

# U8 Verzeichnisse und Sortieren

- Besprechung Aufgabe 6
- Dynamische Speicherverwaltung
- POSIX-Verzeichnis-Systemchnittstelle
- Datei-Attribute in Inodes
- Generisches Sortieren
- Aufgabe 7

# U8-1 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(personen == NULL) ...
```

- ◆ `calloc` initialisiert den Speicher mit 0
- ◆ `malloc` initialisiert den Speicher nicht

## ■ Freigeben von Speicher

```
void free(void *ptr);
```

- ◆ nur Speicher, der mit einer der alloc-Funktionen zuvor angefordert wurde, darf mit `free` freigegeben werden!

# U8-1 Dynamische Speicherverwaltung

- Längenänderung von dynamisch allokierten Feldern:

```
void *realloc(void *ptr, size_t size)
```

- Die Position im Speicher kann sich hierbei evtl. ändern
  - An das Feld angrenzender Speicher ist bereits belegt
  - **realloc** sucht einen neuen, ausreichend großen Speicherbereich und kopiert das ursprüngliche Feld an dessen Anfang
  - **realloc** liefert einen Zeiger auf die neue Position zurück
- Beispiel: Erweiterung des allokierten Felds um 10 weitere Strukturen

```
neu = (struct person *)realloc(personen,
                               (n+10) * sizeof(struct person));
if(neu == NULL) ...
```

- Wenn **realloc** keinen ausreichend großen Bereich findet, wird **NULL** zurückgegeben und der ursprüngliche Bereich bleibt erhalten.

# U8-2 POSIX-Verzeichnis-Systemschnittstelle

- Verzeichnisse öffnen: **opendir(3)**
- Verzeichnisse lesen: **readdir(3)**
- Verzeichnisse schliessen: **closedir(3)**

# 1 opendir / closedir

## ■ Funktions-Prototypen:

```
#include <sys/types.h>
#include <dirent.h>

DIR *opendir(const char *dirname);

int closedir(DIR *dirp);
```

## ■ Argument von opendir

- ◆ **dirname**: Verzeichnisname

## ■ Rückgabewert: Zeiger auf Datenstruktur vom Typ **DIR** oder **NULL**

- initialisiert einen internen Zeiger des directory-Funktionsmoduls auf den ersten Directory-Eintrag (für den ersten readdir-Aufruf)
- closedir schliesst ein geöffnetes Verzeichnis nach Bearbeitungsende

## 2 readdir

- liefert einen Directory-Eintrag (interner Zeiger) und setzt den Zeiger auf den folgenden Eintrag
- Funktions-Prototyp:

```
#include <sys/types.h>
#include <dirent.h>

struct dirent *readdir(DIR *dirp);
```

- Argumente
  - ◆ **dirp**: Zeiger auf **DIR**-Datenstruktur (von **opendir(3)**)
- Rückgabewert: Zeiger auf Datenstruktur vom Typ **struct dirent** oder **NULL**, wenn **EOF** erreicht wurde oder im Fehlerfall
  - bei **EOF** bleibt **errno** unverändert (kritisch, kann vorher beliebigen Wert haben), im Fehlerfall wird **errno** entsprechend gesetzt
  - **errno** vorher auf 0 setzen, sonst kann **EOF** nicht sicher erkannt werden!

## 2 ... readdir

- Problem: Der Speicher für die zurückgelieferte `struct dirent` wird von den dir-Bibliotheksfunktionen selbst angelegt und bei jedem Aufruf wieder verwendet!
  - ◆ werden Daten aus der dirent-Struktur länger benötigt, müssen sie vor dem nächsten readdir-Aufruf in Sicherheit gebracht (kopiert) werden
  - ◆ konzeptionell schlecht
    - aufrufende Funktion arbeitet mit Zeiger auf internen Speicher der readdir-Funktion
  - ◆ in nebenläufigen Programmen (mehrere Threads) nicht einsetzbar!
    - man weiss evtl. nicht, wann der nächste readdir-Aufruf stattfindet
- readdir ist ein klassisches Beispiel für schlecht konzipierte Schnittstellen in der C-Funktionsbibliothek

### 3 struct dirent

- Definition unter Linux (/usr/include/bits/dirent.h)

