4 Lexikalische Konventionen (S. 76)[i]

4.1 Zeichenvorrat von C (S. 76)

A – Z, a – z, 0 – 9, Leerzeichen, Tabulator, ; . () [] <> + - * / % ^ ~ & |= ! ?: ' " { } \ _

KEINE UMLAUTE!

Einzelne Zeichen begrenzt durch einfache Hochkommas -> ’a‘
Zeichenketten begrenzt durch Anführungszeichen -> “Zeichenkette“
Ein Zeichen wird durch ein Byte dargestellt (7-Bit-Code)

4.2 Lexikalische Einheiten (S. 77)

Namen	Variablen, Funktionen, Marken usw.
Reservierte Wörter, auch Schlüsselwörter	sizeof, int, if, void usw.
Literale Konstanten	Ganzzahlige Konstanten vom Typ int usw.
Konstante Zeichenketten	Strings
Operatoren	+ - & [] usw.
Satzzeichen	: [] , * = usw.

Gross- und Kleinschreibung

C ist „case sensitive“. Alpha ≠ alpha
Reservierte Wörter sind immer klein geschrieben
LISTE RESERVIERTER WÖRTER S.81 (4.2.2)

4.2.1 Namen

Namen bestehen aus a-z, A-Z, 0-9, _
Das **erste Zeichen** eines Namens darf **keine Ziffer** sein!

4.2.3 Literale und symbolische Konstanten

Symbolische Konstanten

Symbolische Konstanten haben einen Namen, der ihren Wert repräsentiert. Solche Konstanten werden mit Hilfe des Präprozessors-Befehl #define eingeführt

```
#define PI 3.1415      (ohne Strichpunkt)
```

Literale Konstanten

4.2.3.1 Ganzzahlige Konstanten: (Default: int)

254	035	0x3f	-34	14L	14U	14UL
dez	okt	hex		long	unsigned	unsigned long

14 *int-Konstante in Dezimaldarstellung mit dem Wert 14 dezimal*
014 *int-Konstante in Oktaldarstellung mit dem Wert 12 dezimal*
0x14 *int-Konstante in Hexadezimaldarstellung mit dem Wert 20 dezimal*
14L *long-Konstante in Dezimaldarstellung mit dem Wert 14*
14U *unsigned-Konstante in Dezimaldarstellung mit dem Wert 14*

4.2.3.2 Zeichenkonstanten

'c'	'\n'	'\x4a'	'\14'	'\\'	L'a'
		ASCII hex	ASCII okt		Double Byte

TABELLE MIT ERSATZDARSTELLUNGEN S.86

4.2.3.3 Gleitpunktkonstanten (Default: double)

254.89	-13.0	3.45e23	4.65f	3.14159L
		exp-Schreibweise	float-Schreibweise	long double

Hinweis: *double i=0.0; (Ansonsten vom Compiler zu int optimiert)*

4.2.3.4 Aufzählungskonstanten

Aufzählungskonstanten haben einen konstanten ganzzahligen Wert (int), beginnend bei 0 wenn nichts explizit zugewiesen. Das Etikett, hier test, ist optional.

```
enum test {ALPHA, BETA, GAMMA};          // ALPHA=0, BETA=1, GAMMA=2
enum test {ALPHA=5, BETA=3, GAMMA=7};    // ALPHA=5, BETA=3, GAMMA=7
enum test {ALPHA=4, BETA, GAMMA=3};      // ALPHA=4, BETA=5, GAMMA=3
```

5 Datentypen und Variablen (S.96)

5.1 Typenkonzept

Alle Variablen müssen genau definiert sein und einen Typ (z.B. int, float, double,…) haben.
Hinweis: Wird ein float-Wert einer int-Variable zugewiesen, werden die Nachkommastellen abgeschnitten.

5.2 Einfache Datentypen

Der Datentyp einer Variable legt den **Speicherbedarf** (Anz. Bits), den **Wertebereich** und die **Genauigkeit** (Gleitpunktzahlen)

Integertypen

`char`	Zeichen (immer 8 Bit)
`int`	üblicher Standarttyp (effizienteste Grösse, mind. 16 Bit, Zielsystemabhängig)
`short / short int`	für kleine ganzzahlige Werte
`long / long int`	für grosse ganzzahlige Werte
`enum`	Aufzählungstypen

Gleitpunkttypen

`float`	Gleitpunktzahl (Fliesskommazahl)
`double`	Gleitpunktzahl mit einer höheren Genauigkeit der Darstellung
`long double`	Gleitpunktzahl mit einer noch höheren Genauigkeit der Darstellung (extended precision)

Bemerkung: float: 1 Vorzeichenbit, 8 Bits für Exponenten, 23 Bits für Mantisse
double: 1 Vorzeichenbit, 11 Bits für Exponenten, 52 Bits für Mantisse

ÜBERSICHT ÜBER DIE ELEMENTAREN DATENTYPEN IN C (S.99).

