

# Übungen zu Systemprogrammierung 1 (SP1)

## Ü3 – Sortieren und Tooling

**Jens Schedel, Christoph Erhardt, Jürgen Kleinöder**

Lehrstuhl für Informatik 4  
Verteilte Systeme und Betriebssysteme

Friedrich-Alexander-Universität  
Erlangen-Nürnberg

WS 2014 – 20. bis 24. Oktober 2014

[https://www4.cs.fau.de/Lehre/WS14/V\\_SP1](https://www4.cs.fau.de/Lehre/WS14/V_SP1)



# Agenda

---

- 3.1 Subversion – Teil 2
- 3.2 Abgabesystem: Team-Arbeit
- 3.3 Fehlerbehandlung
- 3.4 Dynamische Speicherverwaltung
- 3.5 Generisches Sortieren
- 3.6 Haupteingang nach Walhall
- 3.7 Aufgabe 1: wsort
- 3.8 Gelerntes anwenden



# Versionierungsschema

---

- Subversion nummeriert fortlaufend ab Revision 0 (1, 2, 3, ...)
- spezielle Revisionschlüsselwörter
  - **HEAD**: aktuelle Version des Repositories (neueste Version)
  - **BASE**: Revision eines Eintrags (Datei oder Verzeichnis) der Arbeitskopie
  - **COMMITTED**: Letzte Änderungsrevision eines Eintrags – meist älter als BASE
  - **PREV**: **COMMITTED** - 1
- Revision zu einem bestimmten Zeitpunkt
  - “{2014-11-03 17:30}“



# Basisoperationen II

- **diff:** Änderungen der Arbeitskopie anzeigen

```
> svn status
M hallo
> svn diff
Index: hallo
=====
--- hallo (revision 23)
+++ hallo (working copy)
@@ -0,0 +1 @@
+welt
```

- **revert:** Noch nicht committete Änderungen an der Arbeitskopie zurücksetzen

```
> svn revert hallo
Reverted 'hallo'
> svn status
>
```



# Basisoperationen II

---

- `list/ls`: Dateien/Verzeichnisse im Repository anzeigen

```
> svn ls  
branches/  
trunk/
```

- `log`: Historie anzeigen

```
> svn log  
-----  
r1 | www-data | 2013-04-19 15:03:14 +0200 (Fri, 19 Apr 2013) | 1 line  
init repository  
-----
```

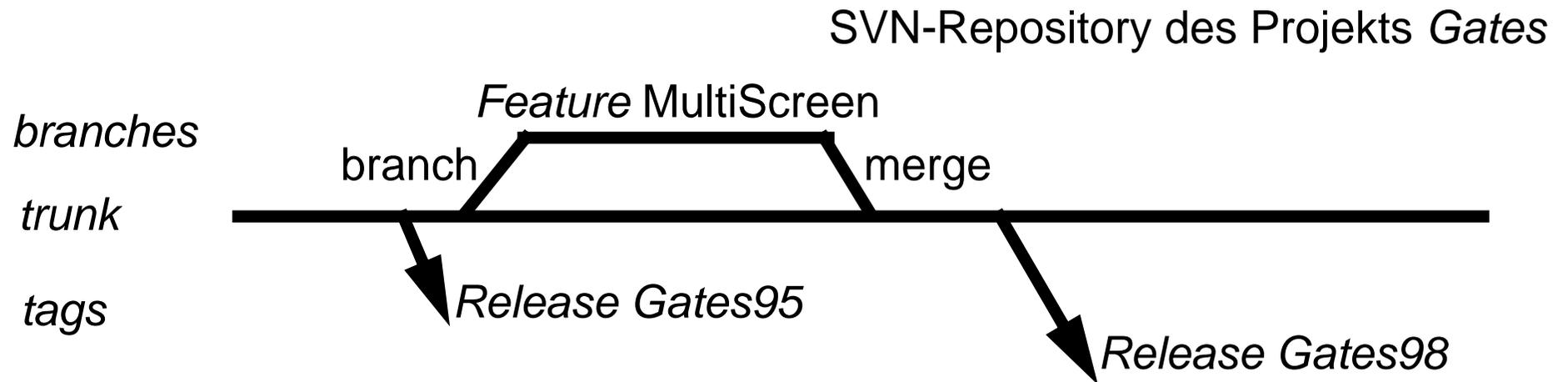
- `move/mv`: Datei umbenennen oder verschieben

- `copy/cp`: Datei/Teilbaum kopieren

```
> svn cp aufgabe2 contest  
> # aufgabe2 wurde in contest kopiert
```



