Grundlagen der Informatik

Vorlesungsskript

Prof. Dr. T. Gervens, Prof. Dr.-Ing. B. Lang, Prof. Dr.-Ing. C. Westerkamp

7 DATEIZUGRIFF 2

7.1 WICHTIGE DATEIFUNKTIONEN 3
7.2 BEISPIELE 9
7.3 STANDARDEINGABE UND STANDARDAUSGABE 11

8 ZEIGER UND FELDER 12

8.1 ZEIGER UND ADRESSEN 12
8.2 ZEIGERARITHMETIK 13
8.3 ZUSAMMENHANG ZWISCHEN FELDERN UND ZEIGERN 15
8.4 FELDER ALS FUNKTIONSPARAMETER 17
8.5 MEHRDIMENSIONALE FELDER 19
8.6 FELDER VON ZEIGERN 22
8.7 ZEIGER AUF FUNKTIONEN 25
8.8 ARGUMENTE AUS DER KOMMANDOZEILE 26
8.9 BEISPIELE FÜR TYPVEREINBARUNGEN 28

9 SPEICHERKLASSEN, ATTRIBUTE UND GÜLTIGKEITSBEREICHE 29

9.1 SPEICHERKLASSEN 29
9.2 ATTRIBUTE 35
9.3 GÜLTIGKEITSBEREICH VON VARIABLEN 36
9.4 ZUSAMMENFASSUNG DER VARIABLENKLASSEN UND IHRER EIGENSCHAFTEN 37

7 Dateizugriff

Zur Eingabe von großen Datenmengen an ein Programm und zur langfristigen Sicherung von Daten eines Programms ist die Ein- und Ausgabe von Daten mittels Dateien unerlässlich.

Bevor man auf eine Datei zugreifen kann, muss sie zunächst geöffnet werden. Anschließend kann man solange auf die Datei zugreifen, bis sie wieder geschlossen wird. Nachfolgend sind wichtige Funktionen aus der *stdio*-Bibliothek zum sequentieller Bearbeiten von Dateien aufgelistet:

`fopen`	Datei öffnen
`fclose`	Datei schließen
`feof`	Überprüfung auf Dateiende (End of File)
`fflush`	Zwischenpuffer in Datei wegschreiben
`fgetc`	Einzelnes Zeichen aus Datei lesen
`fread`	Feld von Zeichen aus Datei lesen
`fgets`	Zeichenkette aus Datei lesen (terminiert mit '\0')
`fscanf`	Formatierte Eingabe aus einer Datei
`fputc`	Einzelnes Zeichen in Datei schreiben
`fwrite`	Feld von Zeichen in Datei schreiben
`fputs`	Zeichenkette in Datei schreiben (terminiert mit '\0')
`fprintf`	Formatierte Ausgabe in eine Datei
`fseek`	Dateizeiger verschieben
`rewind`	Dateizeiger zurücksetzen
`ftell`	Dateizeiger abfragen

Beim Aufruf von **`fopen`** prüft das Betriebssytem, ob man berechtigt ist auf die angegebene Datei zuzugreifen. Wird das Öffnen der Datei vom Betriebssystem erlaubt, erzeugt **`fopen`** eine Kontrollstruktur **`FILE`**, in der alle Informationen über die geöffnete Datei verwaltet werden. Dazu gehört der **Dateizeiger**, der die Position des nächsten zu lesenden oder zu schreibenden Datenbytes in der Datei angibt. Zeigt der Dateizeiger hinter das letzte Byte der Datei, so bewirkt ein Lesen der Datei eine **`EOF`**-Meldung. Beim Schreiben werden in diesem Fall die Datenbytes hinten an die Datei angehängt und damit die Datei erweitert.

Der Dateizeiger wird durch jeden Lese- oder Schreibvorgang um die Anzahl der gelesenen oder geschriebenen Bytes nach vorne bewegt. Somit erfolgt ein sequentieller Dateizugriff. Soll von diesem sequentiellen Zugriff abgewichen werden, kann der Dateizeiger mit **`rewind`** und **`fseek`** direkt bewegt werden. Zusätzlich kann mit **`ftell`** die Position des Zeigers abgefragt werden.

Diese Struktur vom Typ **`FILE`** ist in der Include-Datei „stdio.h“ deklariert. Beim Öffnen einer Datei richtet die Funktion **`fopen`** eine Variable vom Typ **`FILE`** ein, füllt sie mit den zu Datei gehörigen Werten und liefert einen Zeiger auf diese **`FILE`**-Struktur als *return*-Parameter zurück.

7.1 Wichtige Dateifunktionen

7.1.1 Funktion `fopen`

Beschreibung:

Öffnet eine Datei zum Lesen oder zum Schreiben.

Syntax:

```
#include <stdio.h>
FILE *fopen(char *name, char *modus);
```

Bedeutung der Parameter:

`name:` Zeiger auf Zeichenkette, welche den Dateinamen enthält.

`modus:` Zeiger auf Zeichenkette, welche die Zugriffsart auf die Datei beschreibt (Mode-String). Diese kann die Zeichen **`r`**, **`w`**, **`a`**, **`+`**, **`b`** und **`t`** mit folgender Bedeutung enthalten:

- **`r`** Öffnen ausschließlich für Leseoperationen
- **`w`** Erzeugung für Schreiboperationen. Wenn eine Datei dieses Namens bereits existiert, wird sie überschrieben.
- **`a`** Erzeugung für Schreiboperationen oder, falls die Datei bereits existiert, Öffnen für anfügende Schreiboperationen am Dateiende.
- **`r+`** Öffnen einer bereits existierenden Datei für Lese- und Schreiboperationen
- **`w+`** Erzeugung einer neuen Datei für Lese- und Schreiboperationen. Wenn eine Datei dieses Namens bereits existiert, wird sie überschrieben.
- **`a+`** Öffnen einer Datei für Leseoperationen und anfügende Schreiboperationen am Dateiende. Falls die Datei noch nicht existiert, wird sie erzeugt.

Soll eine Datei im Text-Modus geöffnet oder erzeugt werden, wird **`t`** an den String angehängt oder eingefügt.

Beispiele: **`"rt"`**, **`"w+t"`**, **`"rt+"`**

Soll eine Datei im Binär-Modus geöffnet oder erzeugt werden, wird **`b`** an den String angehängt oder eingefügt.

Beispiele **`"wb"`**, **`"a+b"`**, **`"rt+"`**

Rückgabewert:

Im Erfolgsfall liefert **`fopen`** einen Zeiger auf eine gültige **`FILE`**-Struktur zurück. Bei Fehlern wird ein **`NULL`**-Zeiger als *return*-Wert zurückgegeben.

7.1.2 Funktion `fclose`

Beschreibung:

Schließt eine vorher geöffnete Datei.

Syntax:

```
#include <stdio.h>
int fclose(FILE *file);
```

Bedeutung der Parameter:

`file:` Zeiger auf gültige **`FILE`**-Struktur

Rückgabewert:

Bei Erfolg wird der Wert 0 zurückgegeben, ansonsten der Wert EOF.

Bemerkung:

Beim Terminieren eines Programms werden alle Dateien, die im Programm geöffnet aber nicht geschlossen wurden, automatisch vom Betriebssystem geschlossen.

7.1.3 Funktion `feof`

Beschreibung:

Diese Funktion überprüft, ob der Dateizeiger am Ende der Datei steht.

Syntax:

```
#include <stdio.h>
int feof(FILE *file);
```

Bedeutung der Parameter:

`file:` Zeiger auf gültige **`FILE`**-Struktur

Rückgabewert:

Wenn Dateizeiger innerhalb der Datei steht: 0.
Wenn Dateizeiger das Dateiende erreicht hat: Ungleich 0.

7.1.4 Funktion `fflush`

Beschreibung:

Bewirkt das physikalische Schreiben von gepufferten Daten in die zugeordnete Datei.

Syntax:

```
#include <stdio.h>
int fflush(FILE *file);
```

Bedeutung der Parameter:

`file:` Zeiger auf gültige **`FILE`**-Struktur

Rückgabewert:

Bei fehlerfreier Ausführung: 0
im Fehlerfall: EOF

7.1.5 Funktion `fgetc`

Beschreibung:

Liest ein Zeichen aus einer Datei.

Syntax:

```
#include <stdio.h>
int fgetc(FILE *file);
```

Bedeutung der Parameter:

`file`: Zeiger auf gültige **`FILE`**-Struktur

Rückgabewert:

Bei fehlerfreier Ausführung: Das gelesene Zeichen, nachdem es in einen vorzeichenlosen Integer-Wert umgewandelt wurde.

Bei Dateiende oder einem Fehler: EOF

7.1.6 Funktion `fread`

Beschreibung:

Lesen mehrerer Datenobjekte aus einer Datei.