5.3 Variablen (S. 106)

Definition Art der Variablen (Typ, Speicherklasse und Typ-Attribute) Reservierung des Speicherplatzes
Deklaration Art der Variable bzw. die Schnittstelle der Funktion.
Globale Variablen Stehen allen Funktionen zur Verfügung, müssen ausserhalb einer Funktion definiert werden. Automatisch Initialisierung auf Null.
Lokale Variablen Stehen nur in der Funktion zur Verfügung, in der sie definiert sind.
Sichtbarkeit Auf die Variable kann zugegriffen werden. Die Sichtbarkeit wird mein Kompilieren vom Compiler überprüft.
Lebensdauer Während der Lebensdauer besitzt eine Variable einen Speicherplatz

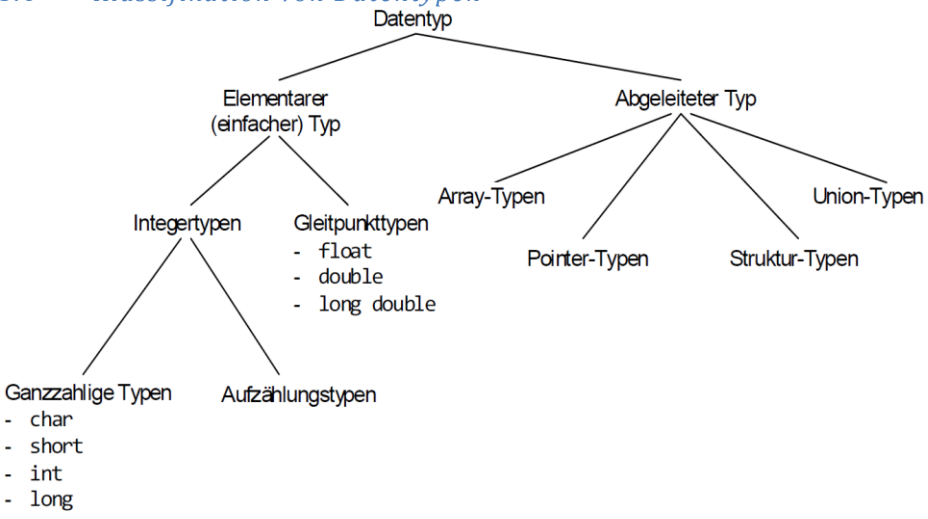
5.4 Typ-Attribute const und volatile

const Variable kann nur initialisiert werden. Weitere Änderungen sind nicht mehr möglich.
const double PI = 3.1415927;

volatile Variable wird nicht (weg-)optimiert durch Compiler, Adressen der Variablen werden nicht geändert. Wenn die Variable immer an derselben Adresse stehen muss.
Beispiel:

```
volatile unsigned char led;
led = 1;
sleep (200);
led = 0;
```

5.6 Klassifikation von Datentypen



6 Pointer und Arrays (S. 116)

6.1 Pointertypen und Pointervariablen

Ein Pointer (Zeiger) ist eine Variable, welche die Adresse einer im Speicher befindlichen Variablen oder Funktion aufnehmen kann. Pointer können auf Datenobjekte, Pointervariablen und Funktionen zeigen.
Der Speicherbedarf einer Pointervariablen ist unabhängig vom Pointer-Typ. Er ist so gross, dass die maximale Adresse Platz findet (z.B. 32 Bit)

Definition von Pointervariablen

Pointer in C sind typisiert, sie zeigen auf eine Variable des definierten Typs.

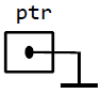
```
int*   ptr1;   Pointer auf int
float* ptr2;   Pointer auf float
```



NULL-Pointer

NULL ist vordefiniert (in <stddef.h>) und setzt den Pointer auf einen definierten NULLWERT. Der NULLWERT ist implementationsabhängig. Besser ist, statt NULL direkt 0 zu verwenden.

```
int*   pointer= 0;
```



Wertzuweisung

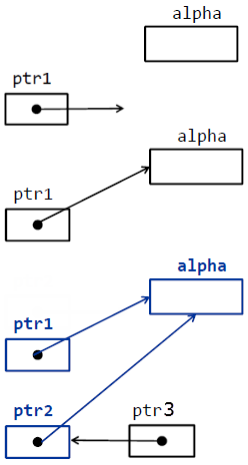
```
int alpha;          // Variable alpha vom Typ int wird definiert

int* ptr1;          // Pointer ptr1 auf den Typ int wird definiert
                    // ptr1 zeigt auf eine nicht definierte Adresse

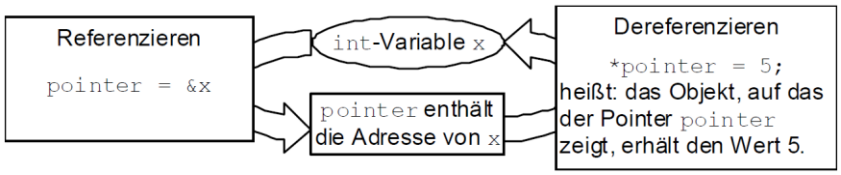
ptr1 = &alpha;      // ptr1 zeigt nun auf die Variable alpha, d.h.
                    // ptr1 enthält die Adresse der Variablen alpha

int* ptr2=ptr1;     //Definieren des ptr2 auf int der auf dieselbe Adresse zeigt wie ptr1
*ptr1=24;           //Dereferenzierung: Der Variable alpha wird 24 zugewiesen

int** ptr3=&ptr1;   //Pointer ptr3 auf den Typ int* wird definiert
                    //und zeigt nun auf ptr1
```



Dereferenzierung. Zugriff auf ein Objekt über einen Pointer



& wird als **Adressoperator** bezeichnet. Mit ihm erhält man die **Adresse eines Objekts**.

* wird als **Dereferenzierungsoperator** bezeichnet. Mit ihm erhält man den **Inhalt des Objektes**, auf das der Pointer zeigt.

