

19.2 Strukturen als Funktionsparameter

- Strukturen können wie normale Variablen an Funktionen übergeben werden
 - ◆ Übergabesemantik: **call by value**
 - Funktion erhält eine Kopie der Struktur
 - auch wenn die Struktur ein Feld enthält, wird dieses komplett kopiert!
 - !!! Unterschied zur direkten Übergabe eines Feldes
- Strukturen können auch Ergebnis einer Funktion sein
 - Möglichkeit mehrere Werte im Rückgabeparameter zu transportieren
- Beispiel

```
struct komplex komp_add(struct komplex x, struct komplex y) {  
    struct komplex ergebnis;  
    ergebnis.re = x.re + y.re;  
    ergebnis.im = x.im + y.im;  
    return(ergebnis);  
}
```

19.3 Felder von Strukturen

- Von Strukturen können — wie von normale Datentypen — Felder gebildet werden
- Beispiel

```
struct student gruppe8[35];
int i;
for (i=0; i<35; i++) {
    printf("Nachname %d. Stud.: ", i);
    scanf("%s", gruppe8[i].nachname);
    ...
    gruppe8[i].gruppe = 8;

    if (gruppe8[i].matrnr < 1500000) {
        gruppe8[i].best = 'y';
    } else {
        gruppe8[i].best = 'n';
    }
}
```

19.4 Zeiger auf Felder von Strukturen

- Ergebnis der Addition/Subtraktion abhängig von Zeigertyp!
- Beispiel

```
struct student gruppe8[35];
struct student *gp1, *gp2;

gp1 = gruppe8; /* gp1 zeigt auf erstes Element des Arrays */
printf("Nachname des ersten Studenten: %s", gp1->nachname);

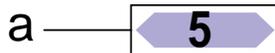
gp2 = gp1 + 1; /* gp2 zeigt auf zweite Element des Arrays */
printf("Nachname des zweiten Studenten: %s", gp2->nachname);

printf("Byte-Differenz: %d", (char*)gp2 - (char*)gp1);
```

19.5 Zusammenfassung

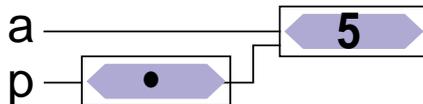
■ Variable

```
int a;
```



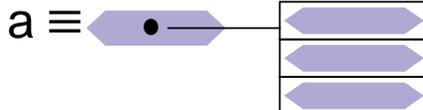
■ Zeiger

```
int *p = &a;
```



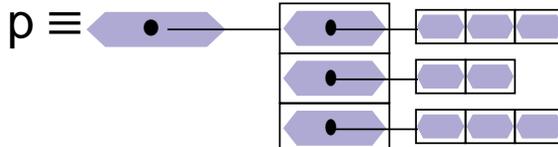
■ Feld

```
int a[3];
```



■ Feld von Zeigern

```
int *p[3];
```



■ Struktur

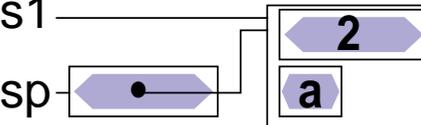
```
struct s{int a; char c;};
```

```
struct s s1 = {2, 'a'};
```



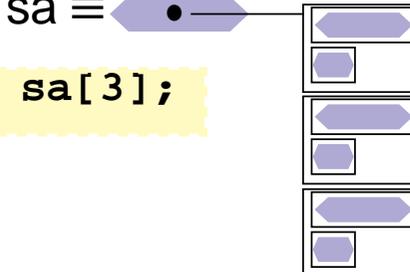
■ Zeiger auf Struktur

```
struct s *sp = &s1;
```



■ Feld von Strukturen

```
struct s sa[3];
```



Zeiger auf Funktionen

■ Datentyp: Zeiger auf Funktion

◆ Variablendef.: *<Rückgabety> (*<Variablenname>)(<Parameter>);*

```
int (*fptr)(int, char*);
```

```
int test1(int a, char *s) { printf("1: %d %s\n", a, s); }  
int test2(int a, char *s) { printf("2: %d %s\n", a, s); }
```

```
fptr = test1;
```

```
fptr(42, "hallo");
```

```
fptr = test2;
```

```
fptr(42, "hallo");
```