# Konventionelles Repository-Layout



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



# Agenda

---

- 3.1 Subversion – Teil 2
- 3.2 Abgabesystem: Team-Arbeit
- 3.3 Fehlerbehandlung
- 3.4 Dynamische Speicherverwaltung
- 3.5 Generisches Sortieren
- 3.6 Haupteingang nach Walhall
- 3.7 Aufgabe 1: wsort
- 3.8 Gelerntes anwenden



# Abgabesystem: Team-Arbeit

---

- Gemeinsame Bearbeitung im Repository eines Teammitglieds
  - Repository-Eigentümer: *alice*
  - Partner (nutzt Repository von *alice*): *bob*
- Abgabe erfolgt ebenfalls im Repository des Eigentümers
  - es ist nur eine Abgabe erforderlich



# Ablauf für den Repository-Eigentümer

- Partner wird für jede Team-Aufgabe separat festgelegt

```
> /proj/i4sp1/bin/set-partner aufgabe2 bob
```

- Hintergrund

- Erzeugung und Commit einer Textdatei `partner` in `trunk/aufgabe2`
- diese Datei enthält den Login-Namen (*bob*) des Partners für diese Aufgabe
- Partner erhält Zugriff auf die relevanten Teile des Repositorys
  - `trunk/aufgabe2`
  - `branches/aufgabe2`

- Abgabe funktioniert wie gewohnt

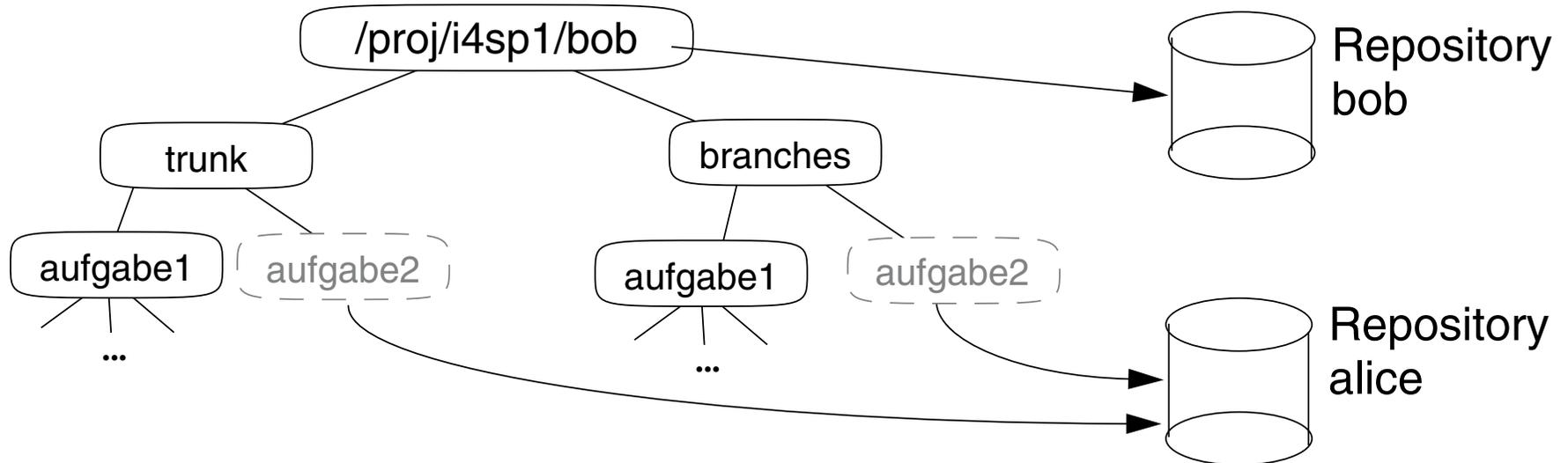
```
> /proj/i4sp1/bin/submit aufgabe2
```



# Ablauf für den Partner

- Partner setzt in seinem Repository einen Verweis auf Hauptrepository

```
> /proj/i4sp1/bin/import-from-partner aufgabe2 alice
```



- Nach Ausführung des Skriptes `svn update` ausführen
- Achtung: Abgabe im eigenen Repository überlagert Partnerabgabe
  - zum Umstieg auf Teamarbeit eigene Abgabe löschen (Übungsleiter hilft)