Beispiel: Adresse einer Variable

```
float zahl = 3.5;
printf("Adresse von zahl: %p", &zahl);   // Ausgabe der Adresse einer Variable
```

6.2 Pointer auf void

Der Pointer auf void ist ein **untypisierter** Pointer. Dieser ist zu allen anderen Pointertypen kompatibel und kann insbesondre in Zuweisungen mit typisierten Pointern gemischt werden. Ein Pointer auf void umgeht also bei einer Zuweisung die Typüberprüfung des Compilers.

6.3 Eindimensionale Arrays

Ein Array bietet eine kompakte Zusammenfassung von mehreren Variablen des gleichen Typs.

Arraydefinition

Syntax: `Typname arrayName[GROESSE];` //GROESSE ist die Anzahl der Array-Elemente

Beispiel 1:
```
int data[1000];          // ein Array von 1000 int-Werten
double zahl[5];          // ein Array von 5 double-Werten
```

Beispiel 2:
```
int alpha [3];    //Arraydefinition, (ein Array von 3 int-Werten)
alpha[0] = 1;     //Das 1. Element alpha[0] hat den Index 0. Der Wert 1 wird zugewiesen.
alpha[1] = 2;     //Das 1. Element alpha[1] hat den Index 1. Der Wert 2 wird zugewiesen.
alpha[2] = 3;     //Das 1. Element alpha[2] hat den Index 2. Der Wert 3 wird zugewiesen.
```

alpha[0] int	alpha[1] int	alpha[2] int
1	2	3

```
alpha[3] = 4;     //Bereichsüberschreitung (möglich, muss vermieden werden!) Buffer overflow
```

Array durchschlaufen

```
#define GROESSE 5          // symbolische Konstante definieren
int alpha[GROESSE];
int i;
for (i = 0; i < GROESSE; i++)
{
       printf("%d \n", alpha[i]);
}
```

10.1.5 Zeichenketten (Strings)

Ein String ist in C ein Array von Zeichen (char-Array). Ein String muss in C immer mit dem Nullzeichen '\0' abgeschlossen werden. DIESES BENÖTIGT EINE STELLE IM ARRAY!

10.1.6 char-Arrays

```
char str[20] = {'Z', 'e', 'i', 'c', 'h', 'e', 'n', 'k', 'e', 't', 't', 'e', '\0'};
char str[20] = "Zeichenkette";
char str[20] = {"Zeichenkette"};
char str[] = "Zeichenkette";
```

Hinweis: Eine direkte Zuweisung einer Zeichenkette an einen Vektor kann nur bei der Initialisierung erfolgen.

10.2.3 Ausgabe

```
Printf(“%s\n“, str);       //String str wird ausgegeben
```

10.6.1 Einige Stringverarbeitungsfunktionen

Für die Stringverarbeitung wird die Bibliothek **string.h** benötigt.
Funktionen, die mit **str** beginnen dienen der Stringverarbeitung, erkennen das '\0'-Zeichen. Funktionen, die mit **mem** beginnen dienen der Speicherverarbeitung, erkennen das '\0'-Zeichen nicht, deshalb muss die Bufferlänge in Byte ebenfalls als Parameter übergeben werden.

strcpy() zum Kopieren von Strings
```
char* strcpy(char* dest, const char* src);
```
Der Inhalt von src wird an die Adresse, auf die dest zeigt, kopiert.

strcat() zum Anhängen eines Strings an einen anderen
```
char* strcat(char* dest, const char* src);
```
Der Inhalt src wird an den String dest angehängt. Das Stringende-Zeichen `\0` von dest wird dabei eintfernt.

strcmp() zum Vergleichen von Strings
strncmp()
```
int strcmp(const char* s1, const char* s2);
int strncmp(const char* s1, const char* s2, size_t n);
```
Vergleicht die beiden Strings s1, s2, bei strncmp() nur bis zum n-ten Zeichen, oder `\0`.
Return: <0 s1 ist lexikographisch kleiner als s2
==0 s1 ist lexikographisch gleich s2
>0 s1 ist lexikographisch grösser als s2

strlen() zum Ermitteln der Stringlänge
```
size_t strlen(const char* s);
```
Bestimmt die länge des Strings. Das Stringende-Zeichen `\0` wird nicht gezählt.

memcpy(), memmove(), memcmp(), memchr(), memset()

DIE AUSFÜHRLICHEN DEFINITIONEN SIND FÜR STR-FUNKTIONEN AB SEITE 275 UND FÜR MEM-FUNKTIONEN AB SEITE 280 ZU FINDEN.

7 Anweisungen, Ausdrücke und Operatoren (S. 136)

7.1 Postfix- und Präfixoperatoren

Postfix-Operatoren sind unäre Operatoren, die **nach** (post) ihrem Operanden stehen.
`printf("%d", i++);` //Rückgabewert von i, als Nebeneffekt wird i inkrementiert
i wird auf den Bildschirm geschrieben, **anschliessend** inkrementiert.

Präfix-Operatoren sind unäre Operatoren, die **vor** (prä) ihrem Operanden stehen.
`printf("%d", ++i);` //Rückgabewert ist i+1, als Nebeneffekt wird i inkrementiert
i wird **zuerst** inkrementiert, dann auf den Bildschirm geschrieben.