Syntax:

```
#include <stdio.h>
size_t fread(void *buffer, size_t size,
             size_t number, FILE *file);
```

Bedeutung der Parameter:

`buffer:`	Speicherbereich zum Einlesen von Daten aus der Datei. Die Größe diese Bereichs muss mindestens **`size`*****`number`** betragen.
`size:`	Größe eines einzelnen zu lesenden Datenobjekts
`number:`	Anzahl der zu lesenden Datenobjekte
`file:`	Zeiger auf gültige **`FILE`**-Struktur

Rückgabewert:

Anzahl der gelesenen Objekte der Größe **`size`**.

7.1.7 Funktion `fgets`

Beschreibung:

Liest eine mit '\0' terminierte Zeichenkette aus einer Datei.

Syntax:

```
#include <stdio.h>
char *fgets(char *buffer, int maxlength,
            FILE *file);
```

Bedeutung der Parameter:

`buffer`	Zeiger auf Puffer zur Aufnahme der Zeichenkette.
`maxlength`	Größe des Puffers.
`file`	Zeiger auf gültige **`FILE`**-Struktur.

Rückgabewert:

Bei erfolgreichem Lesen: Zeiger auf Puffer.

Bei Fehler oder **`EOF`**: **`NULL`**-Zeiger.

7.1.8 Funktion `fscanf`

Beschreibung:

Formatiertes Lesen aus einer Datei.

Syntax:

```
#include <stdio.h>
int fprintf(FILE *file, const char *format, ...);
```

Bedeutung der Parameter:

`file`	Zeiger auf gültige **`FILE`**-Struktur.
`format`	Format-String (entsprechend **`scanf`**).
`...`	Parameteradressen entsprechend dem Format-String.

Rückgabewert:

Anzahl der eingelesenen Parameter.

7.1.9 Funktion `fputc`

Beschreibung:

Schreiben eines Zeichens in eine Datei.

Syntax:

```
#include <stdio.h>
int fputc(int character, FILE *file);
```

Bedeutung der Parameter:

`file`	Zeiger auf gültige **`FILE`**-Struktur.
`character`	Zu schreibendes Zeichen.

Rückgabewert:

Bei fehlerfreier Ausführung: Das geschriebene Zeichen.

Im Fehlerfall: **`EOF`**.

7.1.10 Funktion `fwrite`

Beschreibung:

Schreiben mehrerer Datenobjekte in eine Datei.

Syntax:

```
#include <stdio.h>
size_t fwrite(void *buffer, size_t size,
              size_t number, FILE *file);
```

Bedeutung der Parameter:

`buffer:`	Speicherbereich, der die zu schreibenden Objekte enthält. Die Objekte müssen direkt hintereinander liegen und belegen somit einen Bereich von **`size`*****`number`** Bytes.
`size:`	Größe eines einzelnen zu schreibenden Datenobjekts
`number:`	Anzahl der zu schreibenden Datenobjekte
`file:`	Zeiger auf gültige **`FILE`**-Struktur

Rückgabewert:

Anzahl der geschriebenen Objekte der Größe **`size`**.

7.1.11 Funktion `fputs`

Beschreibung:

Schreiben einer mit '\0' terminierten Zeichenkette in eine Datei.

Syntax:

```
#include <stdio.h>
int fputs(const char *string, FILE *file);
```

Bedeutung der Parameter:

`file:`	Zeiger auf gültige **`FILE`**-Struktur
`string`	Zeiger auf die zu schreibende Zeichenkette.

Rückgabewert:

Bei fehlerfreier Ausführung: Einen positiven Wert.
Im Fehlerfall: **`EOF`**.

7.1.12 Funktion `fprintf`

Beschreibung:

Formatiertes Schreiben in eine Datei.

Syntax:

```
#include <stdio.h>
int fprintf(FILE *file, const char *format, ...);
```

Bedeutung der Parameter:

`file`	Zeiger auf gültige **`FILE`**-Struktur.
`format`	Format-String (entsprechend **`printf`**).
`...`	Parameter entsprechend dem Format-String.

Rückgabewert:

Anzahl der geschriebenen Zeichen.

7.1.13 Funktion `fseek`

Beschreibung:

Verschieben des Dateizeigers.

Syntax:

```
#include <stdio.h>
int fseek(FILE *file, long offset, int mode);
```

Bedeutung der Parameter:

`file`	Zeiger auf gültige **`FILE`**-Struktur.
`offset`	Offset zum Verschieben des Dateizeigers.
`mode`	Folgende Modi sind vorgesehen: **`'SEEK_SET'`**: Der Offset wird vom Dateianfang aus gerechnet. Der Wert des Offset sollte positiv sein. **`'SEEK_CUR'`**: Der Offset wird von der aktuellen Position des Dateizeigers aus gerechnet. **`'SEEK_END'`**: Der Offset wird vom Dateiende aus gerechnet. Der Wert des Offset sollte negativ sein.

Rückgabewert:

Bei fehlerfreier Ausführung: 0.

Im Fehlerfall: ungleich 0.

7.1.14 Funktion `rewind`

Beschreibung:

Dateizeiger auf den Anfang einer Datei setzen.

Syntax:

```
#include <stdio.h>
void rewind(FILE *file);
```

Bedeutung der Parameter:

`file` Zeiger auf gültige **`FILE`**-Struktur.

7.1.15 Funktion `ftell`

Beschreibung:

Abfrage der Position des Dateizeigers.

Syntax:

```
#include <stdio.h>
long ftell(FILE *file);
```

Bedeutung der Parameter:

`file` Zeiger auf gültige **`FILE`**-Struktur.

Rückgabewert:

Bei fehlerfreier Ausführung: Position des Dateizeigers.
Im Fehlerfall: -1L.

7.2 Beispiele

Beispiel: Einlesen von Messwerten und Ermittlung des Mittelwerts

```
#include <stdio.h>
double mittelwert(FILE *fp);
int main(){
    double wert;
    FILE *fp;
    printf("Mittelwertberechnung, Werte\n");
    printf("zeilenweise eingeben, Ende EOF\n\n");
    fp=fopen("datei.dat", "w");
    while(1==scanf("%lf", &wert)){
        fprintf(fp, "%lf\n", wert);
    }
    fclose(fp);
    /* Mittelwert berechnen und ausgeben */
    fp=fopen("datei.dat", "r");
    printf("Mittelwert : %lf", mittelwert(fp));
    return 0;
}
```

```
double mittelwert(FILE *fp)
{
    double summe=0,wert;
    int n=0;
    while (feof(fp)==0){
        if (fscanf(fp, "%lf", &wert)==1) {
           n++;
           summe+=wert;
        }
    }
   return summe/n;
}
```

Beispiel: Einfaches Kopierprogramm

```
#include <stdio.h>
#define BLOCKWEISE_KOPIEREN 1
int main(int argc, char* argv[]) {
    FILE *in, *out;
    int ch;
#if BLOCKWEISE_KOPIEREN
    char buffer[1000];
    size_t n;
#endif
    if (argc!=3) {
        printf("Verwendung: <Quelle> <Ziel>\n");
        return 1;   /* Ende mit Fehler */
    }
    if ((in = fopen(argv[1], "rb"))==NULL) {
        fprintf(stderr, "Kann %s nicht
                         oeffnen\n", argv[1]);
        return 1;   /* Ende mit Fehler */
    }
    if ((out = fopen(argv[2],"wb"))==NULL) {
        fprintf(stderr, "Kann %s nicht
                         oeffnen\n", argv[2]);
        return 1;   /* Ende mit Fehler */
    }
```

```
    while (!feof(in)) {
#if BLOCKWEISE_KOPIEREN
        n = fread(buffer, 1, 1000, in);
        if (n>0)
            fwrite(buffer, 1, n, out);
#else
    ch=fgetc(in); /* Zeichenweise kopieren */
    if (EOF!=ch) {
        fputc(ch, out);
     }
     else
         break;

#endif
    }  /* of while */
    fclose(in);
    fclose(out);
    return 0;     /* normales Ende */
}  /* of main */
```

7.3 Standardeingabe und Standardausgabe

Tastatur und Bildschirm werden in C auch über **FILE**-Strukturen angesprochen.
Jedes C-Programm hat standardmäßig Zugriff auf die folgenden drei Datenströme vom Typ **FILE ***:

stdin	Standardeingabe (Tastatur)
stdout	Standardausgabe (Bildschirm)
stderr	Standardfehlerausgabe (Bildschirm)

Die Deklaration dieser drei FILE-Strukturen findet sich in der Datei *„stdio.h“*
Diese drei (Pseudo)-Dateien werden vor Aufruf der main-Funktion vom Betriebssystem geöffnet. Ohne Aufruf der **fopen**-Funktion kann somit mit den übrigen Dateifunktionen auf diese drei Datenströme zugegriffen werden.