```
struct dirent {  
    __ino_t d_ino;  
    __off_t d_off;  
    unsigned short int d_reclen; /* tatsächl. Länge der Struktur */  
    unsigned char d_type;  
    char d_name[256];  
};
```

- POSIX: **d\_name** ist ein Feld unbestimmter Länge, max. **NAME\_MAX** Zeichen

# U8-3 Datei-Attribute ermitteln: stat

- liefern Datei-Attribute aus dem Inode
- Funktions-Prototyp:

```
#include <sys/types.h>
#include <sys/stat.h>
int stat(const char *path, struct stat *buf);
```

- Argumente:
  - ◆ **path**: Dateiname
  - ◆ **buf**: Zeiger auf Puffer, in den Inode-Informationen eingetragen werden
- Rückgabewert: 0 wenn OK, -1 wenn Fehler
- Beispiel:

```
struct stat buf;
stat("/etc/passwd", &buf); /* Fehlerabfrage ... */
printf("Inode-Nummer: %d\n", buf.st_ino);
```

# 1 stat: Ergebnisrückgabe im Vergleich zur readdir

- problematische Rückgabe auf funktions-internen Speicher wie bei `readdir` gibt es bei `stat` nicht
- Grund: `stat` ist ein Systemaufruf - Vorgehensweise wie bei `readdir` wäre gar nicht möglich
- der logische Adressraum des Anwendungsprogramms ist nur eine Teilmenge (oder sogar komplett disjunkt) von dem logischen Adressraum des Betriebssystems
  - Betriebssystemspeicher ist für Anwendung nicht sichtbar/zugreifbar
  - Funktionen des Kernels (wie `stat`) können keine Zeiger auf ihre internen Datenstrukturen an Anwendungen zurückgeben

# 1 stat / lstat: stat-Struktur

- **`dev_t st_dev;`** Gerätenummer (des Dateisystems) = Partitions-Id
- **`ino_t st_ino;`** Inodenummer (Tupel `st_dev, st_ino` eindeutig im System)
- **`mode_t st_mode;`** Dateimode, u.a. Zugriffs-Bits und Dateityp
- **`nlink_t st_nlink;`** Anzahl der (Hard-) Links auf den Inode (Vorl. 7-32)
- **`uid_t st_uid;`** UID des Besitzers
- **`gid_t st_gid;`** GID der Dateigruppe
- **`dev_t st_rdev;`** DeviceID, nur für Character oder Blockdevices
- **`off_t st_size; Dateigröße in Bytes`**
- **`time_t st_atime;`** Zeit des letzten Zugriffs (in Sekunden seit 1.1.1970)
- **`time_t st_mtime;`** Zeit der letzten Veränderung (in Sekunden ...)
- **`time_t st_ctime;`** Zeit der letzten Änderung der Inode-Information (...)
- **`unsigned long st_blksize;`** Blockgröße des Dateisystems
- **`unsigned long st_blocks;`** Anzahl der von der Datei belegten Blöcke

# U8-4 Generisches Sortieren mit qsort

## 1 Funktion *qsort(3)*

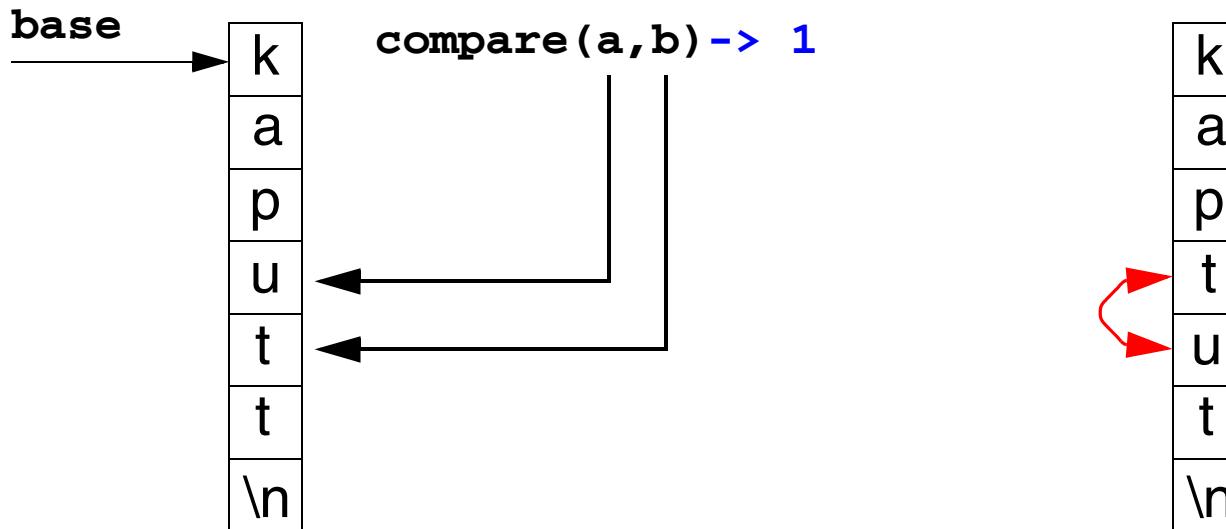
- Prototyp aus `stdlib.h`:

```
void qsort(void *base,
           size_t nel,
           size_t width,
           int (*compare) (const void *, const void *));
```

- Bedeutung der Parameter:
  - ◆ **base** : Zeiger auf das erste Element des Feldes, dessen Elemente sortiert werden sollen
  - ◆ **nel** : Anzahl der Elemente im zu sortierenden Feld
  - ◆ **width**: Größe eines Elements
  - ◆ **compare**: Vergleichsfunktion
- Sortiert ein Feld von beliebigen Elementen

## 2 Arbeitsweise von *qsort(3)*

- **qsort** vergleicht je zwei Elemente mit Hilfe der Vergleichsfunktion **compare**
- sind die Elemente zu vertauschen, dann werden die entsprechenden Felder komplett ausgetauscht, z.B.:



### 3 Vergleichsfunktion

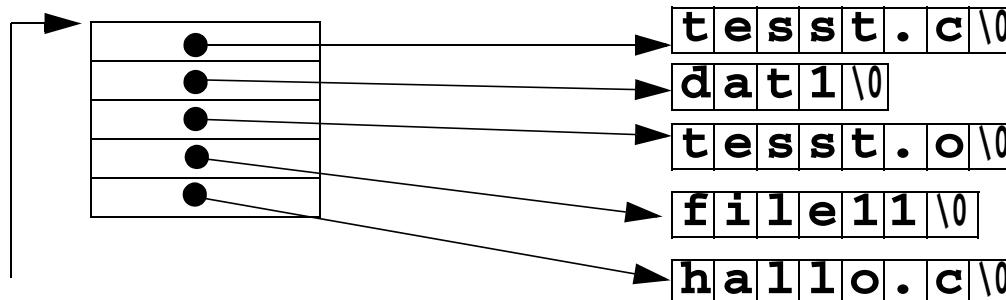
- Die Vergleichsfunktion erhält Zeiger auf Feldelemente, d.h. die übergebenen Zeiger haben denselben Typ wie das Feld
- Die Funktion vergleicht die beiden Elemente und liefert:
  - <0, falls Element 1 kleiner bewertet wird als Element 2
  - 0, falls Element 1 und Element 2 gleich gewertet werden
  - >0, falls Element 1 größer bewertet wird als Element 2

#### ■ Beispiel

- |                |      |
|----------------|------|
| ◆ 'z', 'a'     | ⇒ 1  |
| ◆ 1, 5         | ⇒ -1 |
| ◆ 5,5          | ⇒ 0  |
| ◆ "foo", "bar" | ⇒ 1  |

## 4 Beispiel: Strings sortieren

- Strings: '\0'-terminierte Zeichenfolgen *beliebiger* (unbekannter) Länge
  - **qsort** sortiert Elemente *fester* Länge
  - direkte Sortierung mit **qsort** daher nicht möglich
- Ein Feld aus Zeigern auf die Strings sortieren
  - die Zeiger haben alle die gleiche *feste* Länge
  - Feldelemente haben den Typ `char *`



```
char **strings = malloc(n * sizeof(char *)) ;
```

## 4 Sortieren des Zeigerfeldes mit qsort

### ■ Funktion qsort aufrufen