7.4 Auswertungsreihenfolge

1. Wie in der Mathematik werden als erstes **Teilausdrücke in Klammern** ausgewertet.
2. Dann werden **Ausdrücke mit unären Operatoren** ausgewertet.
   Unäre Operatoren werden *von rechts nach links* angewendet, d.h.
   2.1 **zuerst** werden die **Postfix-Operatoren** auf ihre Operanden
   2.2 und **dann** die **Präfix-Operatoren** auf ihre Operanden angewendet.
3. Abschliessend werden Teilausdrücke mit mehrstelligen Operatoren gemäss der **Priorität** der Operatoren ausgewertet.
4. Bei gleicher Priorität der Operatoren entscheidet die **Assoziativität** (von links nach rechts oder von rechts nach links)

Beispiel:

```
int zahl[4] = {4, 7, 66, 235};
int* p = &zahl[0];
Operation|  *p |    p
---------|-----|---------
Start    |   4 | 0x22cca0
*p++     |   4 | 0x22cca4
*++p     |  66 | 0x22cca8
*(p++)   |  66 | 0x22ccac
(*p)++   | 234 | 0x22ccac
*p       | 235 | 0x22ccac
```

PRIORITÄT UND ASSOZIATIVITÄT DER OPERATOREN VON C (S. 168)

7.5 L-Werte und R-Werte

Ein **Ausdruck** stellt einen **L-Wert** (**lvalue** oder **left value**) dar, wenn er sich auf ein **Speicherobjekt** bezieht. Ein solcher Ausdruck kann links (und rechts) des Zuweisungsoperators stehen.
Ein **lvalue** erfordert immer *Schreibzugriff*.

Ein **Ausdruck**, der sich nicht auf ein Speicherobjekt bezieht, kann nur rechts des Zuweisungsoperators stehen. Er wird als **R-Wert** (**rvalue** oder **right value**) bezeichnet. Einem R-Wert kann nichts zugewiesen werden.
Auf einen **rvalue** wird nur *lesend zugegriffen*.

7.6 Zusammenstellung der Operatoren

7.6.1 Einstellige (unäre) arithmetische Operatoren

Positiver Vorzeichenoperator:	+A
Negativer Vorzeichenoperator:	-A
Postfix-Inkrementoperator:	A++
Präfix-Inkrementoperator:	++A
Postfix-Dekrementoperator:	A--
Präfix-Dekrementoperator:	--A

7.6.2 Einstellige (binäre) arithmetische Operatoren

Additionsoperator:	A + B
Subtraktionsoperator:	A - B
Multiplikationsoperator:	A * B
Divisionsoperator:	A / B
Modulooperator:	A % B

7.6.3 Zuweisungsoperatoren

Zuweisungsoperator:	=

Sowie die kombinierten Zuweisungsoperatoren:

Additions-Zuweisungsoperator:	A += B
Subtraktions-Zuweisungsoperator:	A -= B
Multipl.-Zuweisungsoperator:	A *= B
Divisons-Zuweisungsoperator:	A /= B
Restwert-Zuweisungsoperator:	A %= B

… (S. 149)

7.6.4 Relationale Operatoren

Gleichheitsoperator:	A == B
Ungleichheitsoperator:	A != B
Grösseroperator:	A > B
Kleineroperator:	A < B
Grössergleichoperator:	A >= B
Kleinergleichoperator:	A <= B

Hinweis: Floating-Point Variablen nicht mit == vergleichen. Fehler vom Rechnen -> 1.999 == 2 //FALSE

7.6.5 Logische Operatoren

Logisch UND (AND):	A && B
Logisch ODER (OR):	A || B
Logisch NICHT (NOT):	!A

Hinweis: 0 = false, falsch
1 = true, wahr (genauer: ungleich 0)

7.6.6 Bit-Operatoren

Bitweises AND:	A & B
Bitweises OR:	A | B
Bitweises NOT (Inverter):	~A
Bitweises XOR:	A ^ B
Rechtsshift-Operator:	A >> B
Linksshift-Operator:	A << B

Hinweis: Nur mit unsigned Typen durchführen.
Es werden Nullen nachgeschoben.

7.6.7 Sonstige Operatoren

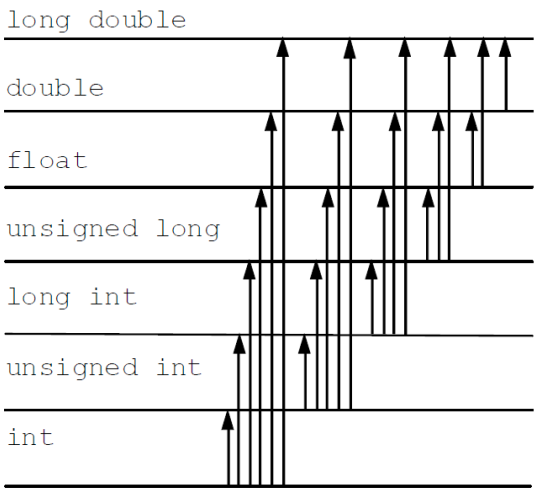
Bedingungsoperator (Ternärer Operator)
A ? B : C

Ist eine verkürzte Schreibweise für

```
if (A)
   B;
else
   C;
```

7.7 Implizite (automatische) Typenumwandlung

Nebst den impliziten (automatischen) Typumwandlungen kann mit Hilfe des cast-Operators eine explizite Typumwandlung bewirkt werden.
! Die Verantwortung liegt dabei beim Programmierer.



Explizite Typenumwandlung

Die explizite Typenumwandlung wird mit Hilfe des cast-Operators durchgeführt.

```
                (Zieltyp)Ausdruck
Beispiel:       int a = (int)4.6; // a == 4
                double d = (double)a; // d == 4.0
```

8 Kontrollstrukturen (S. 184 ff)

9.8.1 Iteration und Rekursion (S. 233)

Theoretisch sind Iteration und Rekursion äquivalent. Die rekursive Form kann eleganter sein, ist aber praktisch immer ineffizienter als die iterative Form. Das **Abbruchkriterium** ist in beiden Formen zentral.

