

U3 3. Übung

- Vorstellen der Aufgabe 1: lilo
- Fehlerbehandlung
- Dynamische Speicherverwaltung
- Debugging mit GDB
- Generisches Sortieren mit **qsort(3)**
- Übersetzen von Projekten mit Hilfe von **make(1)**

U3-1 Fehlerbehandlung

U3-1 Fehlerbehandlung

- Fehler können aus unterschiedlichsten Gründen im Programm auftreten
 - Systemressourcen erschöpft
 - ⇒ **malloc(3)** schlägt fehl
 - Fehlerhafte Benutzereingaben (z.B. nicht existierende Datei)
 - ⇒ **open(2)** schlägt fehl
 - Transiente Fehler (z.B. nicht erreichbarer Server)
 - ...
- Gute Software **erkennt Fehler**, führt eine **angebrachte Behandlung** durch und gibt eine **aussagekräftige Fehlermeldung** aus
- Kann das Programm trotz des Fehlers sinnvoll weiterlaufen?
 - ◆ Beispiel 1: Benutzer gibt ungültige URL in den Browser ein
 - ⇒ Fehlerbehandlung: Fehlermeldung anzeigen, Programm läuft weiter
 - ◆ Beispiel 2: Kopierprogramm: Öffnen der Quelldatei schlägt fehl
 - ⇒ Fehlerbehandlung: Fehlermeldung anzeigen, Kopieren nicht möglich, Programm beenden

1 Fehlerbehandlung: Beendigung des Programmes

- Tritt ein Fehler auf, der ein sinnvolles Weiterarbeiten verhindert, muss das Programm beendet werden und einen Programmabbruch anzeigen.
- Signalisierung des Fehlers an Aufrufer des Programms über den Exitstatus
 - ◆ Exitstatus 0 zeigt erfolgreiche Programmausführung an
 - ◆ Werte ungleich 0 zeigen einen Fehler bei der Ausführung an
 - Die Bedeutung des entsprechenden Wertes ist nicht standardisiert
 - Manchmal enthält die Manpage Informationen über die Bedeutung des Exitstatus
- libc bietet vordefinierte Makros für den Exitstatus an:
 - ◆ `EXIT_SUCCESS`
 - ◆ `EXIT_FAILURE`
- Exitstatus des letzten Befehls ist in der Shell-Variable `$?` gespeichert

2 Erkennung und Ausgabe von Fehlern

- Fehler treten häufig in Funktionen der C-Bibliothek auf
 - erkennbar i.d.R. am Rückgabewert (Manpage, Sektion **RETURN VALUES**)
- Die Fehlerursache wird über die globale Variable `errno` übermittelt
 - Der Wert `errno=0` ist reserviert, alles andere ist ein Fehlercode
 - Bibliotheksfunktionen setzen `errno` im Fehlerfall (sonst nicht *zwingend*)
 - Bekanntmachung im Programm durch Einbinden von `errno.h`
- Fehlercodes können mit `perror(3)` als lesbare Strings ausgegeben werden

```
char *mem = malloc(...); // malloc gibt im Fehlerfall
if(NULL == mem) {        // NULL zurück
    perror("malloc");     // Ausgabe der Fehlerursache
    exit(EXIT_FAILURE);   // Programm mit Fehlercode beenden
}
```

- ◆ `perror(3)` darf nur verwendet werden, wenn die `errno` gesetzt wurde
- ◆ sonst mit Hilfe von `fprintf(3)` eigene Fehlermeldung auf `stderr` ausgeben

3 Fehlererkennung bei fgets

```
while (fgets(buffer, 102, stdin) != NULL) {
    ...
}
// EOF oder Fehler?
```

◆ Rückgabewert `NULL` sowohl im Fehlerfall als auch bei End-of-File

■ Erkennung im Fall von I/O-Streams mit **ferror(3)** und **feof(3)**

```
while (fgets(buffer, 102, stdin) != NULL) {
    ...
}
// EOF oder Fehler?
if (ferror(stdin)) {
    // Fehler
}
```

U3-2 Debuggen mit dem gdb

■ Ein Debugger dient zum Suchen und Finden von Fehlern in Programmen

■ Im Debugger kann man u.a.

