

Ü-SP1

10 Überblick über die 2. Übung

Überblick über die 2. Übung

- Felder
- Strukturen
- Ein- /Ausgabe
- Fehlerbehandlung
- Dynamische Speicherverwaltung, Teil 2
- Portable Programme
- Literatur zur C-Programmierung:
 - ◆ P. A. Darnell, P. E. Margolis. *C: A Software Engineering Approach*. Springer Verlag, Februar 1996, ISBN: 0387946756

Übungen zur Systemprogrammierung 1
© Michael Gollm, Jürgen Kleinöder • Universität Erlangen-Nürnberg • Informatik 4, 2001

2001-10-24 20.52

Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

52

Felder

11 Felder

11.1 Eindimensionale Felder

■ eine Reihe von Daten desselben Typs kann zu einem **Feld** zusammengefaßt werden

■ bei der Definition wird die Anzahl der Feldelemente angegeben, die Anzahl ist konstant!

■ der Zugriff auf die Elemente erfolgt durch **Indizierung**, beginnend bei Null

■ Definition eines Feldes

■ Beispiele:

```
int x[5];
double f[20];
```

Übungen zur Systemprogrammierung 1
© Michael Gollm, Jürgen Kleinöder • Universität Erlangen-Nürnberg • Informatik 4, 2001

2001-10-24 20.52

Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

53

Felder

11.2 Initialisierung eines Feldes

■ Ein Feld kann durch eine Liste von konstanten Ausdrücken, die durch Komma getrennt sind, initialisiert werden

```
int prim[4] = {2, 3, 5, 7};
char name[5] = {'o', 't', 't', 'o', '\0'};
```

■ wird die explizite Felddimensionierung weggelassen, so bestimmt die Zahl der Initialisierungskonstanten die Feldgröße

```
int prim[] = {2, 3, 5, 7};
char name[] = {'o', 't', 't', 'o', '\0'};
```

■ werden zu wenig Initialisierungskonstanten angegeben, so werden die restlichen Elemente mit 0 initialisiert

Übungen zur Systemprogrammierung 1
© Michael Gollm, Jürgen Kleinöder • Universität Erlangen-Nürnberg • Informatik 4, 2001

2001-10-24 20.52

Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

54

11.2 ... Initialisierung eines Feldes (2)

■ Felder des Typs **char** können auch durch String-Konstanten initialisiert werden

```
char name1[5] = "Otto";
char name2[] = "Otto";
```

Übungen zur Systemprogrammierung 1
© Michael Gollm, Jürgen Kleinöder • Universität Erlangen-Nürnberg • Informatik 4, 2001

2001-10-24 20.52

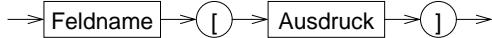
Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

55

11.3 Zugriffe auf Feldelemente

Felder

- Indizierung:



wobei: $0 \leq \text{Wert}(\text{Ausdruck}) < \text{Feldgröße}$

- Beispiele:

```

prim[0] == 2
prim[1] == 3
name[1] == 't'
name[4] == '\0'
  
```

56

2001-10-24 20.52

Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

11.4.1 Zugriffe auf Feldelemente

Felder

- Indizierung:



wobei: $0 \leq A_i < \text{Größe der Dimension } i \text{ des Feldes}$
 $n = \text{Anzahl der Dimensionen des Feldes}$

- Beispiel:

```

int feld[5][8];
feld[2][3] = 10;
  
```

◆ ist äquivalent zu:

```

int feld[5][8];
int *f1;
f1 = (int*)feld;
f1[2*8 + 3] = 10;
  
```

58

2001-10-24 20.52

Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

11.4 Mehrdimensionale Felder

Felder

- neben eindimensionalen Felder kann man auch mehrdimensionale Felder vereinbaren
- Definition eines mehrdimensionalen Feldes



- Beispiel:

```

int matrix[4][4];
  
