Programmierkurs: Strings, Kanäle, Git

Manfred Hauswirth | Open Distributed Systems | Einführung in die Programmierung, WS 25/26

Rückblick

VL 0 „Organisation und Inhalt“: Ablauf der Vorlesung, Termine
VL 1 „Hello World“: „Lebenswichtiges“, Programablauf, Programmierablauf, Kompilierung und Ausführung von Programmen
VL 2 „Die ersten Schritte“: Erstes C-Programm, Elementare C-Strukturen, Datentypen, Operatoren, Schleifen
VL 3 „Kontrollstrukturen & Funktionen“: Syntax, Semantik, bedingte Anweisungen, Blöcke, Sichtbarkeit
VL 4 „Rekursive Funktionen & Bibliotheken“: rekursive Funktionsaufrufe, Modularisierung
VL 5 „Typen“: Einfache und strukturierte Datentypen, Wertebereiche, Typendefinition
VL 6 „Speicher und Adressen“: Speicher, Pointer, Funktionsaufrufe „call by value“ vs. „call by reference“
VL 7 „Speicher und Arrays“: Speicher, Arrays, mehrdimensionale Arrays, Arrays und Pointer
VL 8 „Dynamische Speicherverwaltung“: Speicherallokation, Fehlerbehandlung, Rückgabewerte, Arrays/Pointer/Adressen
VL 9 „Strings, Kanäle, Git“: Strings und Arrays, Zeichensätze, Stringlänge, Ein- und Ausgabe, Arbeiten mit git
VL 10 „Debugging und Stack“: Fehlverhalten/Bugs, Fehlersuche Strategien und Werkzeuge

Technische Universität Berlin

Unsere nächste „Datenstruktur“:

Strings / Zeichenketten

Open Distributed Sytems
ODS

Strings in C

- Strings sind Arrays bestehend aus Zeichen
- Der Datentyp für Zeichen heißt `char` (character)
- Strings haben keine vordefinierte Länge ⇒ anderer Mechanismus notwendig, um die Länge dynamisch zu bestimmen
- Strings werden immer mit ‘`\0`‘ als letztem Zeichen beendet
- Beispiele für Strings:
  - Jeder Text
  - Formatstrings für `printf()`
  - Dateinamen
  - Textdateien
  - …

Strings in C

- Strings sind `char` Arrays, die mit ‘`\0`‘ beendet werden
- In C ist ein String eine Folge von Zeichen, d.h. eine Folge von `char`, die hintereinander im Speicher stehen.
- Jedes Zeichen belegt genau ein Byte.

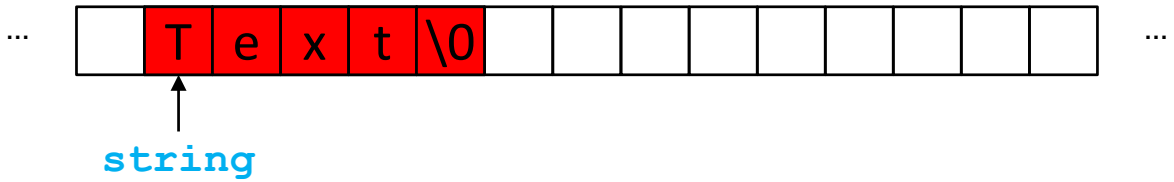
Beispiel:


```
char string[] = "Text";
printf("Der Inhalt von string ist: %s\n", string);
```

Strings in C

- Strings sind `char` Arrays, die mit ‘`\0`‘ beendet werden
- In C sind Strings eine Liste von Zeichen, d.h. eine Liste von char‘s, die hintereinander im Speicher stehen.
- „Stringvariablen“ sind Pointer, die auf den Start dieses Arrays zeigen:

```
char string[] = "Text";
```



Strings in C

Hinweis: Die Kodierung des Zeichensatzes kann unterschiedlich sein

- ASCII-Zeichensatz 7 Bit
- ASCII-Zeichensatz 8 Bit — PC/MS-DOS
- ANSI-Zeichensatz — Windows vor NT / 2000
- UTF-8 (1-4 chars pro Zeichen) — MacOS X, modernes Unix
- UTF-16 (2 chars pro Zeichen) — modernes Windows