- ◆ das Programm schrittweise abarbeiten
- ◆ Variablen- und Speicherinhalte ansehen und modifizieren
- ◆ core dumps (Speicherabbilder beim Programmabsturz) analysieren
 - Erlauben von core dumps (in der laufenden Shell):
z. B. `limit coredumpsize 1024k` oder `limit coredumpsize unlimited`

■ Programm muss mit der Compileroption **-g** übersetzt werden

```
gcc -g -o hello hello.c
```

■ Aufruf des Debuggers mit `gdb <Programmname>`

```
gdb ./hello
```

1 Auffinden von Fehlern mit Hilfe eines Debuggers

```
void initArray(int *array, unsigned int size) {
    int i;
    for ( i=0; i<=size; i++ ) {
        array[i] = 0;
    }
}

int main(int argc, char *argv[]) {
    int *array;
    int buf[8];
    array = buf;

    initArray(buf,8);

    while ( array != buf+8 ) {
        printf("%d\n", *array);
        array++;
    }

    exit(EXIT_SUCCESS);
} /* Auch unter /proj/i4sp1/pub/gdb/gdb.c */
```

2 Referenz: Kommandos des GDBs

- Programmausführung beeinflussen
 - ◆ Breakpoints setzen:
 - **b** [<Dateiname>:]<Funktionsname>
 - **b** <Dateiname>:<Zeilennummer>
 - ◆ Starten des Programms mit **run** (+ evtl. Befehlszeilenparameter)
 - ◆ Fortsetzen der Ausführung bis zum nächsten Stop mit **c** (continue)
 - ◆ schrittweise Abarbeitung auf Ebene der Quellsprache mit
 - **s** (step: läuft in Funktionen hinein)
 - **n** (next: behandelt Funktionsaufrufe als einzelne Anweisung)
 - ◆ Breakpoints anzeigen: **info breakpoints**
 - ◆ Breakpoint löschen: **delete breakpoint#**

2 Referenz: Kommandos des GDBs

- Variableninhalte anzeigen/modifizieren
 - ◆ Anzeigen von Variablen mit: **p *expr***
 - ***expr*** ist ein C-Ausdruck, im einfachsten Fall der Name einer Variable
 - ◆ Automatische Anzeige von Variablen bei jedem Programmstopp (Breakpoint, Step, ...): **display *expr***
 - ◆ Setzen von Variablenwerten mit **set <variablenname>=<wert>**
- Ausgabe des Funktionsaufruf-Stacks (backtrace): **bt**
- Quellcode an aktueller Position anzeigen: **list**
- Watchpoints: Stoppt Ausführung bei Zugriff auf eine bestimmte Variable
 - ◆ **watch *expr***: Stoppt, wenn sich der Wert des C-Ausdrucks ***expr*** ändert
 - ◆ **rwatch *expr***: Stoppt, wenn ***expr*** gelesen wird
 - ◆ **awatch *expr***: Stopp bei jedem Zugriff (kombiniert **watch** und **rwatch**)
 - ◆ Anzeigen und Löschen analog zu den Breakpoints

U3-3 Dynamische Speicherverwaltung - Nachtrag

- Verändern der Größe von Feldern, die durch **malloc()** bzw. **calloc()** erzeugt wurden:

```
int* numbers = malloc( n*sizeof(int) );
if ( numbers == NULL ) { ... }

... // Speicherbedarf gestiegen

neu = realloc( numbers, (n+10) * sizeof(int));
if(neu == NULL) { ... free(numbers); ...}
numbers = neu;
```

- ◆ Neuer Speicherbereich enthält die Daten des ursprünglichen Speicherbereichs (wird automatisch kopiert; aufwändig)
- ◆ Sollte **realloc()** fehlschlagen, wird der ursprüngliche Speicherbereich nicht freigegeben
 - Explizite Freigabe mit **free()** notwendig.

