

U2 2. Übung

- Vorstellen der Aufgabe 1: lilo
- Fehlerbehandlung
- Nachtrag zu Subversion
- Debugging mit GDB und valgrind
- Nachtrag zu dynamischer Speicherverwaltung
- Generisches Sortieren mit **qsort(3)**
- Vorgriff auf die Vorlesung
 - ◆ Argumente in der Kommandozeile
 - ◆ Ein/Ausgabekanäle eines Prozesses
 - ◆ Zeilenweises Einlesen/Ausgeben

U2-1 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)
 - ➡ **connect(2)** schlägt fehl
 - ...
- 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: Ermittlung des Hostnamens zu einer IP-Adresse für Logeintrag
 - ➡ Fehlerbehandlung: IP-Adresse im Log eintragen, Programm läuft weiter
 - ◆ Beispiel 2: Kopierprogramm: Öffnen der Quelldatei schlägt fehl
 - ➡ Fehlerbehandlung: Kopieren nicht möglich, Programm beenden

1 Fehler in Bibliotheksfunktionen

- 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
 - Fehlercode für jeden möglichen Fehler (siehe **errno(3)**)
 - 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 den Funktionen **perror(3)** und **strerror(3)** ausgegeben bzw. in lesbare Strings umgewandelt werden

```
char *mem = malloc(...); // malloc gibt im Fehlerfall
if(NULL == mem) {        // NULL zurück
    perror("malloc");     // Alternative zu strerror + fprintf
    exit(EXIT_FAILURE);   // Programm mit Fehlercode beenden
}
```

2 Fehlererkennung

- Signalisierung von Fehlern normalerweise durch Rückgabewert
- Nicht bei allen Funktionen möglich, z.B. **fgets(3)**

```
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  
}
```

U2-2 Nachtrag zu Subversion

1 Versionierungsschema

- Subversion nummeriert fortlaufend ab Revision 0 (1,2,3,...)
- spezielle Revisionsschlüsselwörter
 - ◆ HEAD: aktuelle Version des Repositories (neuste Version)
 - ◆ BASE: Revision eines Eintrags (Datei, Verzeichnis) der Arbeitskopie
 - ◆ COMMITTED: Letzte Änderungsrevision eines Eintrags älter als BASE
 - ◆ PREV: COMMITTED-1
- Revision zu einem bestimmten Zeitpunkt
 - ◆ {"2010-10-19 14:42"}

2 Kommentar bei commit

- Beim Aufruf von `svn commit` öffnet sich normalerweise ein Editor

- ◆ Im CIP ist dieses der Editor `joe`

- Zum Speichern und Verlassen des Editors `strg-k x` drücken
- Hilfemenü öffnet sich mit `strg-k h`

- ◆ Anderer Editor kann über die Umgebungsvariable `EDITOR` ausgewählt werden

```
alice@fau101[~]$ export EDITOR=vim
```

- Umgebungsvariable ist nur in dieser Shell-Sitzung gültig
- Durch Eintragen des Kommandos in die Konfigurationsdatei der eigenen Shell (z.B. `~/ .bashrc`) wird der Standardeditor für jede neue Shell geändert

- Übergabe des Kommentars als Argument von `svn commit`:

```
alice@fau101[/proj/i4sp/alice]$ svn commit -m "Ich schreibe  
lieber gleich in die Befehlszeile und nicht in den Editor"
```

2 Exkurs: Quoting von Zeichen mit Sonderbedeutung

- Sonderzeichen (wie <, >, &, Leerzeichen) soll als Argument übergeben werden:

```
cd Eigene Dateien
```