Rekursion

Ein Algorithmus heisst rekursiv, wenn er Abschnitte enthält, die sich selbst direkt oder indirekt aufrufen.

Beispiel Fakultät
0! = 1
n! = **(n – 1)!** · n

```
unsigned long
faku(unsigned int n)
{
   if (n > 1)
   {
      return n * faku(n-1);
   }
   else
   {
      return 1UL;
   }
}
```

Iteration

Er heisst iterativ, wenn bestimmte Abschnitte innerhalb einer Ausführung mehrfach durchlaufen werden (Schleife).

Beispiel Fakultät
0! = 1
n! = 1 · 2 · 3 · … · (n – 1) · n

```
unsigned long
faku(unsigned int n)
{
  unsigned long fak = 1UL;
  unsigned int i;
  for (i = 2; i <= n; ++i)
  {
    fak = fak * i;
  }
  return fak;
}
```

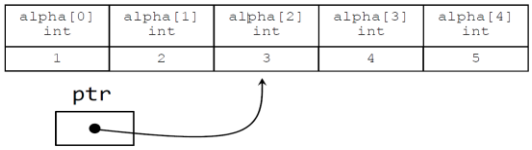
Stack

Bei jedem Aufruf von `faku()` werden die Rücksprungadresse und weitere Verwaltungsinformationen auf einem Stack abgelegt, der durch das Laufzeitsystem verwaltet wird. Auch die übergebenen Parameter (hier nur einer) werden auf diesem Stack abgelegt. Dabei wächst der Stack mit der Rekursionstiefe der Funktion. Der letzte Aufruf von `faku()` mit dem Parameter `n = 1` bewirkt keine weitere Rekursion, da ja die Abbruch-Bedingung erfüllt ist. Der Abbau des Stacks geschieht in umgekehrter Reihenfolge.

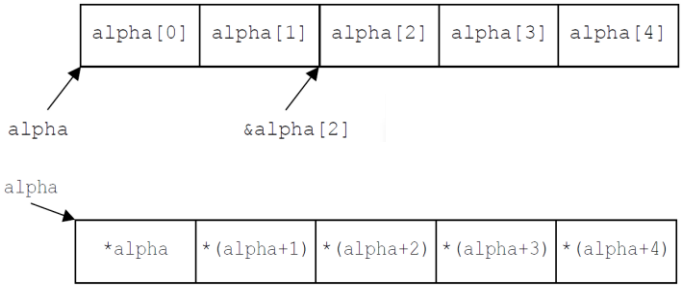
10 Fortgeschrittene Programmierung mit Pointern (S. 252)

10.1 Arrays

```
int alpha[5];     //Definieren des Arrays alpha
int* ptr;         //Definition des Pointers ptr
ptr = &alpha[2];  //Pointer zeigt auf das i-te
                  //Arrayelement
```



Hinweis: Der Name des Arrays kann als konstanter Zeiger auf das erste Element des Arrays verwendet werden.



alpha sei ein Vektor. Dann gilt: `alpha[i] == *(alpha + i)` . Dieses bedeutet, dass die **Arraynotation äquivalent** ist zu einer **Pointernotation** bestehend aus Dereferenzierungsoperator, Pointer und Abstand.

10.2 Übergabe von Arrays und Zeichenketten

Bei der Übergabe eines Arrays an eine Funktion wird als Argument der Arrayname übergeben (i.e. Pointer auf erstes Element des Arrays). Der formale Parameter für die Übergabe eines eindimensionalen Arrays kann ein offenes Array sein oder ein Pointer auf den Komponententyp des Arrays.

Wichtig: Die Grösse des Arrays muss immer explizit mitgegeben werden!

```
enum {groesse = 3};
                                        //Parameter ist Pointer auf Arrayelement
void init(int* alpha, int dim);         //Man sieht nicht sofort ob array oder pointer
void ausgabe(int alpha[], int dim);     //Für Arrayübergabe besser, man sieht‘s gerade
                                        //Parameter ist offenes Array

int main(void)
{
   int arr[groesse];
   init(arr, groesse);                  //Argument ist Name des Arrays
   ausgabe(arr, groesse);
   return 0;
}
```

Ausgabe

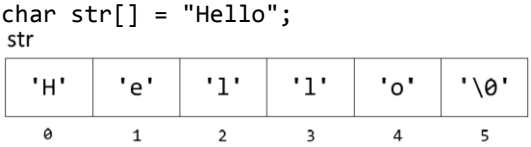
Strings (Zeichenketten) sind char-Arrays, deren letztes Zeichen '\0' ist.

Ausgabe von konstanten Strings ist trivial: `printf("Bin ein konstanter String.\n");`

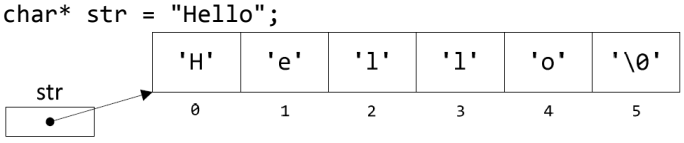
Stringvariablen als Argumente werden von printf() mit Hilfe des Formatelements s ausgegeben.

`char str[] = "Hello"; vs char* str = "Hello";`

`char str[] = "Hello";`



`char* str = "Hello";`



11 Strukturen, Unionen und Bitfelder (S. 304)

11.1 Strukturen

Logisch zusammenhängende Daten können zusammengefast werden. Eine Struktur besteht aus einer festen Anzahl von Komponenten (Felder), die einen verschiedenen Typen haben können.

