildschirm?



# High-Level-I/O (8)

- A. High-Level-I/O
- B. Typisierte Dateien
- C. Low-Level-I/O

## Weitere wichtige Funktionen für die High-Level-I/O

- Entleeren des Puffers, der dem Dateizeiger `fp` zugeordnet ist
  - Syntax: `int fflush( FILE* fp )`
- Dateizeiger auf den Anfang der Datei zurücksetzen (rewind)
  - Syntax: `int rewind( FILE* fp )`
- wahlfreie Positionierung des Dateizeigers `fp` in der Datei
  - Syntax: `int fseek( FILE* fp, int offset, int origin )`
  - `origin` gibt die Stelle in der Datei an, `offset` die Entfernung von `origin`
  - Vordefinierte Konstanten sind `SEEK_SET` (Dateianfang), `SEEK_CUR` (aktuelle Position) und `SEEK_END` (Dateiende).
  - Beispiel: `fseek( fp, -200L, SEEK_END )` → 200 Zeichen vom Ende zurück (L wegen `long int`)
- Ermittlung der Positionierung des Dateizeigers innerhalb der Datei
  - Syntax: `int ftell( FILE* fp )`

# Sonderfall: Ein-/Ausgabe in Zeichenketten

Ähnlich wie die High-Level-I/O funktioniert das Schreiben und Lesen in/aus Zeichenketten.

Syntax: **int sprintf( char\* str, char\* format [, arg] ... )**

- funktioniert wie `printf` bzw. `fprintf`
- Ausgabe erfolgt aber nicht auf Bildschirm oder in eine Datei, sondern in die durch `str` spezifizierte Zeichenkette
- Ausgabe wird mit `'\0'` abgeschlossen
- `str` muss auf einen Speicherbereich mit ausreichender Größe verweisen
- Rückgabe ist Anzahl ausgegebener Zeichen (exklusive der abschließenden `'\0'`)

Syntax: **int sscanf( char\* str, char\* format [, arg] ... )**

- funktioniert wie `scanf` und `fscanf`
- Eingabe wird aus der Zeichenkette `str` gelesen

# Typisierte Dateien (1)

- A. High-Level-I/O
- B. Typisierte Dateien**
- C. Low-Level-I/O

**Regel:** Die zu dieser Gruppe gehörenden Funktionen betrachten die Datei als eine Folge von **identisch strukturierten Datensätzen**.

- z.B. von Verbünden, Unions, Felder oder Zusammenfassungen davon

Funktion zum Schreiben von strukturierten Daten: **fwrite**

- Syntax: **size\_t fwrite( void\* buf, int size, int cnt, FILE\* fp )**
- `fwrite` schreibt `cnt` Elemente jeweils der Größe `size`, die ab der Speicheradresse `buf` stehen, in die Datei, auf die der Dateizeiger `fp` zeigt.
- Rückgabewert ist die Anzahl der tatsächlich geschriebenen Datensätze.
- Im Fehlerfall ist der Rückgabewert kleiner als `cnt`.

Funktion zum Lesen von strukturierten Daten: **fread**

- Syntax: **size\_t fread( void\* buf, int size, int cnt, FILE\* fp )**
- `fread` liest aus der Datei, auf die der Dateizeiger `fp` zeigt, höchstens `cnt` Objekte jeweils der Größe `size` und speichert sie im Speicherbereich ab, der ab Stelle `buf` beginnt.
- Rückgabewert ist die Anzahl der eingelesenen Elemente. Im Fehlerfall oder bei vorzeitigem Dateiende ist der Rückgabewert kleiner als `cnt`.
- **Wichtig:** `buf` muss auf einen ausreichend großen Speicherbereich zeigen!