```

57

2001-10-24 20.52
Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

11.5 Eindimensionale Felder als Funktionsparameter

- ganze Felder können in C **nicht by-value** übergeben werden
- wird einer Funktion der Feldname als Parameter übergeben, kann sie in gleicher Weise wie der Aufrufer auf die Feldelemente zugreifen (und diese verändern!)
- bei der Deklaration des formalen Parameters wird die Feldgröße weggelassen
 - die Feldgröße ist automatisch durch den aktuellen Parameter gegeben
 - ggf. ist die Feldgröße über einen weiteren **int**-Parameter der Funktion explizit mitzuteilen
 - die Länge von Zeichenketten in **char**-Feldern kann normalerweise durch Suche nach dem \0-Zeichen bestimmt werden
- wird ein Feldparameter als **const** deklariert, können die Feldelemente innerhalb der Funktion nicht verändert werden

11.5.1 Beispiele

- Bestimmung der Länge einer Zeichenkette (*String*)

```
int strlen(const char string[])
{
    int i=0;
    while (string[i] != '\0') ++i;
    return(i);
}
```

11.5.1 Beispiele (2)

- Konkatениere Strings

```
void strcat(char to[], const char from[])
{
    int i=0, j=0;
    while (to[i] != '\0') i++;
    while ( (to[i++]) = from[j++]) != '\0' )
    ;
}
```

- Funktionsaufruf mit Feld-Parametern

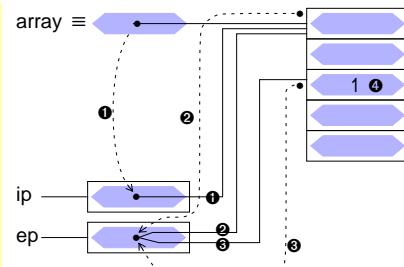
- als aktueller Parameter beim Funktionsaufruf wird einfach der Feldname angegeben

```
char s1[50] = "text1";
char s2[] = "text2";
strcat(s1, s2); /* → s1="text1text2" */
strcat(s1, "text3"); /* → s1="text1text2text3" */
```

11.6 Zeiger und Felder

- ein Feldname ist ein konstanter Zeiger auf das erste Element des Feldes
- im Gegensatz zu einer Zeigervariablen kann sein Wert nicht verändert werden
- es gilt:

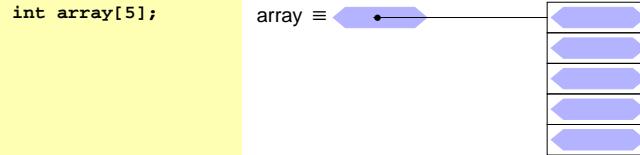
```
int array[5];
int *ip = array; ①
int *ep;
ep = &array[0]; ②
ep = &array[2]; ③
*ep = 1; ④
```



11.6 Zeiger und Felder

Felder

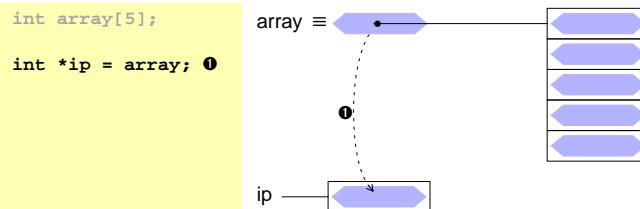
- ein Feldname ist ein konstanter Zeiger auf das erste Element des Feldes
- im Gegensatz zu einer Zeigervariablen kann sein Wert nicht verändert werden
- es gilt:



11.6 Zeiger und Felder

Felder

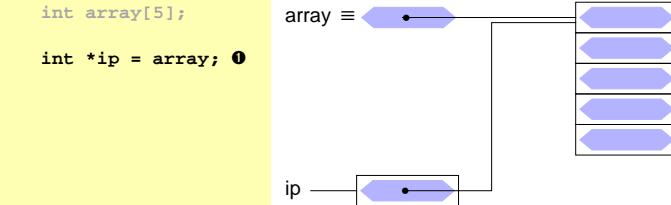
- ein Feldname ist ein konstanter Zeiger auf das erste Element des Feldes
- im Gegensatz zu einer Zeigervariablen kann sein Wert nicht verändert werden
- es gilt:



