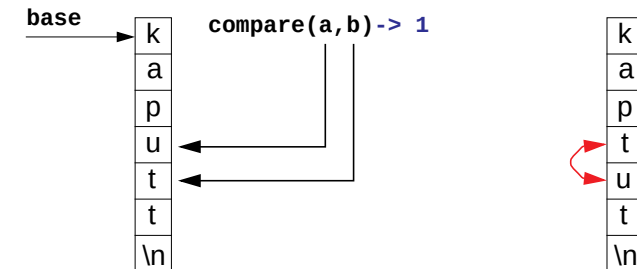


U2-1 Überblick

- Ein-Ausgabefunktionen in C (letzter Abschnitt Vorlesungsstoff ab C.110)
- Aufgabe 2: qsort
- Debugger
- Übersetzen von Projekten mit "make"

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.:



B-2 Aufgabe 2: Sortieren mittels 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

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

B.3 Debuggen mit dem gdb

- Programm muß mit der Compileroption **-g** übersetzt werden

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

- Aufruf des Debuggers mit **gdb <Programmname>**

```
gdb hello
```

- im Debugger kann man u.a.
 - ◆ Breakpoints setzen
 - ◆ das Programm schrittweise abarbeiten
 - ◆ Inhalt Variablen und Speicherinhalte ansehen und modifizieren
- Debugger außerdem zur Analyse von core dumps
 - ◆ Erlauben von core dumps:
z. B. **limit coredumpsize 1024k** oder **limit coredumpsize unlimited**

2 Variablen, Stack

- Anzeigen von Variablen mit **p <variablenname>**
- Automatische Anzeige von Variablen bei jedem Programmhalt (Breakpoint, Step, ...) mit **display <variablenname>**
- Setzen von Variablenwerten mit **set <variablenname>=<wert>**
- Ausgabe des Funktionsaufruf-Stacks: **bt**

1 Breakpoints

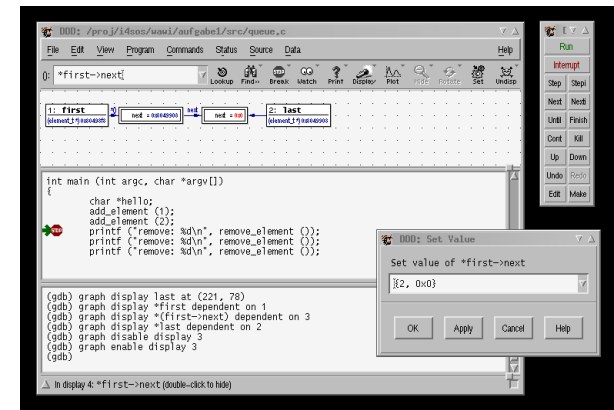
- Breakpoints:
 - ◆ **b <Funktionsname>**
 - ◆ **b <Dateiname>:<Zeilennummer>**
 - ◆ Beispiel: Breakpoint bei main-Funktion

```
b main
```

- Starten des Programms mit **run** (+ evtl. Befehlszeilenparameter)
- Schrittweise Abarbeitung mit
 - ◆ **s** (step: läuft in Funktionen hinein) bzw.
 - ◆ **n** (next: läuft über Funktionsaufrufe ohne in diese hineinzusteppen)
- Fortsetzen bis zum nächsten Breakpoint mit **c** (continue)
- Breakpoint löschen: **delete <breakpoint-nummer>**

3 The Data Display Debugger (DDD)

- Komfortable, grafische Schnittstelle für gdb



4 Emacs und gdb

- gdb lässt sich auch sehr komfortabel im Emacs verwenden
- Aufruf mit "**ESC-x gdb**" und bei der Frage "**Run gdb on file:**" das mit der **-g**-Option übersetzte ausführbare File angeben
- Breakpoints lassen sich (nachdem der gdb gestartet wurde) im Buffer setzen, in welchem das C-File bearbeitet wird: **CTRL-x SPACE**

1 Beispiel

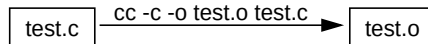
```
test: test.o func.o
cc -o test test.o func.o

test.o: test.c test.h func.h
cc -c test.c

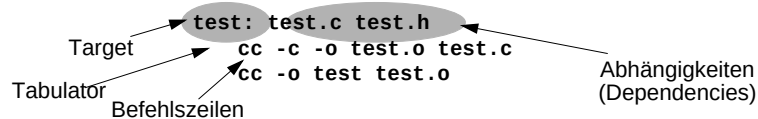
func.o: func.c func.h test.h
cc -c func.c
```

B-4 Make

- Problem: Es gibt Dateien, die aus anderen Dateien generiert werden.
 - ◆ Zum Beispiel kann eine test.o Datei aus einer test.c Datei unter Verwendung des C-Compilers generiert werden.