- Problem: die Shell interpretiert diese Zeichen
 - ◆ im Beispiel das Leerzeichen als Trenner mehrerer Argumente
 - ◆ das Kommando wird mit zwei Argumenten ausgeführt: *Eigene* und *Dateien*
- Lösung: Quoting nimmt Zeichen die Sonderbedeutung:
 - ◆ Voranstellen von \ nimmt genau einem Zeichen die Sonderbedeutung
 \ selbst wird durch \\ eingegeben
 - ◆ Klammern des gesamten Arguments durch " ",
 " selbst wird durch \" angegeben
 - ◆ Klammern des gesamten Arguments durch ' ',
 ' selbst wird durch \' angegeben

```
cd "Eigene Dateien"; cd 'Eigene Dateien'; cd Eigene\ Dateien
```

3 Basisoperationen

- diff: Lokale Änderungen der Arbeitskopie anzeigen

```
alice@fau101[/proj/i4sp/alice/trunk]$ svn status
M      hallo
alice@fau101[/proj/i4sp/alice/trunk]$ svn diff
Index: hallo
=====
--- hallo      (revision 23)
+++ hallo      (working copy)
@@ -0,0 +1 @@
+welt
```

- revert: Änderungen an Arbeitskopie zurücksetzen

```
alice@fau101[/proj/i4sp/alice/trunk]$ svn revert hallo
Reverted 'hallo'
alice@fau101[/proj/i4sp/alice/trunk]$ svn status
alice@fau101[/proj/i4sp/alice/trunk]$
```


3 Basisoperationen

- list/ls: Dateien/Verzeichnisse im Repository anzeigen

```
alice@fau01[/proj/i4sp/alice]$ svn ls
branches/
trunk/
```

- log: Historie anzeigen

```
alice@fau01[/proj/i4sp/alice]$ svn log
-----
r1 | www-data | 2010-04-20 15:03:14 +0200 (Tue, 20 Apr 2010) | 1 line
init repository
-----
```

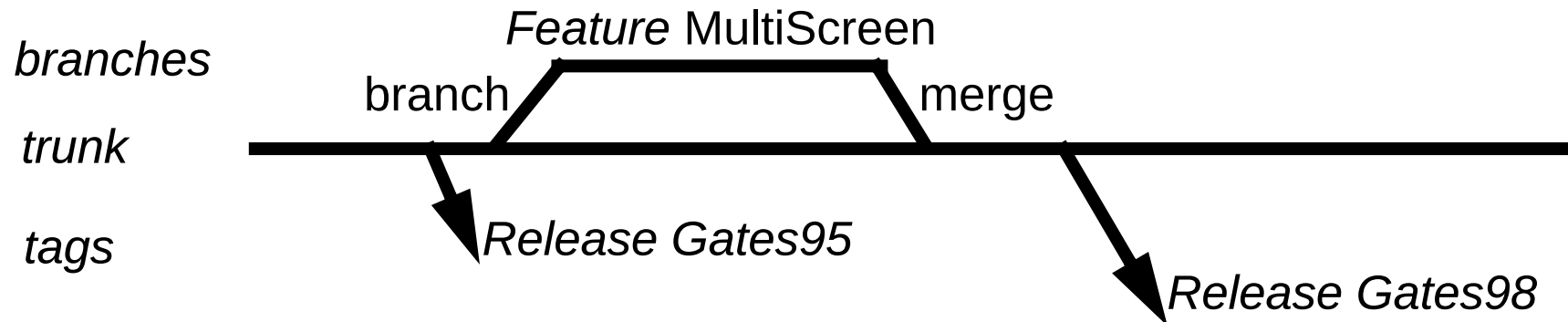
- move/mv: Datei umbenennen oder verschieben

- copy/cp: Datei/Teilbaum kopieren

```
alice@fau01[trunk]$ svn cp aufgabe2 contest
# aufgabe2 wurde in contest kopiert
```