Beispiel:

Der folgende Aufruf der Ausgabefunktion printf:

```
printf("Hallo Welt\n");
```

kann auch wie folgt programmiert werden:

```
fprintf(stdout, "Hallo Welt\n");
```

Beide Programmzeilen sind äquivalent.

8 Zeiger und Felder

8.1 Zeiger und Adressen

Bei einer typischen Maschine kann man sich den Speicherbereich als Bereich von Speicherzellen vorstellen, die fortlaufend nummeriert und adressierbar sind.

Wir haben Zeiger (engl. „pointer“) bereits im Zusammenhang mit Zeichenketten kennengelernt. Zeiger sind Variablen, in der sich die Adresse eines bestimmten Datenobjektes oder einer Funktion befindet. Ihr Speicherbedarf hängt vom Prozessor ab und beträgt meist 2 oder 4 Byte.

Die Verwendung von Zeigern bringt verschiedene Vorteile:

- Speicherersparnis
- Übergabe an Funktionen (call by reference)
- Dynamische Speicherverwaltung
- Rekursive Datenstrukturen
- Effiziente Implementierung von Problemen/Aufgaben (z.B. Sortieren, Zeiger auf Funktionen,...)

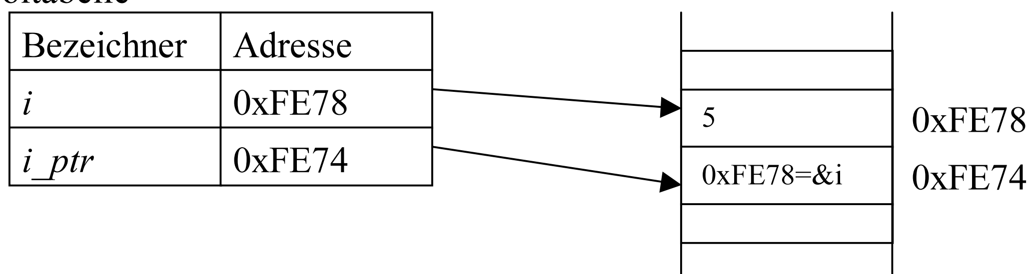
Die Programmierung mit Zeigern erfolgt mit Hilfe der unären Operatoren:

& Adressoperator
***** Inhaltsoperator

Ein Zeiger muss wie jede andere Variable deklariert und definiert werden:

```
int  i;
int  *i_ptr;
i_ptr= &i;
```

Symboltabelle



Bezeichner	Adresse
i	0xFE78
i_ptr	0xFE74

Allgemein:

```
<Typ> *<zeigervariable>;
```

<Typ>: Datentyp des Objektes, auf die der Zeiger verweist.
*****: Kennzeichnet die Variable als Zeigervariable
<zeigervariable>: Name der Zeigervariablen

Die **Initialisierung** eines Zeigers erfolgt durch die Zuweisung einer Adresse. Ein nicht initialisierter Zeiger enthält undefinierte Bitmuster, die bei Dereferenzierung oftmals zu Programmabstürzen führt!

Beispiele:

```
float x=1, y=2;
float *xf1=&x, *xf2;

*xf2=5.0; /* Fehler, xf2 nicht initialisiert */
xf2=NULL; /* NULL initialisieren (Adresse 0) */
y=*xf1;
*xf1=0;
++*xf1;
(*ip)++;
```

Man beachte: Jeder Zeiger zeigt auf einen festgelegten Datentyp. (Ausnahme: Zeiger auf *void*. Ein *void*-Zeiger kann jeden beliebigen Typ aufnehmen.)

8.2 Zeigerarithmetik

Mit Zeigern sind eingeschränkte Rechenoperationen auf der Grundlage der Speichereinheiten des Basistyps möglich.

Beispiel:

```
int *pa, *pb, n;
```

Vergleich zweier Zeiger	`pa<pb`
Addition eines `int`-Wertes zu einem Zeiger	`pa+n, pa++, ++pa`
Subtraktion eines `int`-Wertes von einem Zeiger	`pa-n`
Subtraktion zweier Zeiger gleichen Typs	`pa-pb`

Bei der Addition und Subtraktion von *int*-Werten zu Zeigern sowie beim Erhöhen und Erniedrigen eines Zeigers durch den `++` und `--` Operator wird der Zeiger um den *int*-Wert multipliziert mit der Länge des zugeordneten Datentyps erhöht. Nachfolgendes Beispiel erläutert diesen Zusammenhang.

Beispiel:

```
short *pa=0x2000;    /* pa auf Adresse 0x2000 */
long  *pb=0x2000;    /* pb auf Adresse 0x2000 */
pa++;      /* pa zeigt nun auf Adresse 0x2002 */
pb++;      /* pb zeigt nun auf Adresse 0x2004 */
pa = pa+3; /* pa zeigt nun auf Adresse 0x2008 */
pb = pb+3; /* pb zeigt nun auf Adresse 0x2010 */
```

Weiterhin kann die Differenz zweier Zeiger gebildet werden, wenn beide Zeiger von gleichem Datentyp sind. Das Ergebnis ist vom Typ Integer und gibt den Abstand zwischen den beiden Zeigern in Einheiten der Objekte des zum Zeiger gehörigen Datentyps an.

Beispiel:

```
char *a=0x1000,*b=0x1008;
long *c=0x1000,*b=0x1008;
int i, j;
i=b-a; /* i wird der Wert 8 zugewiesen */
j=d-c; /* j wird der Wert 2 zugewiesen */
```

Beispiel: Neue Variante von strlen:

```
int strlen(char *s){
   char *p=s;
   while(*p!='\0') p++;
   return p-s;
}
```

Zu beachten ist, dass die unären Operatoren **`*`**, **`&`**, **`++`** und **`--`** von rechts nach links ausgewertet werden:

`*pa++`	**`pa++`** liefert als Ergebnis den bisherigen Wert des Zeigers, der danach um eine Typeinheit erhöht wird. **`*pa++`** dereferenziert also das Objekt, auf das **`pa`** vor dem Erhöhen zeigt.
`(*pa)++`	Erst wird der **`*`**-Operator auf **`pa`** angewendet, dann der **`++`**-Operator auf das dereferenzierte Objekt. Der Zeiger bleibt unverändert. Man greift also auf das Objekt zu, auf das **`pa`** gerade zeigt und erhöht dessen Wert um 1.
`*++pa`	Der Wert von **`pa`** wird erst erhöht. Dann wird auf das Objekt zugegriffen, auf welches der erhöhte Zeiger **`pa`** zeigt.
`++*pa`	Das Objekt, auf das **`pa`** zeigt, wird erst um 1 erhöht und dann verwendet.

Weitere Operationen (z.B. Addition und Multiplikation von Zeigern) sind verboten.

Eine Besonderheit ist der **void-Zeiger**. Er enthält die Adresse einer Speicherzelle ohne Festlegung des Datentyps.

Beispiel:

```
void *x, *y;
double d;
long int l;
x=&d;
y=&l;
```

Beliebige Zeiger können ohne Verwendung des cast-Operators in **`void`**-Zeiger umgewandelt und wieder zurückgewandelt werden (sonst ist immer der cast-Operator erforderlich). Erhöhung und Erniedrigen eines **`void`**-Zeigers erfolgt in 1 Byte-Schritten (wie ein Zeiger auf **`char`**).

8.3 Zusammenhang zwischen Feldern und Zeigern

Gleichartige, logische zusammenhängende Objekte können zu Feldern (Array, Vektor) zusammengefasst werden (siehe Abschnitt 4).

Beispiel (siehe Abschnitt 4):

```
int n[5];
n[0]=6;
n[1]=4;
```

Es wird ein Feld namens **n** mit 5 *integer*-Zahlen vereinbart, d.h. es wird ein zusammenhängender Speicherplatz für 5 *integer*-Zahlen reserviert.

n:	n[0]	n[1]	n[2]	n[3]	n[4]
	6	4	?	?	?

Bemerkungen:

- Das erste Feldelement eines Feldes aus N Elementen hat den Index 0, das letzte Element hat den Index N-1.
- In C werden mögliche Bereichüberschreitungen nicht geprüft. Mit:
  ```
  n[5]=7;
  ```
  wird beispielsweise der Speicherplatz hinter **n[4]** (siehe oberes Beispiel) zerstört (→ Laufzeitfehler).
- Der Name des Feldes ohne nachfolgende Indizierung **[...]** hat die Bedeutung einer Adresse. Der Name stellt jedoch einen konstanten Zeiger und keine Zeigervariable dar. Ist **n** der Name eines Feldes sind Ausdrücke wie **n++** und **n=zn** (Mit **zn** als Zeigervariable) somit nicht erlaubt.
- **n** entspricht **&n[0]**.
- **n+i** entspricht **&n[i]**.
- **n[i]** entspricht ***(n+i)**.
- Die (absolute) Adresse des i-ten Feldelementes **n[i]** aus obigem Beispiel lässt sich berechnen zu:

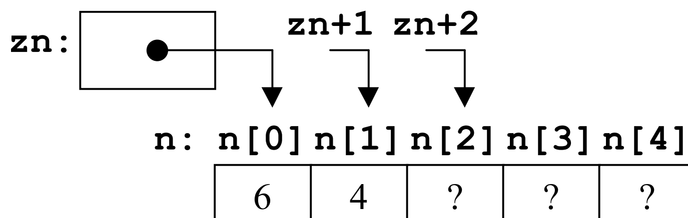
  *Adresswert(n) + i*sizeof(int).*

  Dies ist jedoch kein C-Code!!