# Ablauf für den Partner

- Arbeit in der eigenen Arbeitskopie fast normal möglich
  - Der Befehl `svn commit` übermittelt nur Änderungen an das Repository das für den aktuellen Pfad zuständig ist

```
> svn status
X      aufgabe1
M      aufgabe1/wsort.c
> svn commit
# Es erfolgt kein Commit
> svn status
X      aufgabe1
M      aufgabe1/wsort.c
> cd aufgabe1
> svn commit -m 'Erste Arbeiten an Aufgabe 1 abgeschlossen'
Committed revision 5.
```

- Abgabe funktioniert wie gewohnt

```
> /proj/i4sp1/bin/submit aufgabe1
```



# Agenda

---

- 3.1 Subversion – Teil 2
- 3.2 Abgabesystem: Team-Arbeit
- 3.3 Fehlerbehandlung**
- 3.4 Dynamische Speicherverwaltung
- 3.5 Generisches Sortieren
- 3.6 Haupteingang nach Walhall
- 3.7 Aufgabe 1: wsort
- 3.8 Gelerntes anwenden



- Fehler können aus unterschiedlichsten Gründen im Programm auftreten
  - Systemressourcen erschöpft: `malloc(3)` schlägt fehl
  - Fehlerhafte Benutzereingaben: `fopen(3)` 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



# Fehlerbehandlung: Beendigung des Programms

- Tritt ein Fehler auf, der ein sinnvolles Weiterarbeiten verhindert, muss das Programm beendet werden (`exit(2)`) 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 Man-Page 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