# Typisierte Dateien (2)

- A. High-Level-I/O
- B. Typisierte Dateien**
- C. Low-Level-I/O

## Bemerkungen:

- Hauptanwendungsfall typisierter Dateien ist das effiziente Schreiben und Lesen von zusammengesetzten Datentypen (Verbünde etc.).
- Schreiben und Lesen im Binärformat ist möglich.
- **Warnung:** Achtung ist geboten bei Zeigern, die Bestandteile eines zusammengesetzten Datentyps (z.B. eines `structs`) sind; diese lassen sich nicht einfach so „wegspeichern“, später wieder laden und dann problemlos weiterverwenden.

Beispiel:

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

typedef struct _point3Dstruct
{
    double x, y, z;
    unsigned int cindex;
} point3D;

int main( void )
{
    FILE* fp;
    ...
```

# Typisierte Dateien (3)

- A. High-Level-I/O
- B. Typisierte Dateien**
- C. Low-Level-I/O

Beispiel (Forts.):

```
int main( void )
{
    FILE* fp;
    int i;
    point3D help, data[] = { {2.3, 1.6, -0.6, 4},
                             {-12.1, -1, 0.0, 512},
                             {14.1, 2.2, 1.4, 0} };

    fp = fopen("points.dat", "w"); //Datei zum Schreiben öffnen
    if ( fp == NULL ) //Überprüfung, ob das Öffnen geklappt hat
    {
        printf( "Error opening file %s\n", "points.dat");
        exit(-1);
    }
    if ( fwrite((void*)data, sizeof(point3D), 3, fp) != 3 )
    {
        printf( "Error writing file %s\n", "points.dat");
        exit(-2);
    }
    fclose( fp );
    ...
}
```

# Typisierte Dateien (4)

- A. High-Level-I/O
- B. Typisierte Dateien**
- C. Low-Level-I/O

Beispiel (Forts.):

```
...
if ( fwrite((void*)data, sizeof(point3D), 3, fp) != 3 )
{
    printf( "Error writing file %s\n", "points.dat");
    exit(-2);
}
fclose( fp );

fp = fopen("points.dat", "r"); //Datei zum Lesen öffnen

while ( fread( (void*)&help, sizeof(point3D), 1, fp) == 1 )
    printf( "%lf %lf %lf %d\n",
            help.x, help.y, help.z, help.cindex );

fclose( fp ); //Datei schließen
return 0;
}
```

# Low-Level-I/O (1)

- A. High-Level-I/O
- B. Typisierte Dateien
- C. Low-Level-I/O

## Eigenschaften:

- Die Dateizugriffsfunktionen dieser Gruppe bilden die **Grundlage für alle anderen Dateifunktionen**.
- Sie betrachten jede Art von Dateien als eine **unformatierte Folge von Bytes** und arbeiten ungepuffert.
- Sie dienen im Wesentlichen zum Ein- und Auslesen von **Byte-Sequenzen**.
- Die kennengelernten High-Level-Funktionen sprechen die Datei über einen Zeiger vom Typ `FILE*` an, die Low-Level-Funktionen benutzen dafür **sogenannte Handle**, die vom Typ `int` sind.
- Alle Low-Level-Funktionen sind deklariert in bzw. einzubinden via:  

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
```

# Low-Level-I/O (2)

- A. High-Level-I/O
- B. Typisierte Dateien
- C. Low-Level-I/O

## Öffnen einer Datei

- Syntax: `int open( char* name, int mode )`
- öffnet die mit `name` bezeichnete Datei
- `mode` gibt an, welche Operationen auf der Datei durchgeführt werden können
  - `O_RDONLY` → Datei zum Lesen öffnen
  - `O_WRONLY` → Datei zum Schreiben öffnen
  - `O_CREATE` → Datei wird neu erstellt
  - `O_TRUNC` → Wenn Datei existiert, wird sie geleert.
- Rückgabewert ist ein Datei-Handle (`int`) oder `-1` bei Fehlern

## Schreiben in und Lesen aus einer Datei

- Syntax:  
`int read( int handle, void* buf, unsigned int len)`  
`int write( int handle, void* buf, unsigned int len)`
- Lesen/Schreiben von `len` Bytes aus der bzw. in die durch `handle` gekennzeichnete Datei
- Lese-/Schreibvorgang beginnt an der durch den Zeiger `buf` definierten Stelle im Hauptspeicher
- Rückgabewert:
  - `-1` bei Fehlern, `0` bei Dateiende (`EOF`)
  - sonst: die Anzahl der gelesenen bzw. geschriebenen Zeichen