Strings: Zusammenfassung

- Strings sind eine Folge von Einzelzeichen `char`
- Pointer auf Array von Elementen vom Type char
- String ist terminiert mit ‘`\0`‘
- Speicherbedarf: **Länge + 1 Byte**

Strings und Arrays

- Ein Array (Feld):
  - Ist eine Liste von Daten gleichen Typs
  - Hat eine feste Länge
  - Zugriff auf Arrayelemente mit Index in `[]`

Beispiel:

```
char a[128];
a[0] = 'H';    // Arrayindexanfang ist bei Index 0!
a[1] = 'E';
a[2] = 'L';
a[3] = 'L';
a[4] = 'O';
a[5] = '\0';
```

Strings vs. Arrays

- Strings sind char Arrays

```
char *s = "test";
char c = s[1];  // c = 'e';
```

- Aber es gibt wesentliche Unterschiede
  - Strings müssen mit '`\0`' terminiert werden (d.h. 1 Zeichen länger)
  - Arrays haben feste Länge im Gegensatz zu Strings

- Man speichert Strings in Arrays - mit der Länge vorsichtig sein.
  - Gefahr: Buffer Overflows (überschreiben von anderem Speicher)

Stringlänge berechnen

```
#include <stddef.h>
#include <stdio.h>

int strlen (char *s) {
    int l = 0;                  // initialize length of string

    if (s == NULL) return -1; // no string => return impossible length
    while (*s != '\0') {      // count characters
       l++;                   // increase char count
       s++;                   // go to next character
    }
    return l;
}
```

Stringlänge berechnen

```c
#include <stddef.h>
#include <stdio.h>

int strlen (char *s) {
    int l = 0;

    if (s == NULL) return -1;
    while (*s != '\0') {
       l++;
       s++;
    }
    return l;
}
```

```c
int main() {
        char test[] = "hello world";
        printf ("Länge von '%s' ist: %d\n",
                "", strlen(""));
        printf ("Länge von NULL ist: %d\n",
                strlen(NULL));
        printf ("Länge von '%s' ist: %d\n",
                test, strlen(test));
}
```

Ausgabe:

```
Länge von '' ist: 0
Länge von NULL ist: -1
Länge von 'hello world' ist: 11
```

Stringlänge – viel einfacher

Stringfunktionen aus der Standardbibliothek werden eingebunden mit

```
#include <string.h>

int strlen(char *s)
```

Liefert Länge des String s ohne das ‘`\0`‘

… und es gibt viele weiter …

Ausgabe mittels `printf`

Formatierte Ausgabe: `printf()`

`printf()` gibt die Parameter unter „Steuerung“ des Formatstrings `fmt` auf `stdout` aus

- Aufruf: `int printf(char* fmt, ...)`
- Der Formatstring `fmt` ist eine Zeichenkette
- Die weiteren Parameter müssen den Typ haben, wie er im Formatstring `fmt` angegeben ist

Beispiele

```
printf("Hello world\n");
printf("Wert der Variablen i: %d\n", i);
```

Formatierte Ausgabe: `printf()`

Technische Universität Berlin

Weitere Beispiele

```
printf("a(%d)+b(%d) ist: %d\n", a, b, a+b);

printf("a(%d)/b(%d) ist: %d\n", a, b, a/b);

printf("a(%f)/b(%f) ist: %f\n", a, b, a / (float) b);

printf("Die Kodierung von %c ist %d\n",'a', 'a');
```

`printf()`: Formatzeichen

`%c`	**Einzelzeichen**	`char`
`%d`	**Integer**	`int`
`%f`	**Gleitkommazahl**	`float`
`%p`	**Pointer**	`void *`
`%s`	**Zeichenkette/String**	`char *`

… und viele mehr …

Ein-/Ausgabekanäle

Ein-/Ausgabe

Jeder Unix-Prozess hat voreingestellt drei Kanäle für Ein-/Ausgabe:

- `stdin` Standardeingabe, meist Tastatur
- `stdout` Standardausgabe, meist Bildschirm
- `stderr` Standardfehlerausgabe, meist Bildschirm

- Die Standardkanäle sind umlenkbar:

```
$ ./meinprog < InFile
$ ./meinprog > OutFile
```

- Die Standardkanäle sind kombinierbar:

```
$ ./meinprog1 | sort > OutFile
```

  - ➢ Ausgabe von `./meinprog1` als Eingabe für `sort` verwenden

Technische Universität Berlin

Versionsmanagement mit

Open Distributed Sytems
ODS

Was ist git?