# Erkennung und Ausgabe von Fehlern

- Fehler treten häufig in Funktionen der C-Bibliothek auf
  - erkennbar i. d. R. am Rückgabewert (Man-Page, **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 als lesbare Strings ausgegeben mit `perror(3)`

```
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)` nur verwenden, wenn die `errno` gesetzt wurde
- sonst mit Hilfe von `fprintf(3)` eigene Fehlermeldung ausgeben



# Agenda

---

- 3.1 Subversion – Teil 2
- 3.2 Abgabesystem: Team-Arbeit
- 3.3 Fehlerbehandlung
- 3.4 Dynamische Speicherverwaltung**
- 3.5 Generisches Sortieren
- 3.6 Haupteingang nach Walhall
- 3.7 Aufgabe 1: wsort
- 3.8 Gelerntes anwenden



# Dynamische Speicherverwaltung – Teil 2

- Verändern der Größe von Feldern, die durch `malloc(3)` bzw. `calloc(3)` erzeugt wurden:

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

- Neuer Speicherbereich enthält die Daten des ursprünglichen Speicherbereichs (wird automatisch kopiert; aufwändig)
- Sollte `realloc(3)` fehlschlagen, wird der ursprüngliche Speicherbereich nicht freigegeben
  - Explizite Freigabe mit `free(3)` notwendig.



# Agenda

---

- 3.1 Subversion – Teil 2
- 3.2 Abgabesystem: Team-Arbeit
- 3.3 Fehlerbehandlung
- 3.4 Dynamische Speicherverwaltung
- 3.5 Generisches Sortieren**
- 3.6 Haupteingang nach Walhall
- 3.7 Aufgabe 1: wsort
- 3.8 Gelerntes anwenden



# 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
  - Fachbegriff für dieses Programmierschema: *Rückruf (Callback)*
- Prototyp aus `stdlib.h`:

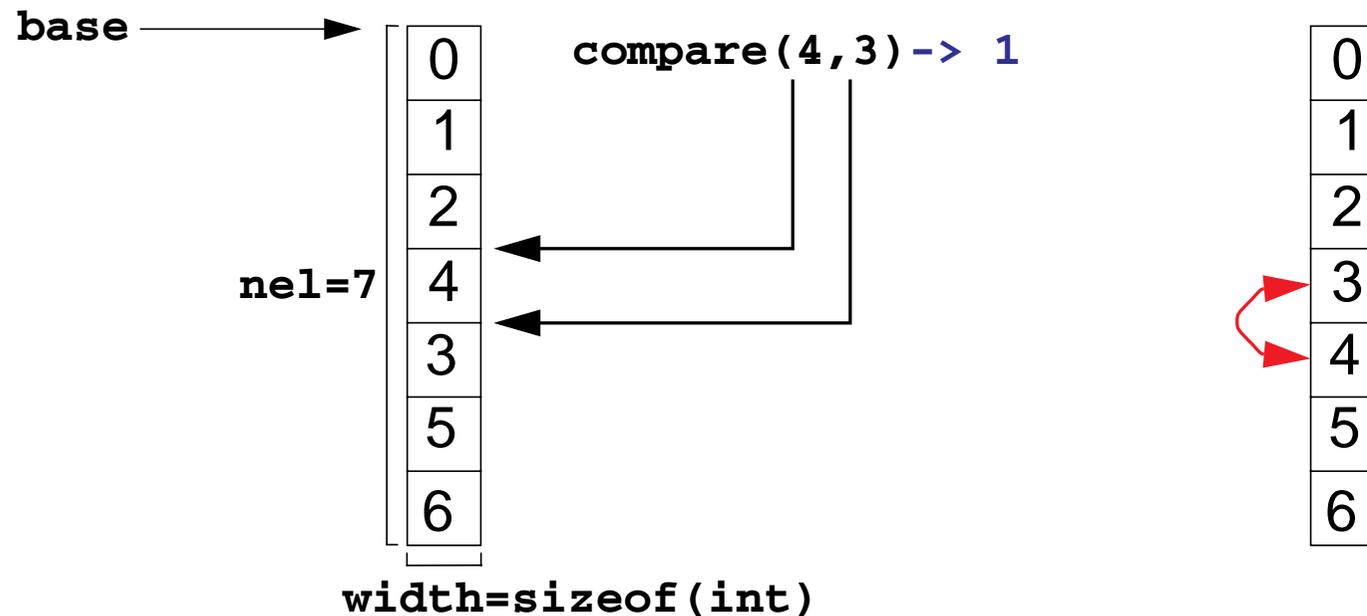
```
void qsort(void *base,  
           size_t nel,  
           size_t width,  
           int (*compare) (const void *, const void *));
```

- `base`: Zeiger auf das erste Element des zu sortierenden Feldes
- `nel`: Anzahl der Elemente im zu sortierenden Feld
- `width`: Größe eines Elements
- `compare`: Vergleichsfunktion



# Arbeitsweise von qsort

- qsort vergleicht je zwei Elemente mit Hilfe der Vergleichsfunktion



- Die Funktion vergleicht die beiden Elemente und liefert:
  - < 0 falls Element 1 kleiner gewertet wird als Element 2
  - 0 falls Element 1 und Element 2 gleich gewertet werden
  - > 0 falls Element 1 größer gewertet wird als Element 2



# Vergleichsfunktion

- Die Vergleichsfunktion erhält Zeiger auf Feldelemente

- Beispiel: Vergleichsfunktion für int

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

- const-Zusicherung: Funktion ändert die verglichenen Werte nicht

- `qsort(3)` kennt den tatsächlichen Datentyp nicht

- Prototyp ist generisch mit void-Zeigern parametrisiert

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

- Cast erforderlich

- entweder innerhalb einer *Wrapper*-Funktion
- oder bei der Übergabe des Funktionszeigers an `qsort(3)` (schöner!)



# Agenda

---

- 3.1 Subversion – Teil 2
- 3.2 Abgabesystem: Team-Arbeit
- 3.3 Fehlerbehandlung
- 3.4 Dynamische Speicherverwaltung
- 3.5 Generisches Sortieren
- 3.6 Haupteingang nach Walhall**
- 3.7 Aufgabe 1: wsort
- 3.8 Gelerntes anwenden



- Baukasten von Debugging- und Profiling-Werkzeugen
- Für uns relevant: *memcheck*
  - Erkennt Speicherzugriff-Probleme:
    - Nutzung von nicht-initialisiertem Speicher
    - Zugriff auf freigegeben Speicher
    - Zugriff über das Ende von allozierten Speicherbereichen



- Zugriffe auf nicht allozierten Speicher finden

```
=711= Invalid read of size 4
=711=    at 0x804841B: main (test.c:19)
=711= Address 0x0 is not stack'd, malloc'd or (recently) free'd
=711=
=711= Process terminating with default action of signal 11 (SIGSEGV)
=711= Access not within mapped region at address 0x0
```

- In Zeile 19 wird lesend auf die Adresse 0x0 zugegriffen
- Der Prozess wird auf Grund einer Speicherzugriffsverletzung (SIGSEGV) beendet

```
=787= Invalid write of size 1
=787=    at 0x48DC9EC: memcpy (mc_replace_strmem.c:497)
=787=    by 0x80485A2: test_malloc (test.c:57)
=787=    by 0x80484A8: main (test.c:22)
=787= Address 0x6d1f02d is 0 bytes after a block of size 5 alloc'd
```

- In Zeile 57 wird memcpy aufgerufen, welches ein Byte an eine *ungültige* Adresse schreibt



## ■ Auffinden von nicht freigegebenem Speicher

```
=787= HEAP SUMMARY:  
=787=      in use at exit: 5 bytes in 1 blocks  
=787=    total heap usage: 1 allocs, 0 frees, 5 bytes allocated
```

- Bei Programmende ist noch ein Speicherbereich (Block) belegt
- Während der Programmausführung wurde einmal `malloc()` und keinmal `free()` aufgerufen
- Mit Hilfe der Option `--leak-check=full --show-reachable=yes` wird angezeigt, wo der Speicher angelegt wurde, der nicht freigegeben wurde.