Ein-/Ausgabe

- E-/A-Funktionalität nicht Teil der Programmiersprache
- Realisierung durch "normale" Funktionen
 - Bestandteil der Standard-Funktionsbibliothek
 - einfache Programmierschnittstelle
 - effizient
 - portabel
 - betriebssystemnah
- Funktionsumfang
 - Öffnen/Schließen von Dateien
 - Lesen/Schreiben von Zeichen, Zeilen oder beliebigen Datenblöcken
 - Formatierte Ein-/Ausgabe

21.1 Standard Ein-/Ausgabe

■ Jedes C-Programm erhält beim Start automatisch 3 E-/A-Kanäle:

◆ **stdin** Standardeingabe

- normalerweise mit der Tastatur verbunden
- Dateiende (**EOF**) wird durch Eingabe von **CTRL-D** am Zeilenanfang signalisiert
- bei Programmaufruf in der Shell auf Datei umlenkbar
prog <eingabedatei
(bei Erreichen des Dateiendes wird **EOF** signalisiert)

◆ **stdout** Standardausgabe

- normalerweise mit dem Bildschirm (bzw. dem Fenster, in dem das Programm gestartet wurde) verbunden
- bei Programmaufruf in der Shell auf Datei umlenkbar
prog >ausgabedatei

◆ **stderr** Ausgabekanal für Fehlermeldungen

- normalerweise ebenfalls mit Bildschirm verbunden

21.1 Standard Ein-/Ausgabe (2)

■ Pipes

- ◆ die Standardausgabe eines Programms kann mit der Standardeingabe eines anderen Programms verbunden werden

- Aufruf

```
prog1 | prog2
```

- ! Die Umlenkung von Standard-E/A-Kanäle ist für die aufgerufenen Programme völlig unsichtbar

■ automatische Pufferung

- ◆ Eingabe von der Tastatur wird normalerweise vom Betriebssystem zeilenweise zwischengespeichert und erst bei einem **NEWLINE**-Zeichen (`'\n'`) an das Programm übergeben!

21.2 Öffnen und Schließen von Dateien

- Neben den Standard-E/A-Kanälen kann ein Programm selbst weitere E/A-Kanäle öffnen

- Zugriff auf Dateien

- Öffnen eines E/A-Kanals

- Funktion `fopen`:

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

name Pfadname der zu öffnenden Datei

mode Art, wie die Datei geöffnet werden soll

"r" zum Lesen

"w" zum Schreiben

"a" append: Öffnen zum Schreiben am Dateiende

"rw" zum Lesen und Schreiben

- Ergebnis von `fopen`:

Zeiger auf einen Datentyp `FILE`, der einen Dateikanal beschreibt
im Fehlerfall wird ein `NULL`-Zeiger geliefert

21.2 Öffnen und Schließen von Dateien (2)

■ Beispiel:

```
#include <stdio.h>

int main(int argc, char *argv[]) {
    FILE *eingabe;

    if (argv[1] == NULL) {
        fprintf(stderr, "keine Eingabedatei angegeben\n");
        exit(1);          /* Programm abbrechen */
    }

    if ((eingabe = fopen(argv[1], "r")) == NULL) {
        perror(argv[1]); /* Fehlermeldung ausgeben */
        exit(1);        /* Programm abbrechen */
    }

    ... /* Programm kann jetzt von eingabe lesen */
}
```

■ Schließen eines E/A-Kanals

```
int fclose(FILE *fp)
```

➤ schließt E/A-Kanal `fp`

21.3 Zeichenweise Lesen und Schreiben

■ Lesen eines einzelnen Zeichens

◆ von der Standardeingabe

```
int getchar( )
```

- lesen das nächste Zeichen
- geben das gelesene Zeichen als `int`-Wert zurück
- geben bei Eingabe von `CTRL-D` bzw. am Ende der Datei `EOF` als Ergebnis zurück