Man kann auch mit Hilfe von Zeigervariablen auf die Elemente eines Feldes zugreifen.

Beispiel:

```
int n[5];
int *zn;
zn=n;         /* oder zn=&n[0] */
*zn     = 6;
*(zn+1) = 4;
```



Der Zeigerwert **zn** zeigt auf das erste Element **n[0]**, **zn+1** zeigt auf das zweite Element **n[1]** usw. Somit entspricht ***(zn)** dem Feldzugriff **n[0]** und ***(zn+1)** entspricht **n[1]** usw.
Wichtig: Feldname und Zeigervariable sind zu unterscheiden. Der Zeiger ist eine Variable mit zugewiesenem Speicherplatz. Ausdrücke wie **zn=n** und **zn++** sind erlaubt.

Ein Feldname stellt in C einen konstanten Zeiger dar. Zum Zugriff auf ein Feldelement wird beim **[]**-Operator der Feldname und der Index des gewünschten Elements angegeben (z.B. **n[3]**). Der []-Operator fordert jedoch keineswegs einen konstanten Zeiger, sondern es kann ein beliebiger Zeiger angegeben werden. Somit kann z.B. bei gegebener Zeigervariablen **zn** statt mit ***(zn+3)** auch mittels **zn[3]** auf das dritte Feldelement zugegriffen werden.

Beispiel:

```
int main(){
   int n[10], *zn;
   zn=n;   /* Wichtig: Zeiger initialisieren */
   zn[0]=6;
   zn[1]=7;
   printf("Ergebnis %i %i\n", n[0], n[1]);
   return 0;
}
```

Wichtig ist jedoch, dass die Zeigervariable vorher korrekt initialisiert wurde und wirklich auf ein Feld im Speicher schreibt. Ansonsten erfolgt der Feldzugriff auf eine zufällige Stelle im Speicher (→ Laufzeitfehler).

8.4 Felder als Funktionsparameter

Bei der Übergabe von Feldern an Funktionen gelten die folgenden Bemerkungen:

- Von einem Feld wird die Adresse als Parameter an eine Funktion übergeben.
- Zur Deklaration des zu einem Feld zugehörigen Funktionsparameters sind zwei Formen äquivalent: Eine Zeigerdeklaration oder eine Felddeklaration ohne spezifizierte Größe.
  Beispiel: „**`double *v`**" und „**`double v[]`**" sind äquivalent.
- Die Dimension von Feldern wird nicht implizit mit übergeben. Wird die Dimension eines Feldes in der Funktion benötigt, muss sie über einen zusätzlichen Parameter separat übergeben werden.

Programmbeispiel zur Übergabe eines Feldes an eine Funktion:

```
#define MAXDIM 10

double skalarprodukt(double *, double *, int);

int main() {
    double x[MAXDIM], y[MAXDIM], produkt;
    int result, n=0;
    while(n<MAXDIM && EOF!=scanf("%lf %lf", &x[n],&y[n])) {
        ++n;
    }
    produkt=skalarprodukt(x, y, n);
    printf("%lf", produkt);
    return 0;
}

double skalarprodukt(double *v, double *w,
                     int dim) {
   double skprod=0;
   int i;
   for(i=0; i<dim; i++) {
       skprod+=v[i]*w[i];
   }
   return (skprod);
}
```

Beispiel: Stack mit den Funktionen *pop*, *push* und *stapel_aus*

```
#include <stdio.h>
#define MAX 10

int stapel[MAX], top=0;  /* globale Variable */
int push(int wert);
int pop(void);
int stapel_aus(void);

int main() {
    int zahl;
    char ch;
    printf("Stapel, Auswahl durch grossen Buchstaben
            des Befehls:\n\n");
    while (1) {
        printf("pUsh, pOp, Ausgabe, Exit: ");
        scanf("%c", &ch);
        if (ch=='\n') {
            scanf("%c", &ch);
        }
        if(ch=='u' || ch=='U') {
            printf("Bitte Zahl eingeben     : ");
            scanf("%d", &zahl);
            push(zahl);
        }
        else if(ch=='o' || ch=='O') {
            printf("oberstes Element: %d\n", pop());
        }
        else if (ch=='a' || ch=='A') {
            stapel_aus();
        }
        else if(ch== 'e' || ch=='E') {
            return 0;
        }
        else {
            printf("\nFehler: Falsche Eingabe\n");
        }
        return 0;
    }  /* of while */
} /* of main
```

```
int push(int wert){
   if(top==MAX) {
      printf("Stapelueberlauf! ");
      return 1;
   }
   else {
      stapel[top++]=wert;
      return 0;
   }
}
int pop(void){
   if(top == 0) {
      puts("Stapel ist leer");
      return 2;
   }
   else {
      return stapel[--top];
   }
}
int stapel_aus(void) {
   int zaehl;
   if (top==0) {
      puts("Stapel ist leer\n");
      return 1;
   }
   else {
      printf("\nInhalt des Stapels:    \n");
      for(zaehl=top-1; zaehl>=0; zaehl--){
         printf("              %d\n", stapel[zaehl]);
      }
      printf("\n");
      return 0;
   }  /* of if
}   /* of stapel_aus */
```

8.5 Mehrdimensionale Felder

Mehrdimensionale Felder sind in C als Felder von Feldern definiert.

Syntax (2-dimensionales Feld):

Definition (*n*, *m*: int-Werte):

`Speicherklasse Typ Name [n][m];`

Aufruf (Zeilenindex *i*: $0 \leq i < n$; Spaltenindex *j*: $0 \leq j < m$):

`Name[i][j]`

Beispiel:

```
int n[2][3];
n[0][0]=7;
n[0][1]=5;
n[0][2]=7;
n[1][0]=8;
n[1][1]=1;
n[1][2]=4;
```

$$n = \begin{pmatrix} 7 & 5 & 7 \\ 8 & 1 & 4 \end{pmatrix}$$

Die Elemente des mehrdimensionalen Feldes werden, beginnend mit der niedrigsten, linksstehenden Dimension zeilenweise gespeichert:

	1000	1004	1008	1012	1016	1020	Adresse
n→	7	5	7	8	1	4	Wert
	n[0][0]	**n[0][1]**	**n[0][2]**	**n[1][0]**	**n[1][1]**	**n[1][2]**	
	↑			↑			
	n[0]			**n[1]**			

Bedeutung des Namens **n** bei mehrdimensionalen Feldern

Der Feldname **n** in obigem Beispiel steht (wie in Abschnitt 8.3 erläutert) für die Adresse des ersten Feldelements. Da **n** im zweidimensionalen Fall ein Feld von Feldern ist, zeigt dieser konstante Zeiger **n** im Beispiel auf ein Feld von 3 Elementen vom *integer*-Typ. Die Variable **n** ist daher vom Typ „Zeiger auf Feld von 3 Elementen", welcher wie folgt in C beschrieben werden kann:

```
int *[3]
```

oder

```
int [][3].
```

Der Ausdruck ***n** liefert einen Zeiger auf das Feld **n[0]**, der Ausdruck ***(n+1)** zeigt entsprechend auf das Feld **n[1]**. Allgemein referenziert der Ausdruck **n[i]** das i+1-te Feldelement von **n**. Im Beispiel referenziert der Ausdruck **n[i]** ein eindimensionales Feld aus *integer*-Elementen. Er steht somit für einen konstanten Zeiger auf Elemente des Typs *integer*, der auf die Adresse des ersten Elements **n[i][0]** zeigt. Auf die skalare Komponente **n[i][j]** des zweidimensionalen Feldes kann somit alternativ wie folgt zugegriffen werden:

n[i][j]	Feldelement **j** von Feld (Feldelement **i** von Feld **n**)
***(*(n+i)+j)**	(Inhalt des Elements mit Abstand **j** von Zeiger (Inhalt des Elements mit Abstand **i** von Zeiger **n**))
(*(n+i))[j])	Feldelement **j** von Feld (Inhalt des Elements mit Abstand **i** von Zeiger **n**)
***(n[i]+j)**	(Inhalt des Elements mit Abstand **j** von Zeiger (Feldelement **i** von Feld **n**))

Der Bezeichner **n** und der Ausdruck **n[0]** repräsentieren beide konstante Zeiger, welche auf die gleiche physikalische Adresse zeigen. Sie sind jedoch keineswegs äquivalent, da sie unterschiedlichen Typen zugeordnet sind:

Typ von **n**: **int [][3]** oder **int *[3]**
Typ von **n[i]**: **int [3]**

Der erste Zeiger **n** zeigt im Beispiel auf die Adresse des ersten Feldes **n[0]** aus *integer*-Elementen. Dieses Feld beginnt im Speicher an der Adresse seines ersten Elementes **n[0][0]**. Der zweite Zeiger **n[0]** zeigt auf die Adresse seines ersten Elements **n[0][0]**. Somit wird beiden Zeigern der gleiche Adresswert zugeordnet, obwohl sie von unterschiedlichem Typ sind.
Der unterschiedliche Typ kommt jedoch zum Tragen, wenn zum Zeigerwert z.B. der Wert 1 hinzu addiert wird (Zugriff auf das zweite Element).