```
=799= 5 bytes in 1 blocks are definitely lost in loss record 1  
=799=   at 0x48DAF50: malloc (vg_replace_malloc.c:236)  
=799=   by 0x8048576: test_malloc (test.c:52)  
=799=   by 0x80484A8: main (test.c:22)
```

- In Zeile 52 wurde der Speicher angefordert
- Im Quellcode Stellen identifizieren, an denen `free()`-Aufrufe fehlen



## ■ Auffinden uninitialisierten Speichers

```
=799= Use of uninitialised value of size 4
=799=   at 0x4964316: _itoa_word (_itoa.c:195)
=799=   by 0x4967C59: vfprintf (vfprintf.c:1616)
=799=   by 0x496F3DF: printf (printf.c:35)
=799=   by 0x8048562: test_int (test.c:48)
=799=   by 0x8048484: main (test.c:15)
```

- In Zeile 48 wird auf uninitialisierten Speicher zugegriffen
- Mit Hilfe der Option `--track-origins=yes` wird angezeigt, wo der uninitialisierte Speicher angelegt wurde

```
=683= Use of uninitialised value of size 4
=683=   at 0x4964316: _itoa_word (_itoa.c:195)
=683=   by 0x4967C59: vfprintf (vfprintf.c:1616)
=683=   by 0x496F3DF: printf (printf.c:35)
=683=   by 0x8048562: test_int (test.c:48)
=683=   by 0x8048484: main (test.c:15)
=683= Uninitialised value was created by a stack allocation
=683=   at 0x804846A: main (test.c:10)
```



- Spezialfall: Zugriff auf uninitialisierten Speicher bei Bedingungsprüfungen

```
=683= Conditional jump or move depends on uninitialised value(s)
=683=   at 0x48DC0E7: __GI_strlen (mc_replace_strmem.c:284)
=683=   by 0x496886E: vfprintf (vfprintf.c:1617)
=683=   by 0x496F3DF: printf (printf.c:35)
=683=   by 0x8048562: test_int (test.c:48)
=683=   by 0x8048484: main (test.c:15)
```



# Agenda

---

- 3.1 Subversion – Teil 2
- 3.2 Abgabesystem: Team-Arbeit
- 3.3 Fehlerbehandlung
- 3.4 Dynamische Speicherverwaltung
- 3.5 Generisches Sortieren
- 3.6 Haupteingang nach Walhall
- 3.7 Aufgabe 1: wsort**
- 3.8 Gelerntes anwenden



# Aufgabe 1: wsort

- Lernziele
  - Einlesen von der Standardeingabe (`stdin`)
  - Umgang mit dynamischer Speicherverwaltung (`realloc(3)`)
  - Verwendung von Debug-Werkzeugen
- Ausprobieren eures Programmes
  - Beispiel-Eingabedateien in `/proj/i4sp1/pub/aufgabe1`
  - Vergleichen der Ausgabe mit vorgegebenem Binary
    - Hier am Beispiel der `wlist0`

```
> ./wsort < /proj/i4sp1/pub/aufgabe1/wlist0 > wlist0.mine
> /proj/i4sp1/pub/aufgabe1/wsort < \
    /proj/i4sp1/pub/aufgabe1/wlist0 > wlist0.spteam
> diff wlist0.mine wlist0.spteam
```

- Auch graphische Diff-Werkzeuge möglich: `kompare`, `meld`, etc.



# Agenda

---

- 3.1 Subversion – Teil 2
- 3.2 Abgabesystem: Team-Arbeit
- 3.3 Fehlerbehandlung
- 3.4 Dynamische Speicherverwaltung
- 3.5 Generisches Sortieren
- 3.6 Haupteingang nach Walhall
- 3.7 Aufgabe 1: wsort
- 3.8 Gelerntes anwenden



## „Aufgabenstellung“

- `isort` Programm, welches als Argumente übergebene Zahlen sortiert