11.6 Zeiger und Felder

Felder

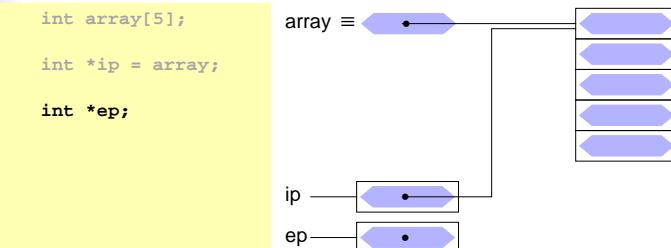
- ein Feldname ist ein konstanter Zeiger auf das erste Element des Feldes
- im Gegensatz zu einer Zeigervariablen kann sein Wert nicht verändert werden
- es gilt:



11.6 Zeiger und Felder

Felder

- ein Feldname ist ein konstanter Zeiger auf das erste Element des Feldes
- im Gegensatz zu einer Zeigervariablen kann sein Wert nicht verändert werden
- es gilt:

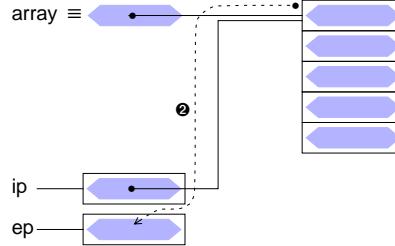


11.6 Zeiger und Felder

Felder

- ein Feldname ist ein konstanter Zeiger auf das erste Element des Feldes
- im Gegensatz zu einer Zeigervariablen kann sein Wert nicht verändert werden
- es gilt:

```
int array[5];
int *ip = array;
int *ep;
ep = &array[0]; ②
```

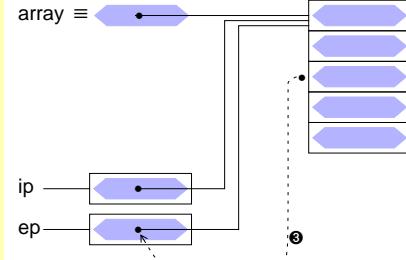


Felder

11.6 Zeiger und Felder

- ein Feldname ist ein konstanter Zeiger auf das erste Element des Feldes
- im Gegensatz zu einer Zeigervariablen kann sein Wert nicht verändert werden
- es gilt:

```
int array[5];
int *ip = array;
int *ep;
ep = &array[0];
ep = &array[2]; ③
```

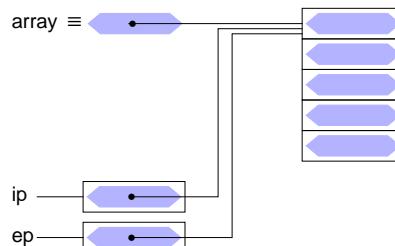


11.6 Zeiger und Felder

Felder

- ein Feldname ist ein konstanter Zeiger auf das erste Element des Feldes
- im Gegensatz zu einer Zeigervariablen kann sein Wert nicht verändert werden
- es gilt:

```
int array[5];
int *ip = array;
int *ep;
ep = &array[0]; ②
```

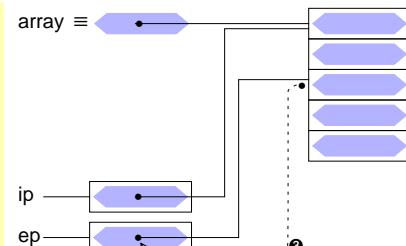


Felder

11.6 Zeiger und Felder

- ein Feldname ist ein konstanter Zeiger auf das erste Element des Feldes
- im Gegensatz zu einer Zeigervariablen kann sein Wert nicht verändert werden
- es gilt:

```
int array[5];
int *ip = array;
int *ep;
ep = &array[0];
ep = &array[2]; ③
```