<u>Beispiel:</u>

Annahme: **n** und **n[0]** zeigen auf Adresse *1000*.
n+1 entspricht dann der Adresse *1012*
n[0]+1 entspricht dann der Adresse *1004.*

<u>Beispiel:</u> Initialisierung eines Feldes und Zugriff über verschiedene Ausdrücke

```
#include <stdio.h>

int main() {
   double a[2][3]={{.1,12.0,1.3},{2.1,2.2,2.3}};
   printf("%f=%f=%f=%f\n", a[1][1], (*(a+1))[1],
          *(a[1]+1),*(*(a+1)+1));
}
```

<u>Beispiel:</u> Transponieren einer Matrix

```
#define MAXDIM 100 /* Anzahl Zeilen/Spalten */

void transpo(int *, int *, double [][MAXDIM]);

int main() {
   int zeilen, spalten;
   double A[MAXDIM][MAXDIM]);
   ...
   transpo(&zeilen, &spalten, A);
   ...
}
```

```
void transpo(int *n, int *m, double A[][MAXDIM]){
                     /* oder double (*A)[MAXDIM] */
   int i, j, max=(*m>*n)? *m : *n;
   double hilf;
   for (i=0; i<max; i++){
      for (j=0; j<i; j++) {
         hilf=A[i][j];
         A[i][j]=A[j][i]; /* Transponieren */
         A[j][i]=hilf;
      }
   }
   i=*n;
   *n=*m; /* Austausch der Dimensionen */
   *m=i;
}
```

8.6 Felder von Zeigern

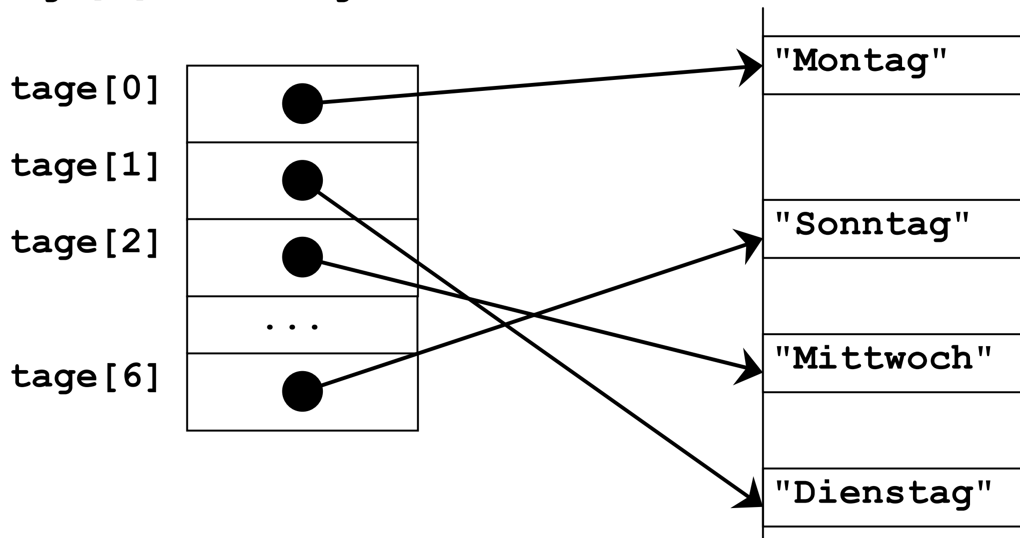
In C können auch Felder von Zeigern definiert werden. Felder von Zeigern werden in C sehr häufig verwendet. Ein solches Feld enthält nur Adressen. Die Adressen müssen alle vom selben Typ sein.
Die Definition:

```
short *aptr[10];
```

ist beispielsweise die Definition eines Feldes, das 10 Zeiger vom Typ „Zeiger auf *short*-Variable“ enthält. Jedes Element dieses Feldes (**aptr[0]**, ..., **aptr[9]**) ist vom Typ **short ***.
Felder von Zeigern werden häufig im Zusammenhang mit Zeichenketten verwendet (siehe Abschnitt 4). Dort hat jedes Feldelement den Typ „Zeiger auf *character*-Variable“ (bzw. **char***) und kann auf das erste Element einer Zeichenkette verweisen.

Beispiel: Zeigerfeld für Zeichenketten

```
char *tage[7];
tage[0]="Montag";
tage[1]="Dienstag";
tage[2]="Mittwoch";
tage[3]="Donnerstag";
tage[4]="Freitag";
tage[5]="Samstag";
tage[6]="Sonntag";
```



Der Zugriff auf die Texte geschieht, wie für Feldern fester Größe oben erläutert, mittels Indizierung oder mittels Zeigerarithmetik.

	Ausgabe der `printf`-Anweisungen:
`printf("%s\n", tage[0]);`	`Montag`
`printf("%s\n", *(tage+1));`	`Dienstag`
`printf("%c", *tage[0]);`	`M`
`printf("%c", *(tage[3]+1));`	`Mo`
`printf("%c", *(*(tage+1)+3));`	`Mon`
`printf("%c", tage[3][0]);`	`MonD`
	↑(Hier steht der Cursor)

Ein Feld von *character*-Zeigern auf Zeichenketten ist ein spezieller Fall des Feldes von Zeigern, die auf die Anfangselemente von Feldern verweisen. Diese Art der Verwaltung von Feldern (z.B. Zeichenketten) bietet einige Vorteile bei der Verarbeitung von Feldern:

- Felder unterschiedlicher und beliebiger Länge können über das Zeigerfeld zusammengefasst werden.
- Beim Sortieren der Felder ist nur ein Vertauschen von Zeigerinhalten (Adressen) im Zeigerfeld erforderlich.
- Es können mehrere Zeigerfelder definiert werden, welche auf die gleichen Felder

zeigen, diese aber nach unterschiedlichen Kriterien sortieren.

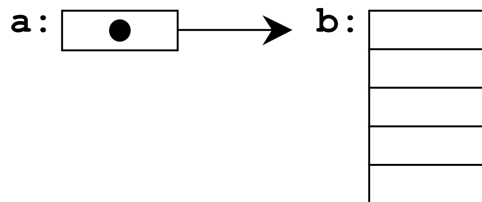
Nicht zu verwechseln ist ein Feld von Zeigern mit mehrdimensionalen Feldern:

```
char  a[10][20]; /* Speicherung von 200 char-Werten */
char *a[10];     /* Speicherung von 10 Zeigern auf char, Texte
                    können unterschiedlich lang sein */
```

In beiden Fällen wird auf das Element mit Index (2,3) mit **a[2][3]** zugegriffen.

<u>Zeiger auf Felder:</u>
In Ergänzung zum Feldnamen, der einen konstanten Zeiger repräsentiert, kann auch ein variabler Zeiger auf ein Feld vereinbart werden.
<u>Beispiel:</u>

```
long int (*a)[5]; /* Zeiger auf ein Feld
                     mit 5 Elementen.
                     Achtung: Es wird kein
                     Feld definiert */
long int b[5];    /* Feld mit 5 Elementen */
...
a=&b; /* Zuweisung der Feldadresse zum Zeiger*/
b[1]=(*a)[0]; /* (Element 1) = (Element 0) */
```



Zum Abschluss soll nochmals auf den Unterschied zwischen einem <u>Feld von Zeigern:</u>

```
long int *f[3];   /* ein Feld von Zeigern */
```

und einem <u>Zeiger auf ein Feld:</u>

```
long int (*z)[3]; /* Zeiger auf ein Feld  */
```

hingewiesen werden. Nachfolgendes Diagramm verdeutlicht den Unterschied:

<u>Feld von Zeigern</u> <u>Zeiger auf Feld</u>

8.7 Zeiger auf Funktionen

In C gibt es neben Zeiger auf Datenobjekte auch Zeiger auf Funktionen.
Ein Zeiger auf eine Funktion hat als Wert die Anfangsadresse des Unterprogramms in Maschinensprache, das die Funktion ausführt.