- Ausführung von *Update*-Operationen
- **Makefile**: enthält Abhängigkeiten und Update-Regeln (Befehlszeilen)



2 Allgemeines

- Kommentare beginnen mit # (bis Zeilenende)
- Befehlszeilen müssen mit TAB beginnen
- das zu erstellende Target kann beim **make**-Aufruf angegeben werden (z.B. **make test**)
 - ◆ wenn kein Target angegeben wird, bearbeitet make das erste Target im Makefile
- beginnt eine Befehlszeile mit @ wird sie nicht ausgegeben
- jede Zeile wird mit einer neuen Shell ausgeführt (d.h. z.B. **cd** in einer Zeile hat keine Auswirkung auf die nächste Zeile)

3 Makros

- in einem Makefile können Makros definiert werden
`SOURCE = test.c func.c`
- Verwendung der Makros mit `$(NAME)` oder `${NAME}`
`test: $(SOURCE)`
`cc -o test $(SOURCE)`

5 ... Makros

- Erzeugung neuer Makros durch Konkatenation
`OBJS += hallo.o`
oder
`OBJS = $(OBJS) hallo.o`
- Erzeugen neuer Makros durch Ersetzung in existierenden Makros
`OBJS_SOLARIS = $(OBJS:test.o=test_solaris.o)`
- Ersetzen mit Pattern-Matching
`SOURCE = test.c func.c`
`OBJS = $(SOURCE:%.c=%.o)`
- Benutzen von Befehlsausgaben
`WORKDIR = $(shell pwd)`

4 Dynamische Makros

- `$@` Name des Targets
`test: $(SOURCE)`
`cc -o $@ $(SOURCE)`
- `$*` Basisname des Targets
`test.o: test.c test.h`
`cc -c $*.c`
- `$?` Abhängigkeiten, die jünger als das Target sind
- `$<` Name einer Abhängigkeit (in impliziten Regeln)

6 Eingebaute Regeln und Makros

- `make` enthält eingebaute Regeln und Makros (`make -p` zeigt diese an)
- Wichtige Makros:
 - ◆ `CC` C-Compiler Befehl
 - ◆ `CFLAGS` Optionen für den C-Compiler
 - ◆ `LD` Linker Befehl
(in der Praxis wird aber meist `cc` verwendet, weil direkter Aufruf von `ld` die Standard-Bibliotheken nicht mit einbindet - `cc` ruft intern bei Bedarf automatisch `ld` auf)
 - ◆ `LDFLAGS` Optionen für den Linker
- Wichtige Regeln:
 - ◆ `.c.o` C-Datei in Objektdatei übersetzen
 - ◆ `.c` C-Datei übersetzen und linken

7 Suffix Regeln

- Eine Suffix Regel kann verwendet werden, wenn **make** eine Datei mit einer bestimmten Endung (z.B. **test.o**) benötigt und eine andere Datei gleichen Namens mit einer anderen Endung (z.B. **test.c**) vorhanden ist.

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

- Suffixe müssen deklariert werden
- ```
.SUFFIXES: .c .o $(SUFFIXES)
```

- Explizite Regeln überschreiben die Suffix-Regeln

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

## 9 Pseudo-Targets (PHONY)

- .PHONY-Targets

- Pseudo-Targets, die nicht die Erzeugung einer gleichnamigen Datei zum Ziel haben, sondern nur zum Aufruf einer Reihe von Kommandos dienen

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

- Aufräumen mit **make clean**

```
clean:
 rm -f $(OBJS)
```

- Projekt bauen mit **make all**

```
all: test
```

- Installieren mit **make install**

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

## 8 Beispiel verbessert

```
SOURCE = test.c func.c
OBJS = $(SOURCE:%.c=%.o)
HEADER = test.h func.h
```

```
test: $(OBJS)
 @echo Folgende Dateien erzwingen neu-linken von $@: $?
 $(CC) $(LDFLAGS) -o $@ $(OBJS)
```

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

```
test.o: test.c $(HEADER)
```

```
func.o: func.c $(HEADER)
```