# Low-Level-I/O (3)

- A. High-Level-I/O
- B. Typisierte Dateien
- C. Low-Level-I/O

## Weitere grundlegende Low-Level-I/O-Funktionen

- Syntax:  

```
long lseek ( int handle, long offset, int pos)
int unlink( char* name )
int close( handle );
```
- `lseek` dient zum Bewegen des Dateizeigers in der geöffneten Datei
  - Er wird um `offset` Bytes, beginnend bei der durch `pos` angegebenen Position bewegt
  - `pos` kann die Konstanten `SEEK_SET`, `SEEK_CUR` und `SEEK_END` annehmen
  - Rückgabewert ist der Offset der neuen Position des Dateizeigers und 0 im Fehlerfall
- `close` schließt die Datei
  - Rückgabewert ist -1 bei Fehlern, sonst 0
- `unlink` löscht die angegebene Datei
  - Rückgabewert ist -1 bei Fehlern, sonst 0

# Low-Level-I/O (4)

- A. High-Level-I/O
- B. Typisierte Dateien
- C. Low-Level-I/O

## Weitere, betriebssystemnahe Low-Level-I/O-Funktionen

- **access()** : ermittelt die Zugriffsmöglichkeiten auf eine Datei
- **chmod()** : setzt die Zugriffsrechte einer Datei
- **creatnew()** : erzeugt und öffnet eine neue Datei zum Lesen und Schreiben im Binärmodus
- **creattmp()** : erzeugt eine temporäre Datei, deren Name eindeutig ist
- **dup()** : verdoppelt ein Datei-Handle
- **filelength()** : ermittelt die Größe einer Datei in Bytes
- **getftime()** : ermittelt Datum und Zeit einer Datei
- **ioctl()** : dient der direkten Steuerung von Peripheriegeräten
- **isatty()** : prüft den Gerätetyp
- **remove()** : löscht eine Datei
- **rename()** : ändert den Namen einer Datei

# 4. Effizientes Programmieren in C

1. Felder und Zeichenketten
2. Standardein- und -ausgabe
3. Zeiger
4. Felder und Zeiger
5. Benutzerdefinierte Typen
6. Anwendungsbeispiel “Verkettete Listen”
7. Kommandozeilenparameter
8. Dateiein- und -ausgabe
9. **Programmerzeugung und Präprozessor**
10. “Beliebte” Fehler



# Programmübersetzung (1)

## Die **Einschritt-Variante**

- **nur** für kleine Programme (die nur aus einer einzelnen Datei bestehen) geeignet

- Quellcode (editierbar) → test.c



kompilieren  
und linken

```
gcc test.c -pedantic -g -lm -o test
```

- Programm-Code (lauffähig) → test

*Alle blauen Optionen  
sind beispielhaft!*

## Die **Zweischritt-Variante**

- auch für Programme mit mehreren Quellcodedateien verwendbar

- Quellcode (editierbar) → test.c



kompilieren

```
gcc -c test.c -pedantic -g test.o
```

- Object-Code (ähnlich Bibliotheken) → test.o



linken

```
gcc test.o -lm -o test
```

- Programm-Code (lauffähig) → test

# Programmübersetzung (2)

Programmübersetzung bei komplexen Programmen mit vielen Quellkodedateien

Prinzip der **Dateinamenvergabe**:

- Partitioniere alle Funktionen entsprechend ihrer Funktionalität in Dateien,  
→ erhalten **Endung .c**
- Lagere alle Deklarationen von Objekten/Funktionen, die in mehreren .c-Dateien benutzt werden, in sogenannte Header-Dateien aus  
→ erhalten **Endung .h**