```
void qsort(void *base,  
          size_t nel,  
          size_t width,  
          int (*compare) (const void *, const void *));
```

### ■ Bedeutung der Parameter:

- ◆ **base** : Zeiger auf erstes Element des zu sortierenden Feldes (*Zeiger auf den Zeiger auf ersten String*)
- ◆ **nel** : Anzahl der Elemente im zu sortierenden Feld (*Zahl der Zeiger/Strings*)
- ◆ **width**: Größe eines Elements (*Größe eines char-Zeigers - sizeof*)
- ◆ **compare**: Zeiger auf Vergleichsfunktion

## 4 Beispiel: Compare Funktion

### ■ Lösung für die compare-Funktion:

- **qsort** übergibt der compare-Funktion Zeiger auf Elemente
- **compare** wird also mit Zeigern auf **char \*** aufgerufen ➔ **char \*\***
- **strcmp(3)** wird verwendet, um die Strings zu vergleichen

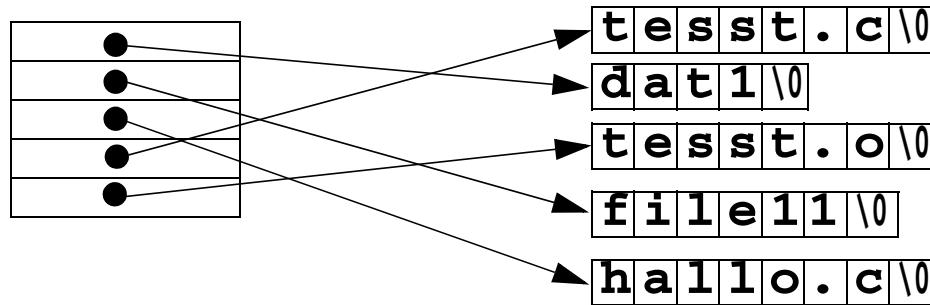
```
int compare(const char **a, const char **b) {
    return strcmp( *a, *b );
}
```

### ■ Aufruf von qsort (**nel=5**) im Beispiel

```
qsort(strings, nel, sizeof(char *),
      (int (*) (const void *, const void *)) compare);
```

- Funktionstyp der **compare**-Funktion ist generisch (mit **void \***)
- Typ der **compare**-Funktion muss für den **qsort**-Aufruf gecastet werden

## 4 Resultat nach dem Aufruf von qsort



- die Strings stehen unverändert, nur die Zeiger im Feld wurden sortiert

# U8-5 Aufgabe 7

## 1 Teilaufgabe a)

- Argumente aus der Kommandozeile ausgeben
- Vorlesung F.36 ff

## 2 Teilaufgabe b)

- Directory öffnen (**opendir(3)**, Vorlesung J.20)
- Schleife: Einträge lesen (**readdir(3)**, Vorlesung J.21)
- Inode-Information des Eintrags lesen (**stat(2)**)
- Dateinamen und -größe ausgeben (**printf(3)**, `d_name` aus dirent-Struktur, `st_size` aus stat-Struktur)

### 3 Teilaufgabe c)

#### ■ Problem

- Rückgabewert von readdir nur bis zum nächsten Aufruf von readdir gültig
- Daten müssen aus dirent-Struktur wegkopiert werden

#### ■ Lösung

- Speicher für Dateinamen mit *malloc(3)* besorgen
- benötigten Speicherplatz mir *strlen(3)* ermitteln
- Platz für abschließendes ' \0 ' -Zeichen nicht vergessen!
- Dateiname in neuen Speicher umkopieren (*strcpy(3)*)

### 3 Teilaufgabe c)

#### ■ Problem 2

- Einträge in einem Directory sind nicht sortiert
- Einträge sollen nach Dateigröße sortiert werden

#### ■ Lösung

- Dateigröße zusammen mit den Namen in einer Struktur speichern
- Feld mit Zeigern auf diese Strukturen mit der Funktion **qsort(3)** sortieren

#### ■ nach dem Sortieren

- ◆ mit `printf( "\t%ld\t%s\n" , . . . )` alle Einträge des Feldes ausgeben
- ◆ dynamisch allokierten Speicher wieder freigeben

### 4 Teilaufgabe d)

#### ■ beliebige viele Verzeichniseinträge sortieren

- Lösung: Zeigerfeld mit **realloc(3)** nach Bedarf vergrössern