11.1.1 Definition von Strukturtypen und Strukturvariablen

```
struct StructName
{
   FeldTyp1 feld1;
   FeldTyp2 feld2;
   FeldTyp3 feld3;
   ...
   FeldTypN feldN;
};
```

Beispielstruktur für einen Studenten

```
struct Adresse
{
   char strasse[20];
   int hausnummer;
   int plz;
   char stadt[20];
};
```

```
struct Student
{
   int matrikelnummer;
   char name[20];
   char vorname[20];
   struct Adresse wohnort;
};
```

Die erstellten Strukturen können mehrfach verwendet werden.

```
struct Student mueller;
struct Student keller;
```

11.1.3 Selektion der Komponenten

Der Zugriff auf ein Feld einer **Strukturvariablen** erfolgt über den Namen der Strukturvariablen, gefolgt von einem Punkt und dem Namen des Feldes.

```
mueller.matrikelnummer = 12345;
strcpy(mueller.vorname, "Heinz");
```

Wenn der Zugriff über einen **Pointer** auf eine Strukturvariable erfolgt, über den Namen des Pointers, gefolgt von einem Pfeil (->) und dem Namen des Feldes.

```
struct Student* ptr = &keller;
ptr -> matrikelnummer = 67890;     //einfache Form mit Pointer
(*ptr).matrikelnummer = 67891;     //alternative From
```

Alignment

Gewisse Datentypen verlangen unter Umständen, dass Sie auf eine Wortgrenze (gerade Adresse) gelegt werden. Durch das Alignment kann es vorkommen, dass einzelne Bytes nicht verwendet werden, d.h. im Speicher ausgelassen werden. Die Grösse einer Strukturvariablen kann nicht durch Addieren der Grössen der Felder ermittelt werden, nur sizeof() liefert den genauen Wert.

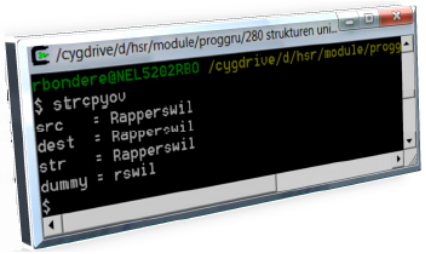
```
struct B
{
  int   wert;
  char  text[3];
  int   zahl;
};
```

Das `int`-Feld `zahl` muss auf einer geraden Adresse beginnen. Deshalb wird die Adresse 7 hier nicht verwendet.

11	zahlLoLo
10	zahlLoHi
9	zahlHiLo
8	zahlHiHi
7	Not Used
6	text[2]
5	text[1]
4	text[0]
3	wertLoLo
2	wertLoHi
1	wertHiLo
0	wertHiHi

Buffer overflow

```
/* Datei: strcpyov.c */
#include <stdio.h>
#include <string.h>
int main(void)
{
struct
{
char dest[5];
char dummy[8];
} s;
char src[] = "Rapperswil";          // benötigt 11 Bytes
char* str;
str = strcpy(s.dest, src);          // s.dest zu klein!
printf("src = %s\n", src);          //src = Rapperswil
printf("dest = %s\n", s.dest);      //dest = Rapperswil
printf("str = %s\n", str);          //str = Rapperswil
printf("dummy = %s\n", s.dummy);    //dummy = rswil
return
```



dummy[7]	
dummy[6]	
dummy[5]	'\0'
dummy[4]	'l'
dummy[3]	'i'
dummy[2]	'w'
dummy[1]	's'
dummy[0]	'r'
dest[4]	'e'
dest[3]	'p'
dest[2]	'p'
dest[1]	'a'
dest[0]	'R'

11.1.4 Übergabe von Strukturvariablen an Funktionen und Rückgabe durch return

Strukturvariablen können komplett an Funktionen übergeben werden. Der Rückgabetyp einer Funktion kann eine Struktur sein. Dabei wird die Strukturvariable direkt komplett übergeben. Zu beachten ist der Kopieraufwand bei der Übergabe, bzw. Rückgabe eines Wertes. In der Praxis wird deshalb häufig mit Pointern gearbeitet.

```
void foo(struct Angestellter a);                // grosser Kopieraufwand, nicht ideal
void foo(struct Angestellter* pa);              // nur Pointerübergabe, effizient
void fooRead(const struct Angestellter* pa)     // nur Pointerübergabe, read only durch const
```

11.2 Unionen

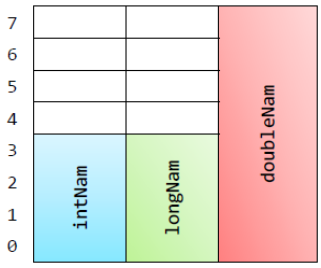
Eine Union ist gleich aufgebaut wie eine Struktur. Im Gegensatz zur Struktur ist aber zu einem bestimmten Zeitpunkt jeweils nur eine einzige Komponente einer Reihe von alternativen Komponenten gespeichert.