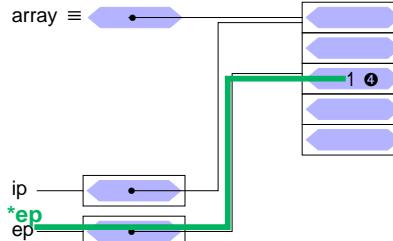


11.6 Zeiger und Felder

Felder

- ein Feldname ist ein konstanter Zeiger auf das erste Element des Feldes
- im Gegensatz zu einer Zeigervariablen kann sein Wert nicht verändert werden
- es gilt:

```
int array[5];
int *ip = array;
int *ep;
ep = &array[0];
ep = &array[2];
*ep = 1; ❶
```

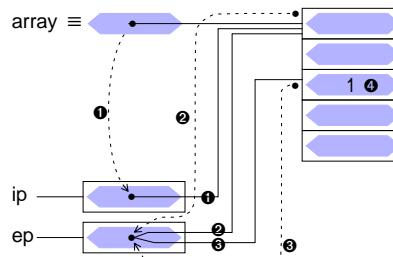


11.6 Zeiger und Felder

Felder

- ein Feldname ist ein konstanter Zeiger auf das erste Element des Feldes
- im Gegensatz zu einer Zeigervariablen kann sein Wert nicht verändert werden
- es gilt:

```
int array[5];
int *ip = array; ❶
int *ep;
ep = &array[0]; ❷
ep = &array[2]; ❸
*ep = 1; ❹
```

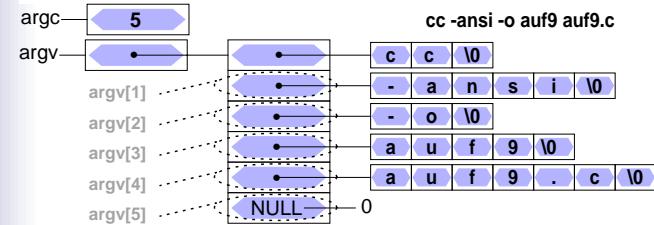


Felder

11.7 Kommandozeilenparameter

- das folgende Programmstück gibt alle Argumente der Kommandozeile aus (außer dem Kommandonamen)

```
int main (int argc, char *argv[])
{
    int i;
    for (i=1; i<argc; i++) {
        printf ("%s\n", argv[i],
               (i < argc-1) ? ' ':'\n');
    }
}
```



Strukturen

12 Strukturen

- Initialisierung
- Strukturen als Funktionsparameter
- Felder von Strukturen
- Zeiger auf Strukturen

12.1 Initialisieren von Strukturen

- Strukturen können — wie Variablen und Felder — bei der Definition initialisiert werden

- Beispiele

```
struct student stud1 = {
    "Meier", "Hans", "24.01.1970", 1533180, 5, 'n'
};

struct komplex c1 = {1.2, 0.8}, c2 = {0.5, 0.33};
```

!!! Vorsicht

bei Zugriffen auf eine Struktur werden die Komponenten durch die Komponentennamen identifiziert,

bei der Initialisierung jedoch nur durch die Position

→ potentielle Fehlerquelle bei Änderungen der Strukturtyp-Deklaration

12.2 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);
}
```

12.3 Felder von Strukturen

- Von Strukturen können — wie von normalen 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';
    }
}
```

12.4 Zeiger auf Felder von Strukturen

- Ergebnis der Addition/Subtraktion abhängig von Zeigertyp!

- Beispiel

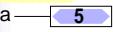
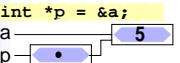
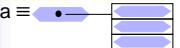
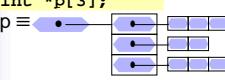
```
struct student gruppe8[35];
struct student *gp1, *gp2;

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);

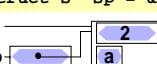
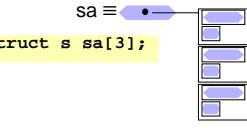
printf("Byte-Differenz: %d", (char*)gp2 - (char*)gp1);
```