Beispiel:

Die Definition:

```
int (*f) (double);
```

bezeichnet einen Zeiger mit Bezeichner **f** auf eine Funktion, die einen **double**-Wert als Argument verlangt und einen **int**-Wert als Rückgabewert zurückliefert.

Die Klammerung **(*f)** ist notwendig, da der **()**-Operator stärker als der *****-Operator bindet. Die sonst entstehende Definition **int *f(double)** bezeichnet eine Funktion, die einen *integer*-Zeiger als Rückgabewert liefert.

Zeiger auf Funktionen werden dort angewendet, wo Funktionen als Eingabeparameter erforderlich sind (z.B. bei numerischen Routinen wie Integralberechnung, Differentialgleichungs-Lösern,...) .

Der Name einer C-Funktion hat die Bedeutung eines konstanten Zeigers auf die Funktion und gibt somit die Startadresse der Funktion an.

Beispiel:

In der folgenden Definition:

```
double g(double x) {...}
```

steht der Name g für einen konstanten Zeiger auf die Startadresse der Funktion.

Beispiel: Numerische Differentiation:

$$f'(x) \approx \frac{f(x+h) - f(x-h)}{2h}$$

Die Funktion mit Namen **diff** berechnet die Näherungswerte für die Ableitung einer beliebigen, differenzierbaren Funktion **f** an einer Stelle **x**. Für die Berechnung benötigt die Funktion **diff** die folgenden Eingaben:

- Startwert *x*.
- Eine Funktion *f(x)*.
- Der Wert h wird fest vorgegeben.

Der Rückgabewert der Funktion ist die Näherung für *f'(x)*.

```
#include <math.h>
#include <stdio.h>
#define h  0.001

double test(double);

double diff(double (*)(), double);
```

```
int main(){
   double a;
   scanf("%lf", &a);
   printf("Ableitung von f ist %lf", diff(f, a));
}
double test(double x){
   return cos(2*x*x);
}
double diff( double (*f)(double), double x){
   double ergebnis;
   ergebnis=((*f)(x-h)+(*f)(x+h))/(2*h);
   return ergebnis;
}
```

8.8 Argumente aus der Kommandozeile

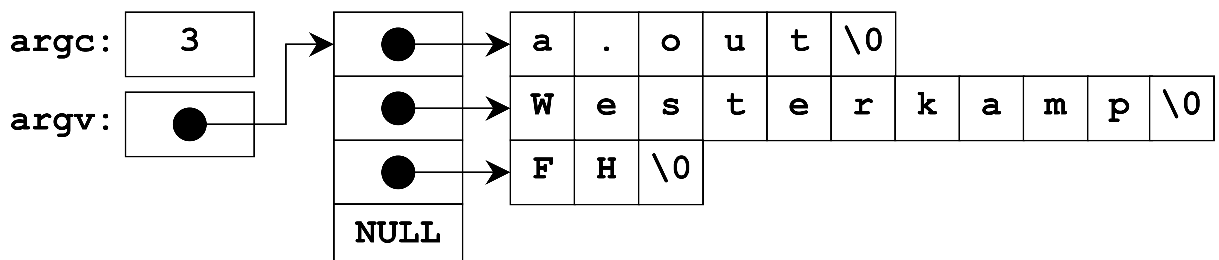
Einem ausführbaren C-Programm können (so wie den meisten Kommandos unter Unix oder DOS) beim Aufruf durch Leerzeichen getrennte Argumente über die Kommandozeile übergeben werden. Diese Argumente werden vom Betriebssystem gelesen und der **`main`**-Funktion beim Aufruf als Parameter übergeben.
Die **`main`**-Funktion muss dann wie folgt deklariert werden:

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

Der erste Parameter **`argc`** enthält die Anzahl der Argumente. Der zweite Parameter **`argv`** ist ein Zeiger auf ein Feld von Zeigern auf Zeichenketten. Die Größe des Feldes ist durch den Wert des ersten Parameters **`argc`** spezifiziert. Jeder Zeiger des Feldes zeigt auf eine Zeichenkette, welche als Argument beim Aufruf des Programms auf der Kommandozeile spezifiziert wurde.

Beispiel:

Beim Aufruf des Programms *a.out* mit den Argumenten ''Westerkamp'' und ''FH'' ergibt sich folgende Struktur für *argv*:



Im Beispiel ist dargestellt, dass der Zeiger Feld, auf den **`argv`** zeigt, um ein Element größer ist als der in **`argc`** spezifizierte Wert. Das letzte Argument im Zeigerfeld (also **`argv[argc]`**) enthält immer den NULL-Zeiger.

Beispiel: Echo Kommando

Folgendes Beispiel simuliert das Echo-Kommando. Es druckt alle Argumente inklusive des Kommandonamens aus.

```
#include <stdio.h>
int main(int argc, char *argv[]){
    int i;
    for(i=0; i<argc; ++i) {
        printf("%s"', argv[i]);
    }
    printf("\n");
}
```

Da das letzte Element des Zeigerfeldes immer den Wert NULL enthält, kann das obige Progamm auch als nachfolgende Zeigervariante realisiert werden.

```
#include <stdio.h>
int main(int argc, char *argv[]){
    do {
       printf("%s ", *argv);
    }
    while(NULL != *++argv);
    printf("\n");
    return 0;
}
```

Die erste Zeichenkette, auf die der Zeiger **argv[0]** zeigt, enthält den Kommandonamen, mit dem das Programm gestartet wurde. Unter UNIX entspricht er dem Namen der Datei, die den ausführbaren Programmcode enthält (z. B. a.out).
Der Kommandoname wird bei der Anzahl der Argumente in **argc** mitgezählt.

An ein C-Programm können nur Zeichenketten übergeben werden. Will man eine Zahl übergeben, muss die zugehörige Zeichenkette in eine Zahl gewandelt werden. Dazu dienen die Funktionen:

double atof(char *) oder **sscanf (char *, char *,...).**

Beispiel:

Das nächste Programm erwartet eine Dezimalzahl als Kommandozeilen-Argument. Die Zeichenkette wird in das *double*-Format umgewandelt.

```
#include <stdio.h>
#include <math.h>
#include <stdlib.h>
int main(int arg c, char *argv[]) {
    double y;
    if (argv[1])  {
        y=atof(argv[1]);
        printf("\nsin(%lf)=%lf, cos(%lf)=%lf\n",
```

```
                  y, sin(y), y, cos(y));
        } else printf("\n Argument fehlt\n");
    }
```

8.9 Beispiele für Typvereinbarungen

Deklaration	Bedeutung
`int a;`	*integer*-Variable
`double *a;`	Zeiger auf *double*
`float mat[10];`	Feld mit 10 Komponenten vom Typ *float*
`double Zahl[10][20];`	2-dimensionales Feld mit 10 Zeilen und 20 Spalten
`long int (*FD)[100];`	Zeiger auf ein Feld mit 100 Komponenten
`char *Zeilen[100];`	Feld von 100 Zeigern auf *character*
`short *g(int b);`	Funktion, die einen *integer*-Wert als Parameter fordert und einen Zeiger auf *short* zurückgibt.
`double (*g)` `(float x1,float x2);`	`g` ist ein Zeiger auf eine Funktion, die als Ergebnis einen Zeiger auf *double* liefert und als Argumente zwei *float*-Werte erwartet.
`int **a;`	Zeiger auf Zeiger auf *integer*-Wert
`long **f()`	`f` ist eine Funktion mit Rückgabe Zeiger auf Zeiger auf *long*-Wert.
`long *(*f)()`	`f` ist ein Zeiger auf eine Funktion mit Rückgabewert Zeiger auf *long integer*.
`long (**f)()`	`f` ist ein Zeiger auf einen Zeiger auf eine Funktion mit Rückgabewert *long integer*.
`long **(**f)()`	`f` ist ein Zeiger auf einen Zeiger auf eine Funktion mit Rückgabewert Zeiger auf Zeiger auf *long*.
`int(*f(int(*)()))();`	Funktion `f`, die als Argument einen Zeiger auf eine Funktion mit Rückgabewert *integer* erwartet und einen Zeiger auf eine ebensolche Funktion zurückliefert.

Bemerkung:

Es gibt in C

- keine Felder, deren Komponenten Funktionen sind und
- keine Funktionen, die ein Feld als Rückgabewert haben.

Es gibt aber

- Felder, deren Komponenten Zeiger auf Funktionen sind und
- Funktionen, die einen Zeiger auf ein Feld zurückgeben.

9 Speicherklassen, Attribute und Gültigkeitsbereiche

9.1 Speicherklassen

Ein Programm kann Variablen mit unterschiedlichen Eigenschaften enthalten.