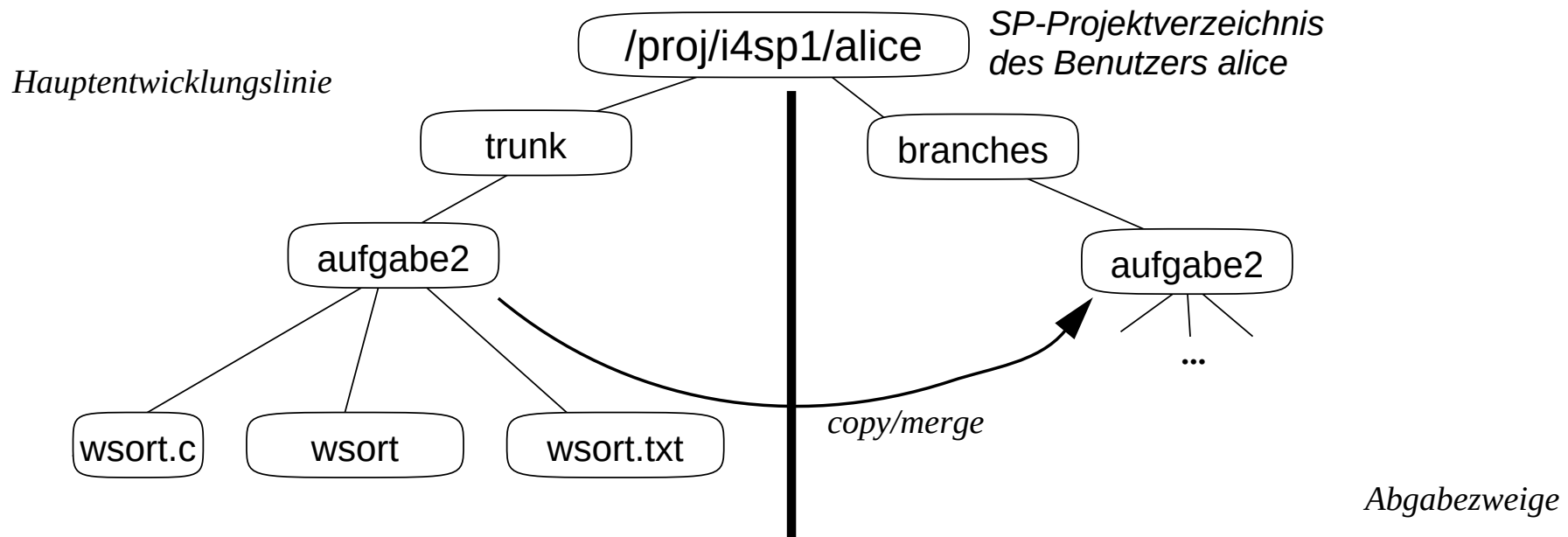
4 Konventionelles Repository-Layout

SVN-Repository des Projekts *Gates*



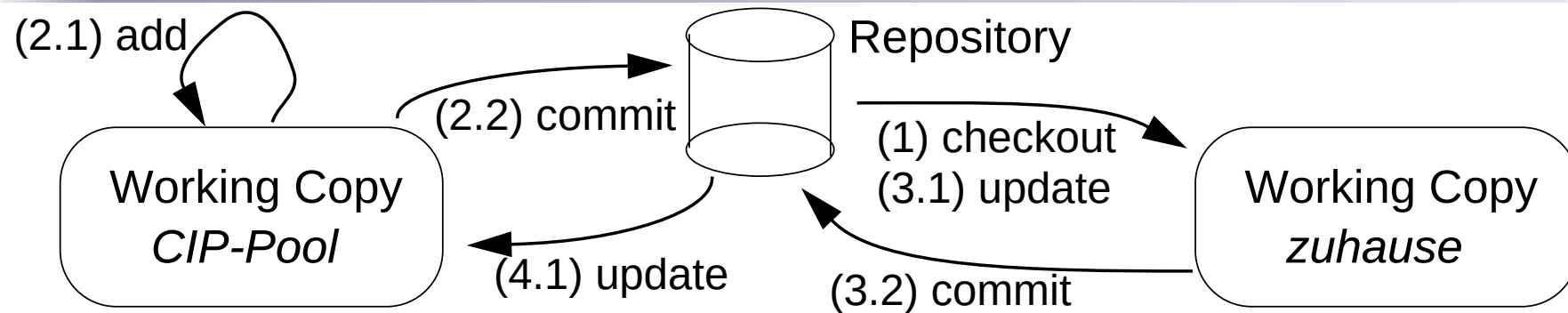
- Unterteilung des Wurzelverzeichnisses
 - ◆ Hauptentwicklungslinie: *trunk*
 - ◆ Verzeichnis mit Entwicklungszweigen: *branches*
 - ◆ Eingefrorene Versionen: *tags*
- Größere Features können von der Hauptlinie entkoppelt in einem eigenen Zweig (*branch*) entwickelt werden und nach Fertigstellung wieder in die Hauptlinie eingebracht (*merge*) werden
- Besondere Versionen können benannt (*getaggt*) werden (z.B. Release)