◆ von einem Dateikanal

```
int getc(FILE *fp )
```

■ Schreiben eines einzelnen Zeichens

◆ auf die Standardausgabe

```
int putchar(int c)
```

◆ auf einen Dateikanal

```
int putc(int c, FILE *fp )
```

- schreiben das im Parameter `c` übergebene Zeichen
- geben gleichzeitig das geschriebene Zeichen als Ergebnis zurück

21.3 Zeichenweise Lesen und Schreiben (2)

- Beispiel: copy-Programm, Aufruf: `copy Quelldatei Zieldatei`

```
#include <stdio.h>
int main(int argc, char *argv[]) {
    FILE *quelle, *ziel;
    int c;          /* gerade kopiertes Zeichen */

    if (argc < 3) { /* Fehlermeldung, Abbruch */ }

    if ((quelle = fopen(argv[1], "r")) == NULL) {
        perror(argv[1]); /* Fehlermeldung ausgeben */
        exit(EXIT_FAILURE); /* Programm abbrechen */
    }

    if ((ziel = fopen(argv[2], "w")) == NULL) {
        /* Fehlermeldung, Abbruch */
    }

    while ( (c = getc(quelle)) != EOF ) {
        putc(c, ziel);
    }

    fclose(quelle);
    fclose(ziel);
}
```

Teil 1: Aufrufargumente auswerten

21.3 Zeilenweise Lesen und Schreiben

■ Lesen einer Zeile von der Standardeingabe

```
char *fgets(char *s, int n, FILE *fp)
```

- liest Zeichen von Dateikanal `fp` in das Feld `s` bis entweder `n-1` Zeichen gelesen wurden oder `'\n'` oder `EOF` gelesen wurde
- `s` wird mit `'\0'` abgeschlossen (`'\n'` wird nicht entfernt)
- gibt bei `EOF` oder Fehler `NULL` zurück, sonst `s`
- für `fp` kann `stdin` eingesetzt werden, um von der Standardeingabe zu lesen

■ Schreiben einer Zeile

```
int fputs(char *s, FILE *fp)
```

- schreibt die Zeichen im Feld `s` auf Dateikanal `fp`
- für `fp` kann auch `stdout` oder `stderr` eingesetzt werden
- als Ergebnis wird die Anzahl der geschriebenen Zeichen geliefert

21.4 Formatierte Ausgabe

■ Bibliotheksfunktionen — Prototypen (Schnittstelle)

```
int printf(char *format, /* Parameter */ ... );  
int fprintf(FILE *fp, char *format, /* Parameter */ ... );  
int sprintf(char *s, char *format, /* Parameter */ ... );  
int snprintf(char *s, int n, char *format, /* Parameter */ ... );
```

■ Die statt ... angegebenen Parameter werden entsprechend der Angaben im `format`-String ausgegeben

- bei `printf` auf der Standardausgabe
- bei `fprintf` auf dem Dateikanal `fp`
(für `fp` kann auch `stdout` oder `stderr` eingesetzt werden)
- `sprintf` schreibt die Ausgabe in das `char`-Feld `s`
(achtet dabei aber nicht auf das Feldende -> Pufferüberlauf möglich!)
- `snprintf` arbeitet analog, schreibt aber maximal nur `n` Zeichen
(`n` sollte natürlich nicht größer als die Feldgröße sein)

21.4 Formatierte Ausgabe (2)

- Zeichen im `format`-String können verschiedene Bedeutung haben
 - normale Zeichen: werden einfach auf die Ausgabe kopiert
 - Escape-Zeichen: z. B. `\n` oder `\t`, werden durch die entsprechenden Zeichen (hier Zeilenvorschub bzw. Tabulator) bei der Ausgabe ersetzt
 - Format-Anweisungen: beginnen mit `%`-Zeichen und beschreiben, wie der dazugehörige Parameter in der Liste nach dem `format`-String aufbereitet werden soll