12.5 Zusammenfassung

- Variable
`int a;`

- Zeiger
`int *p = &a;`

- Feld
`int a[3];`

- Feld von Zeigern
`int *p[3];`


Strukturen

- 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];`


80

Zeiger auf Funktionen

- Datentyp: Zeiger auf Funktion
- ◆ Variablendef.: `<Rückgabetyp> (*<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");

```

14 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

14.1 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

14.1 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!

Ein-/Ausgabe

```
#include <stdio.h>

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

    if (argc[1] == NULL) {
        fprintf(stderr, "keine Eingabedatei angegeben\n");
        exit(1);
    }

    if ((eingabe = fopen(argv[1], "r")) == NULL) {
        /* eingabe konnte nicht geöffnet werden */
        perror(argv[1]);
        /* Fehlermeldung ausgeben */
        exit(1);
    }

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

Übungen zur Systemprogrammierung 1
© Michael Gollm, Jürgen Kleinöder • Universität Erlangen-Nürnberg • Informatik 4, 2001
Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

Übungen zur Systemprogrammierung 1
© Michael Gollm, Jürgen Kleinöder • Universität Erlangen-Nürnberg • Informatik 4, 2001
Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

14.2 Öffnen und Schließen von Dateien (2)

■ Beispiel:

```
#include <stdio.h>

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

    if (argc[1] == NULL) {
        fprintf(stderr, "keine Eingabedatei angegeben\n");
        exit(1);
    }

    if ((eingabe = fopen(argv[1], "r")) == NULL) {
        /* eingabe konnte nicht geöffnet werden */
        perror(argv[1]);
        /* Fehlermeldung ausgeben */
        exit(1);
    }

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

■ Schließen eines E/A-Kanals

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

► schließt E/A-Kanal fp

Übungen zur Systemprogrammierung 1
© Michael Gollm, Jürgen Kleinöder • Universität Erlangen-Nürnberg • Informatik 4, 2001
Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

Übungen zur Systemprogrammierung 1
© Michael Gollm, Jürgen Kleinöder • Universität Erlangen-Nürnberg • Informatik 4, 2001
Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

14.2 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

14.3 Zeichenweise Lesen und Schreiben

■ Lesen eines einzelnen Zeichens

- ◆ von der Standardeingabe
- ◆ von einem Dateikanal

<pre>int getchar()</pre>	<pre>int getc(FILE *fp)</pre>
--------------------------	-------------------------------

► 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

■ Schreiben eines einzelnen Zeichens

- ◆ auf die Standardausgabe
- ◆ auf einen Dateikanal

<pre>int putchar(int c)</pre>	<pre>int putc(int c, FILE *fp)</pre>
-------------------------------	--------------------------------------

► schreiben das im Parameter c übergeben Zeichen

► geben gleichzeitig das geschriebene Zeichen als Ergebnis zurück

Übungen zur Systemprogrammierung 1
© Michael Gollm, Jürgen Kleinöder • Universität Erlangen-Nürnberg • Informatik 4, 2001
Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

Übungen zur Systemprogrammierung 1
© Michael Gollm, Jürgen Kleinöder • Universität Erlangen-Nürnberg • Informatik 4, 2001
Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

14.6 Formatierte Eingabe

Ein-/Ausgabe

- 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

14.6 Formatierte Eingabe

Ein-/Ausgabe

<code>%d</code>	int
<code>%hd</code>	short
<code>%ld</code>	long int
<code>%lld</code>	long long int
<code>%f</code>	float
<code>%lf</code>	double
<code>%Lf</code>	long double
	analog auch <code>%e</code> oder <code>%g</code>

<code>%c</code>	char
<code>%s</code>	String, wird automatisch mit <code>'\0'</code> 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] = "XXXXXXXX", 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"
```

14.6 Formatierte Eingabe

Ein-/Ausgabe

- *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

15 Fehlerbehandlung

Fehlerbehandlung

- Fast jeder Systemcall/Bibliotheksauftruf kann fehlschlagen
 - ◆ Fehlerbehandlung unumgänglich!
- Vorgehensweise:
 - ◆ Rückgabewerte von Systemcalls/Bibliotheksaufträufen 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);
```

16 Dynamische Speicherverwaltung

- Erzeugen von Feldern der Länge `n`:

- ◆ mittels: `void *malloc(size_t size)`