Hinweis: Der Programmierer muss verfolgen, welcher Tap jeweils in der Union gespeichert ist. Der Datentyp, der entnommen wird, muss der sein, der zuletzt gespeichert wurde.

```
union UnionName
{
   FeldTyp1 feld1;
   FeldTyp2 feld2;
   FeldTyp3 feld3;
   ...
   FeldTypN feldN;
};
```

Beispiel Definition

```
union Vario
{
   int intNam;
   long longNam;
   double doubleNam;
};
```



12 Komplexere Datentypen, eigene Typennamen und Namensräume *(S. 336)*

12.1 Komplexere Vereinbarungen(S. 336)

Array von Pointern

```
int* alpha[8];       //alpha ist ein Array von 8 int-Pointern
```

[] hat Vorrang vor *

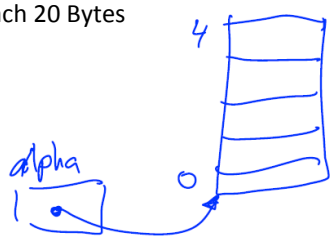
Pointer auf ein Array

```
int (*alpha)[8];     //alpha ist ein Pointer auf ein Array von 8 int-Werten
int* (*beta)[8];     //beta ist ein Pointer auf ein Array von 8 int-Pointern
```

Wie viel höher ist die Adresse von (alpha+1)

```
#include <stdio.h>
int main(void)
{
  int arr[5];
  int (*alpha)[5] = &arr;
  printf("Adresse von alpha : %p\n", alpha);
  printf("Adresse von (alpha+1): %p\n",
  alpha+1);
  return 0;
}
```

`alpha` ist ein Pointer auf einen Array von 5 `int`s, d.h. `alpha` zeigt auf einen Bereich, der 5 · 4 = 20 Bytes gross ist. `(alpha+1)` liegt demnach 20 Bytes höher.



Funktion mit Pointer als Rückgabewert

```
int* foo(....);      //foo ist eine Funktion, welche einen int-Pointer zurückgibt
```

() hat immer Vorrang

Pointer auf eine Funktion

```
int (*pFunc)(....); //pFunc ist ein Pointer auf eine Funktion, die ein int zurückgibt
int* (*pFoo)(....); //pFoo ist ein Pointer auf eine Funktion,
                    //die einen int-Pointer zurückgibt
```

() hat immer Vorrang, dann Assoziativität

Funktion mit einem Rückgabewert „Pointer auf ein Array von Pointern auf Zeichen“

```
char* (*delta(....))[10];  // delta ist eine Funktion mit der Parameterliste (....) und gibt
einen Pointer auf ein Array von 10 Pointern auf char zurück
```

() hat immer Vorrang, [] hat Vorrang vor *

12.2 Komplexere Typen (S. 338)

12.3 typedef zur Vereinbarung eigener Typnamen (S. 338ff)

Eigene Typnamen als Aliasnamen zu bestehenden Datentypennamen können in C mit hilfe von typedef vereinbart werden. Typedefs werden im Normalfall global definiert.

Strukturdefinition mit typedef

```
typedef struct
{
   int nummer;       // Artikelnummer
   char name[80];    // Artikelname (Bezeichnung)
   double preis;     // Artikelpreis
} Artikel;

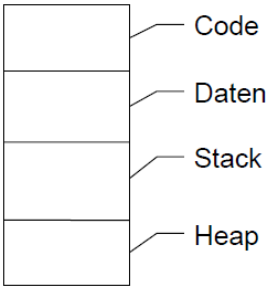
int main(void)
{
   Artikel art;      //Bei der Definition einer Strukturvariable muss das Schlüsselwort
   Artikel* pArt;    //struct nicht mehr gesetzt werden.
}
```

Hinweis: Ein `typedef` ist eine reine Textersetzung.

13 Speicherklassen (S. 344)

13.1 Adressraum eines Programms (S. 344)

Der Adressraum eines ablauffähigen C-Programms – also einer Datei programm.exe – besteht aus den vier Segmenten (Regionen) Code, Daten, Heap und Stack.



Code
Programm in Maschinencode
Häufig werden auch Konstante abgelegt
Daten
Globale Variablen, static Variablen
Stack
Lokale Variablen
Parameter einer Funktion
Rücksprungadressen
Heap
Dynamische Variablen (Speicherplatz wird erst zur Laufzeit angefordert)

13.2 Programme aus mehreren Dateien – Adressen (S. 346)

C unterstützt eine getrennte Kompilierung. Ein grosses Programm kann in Module zerlegt werden, die getrennt kompiliert werden. Der **Compiler** erzeugt aus jeder Quelldatei eine Object-Datei mit Maschinencode.
Der **Linker** bindet die kompilierten Object-Dateien, die aufgerufenen Bibliotheksfunktionen und das Laufzeitsystem (Fehlerfunktionen, Speicherverwaltungsroutinen, etc.) zu einem ablauffähigen Programm (executable program).

Buildprozess

```
gcc -c foo.c                          //Quelldateien foo.c kompilieren
gcc -c goo.c                          //Quelldateien goo.c kompilieren
gcc -c main.c                         //Quelldateien main.c kompilieren
gcc -o prog.exe foo.o goo.o main.o    //Object-Dateien Linken und Programm erzeugen
```

Um diesen Schritt zu automatisieren werden in der Praxis oft ein Buildscript oder eine Buildtool eingesetzt.