Prinzip der Programmübersetzung

- Kompiliere alle .c-Dateien (vgl. Zweischritt-Variante, Schritt 1)  
→ mehrere Objektdaten mit **Endung .o** entstehen
- Binde alle Objektdaten zusammen mithilfe des Linkers (vgl. Zweischritt-Variante, Schritt 2)  
→ ein **ausführbares Programm** entsteht
- Berücksichtige bei zukünftigen Übersetzungsvorgängen (zeitliche) Abhängigkeiten



Nutzung des Werkzeuges **make** für diese Zwecke (später)

# Präprozessor (1)

Erinnerung: Frage: Was macht folgende Zeile

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

Vor dem Übersetzen des Programms wird der Präprozessor aktiv und behandelt alle Zeilen startend mit #

- hier: Einfügen von Funktionsdefinitionen aus einer Bibliothek (Standardein-/ausgabe)
- Nicht die Implementierung der Funktion wird eingefügt!
- konkret: Einbindung von `stdio.h` ermöglicht, die Funktion `printf()` (sowie diverse weitere dort definierte Funktionen) zu benutzen
- Beispiel: 

```
#include <filename.h> //aus Standardverzeichnissen  
#include "filename.h" //zuerst im aktuellen Verzeich-  
//nis (Arbeitsverzeichnis), dann  
//in Standardverzeichnissen
```

Funktionen des Präprozessors:

- textuelles Einfügen von Programmdateien → sh. oben via `#include`
- Definition von Markos (im einfachsten Fall: Konstanten)
- bedingte Übersetzung

# Präprozessor (2)

## Makrovereinbarung: Konstantenvereinbarung

- Syntax: **#define NAME constterm**
- `#define` wird sehr häufig benutzt, meistens um symbolische Konstanten zu definieren.
- Der Präprozessor substituiert den String „100“ für alle auftretenden `MAX_LENGTH`
- Achtung: Es erfolgt ein textuelles Ersetzen aller im Programm folgenden Bezeichner mit Name **NAME** durch **constterm**
- Beispiel: **#define MAX\_LENGTH 100**

Beispiel:

```
/* later... */

int i;

/* later... */

if (i > MAX_LENGTH) {
    printf("Whoa there!\n");
}

/* That code expands into: */
if (i > 100)
{
    printf("Whoa there!\n");
}
```

# Präprozessor (3)

Achtung: **Reine Textersetzung** → keine Typüberprüfung (type-checking) erfolgt!

**Ergebnis: Alle** Aufkommen von `MAX_LENGTH` werden mit 100 ersetzt.

Frage: Warum nicht 100 direkt schreiben?

- Änderungen müssen nun nur an einer Stelle des Programms gemacht werden.
- Fest kodierte Werte werden als „magic numbers“ bezeichnet.
- wiederholen sich häufig im Programm
- müssen an vielen Zeilen des Programms geändert werden

Wir ermöglichen damit:

```
#define SOME_CONSTANT 100
```

Bessere Alternative (kennen wir bereits):

```
const int SOME_CONSTANT = 100;
```

**Frage:** Warum ist das besser? → **Antwort:** Wegen Typsicherheit (type checking).



# Präprozessor (4)

## Makrovereinbarung

- Syntax: **#define MACRONAME (parameterliste) macrorumpf**
- **MACRONAME** verwendet den oder die Parameter; auch hier erfolgt komplette textuelle Ersetzung.
- Konvention: `#define ALL_CAPITAL_LETTERS` wird stets mit Grossbuchstaben geschrieben (anders als bei Variablen)
- Zeilenumbrüche sind erlaubt und müssen mit `\` gekennzeichnet werden.

Beispiel 1: **#define CIRCLEAREA(r) 3.1415\*r\*r**  
...  
**float radius = 1.5;**  
**printf("Area of circle with radius %f is %f.\n",**  
**radius, CIRCLEAREA(radius) );**  
...