5 Funktionsweise der SP-Abgabe



- Zur Abgabe wird ein Abgabezweig für jede Aufgabe in *branches* erzeugt
- ◆ Eine erfolgreiche Abgabe ist durch Aufruf der folgenden URL erkennbar:
 - `https://www4.informatik.uni-erlangen.de/i4sp/ss11/sp1/<login>/branches/<aufgabe>/`
 - Dort werden die Dateien angezeigt, die zur Bewertung herangezogen werden
- ◆ unterhalb von *branches* sollte nicht von Hand editiert/eingecheckt werden

6 Arbeiten zuhause



- (1) Zusätzliche Arbeitskopie(n) erstellen (*checkout*, einmalig)
- (2) Start der Arbeit an einer Aufgabe im CIP-Pool
 - ◆ angelegte Dateien und Verzeichnisse unter Versionskontrolle stellen (*add*)
 - ◆ Zwischenstand ins Repository einchecken (*commit*)
- (3) Arbeit zuhause fortsetzen
 - ◆ Arbeitskopie zunächst auf den aktuellen Stand bringen (*update*)
 - ◆ Zwischenstand ins Repository einchecken (*commit*)
- (4) Arbeit im CIP-Pool fortsetzen
 - ◆ Arbeitskopie zunächst auf den aktuellen Stand bringen (*update*)
 - ◆ ...

U2-3 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

U2-4 valgrind

- Nicht immer ist der Debugger das richtige Werkzeug zum Finden von Fehlern (Quellcode ab Mittwoch unter: `/proj/i4sp1/pub/gdb/lilo.c`)
- Baukasten von Debugging- und Profiling-Werkzeugen (ausführbarer Code wird durch synthetische CPU auf Softwareebene interpretiert → Ausführung erheblich langsamer!)
 - ◆ Memcheck: erkennt Speicherzugriff-Probleme
 - Nutzung von nicht-initialisiertem Speicher
 - Zugriff auf freigegebenen Speicher
 - Zugriff über das Ende von allokierten Speicherbereichen
 - Zugriff auf ungültige Stack-Bereiche
 - ...
 - ◆ Helgrind: erkennt Koordinierungsprobleme zwischen mehreren Threads
- Aufrufbeispiel: **valgrind ./wsort**

1 Fehler finden mit valgrind

■ Zugriffe auf nicht allokierten Speicher finden

```
==11711== Invalid read of size 4
==11711==      at 0x804841B: main (gdb.c:19)
==11711==   Address 0x0 is not stack'd, malloc'd or (recently) free'd
==11711==
==11711==
==11711== Process terminating with default action of signal 11 (SIGSEGV)
==11711==   Access not within mapped region at address 0x0
==11711==      at 0x804841B: main (gdb.c:19)
```

- ◆ In Zeile 19 wurden 4 Byte von Adresse 0x0 gelesen
 - Auch schreibende Zugriffe können entdeckt werden
- ◆ Darauf hin wurde der Prozess beendet

1 Fehler finden mit valgrind

■ Auffinden von nicht freigegebenem Speicher

```
==5344== HEAP SUMMARY:  
==5344==      in use at exit: 16 bytes in 2 blocks  
==5344==    total heap usage: 4 allocs, 2 frees, 32 bytes allocated
```