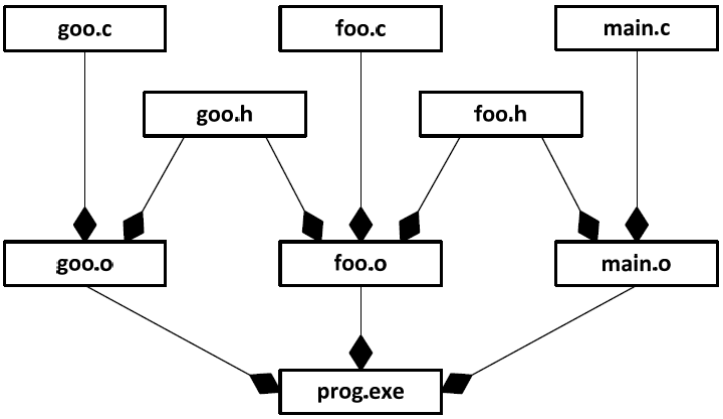
make-File

In einem make-File können Abhängigkeiten zwischen Dateien definiert werden. Wenn eine Datei geändert wird, dann werden alle Operationen ausgeführt, welche von dieser geänderten Datei abhängig sind.

```
# makefile
cc=gcc
CFLAGS= -c -Wall
LFLAGS= -Wall
OBJ=foo.o goo.o main.o
EXE=prog.exe

${EXE}: ${OBJ}
        ${cc} ${LFLAGS} -o ${@} ${OBJ}
foo.o: foo.h goo.h foo.c
        ${cc} ${CFLAGS} -o ${@} foo.c
goo.o: goo.h goo.c
        ${cc} ${CFLAGS} -o ${@} goo.c
main.o: foo.h main.c
        ${cc} ${CFLAGS} -o ${@} main.c

clean:
        rm -f ${OBJ} ${EXE}
```



Targets: Diese können mit make angesprungen werden, z.B. make goo.o
Abhängigkeitsliste, d.h. von diesen Dateien ist das Target abhängig
Befehl, der ausgeführt werden muss, falls etwas geändert hat. **Wichtig:** erstes Zeichen muss ein TAB sein.

13.3 Programme aus mehreren Dateien – die Speicherklasse extern (S. 347)

13.4 Programme aus mehreren Dateien – die Speicherklasse static

Information Hiding in C-Programmen

Funktionen und globale Variablen, die als `static` definiert werden, sind nur in ihrer eigenen Datei sichtbar. Funktionen aus anderen Dateien können auf diese Funktionen und Variablen nicht zugreifen.
(Ausführliche Behandlung dieses Aspektes erfolgt im Kapitel 19)

13.5.1 Automatische Variablen

13.5.2 Speicherklasse static für lokale Variablen (S. 357)

13.8 Tabellarischer Überblick über die Speicherklassen

Die folgende Tabelle zeigt die Gültigkeit und die Lebensdauer von Variablen der verschiedenen Speicherklassen.

Speicherklasse	Gültigkeit	Lebensdauer	Automat. Initialisierung	Segment
`register`	Block	Block	nein	Stack oder in Register
`auto`	Block	Block	nein	Stack
`static` lokale Variable	Block	Programm	mit 0	Daten
`static` externe Variable	In Datei ab Definition, in anderen Dateien nicht	Programm	mit 0	Daten
`extern` (nicht `static`)	In Programm ab Definition bzw. ab `extern`-Deklaration	Programm	mit 0	Daten

Nachfolgende Tabelle zeigt, welche Speicherklasse eingesetzt werden können, wenn man einen bestimmten Gültigkeitsbereich für Variablen erzeugen möchte.

Gültigkeitsbereich	Speicherklasse

Nur in Funktion oder Block	`auto`
	`auto static` (permantent lokal)
	`register`

Externer Variablen nur in Datei	`static` bei Definition
Externer Variablen im ganzen Programm	Nicht `static` bei Definition und extern in anderen Dateien

Nice to know / Nice to have

1 Kilobyte = 1 kByte = 2^{10} Bytes = 1024 Bytes
1 Megabyte = 1 MByte = 2^{20} Bytes = 1048576 Bytes
1 Gigabyte = 1 GByte = 2^{30} Bytes = 1073741824 Bytes

2^{10}	2^9	2^8	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
1024	512	256	128	64	32	16	8	4	2	1

Notizen

Code-Stile

Variablen

- mit Kleinbuchstaben beginnen
- erster Buchstaben von zusammengesetzten Wörtern ist gross (Camelcase)
- keine Underscores
- immer nur eine Variable pro Zeile definieren
- solche Feinheiten, bei denen sich zwei Namen nur in der Gross- bzw. Kleinschreibung einzelner Buchstaben unterscheiden, sollen unbedingt vermieden werden (fehleranfällig)
- Beispiele: `counter, maxSpeed` -> NICHT `max_speed`
- Variablen so lokal wie möglich definieren, d.h. im innersten möglichen Bloc

Globale Variablen

- so wenig wie möglich verwenden
- am Anfang der Datei definieren, d.h. auf jeden Fall vor der ersten Funktion
- müssen speziell gekennzeichnet werden. Sie sollen mit einem Präfix (Modulkürzel) gefolgt von einem underscore character (_) beginnen. Dadurch werden die Namen eindeutig.
  Beispiel: Die globale int-Variable counter im Modul Foo muss wie folgt definiert werden:
  ```
  int Foo_counter;
  ```

Funktionen

- mit Kleinbuchstaben beginnen
- erster Buchstaben von zusammengesetzten Wörtern ist gross
- keine Underscores
- Beispiele: `getCount(), init(), setMaxSpeed()`

Konstante

- Symbolische #define-Konstanten werden aus Grossbuchstaben und Underscores gebildet (keine Kleinbuchstaben)

Strukturen, Unionen

- Strukturname und Unionname mit einem grossen Buchstaben beginnen
  ```
  struct Angestellter;
  union Vario;
  ```
- Struktur- und Unionvariablen mit einem kleinen Buchstaben beginnen
- Bei Feldern von Strukturen und Union soll kein Präfix bei den Feldnamen verwendet werden

[i] Alle Seitenabgaben beziehen sich auf: Lehrbuch: „C als erste Programmiersprache“ (B. G. Teubner Verlag), 7. Auflage