Beispiel 2: **#define MAX(a, b) \**  
**((a) > (b)) ? (a) : (b))**

# Präprozessor (5)

## Bedingte Übersetzung des Programmes

- Unter bedingter Übersetzung versteht man die Fähigkeit, nur bestimmte Teile des Programms zu kompilieren.
- Die Entscheidung, ob ein folgender Teil mit zu kompilieren ist, ist dabei von einer Bedingung abhängig, die zur Übersetzungszeit ausgewertet wird.
- Syntax:

<b>#ifdef macroname</b>	<b>#if bedingungsterm</b>
...	...
<b>#else //optionaler Zweig</b>	<b>#else</b>
...	...
<b>#endif</b>	<b>#endif</b>
- In Variante 1 (links) prüft der Compiler zunächst, ob in dieser Datei oder in einer per `include` eingebundenen Datei ein Makro mit Namen `macroname` definiert wurde.
- In Variante 2 (rechts) prüft der Compiler, ob der Bedingungsterm wahr ist.

<b>#ifdef WINDOWS</b>
<b>#include &lt;windows.h&gt;</b>
<b>#else</b>
<b>#include &lt;X11/X.h&gt;</b>
<b>#endif</b>
- Der `#else`-Zweig ist optional, das abschließende `#endif` ist obligatorisch.
- Beispiel: (rechts)

# Präprozessor (6)

## Bedingte Übersetzung des Programmes

- Manchmal besteht der Wunsch, Code nur unter bestimmten Bedingungen zu übersetzen.

Beispiel:

- Plattformabhängiges Übersetzen (sich je nach OS unterscheidende Code-Bereiche)
- Übersetzen zu Debug-Zwecken

```
#define DEBUG
    int value = 10;
#ifdef DEBUG
    printf("value = %d\n", value);
#endif
```

Entscheidung wird zur Kompilation gefällt

```
% gcc -DDEBUG foo.c -o foo
```

-DDEBUG heißt `#define DEBUG`

# Präprozessor (7)

## Bedingte Übersetzung des Programmes

- `#if` testet Integer-Variablen, z.B. Revisions- und Versionsnummern

```
#if REVISION == 1
/* revision 1 code */
#elif REVISION == 2
/* revision 2 code */
#else
/* generic code */
#endif
```

- Möglichkeit: Benutze `#if 0`, um große Code-Blöcke auszukommentieren

```
#if 0
/* This doesn't get compiled. */
#endif
```

➔ Interessante Möglichkeit, da `/* ... */` nicht geschachtelt werden können

# Präprozessor (8)

## Beispiel:

- zu Konstanten  
und bedingter Übersetzung

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

#define n 10*10
#define m n*10

int main(void)
{
    printf("%d %d\n", n, m);
    #if 0
        printf("%d\n", n);
        ...
    #endif
    ...
    #ifdef DEBUG
        ...
    #else
        ...
    #endif
    return 0;
}
```

# Präprozessor (9)

Beispiel:

- zu Makros

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

#define SWAP(x,y) x=x^y; \
                  y=x^y; \
                  x=x^y;

int main(void)
{
    int a, b;
    char c='a', d='c';

    SWAP(a,b);
    SWAP(c,d);

    return 0;
}
```

Hinweise:

- Mit Compiler-Aufruf **gcc -E datei.c** erhält man als Ausgabe das Ergebnis der Arbeit des Präprozessors. Ausgabe erfolgt nach `stdout`. ➔ nützlich für Fehlersuche!

# Präprozessor (10)

Mehrfacheinbindung von Header-Dateien kann zu Problemen führen

- z.B. Mehrfachdefinitionen von Verbünden oder gleich benannten Datentypen

Dies ist recht **schwierig zu unterdrücken**, da Header-Dateien wiederum andere Header-Dateien einbinden (dürfen).

Folgender – **sehr verbreiteter** – Mechanismus schafft Abhilfe: „Include Guards“

```
/* header file "foo.h": */  
#ifndef __FOO_H_  
#define __FOO_H_  
  
/* contents of file */  
  
#endif /* __FOO_H_ */
```

Ergebnis: Der Inhalt von `foo.h` wird **nur einmal** eingebunden!