- ◆ Bei Programmende sind noch 2 Speicherbereiche (Blöcke) belegt
- ◆ Während der Programmausführung wurde viermal **malloc()** und nur zweimal **free()** aufgerufen
- ◆ Mit Hilfe der Option **--leak-check=full --show-reachable=yes** wird angezeigt, wo der Speicher angelegt wurde, der nicht freigegeben wurde.

```
==5865== 8 bytes in 1 blocks are still reachable in loss record 1 of 2  
==5865==    at 0x48DAF50: malloc (vg_replace_malloc.c:236)  
==5865==    by 0x80484B9: insertElement (lilo.c:28)  
==5865==    by 0x8048599: main (lilo.c:61)
```

- In der Zeile 28 wurde der Speicher angelegt
- Nun im Quellcode Stellen identifizieren, an denen **free()**-Aufrufe fehlen

1 Fehler finden mit valgrind

■ Auffinden uninitialisierten Speichers

```
==18108== Conditional jump or move depends on uninitialised value(s)
==18108==      at 0x80484AC: insertElement (lilo.c:19)
==18108==      by 0x8048566: main (lilo.c:59)
```

- ◆ In der Funktion `insertElement()` in Zeile 19 wird auf uninitialisierten Speicher zugegriffen.
- ◆ Mit Hilfe der Option `--track-origins=yes` wird angezeigt, wo der uninitialisierte Speicher angelegt wurde.

```
==18741== Conditional jump or move depends on uninitialised value(s)
==18741==      at 0x80484AC: insertElement (lilo.c:19)
==18741==      by 0x8048566: main (lilo.c:59)
==18741== Uninitialised value was created by a heap allocation
==18741==      at 0x48DAF50: malloc (vg_replace_malloc.c:236)
==18741==      by 0x80484B9: insertElement (lilo.c:28)
==18741==      by 0x8048549: main (lilo.c:58)
```

U2-5 Dynamische Speicherverwaltung

- Mit `malloc()` angeforderter Speicher ist nicht initialisiert (U2.19)
- explizite Initialisierung mit `void *memset(void *s, int c, size_t n)`

```
memset(numbers, 0, sizeof(int)*n);
```

- Vorinitialisierter Speicher kann mit `calloc()` angefordert werden:
`void *calloc(size_t nelem, size_t elsize)`

```
int *numbers;  
numbers = (int *)calloc(n, sizeof(int));  
if(numbers == NULL) ...
```