- Format-Anweisungen
 - `%d, %i` `int` Parameter als Dezimalzahl ausgeben
 - `%f` `float` Parameter wird als Fließkommazahl (z. B. 271.456789) ausgegeben
 - `%e` `float` Parameter wird als Fließkommazahl in 10er-Potenz-Schreibweise (z. B. 2.714567e+02) ausgegeben
 - `%c` `char`-Parameter wird als einzelnes Zeichen ausgegeben
 - `%s` `char`-Feld wird ausgegeben, bis `'\0'` erreicht ist

21.5 Formatierte Eingabe

■ Bibliotheksfunktionen — Prototypen (Schnittstelle)

```
int scanf(char *format, /* Parameter */ ...);  
int fscanf(FILE *fp, char *format, /* Parameter */ ...);  
int sscanf(char *s, const char *format, /* Parameter */ ...);
```

- Die Funktionen lesen Zeichen von `stdin` (`scanf`), `fp` (`fscanf`) bzw. aus dem `char`-Feld `s`.
- `format` gibt an, welche Daten hiervon extrahiert und in welchen Datentyp konvertiert werden sollen
- Die folgenden Parameter sind Zeiger auf Variablen der passenden Datentypen (bzw. `char`-Felder bei Format `%s`), in die die Resultate eingetragen werden
- relativ komplexe Funktionalität, hier nur Kurzüberblick für Details siehe Manual-Seiten

21.5 Formatierte Eingabe (2)

- *White space* (Space, Tabulator oder Newline `\n`) bildet jeweils die Grenze zwischen Daten, die interpretiert werden
 - *white space* wird in beliebiger Menge einfach überlesen
 - Ausnahme: bei Format-Anweisung `%c` wird auch *white space* eingelesen

- Alle anderen Daten in der Eingabe müssen zum `format`-String passen oder die Interpretation der Eingabe wird abgebrochen
 - wenn im `format`-String normale Zeichen angegeben sind, müssen diese exakt so in der Eingabe auftauchen
 - wenn im `Format`-String eine `Format-Anweisung` (`%...`) angegeben ist, muss in der Eingabe etwas hierauf passendes auftauchen
 - ↳ diese Daten werden dann in den entsprechenden Typ konvertiert und über den zugehörigen Zeiger-Parameter der Variablen zugewiesen

- Die `scanf`-Funktionen liefern als Ergebnis die Zahl der erfolgreich an die Parameter zugewiesenen Werte

21.5 Formatierte Eingabe (3)

`%d` int
`%hd` short
`%ld` long int
`%lld` long long int

`%f` float
`%lf` double
`%Lf` long double
analog auch `%e` oder `%g`

`%c` char
`%s` String, wird
automatisch mit
'\0' abgeschl.

■ nach % kann eine Zahl folgen, die die maximale Feldbreite angibt

`%3d` = 3 Ziffern lesen

`%5c` = 5 char lesen (Parameter muss dann Zeiger auf char-Feld sein)

➤ `%5c` überträgt exakt 5 char (hängt aber kein '\0' an!)

➤ `%5s` liest max. 5 char (bis white space) und hängt '\0' an

■ Beispiele:

```
int a, b, c, d, n;  
char s1[20]="xxxxxx", s2[20];  
n = scanf("%d %2d %3d %5c %s %d",  
          &a, &b, &c, s1, s2, &d);
```

Eingabe: 12 1234567 sowas hmm

Ergebnis: n=5, a=12, b=12, c=345

s1="67 soX", s2="was"

21.6 Fehlerbehandlung

- Fast jeder Systemcall/Bibliotheksaufruf kann fehlschlagen
 - ◆ Fehlerbehandlung unumgänglich!

- Vorgehensweise:
 - ◆ Rückgabewerte von Systemcalls/Bibliotheksaufrufen abfragen
 - ◆ Im Fehlerfall (meist durch Rückgabewert -1 angezeigt): Fehlercode steht in der globalen Variable `errno`

- Fehlermeldung kann mit der Funktion `perror` auf die Fehlerausgabe ausgegeben werden:

```
#include <errno.h>
void perror(const char *s);
```