```
struct person *personen;
personen = (struct person *)malloc(sizeof(struct person)*n);
if(personen == NULL) ...
```

- ◆ mittels: `void *calloc(size_t nelem, size_t elsize)`

```
struct person *personen;
personen = (struct person *)calloc(n, sizeof(struct person));
if(person == NULL) ...
```

- ◆ `calloc` initialisiert den Speicher mit 0

- ◆ `malloc` initialisiert den Speicher nicht

- ◆ explizite Initialisierung mit `void *memset(void *s, int c, size_t n)`

```
memset(personen, 0, sizeof(struct person)*n);
```

17 Portable Programme

- 1. Verwenden der standardisierten Programmiersprache ANSI-C

- ◆ gcc-Aufrufoptionen

```
-ansi -pedantic
```

- 2. Verwenden einer standardisierten Betriebssystemschnittstelle, z.B. POSIX

- ◆ gcc-Aufrufoption

```
-D_POSIX_SOURCE
```

- ◆ oder `#define` im Programmtext

```
#define _POSIX_SOURCE
```

- Programm sollte sich mit folgenden gcc-Aufruf compilieren lassen

```
gcc -ansi -pedantic-errors -D_POSIX_SOURCE -Wall -Werror
```

17.1 POSIX

- Standardisierung der Betriebssystemschnittstelle:

Portable Operating System Interface (IEEE Standard 10003.1)

- POSIX.1 wird von verschiedenen Betriebssystemen implementiert:

- ◆ SUN Solaris 2.6

- ◆ SGI Irix 6.2/6.4

- ◆ DIGITAL Unix 4.0

- ◆ Linux (größtenteils POSIX, zertifizierte Version von Fa. Unifix)

- ◆ Windows NT (Posix Subsystem)

- ◆ ...



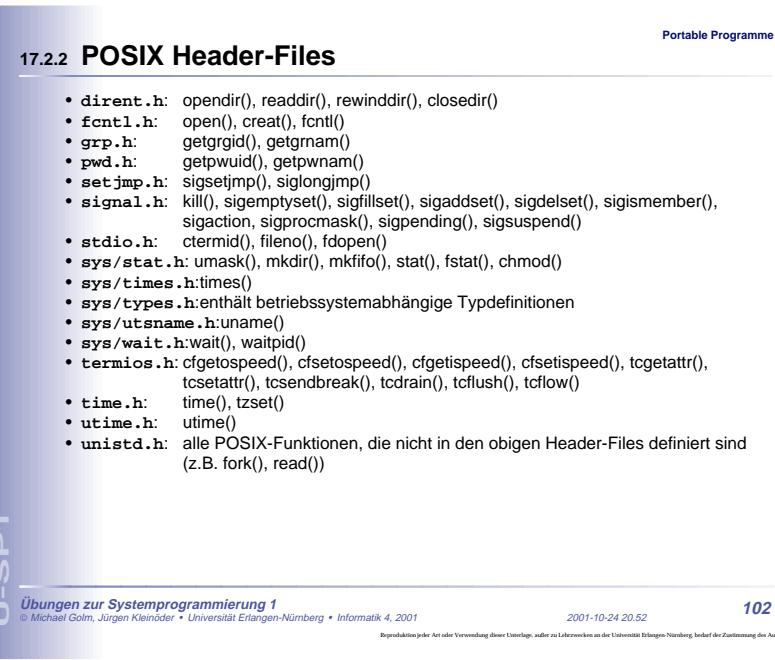
17.2 Header-Files: ANSI und POSIX

- In den Standards ANSI-C und POSIX.1 sind Header-Files definiert, mit
 - ◆ Funktionsprototypen
 - ◆ typedefs
 - ◆ Makros und Defines
- ◆ Wenn in der Aufgabenstellung nicht anders angegeben, sollen ausschließlich diese Header-Files verwendet werden.

Portable Programme

100

U-SP1



17.2.2 POSIX Header-Files

- dirent.h: opendir(), readdir(), rewinddir(), closedir()
- fcntl.h: open(), creat(), fcntl()
- grp.h: getgrgid(), getgrnam()
- pwd.h: getpwuid(), getpwnam()
- setjmp.h: sigsetjmp(), siglongjmp()
- signal.h: kill(), sigemptyset(), sigfillset(), sigaddset(), sigdelset(), sigismember(), sigaction, sigprocmask(), sigpending(), sigsuspend()
- stdio.h: ctermid(), fileno(), fdopen()
- sys/stat.h: umask(), mkdir(), mkfifo(), stat(), fstat(), chmod()
- sys/times.h: times()
- sys/types.h: enthält betriebssystemabhängige Typdefinitionen