U3-4 Generisches Sortieren mit qsort

- Vergleich nahezu beliebiger Daten
 - ◆ alle Daten müssen die gleiche Größe haben
- qsort weiß nicht, was es sortiert (wie der Vergleich zu bewerkstelligen ist)
 - ◆ Aufrufer stellt Routine zum Vergleich zweier Elemente zur Verfügung

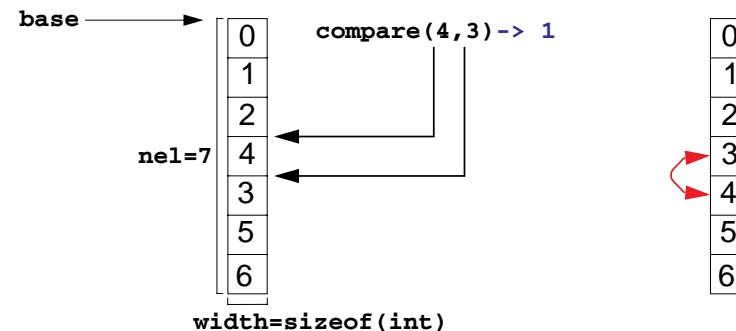
- Prototyp aus `stdlib.h`:

```
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 zu sortierenden Feldes
 - ◆ **nel**: Anzahl der Elemente im zu sortierenden Feld
 - ◆ **width**: Größe eines Elements
 - ◆ **compare**: Vergleichsfunktion

1 Arbeitsweise von qsort(3)

- qsort vergleicht je zwei Elemente mit Hilfe der Vergleichsfunktion
 - ◆ sind die beiden Elemente falsch angeordnet, werden diese getauscht



- ◆ Die Funktion vergleicht die beiden Elemente und liefert:
 - < 0, falls Element 1 kleiner gewertet wird als Element 2(1, 5 : -4)
 - 0, falls Element 1 und Element 2 gleich gewertet werden (5, 5 : 0)
 - > 0, falls Element 1 größer gewertet wird als Element 2(9, 1 : 8)

2 Vergleichsfunktion

- Die Vergleichsfunktion erhält Zeiger auf Feldelemente

- ◆ Beispiel: Vergleichsfunktion für `int`

```
int intCompare(const int *, const int *);
```

- `qsort()` kennt den tatsächlichen Datentyp nicht

- ◆ Prototyp ist generisch mit `void`-Zeigern parametrisiert

```
void qsort(..., int (*compare) (const void *, const void *));
```

- Cast erforderlich

- ◆ entweder innerhalb der Funktion `intCompare()`

- ◆ oder bei der Übergabe des Funktionszeigers an `qsort()`

- `const`-Zusicherung: Funktion ändert die verglichenen Werte nicht

3 Exkurs: Funktionszeiger

- Notation von Funktionstypen

```
int (*) (const void *, const void *)
```

Rückgabetypp

Funktionszeiger

Parameterliste

- ◆ wird ein Name benötigt, wird dieser hinter dem geklammerten `*` notiert

- Cast wie bei allen anderen Datentypen

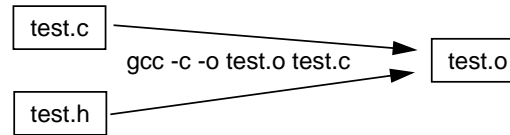
```
int intCompare(const int *, const int *); // Prototyp
```

```
int (*compare) (const void *, const void *) =  
    (int (*) (const void *, const void *)) intCompare;
```

- ◆ Funktionszeiger-Variablen mit Namen `compare` wird die Adresse der typinkompatiblen Funktion `intCompare` zugewiesen

U3-5 Make

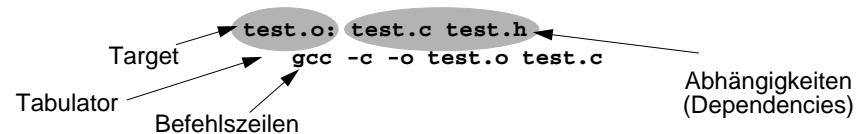
- Grundsätzlich: Erzeugung von Dateien aus anderen Dateien
 - ◆ für uns interessant: Erzeugung einer .o-Datei aus einer .c-Datei



- Ausführung von *Update*-Operationen (auf Basis der Modifikationszeit)

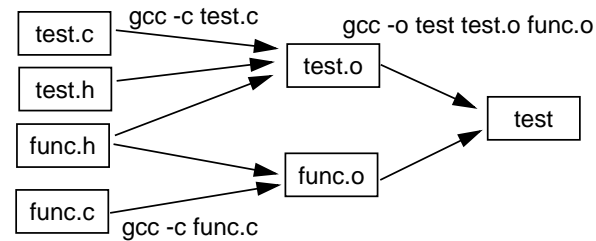
1 Funktionsweise

- Regeldatei mit dem Namen **Makefile**
 - ◆ Targets (was wird erzeugt?, hier: test.o)
 - ◆ Abhängigkeiten (woraus?, hier: test.c, test.h)
 - ◆ Befehlszeilen (wie?, hier: entsprechendes GCC-Kommando)



- ◆ jede Zeile wird in einer neuen Shell ausgeführt
 - **cd** in einer Zeile hat keine Auswirkung auf die nächste Zeile
- das zu erstellende Target kann beim **make**-Aufruf angegeben werden (z.B. **make test.o**)
 - ◆ ohne explizite Target-Angabe bearbeitet make das **erste** Target im Makefile
 - Dieses ist normalerweise das Target **all**

2 Beispiel mit mehreren Modulen