- **Automatische Variable** (transiente Variable) werden lokal angelegt. Nach Abarbeitung eines zugehörigen Programmteils wird der zugeordnete Speicher wieder freigegeben.
- **Globale und statische Variable** (permanente Variable) werden beim Starten des Programmes angelegt, der zugeordnete Speicher bleibt während der gesamten Programmlaufzeit der Variablen zugeordnet.
- **Dynamische Variable** werden während der Laufzeit durch den Programmcode angelegt und auch wieder freigegeben. Ihnen wird kein Variablenname zugeordnet, der Zugriff erfolgt über (automatische, statische oder globale) Zeigervariable.
- Schließlich besitzt ein Programm noch **konstante Werte**, die durch den Programmcode nicht verändert werden können.

Diese verschiedenen Speicherklassen der Variablen, auch als Variablenklassen bezeichnet, werden auf unterschiedliche Speicherbereiche (Speichersegmente) eines Programms abgebildet. Nachfolgende Abbildung zeigt diesen Zusammenhang:

Hauptspeichersegmente	Zugeordnete Speicherklassen
Code-Segment (text)	Programmcode, Konstanten, Initialisierungswerte für Datensegement
Daten-Segment	Globale und statische Variable
Stack-Segment	Automatische Variable
Heap-Segment	Dynamische Variable

Beim Start eines Programmes wird zunächst vom Betriebssystem das Code-Segment (häufig auch als text-Segment bezeichnet) geladen und der Speicherplatz für die anderen Segmente bereitgestellt. Dann wird vom Betriebssystem der Programmcode im Code-Segment angesprungen. Durch Routinen, die mit dem Compiler mitgeliefert und beim Binden zu den Anwenderfunktionen hinzugefügt werden, wird zunächst das Datensegment initialisiert. Diese Routinen rufen anschließend die **`main`**-Funktion auf und starten damit den vom Programmierer in der Sprache C spezifizierten Code.

9.1.1 Automatische Variable

Alle bisher betrachteten Variablen wurden im Funktionsrumpfes zu Beginn im Vereinbarungsteil definiert. Diese Variablen sind lokal innerhalb des gesamten Funktionsrumpfes gültig.
Möchte man Variablen noch detaillierter einem eingeschränkten, lokalen Code-Bereich zuordnen, kann man auch am Beginn jeder Verbundanweisung (siehe Abschnitt 4) einen Vereinbarungsteil vorsehen.
Beide Variablendefinitionen erzeugen **automatische** Variable. Eine automatische Variable wird beim Aufruf einer Funktion bzw. beim Eintritt des Programmes in eine Verbundanweisung jedesmal neu auf dem Stack (siehe Abschnitt 5) angelegt. Nach dem Anlegen ist der Wert der Variable undefiniert. Der Wert der Variablen aus einem früheren Funktionsaufruf oder einem früheren Durchlauf der Verbundanweisung ist nicht mehr vorhanden. Vor einer sinnvollen Verwendung muss daher der Variablen ein Wert zugewiesen werden. Diese Wertzuweisung kann innerhalb der Definition oder später in einer separaten Zuweisung erfolgen.

Beispiel:

```
#include <stdio.h>

int main(){
   int i;
   printf("Wert von i: %d\n", i);
}
```

Der Wert von **i** ist zufällig!!

Beispiel:

```
#include <stdio.h>

int test(int v);

int main(){
   printf("Wert1 test: %d\n", test(1));
   printf("Wert2 test: %d\n", test(1));
}

int test(int v){
   int i = 0; /* i lokal in Funktion test */
   i = i+v;
   {  int j;  /* j lokal in Verbundanweisung */
      for (j=0; j<10; j++) {
         i = i+j;
      }
   }
   return i;
}
```

Das Beispiel zeigt die Verwendung einer lokalen, automatischen Variablen (Variable **`j`**), die zu Beginn eines Blocks definiert wird. Die Variable **`i`** wird bei jedem Aufruf der Funktion Test neu auf den Wert 0 initialisiert. Somit führt der zweimalige Aufruf der Funktion **`test`** jedes Mal zum gleichen Ergebnis und die beiden obigen **`printf`**-Anweisungen geben den gleichen Wert aus. (Übung: Welches Ergebnis liefert die Funktion **`test`**?)

Da die automatischen Variablen auf dem Stack angelegt werden, sind ihre Adressen nach dem Übersetzen und Binden des Programmes nicht bekannt. Sie ergeben sich erst beim Start der Funktion zur Laufzeit. Auch variieren die Speicherpositionen der Variablen auf dem Stack, je nachdem von wo aus eine Funktion aufgerufen wird.
Automatische Variable können durch Voranstellen des Schlüsselwortes **`auto`** als automatische Variable gekennzeichnet werden.
Die C-Syntax erlaubt eine zweite Kennzeichnung für häufig verwendete automatische Variable, die zur Optimierung des Laufzeitverhaltens in Registern der CPU gehalten werden sollten. Diese Variable werden durch Voranstellen des Schlüsselwortes **`register`** gekennzeichnet. Es wird jedoch nicht garantiert, dass solche Variable wirklich in Register gelegt werden, sie können vom Compiler auch auf dem Stack abgelegt werden. Die Kennzeichnung dient lediglich als Hinweis für den Optimierungslauf des Compilers.

<u>Beispiel:</u>

```
long a, b;          /* Automatische Variable,
                       auch ohne Kennzeichnung */
auto double f, g;   /* Explizite Kennzeichnung
                       als automatische Variable */
register int i, j;  /* Automatische Register-
                       Variable */
```

9.1.2 Statische, lokale Variable

Statische, lokale Variable werden, wie auch die automatischen Variablen, im Vereinbarungsteil am Beginn eines Funktionsrumpfes oder einer Verbundanweisung definiert. Sie werden durch das Schlüsselwort **`static`** gekennzeichnet.
Statische, lokale Variable werden im Datensegment angelegt, daher ist jeder dieser Variablen immer eine eindeutige Speicherzelle und damit eine eindeutige Adresse zugeordnet. Ein zugewiesener Wert bleibt erhalten, bis ein neuer Wert zugewiesen wird.
Einer lokalen, statischen Variablen kann bei der Definition ein Initialisierungswert zugewiesen werden. Der Initialisierungswert wird der Variablen vor dem Start der **`main`**-Funktion einmal zugewiesen. Wird kein Initialisierungswert spezifiziert, wird eine statische, lokale Variable zu 0 initialisiert.

Beispiel:

```
#include <stdio.h>
int test();

int main(){
    printf("test: %d\n", test());
    printf("test: %d\n", test());
    printf("test: %d\n", test());
    return 0;
}
int test(){
    static int i=0;
    return i++;
} /* i ueberlebt das Ende der Verbundanweisung */
```

Das Programm ergibt folgende Ausgabe:

```
test: 0
test: 1
test: 2
```

Rekursive Funktionen: Bei rekursiven Funktionen ist zu beachten, dass für jede rekursive Instanz einen eigener Satz automatischer Variablen auf dem Stack angelegt wird, alle Instanzen jedoch auf die gleichen lokalen, statischen Variablen zugreifen.

Beispiel:

```
#include <stdio.h>

void test(int n);

int main(){
   test(3);
   return 0;
}

void test(int n){
   int i=0;
   static int j=0;
   if (n) {
      test(n-1);
   }
   printf("n, i, j: %d, %d, %d\n", n, i++, j++);
}
```

Das Programm führt zu folgender Ausgabe auf dem Bildschirm:

```
n, i, j: 0, 0, 0
n, i, j: 1, 0, 1
n, i, j: 2, 0, 2
n, i, j: 3, 0, 3
```

9.1.3 Globale Variable

Globale Variable werden beim Programmstart außerhalb von Funktionen eingerichtet und bleiben bis zum Ende des Programms verfügbar. Aus unterschiedlichen Funktionen kann direkt auf die globale Variable zugegriffen werden, ohne diese mittels Zeiger an die Funktionen zu übergeben. In C gibt es für globale Variable eine Sichtbarkeit innerhalb einer Datei und eine Sichtbarkeit im gesamten Programm.

Globale Variable mit Sichtbarkeit im gesamten Programm

Die Vereinbarung einer Variablen außerhalb einer Funktion ohne Schlüsselwort für die Speicherklasse definiert eine globale Variable, die im gesamten Programm sichtbar ist. Da der Code eines Programmes über mehrere Dateien aufgeteilt werden kann, eine Variable jedoch nur einmal definiert werden darf, muss eine Verbindung der globalen Variablen zu den übrigen Dateien hergestellt werden. Dazu dient das Schlüsselwort **`extern`**. Es deklariert die Variable, führt aber nicht zur Zuweisung von Speicher.

Globale Variable mit Sichtbarkeit innerhalb einer Datei

Die Vereinbarung einer Variablen außerhalb einer Funktion mit Schlüsselwort **`static`** definiert eine globale Variable, die innerhalb der Datei sichtbar ist.