# 4. Effizientes Programmieren in C

1. Felder und Zeichenketten
2. Standardein- und -ausgabe
3. Zeiger
4. Felder und Zeiger
5. Benutzerdefinierte Typen
6. Anwendungsbeispiel “Verkettete Listen”
7. Kommandozeilenparameter
8. Dateiein- und -ausgabe
9. Programmerzeugung und Präprozessor
10. **“Beliebte” Fehler**





# Beliebte Fehler (1)

## Parameterfehler bei der Ein-/Ausgabe

- Beispiel: [Compiler gibt i.d.R. Warnmeldung aus; aber Kompilation erfolgt]

```
int main( void )
{
    int a = -2, b = 4;
    double c = 12.445325E-12;

    printf( "%u  %d  \n", a, b );
    printf( "%d  %d  %d  \n", a, c, b );
    return 0;
}
```

## Probleme:

- falsche Umwandlung des internen Formats
- der Übergabestack wird in falschen Positionen ausgelesen
- falsche Ausgaben entstehen
- unerwünschtes Programmverhalten → Speicherfehler, evtl. Absturz

# Beliebte Fehler (2)

## Fehlerhafte Freigabe von Speicher

- Beispiel: [Compiler gibt i.d.R. Warnmeldung aus; aber Kompilation erfolgt]

```
int main( void )  
{  
    int* ptr;  
    int array[4];  
  
    ptr = array;  
    free(ptr) ;  
  
    return 0;  
}
```

## Probleme:

- statischer Speicherplatz wird versucht freizugeben
- Freispeicherliste falsch
- Speicherfehler (Segmentation Fault)

# Beliebte Fehler (3)

## Fehlerhafte Freigabe von Speicher

- Beispiel:  
(kompiliert ohne  
Warnmeldung)

```
#include <stdlib.h>
#include <stdio.h>
int main( void )
{
    int* ptr1;
    int* ptr2;
    ptr1 = (int*) malloc( sizeof(int) );
    ptr2 = ptr1;
    free(ptr1);
    free(ptr1); //oder auch free(ptr2)
    return 0;
}
```

```
*** Error in `./test': double free or corruption (fasttop): 0x0000000001d9c010 ***
Abgebrochen (Speicherabzug geschrieben)
```

## Probleme:

- Speicherbereich wird doppelt freigegeben
- Freispeicherliste wird korrumpiert
- sehr ärgerlich: Der Programmabsturz kann womöglich auch erst viel später erfolgen!

# Beliebte Fehler (4)

## Indexfehler

- Paararbeit
- Wo sind der/  
die Fehler?

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

int a[4];

int main( void )
{
    int i;
    int* b;
    unsigned int j;
    b = a+1;
    j = -1;
    for ( i=0; i<=4; i++ ) //Index zu groß
        a[i] = j;          //a[4] != mein Speicher
    a[-1] = -1;            //Index zu niedrig
    b[-1] = 42;            //Ok, da b[-1]==a[0]
    b[3] = 10;             //Index zu groß: b[-1]..b[2]
    a[j] = 35;             //Index zu niedrig bzw. zu hoch
                          //da j unsigned int ist

    return 0;
}
```

# Beliebte Fehler (5)

## Indexfehler

- Beispiel: siehe rechts
- Frage: Welche Ausgabe ergibt sich?
- Antwort:  
1      2  
(D.h.: Nicht 1 4 und auch nicht 2 4!)
- Erklärung:
  - lokale Variablen (b, c) stehen hintereinander im Speicher; Felder danach
  - a[-1] überschreibt damit c
- Probleme:
  - Änderung anderer Daten & Änderung des Programmes → Effekt tritt ggf. erst verzögert zutage
  - sehr subtiler Fehler, evtl. direkter Absturz
  - bei komplexeren Programmen ist die Fehlerquelle nur sehr schwer aufzufinden

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

int main( void )
{
    int b = 1;
    int a[4];
    int c = 4;

    a[-1] = 2;

    printf( "%d  %d  \n", b, c );

    return 0;
}
```