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

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

U2-6 Generisches Sortieren mit qsort

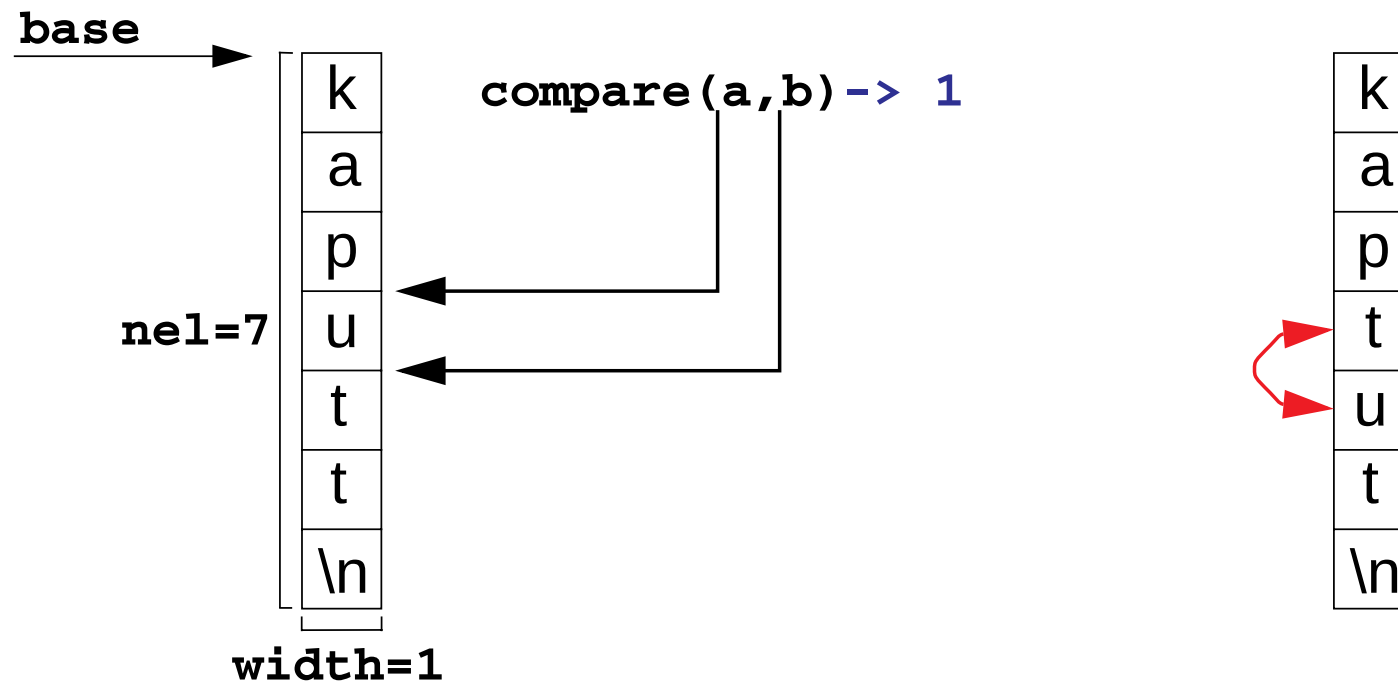
- Vergleich nahezu beliebiger Daten
 - alle Daten müssen die gleiche Größe haben
- qsort weiß nicht, was es sortiert (wie der Vergleich zu bewerkstelligen ist)
 - Aufrufer stellt Routine zum Vergleich zweier Elemente zur Verfügung
- Prototyp aus `stdlib.h`:

```
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
- Beispiel:



2 Vergleichsfunktion

■ Die Vergleichsfunktion erhält Zeiger auf Feldelemente

- da qsort den tatsächlichen Datentyp nicht kennt, ist der generische Prototyp mit void-Zeigern parametrisiert

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

- die übergebenen Zeiger haben tatsächlich denselben Typ wie das Feld

```
int charcompare(const char *, const char *);
```

- Cast erforderlich

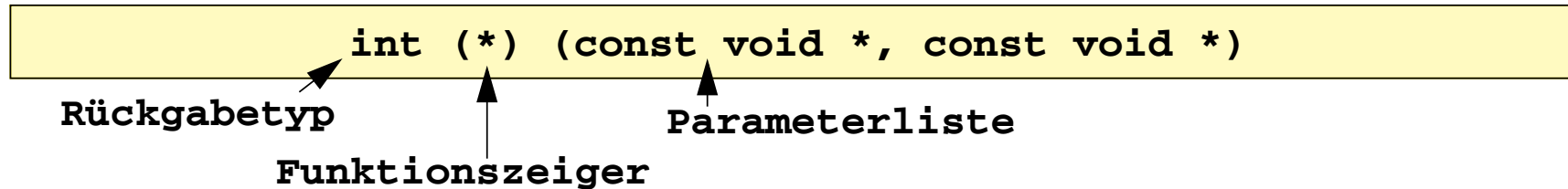
■ const-Zusicherung: Funktion ändert die verglichenen Werte nicht

■ 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 ('z', 'a' : 25)

3 Vergleichsfunktion - Cast

■ Notation von Funktionstypen



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

■ Cast dann wie bei allen anderen Datentypen

```
int charcompare(const char *, const char *); // Prototyp

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

◆ Funktionszeiger-Variable mit Namen `compare`

◆ wird die Adresse der typ-inkompatiblen Funktion `charcompare` zugewiesen