Globale Variable werden in C grundsätzlich initialisiert. Werden keine Initialisierungswerte angegeben, wird eine Initialisierung mit 0 durchgeführt.

Beispiel:

Datei main.c:

```
#include <stdio.h>
void test(int n);
int g;

int main(){
   printf("1: g=%d\n", g);
   test(3);
   printf("2: g=%d\n", g);
   test(7);
   printf("3: g=%d\n", g);
   test(0);
   printf("4: g=%d\n", g);
   return 0;
}
```

Datei test.c

```
extern int g;
static int d

void test(int n){
   g=d;
   d=n;
}
```

9.1.4 Dynamische Variable

Dynamische Variable werden durch Programmcode angefordert. Sie besitzen keinen Variablennamen, sondern werden nur über zugeordnete Zeiger angesprochen. Die Lebensdauer ist somit ebenfalls durch den Programmcode bestimmt. Eine Initialisierung dynamischer erfolgt nicht bei der Anforderung, sondern muss explizit durch den Programmcode erfolgen.
Eine genaue Betrachtung dynamischer Datenobjekte erfolgt in Abschnitt 10.

9.1.5 Konstante Variable

ANSI-C erlaubt die Spezifikation von konstanten Variablen. Sie werden wie Variablen vereinbart, der Definition wird jedoch das Schlüsselwort `const` vorangestellt. Damit erhält die Variable ein Attribut, dass sie nicht geändert werden soll. Der Compiler verhindert dann das Ändern des Wertes (oder warnt zumindest davor). Die Speicherklasse ist äquivalent zur Speicherklasse, die sich auch ohne das Attribut ergeben würde. Somit gelten die bisherigen Ausführungen zu lokalen und globalen Variablen auch für konstante Variable.

Beispiel:

```
const int Anzahl = 100;
const double pi =3.1415927;
```

Eine konstante Variable muss bei der Definition initialisiert werden, da eine spätere Wertzuweisung ja durch das `const`-Attribut verhindert werden soll.

Bei der Definition konstanter Zeigervariablen bestehen zwei Möglichkeiten. Zum einen kann man einen Zeiger auf eine konstante Variable vereinbaren, dann kann zwar der Zeiger verändert werden, nicht jedoch der Inhalt der Variablen, auf die der Zeiger zeigt. Zum zweiten kann ein konstanter Zeiger vereinbart werden, der bei der Definition auf eine Variable gerichtet wird und danach nicht verändert werden darf. Nachfolgendes Beispiel zeigt diese Unterschiede.

Beispiel:

```
int          Variable;
const int    Konst_Var = 10;
const int   *Zeiger_auf_Konst_Var = &Konst_Var;
int * const  Konst_Zeiger_auf_Var = &Variable;
const int * const  Konst_Zeiger_auf_Konst_Var
                                  = &Konst_Var;
```

Symbolische und literale Konstanten

Symbolische Konstanten mit dem `#define`-Befehl des Präprozessors vereinbart. Beim Präprozessor-Lauf werden alle Vorkommen symbolischer Konstanten im Programmcode in literale Konstanten aufgelöst. Literale Konstanten haben keinen Namen, sie werden durch ihren Wert dargestellt. Da sie somit nicht geändert werden können, werden sie sinnvollerweise vom Compiler im Code-Segment angelegt.

Beispiel:

```
Zeile 1: #define ANZAHL 100
         ...
Zeile 2: double f[ANZAHL], g[ANZAHL];
         ...
Zeile 3: for (i=0; i<ANZAHL; i=i+1) {
Zeile 4:     f[i]=23.0*g[i];
         }
         ...
```

In Zeile 1 wird eine symbolische Konstante **`ANZAHL`** vereinbart, die dann in den Zeilen 2 und 3 verwendet (referenziert) wird. Dieses Vorgehen ist sehr hilfreich bei Änderungen, da **`ANZAHL`** nur an einer Stelle angepasst werden muss.
In Zeile 3 werden die Literalen Konstanten 0 und 1 verwendet, in Zeile 4 die literale Konstante 23.0.

9.2 Attribute

In C existieren die beiden Attribute **`const`** und **`volatile`**, die einer Variablendefinition hinzugefügt werden können.

`const`-Attribut

In obigem Abschnitt 9.1.5 wurde das **`const`**-Attribut für Variable bereits erläutert.

`volatile`-Attribut

Das Attribut **`volatile`** wird häufig mit der Bezeichnung *„Zerbrechlich"* ins Deutsche übersetzt. Mit diesem Attribut werden Speicherzellen gekennzeichnet, welche z.B. zwischen zwei Lesezyklen ihren Wert ändern können.
Beispielsweise hat man dieses Verhalten bei Variablen, auf die zwei asynchrone Softwareprozesse zugreifen (z.B. Semaphore). Oder ein Programm greift auf Register von Peripheriebausteinen zu, die in den Speicherbereich des Prozessors eingeblendet sind (Memory-mapped IO). Aufgrund von Hardwareereignissen kann sich der Inhalt eines solchen Registers ändern.
Moderne optimierende Compiler merken sich jedoch Inhalte von häufig verwendeten Variablen in Registern, um damit Zugriffe auf den externen Speicher zu sparen. Bei **`volatile`**-Variablen muss der Compiler diese Optimierung abschalten und bei jedem Zugriff auf die Variable im Speicher zugreifen.

Beispiel:

```
volatile int *hardware_reg = 0x03f8
...
while (0==*hardware_reg) { /* HW lesen */
   /* warten */
}
*hardware_reg=0; /* HW schreiben */
```

Durch das **`volatile`**-Attribut muss der Compiler gewährleisten, dass bei jedem Durchlauf der **`while`**-Schleife erneut die Hardware gelesen wird. Ansonsten könnte eine unendliche Schleife entstehen.

9.3 Gültigkeitsbereich von Variablen

Bei der Definition einer Variablen besitzt eine Variable einen Gültigkeitsbereich, er wird im Englischen als *Scope* bezeichnet. In diesem Bereich ist die Variable sichtbar und es kann auf sie zugegriffen werden. Werden mehrere Variable gleichen Namens definiert, die sich in ihren Gültigkeitsbereichen unterscheiden, müssen eindeutige Regeln entscheiden, auf welche Variable zugegriffen wird.

- Innerhalb einer Vereinbarungsebene (z.B. globaler Vereinbarungsteil, Vereinbarungsteil eines Blocks) dürfen Variablennamen nur einmal vergeben werden.
  Beispiel:
  ```
  int main() {
     int x;
     double x;  /* Fehler: Gleicher Name ist verboten  */
  }
  ```
- Ist auf einer höheren Ebene eine Variable vereinbart, wird diese durch eine Variable gleichen Namens in einer tieferen Ebenen verdeckt. Es wird dann immer auf die Variable in der tieferen Ebene zugegriffen.

Beispiel:

```
#include <stdio.h>
int i=10;
int test();
int main () {
   int i=11;
   { int i=12;
      printf("a: i=%d\n", i);
      printf("b: i=%d\n", test());
   }
   printf("c: i=%d\n", i);
   printf("d: i=%d\n", test());
   return 0;
}
int test(){
   return i;
}
```

Folgende Ausgabe erscheint auf dem Bildschirm:

```
a: i=12
b: i=10
c: i=11
d: i=10
```

Eine Variable ist somit im Block gültig, in dem sie definiert wird. Weiterhin ist sie in hierarchisch darunter liegenden Blöcken gültig, sofern dort keine Variable mit gleichem Namen definiert ist.

9.4 Zusammenfassung der Variablenklassen und ihrer Eigenschaften

<table>
<tr><th>Variablentyp</th><th>Schlüssel-wort</th><th>Gültigkeit</th><th>Lebens-dauer</th><th>Initiali-sierung</th><th>Segment</th></tr>
<tr><td rowspan="3">Automatisch</td><td>-</td><td rowspan="3">Block</td><td rowspan="3">Block</td><td rowspan="3">nein</td><td rowspan="3">Stack</td></tr>
<tr><td><code>auto</code></td></tr>
<tr><td><code>register</code></td></tr>
<tr><td>Lokal, statisch</td><td><code>static</code></td><td>Block</td><td>Programm</td><td>ja</td><td>Daten</td></tr>
<tr><td rowspan="3">Global</td><td><code>static</code></td><td>Datei</td><td>Programm</td><td>ja</td><td>Daten</td></tr>
<tr><td>-</td><td rowspan="2">Programm</td><td rowspan="2">Programm</td><td>ja</td><td rowspan="2">Daten</td></tr>
<tr><td><code>extern</code></td><td>-</td></tr>
<tr><td>Dynamisch</td><td>-</td><td>(kein Name)</td><td>Code-abhängig</td><td>nein</td><td>Heap</td></tr>
<tr><td>Symbolische und literale Konstanten</td><td>-</td><td>-</td><td>-</td><td>-</td><td>Code</td></tr>
</table>