- sys/utname.h: uname()
- sys/wait.h: wait(), waitpid()
- termios.h: cfgetospeed(), cfsetospeed(), cfgetispeed(), cfsetispeed(), tcgetattr(), tcsetattr(), tcsendbreak(), tcdrain(), tcflush(), tcflow()
- time.h: time(), tzset()
- utime.h: utime()
- unistd.h: alle POSIX-Funktionen, die nicht in den obigen Header-Files definiert sind (z.B. fork(), read())

Portable Programme

102

U-SP1



17.2.1 ANSI-C Header-Files

Portable Programme

- assert.h: assert()-Makro
- ctype.h: Makros und Funktionen für Characters (z.B. tolower(), isalpha())
- errno.h: Fehlerauswertung (z.B. errno-Variable)
- float.h: Makros für Fließkommazahlen
- limits.h: Enthält Definitionen für Systemschranken
- locale.h: Funktion setlocale()
- math.h: Mathematische Funktionen für double
- setjmp.h: Funktionen setjmp(), longjmp()
- signal.h: Signalbehandlung
- stdarg.h: Funktionen und Makros für variable Argumentlisten
- stddef.h: Def. von ptrdiff_t, NULL, size_t, wchar_t, offsetp, errno
- stdio.h: I/O Funktionen (z.B. printf(), scanf(), fgets())
- stdlib.h: Hilfsfunktionen (z.B. malloc(), getenv(), rand())
- string.h: Stringmanipulation (z.B. strcpy())
- time.h: Zeitmanipulation (z.B. time(), ctime(), strftime())

101

U-SP1



17.2.3 POSIX Datentypen

Portable Programme

- Betriebssystemabhängige Typen aus <sys/types.h>:

- dev_t: Gerätenummer
- gid_t: Gruppen-ID
- ino_t: Seriennummer von Dateien (Inodenummer)
- mode_t: Dateiattribute (Typ, Zugriffsrechte)
- nlink_t: Hardlink-Zähler
- off_t: Dateigrößen
- pid_t: Prozeß-ID
- size_t: entspricht dem ANSI-C size_t
- ssize_t: Anzahl von Bytes oder -1
- uid_t: User-ID

103

U-SP1

17.3 XOPEN / Unix98

- Die Open Group
 - ◆ Eigentümer des Markenzeichens "UNIX"
 - ◆ Erstellen Spezifikationen (Systemaufruf-Schnittstellen, Programme, ...)
- Hersteller können für ihr Betriebssystem ein "Branding" erwerben
- The Single UNIX® Specification (UNIX 95)
 - ◆ enthält STREAMS, Sockets, XTI, POSIX.1, BSD und SVID Schnittstellen
 - ◆ Solaris 2.5 and 2.5.1, HP-UX 10.10, IBM AIX 4.2, Digital Unix 4
- The Single UNIX® Specification, Version 2 (Unix98)
 - ◆ <http://www.opengroup.org/onlinepubs/007908799/>
 - ◆ http://www4/Lehre/WS00/V_SP1/Uebung/xopen/susv2/
 - ◆ Unix98-Erweiterungen mit: `#define _XOPEN_SOURCE 500`

18 Systemaufrufe vs. Bibliotheksaufrufe