```
all: test
```

```
test: test.o func.o
    gcc -o test test.o func.o
```

```
test.o: test.c test.h func.h
    gcc -c test.c
```

```
func.o: func.c func.h
    gcc -c func.c
```

3 Makros

- in einem Makefile können Makros definiert werden

```
SOURCE = test.c func.c
```

- Verwendung der Makros mit \$(NAME) oder \${NAME}

```
test: $(SOURCE)
    gcc -o test $(SOURCE)
```

- Erzeugen neuer Makros durch Ersetzung in existierenden Makros

```
OBJS = $(SOURCE:%.c=%.o)
```

- ◆ In allen Wörtern, die auf den Suchstring .c enden, wird dieser durch .o ersetzt

- Erzeugung neuer Makros durch Konkatenation

```
◆ ALLOBJS = $(OBJS) hallo.o
```

- Gängige Makros:

- ◆ cc C-Compiler-Befehl
- ◆ CFLAGS Optionen für den C-Compiler

4 Dynamische Makros

- `$@` Name des Targets (hier: `test`)

```
test: test.c
$(CC) -o $@ test.c
```

- `$*` Basisname des Targets (ohne Dateiendung, hier `test`)

```
test.o: test.c test.h
$(CC) -c $*.c
```

- `$<` Name der ersten Abhängigkeit

```
test.o: test.c test.h
$(CC) -c $<
```

- `$^` Mit Leerzeichen getrennte Liste aller Abhängigkeiten

```
test: test.o func.o
$(CC) -o $@ $^
```

5 Suffix-Regeln

- Allgemeine Regel zur Erzeugung einer Datei mit einer bestimmten Endung aus einer gleichnamigen Datei mit einer anderen Endung.

- Beispiel: Erzeugung von `.o`-Dateien aus `.c`-Dateien

```
%.o: %.c
$(CC) $(CFLAGS) -c $<
```

- Explizite Regeln überschreiben die Suffix-Regeln

```
test.o: test.c
$(CC) $(CFLAGS) -DXYZ -c $<
```

- Regeln ohne Kommandos können Abhängigkeiten überschreiben

```
test.o: test.c test.h func.h
```

- ◆ die Suffix-Regel wird weiterhin zur Erzeugung herangezogen

6 Pseudo-Targets

- Dienen nicht der Erzeugung einer gleichnamigen Datei
- Deklaration als Abhängigkeiten des Spezial-Targets **.PHONY**

```
.PHONY: all clean install
```

 - ◆ so deklarierte Targets werden immer gebaut, auch wenn eine gleichnamige Datei bereits existiert, die aktueller als die Abhängigkeiten ist
- Aufräumen mit **make clean**

```
clean:
    rm -f $(OBJS) test
```
- Projekt bauen mit **make all** (Konvention: **all** ist immer erstes Target)

```
all: test
```
- Installieren mit **make install**

```
install: all
    cp test /usr/local/bin
```

7 Beispiel verbessert

```
SOURCE = test.c func.c
OBJS = $(SOURCE:%.c=%.o)
HEADER = $(SOURCE:%.c=%.h)
CC = gcc
CFLAGS = -Wall -Werror -std=c99 -pedantic -D_XOPEN_SOURCE=600

.PHONY: all

all: test

test: $(OBJS)
    $(CC) $(LDFLAGS) -o $@ $(OBJS)

%.o: %.c
    @echo Folgende C-Datei wird neu uebersetzt: $<
    $(CC) $(CFLAGS) -c $<

# korrekte Abhaengigkeiten
test.o: test.c $(HEADER)
func.o: func.c func.h
```