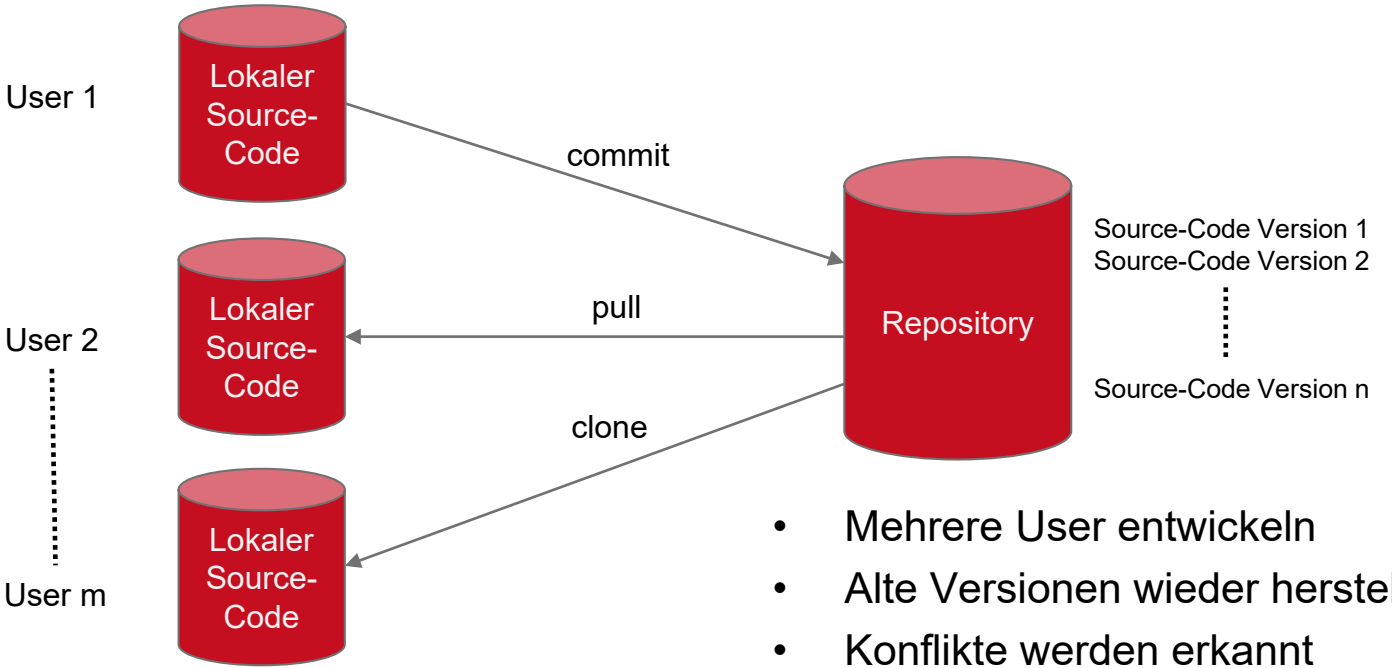
- Freie Software zur verteilten Versionsverwaltung von Dateien (Sourcecode)
- Durch Linus Torvalds (⇒ Linux) initiiert
- Name: ein Witz von Linus Torvalds
  - *“I’m an egotistical bastard, and I name all my projects after myself. First ‘Linux’, now ‘Git’.”*
  - Git (Brit. ugs): Blödmann, Mistkerl

Warum Versionsverwaltung?

- Verteilter Zugriff
- Sicherung
- Zurückgehen auf alte Versionen
- Unterschiede zwischen Versionen
- Ein Versionsverwaltungssystem „beschützt“ vor Fehlern

- **For the rest of your software development life:**
  Git oder eine andere Versionsverwaltung wird verwendet werden!!!

Versionsverwaltung: Überblick



- Mehrere User entwickeln
- Alte Versionen wieder herstellbar
- Konflikte werden erkannt

Git – step by step: Administration

- Wir verwenden nur einen Bruchteil der Funktionalität

- Git initialisieren (bei der ersten Verwendung, 1x)

```
git config --global user.name '<Ihr Name>'
git config --global user.email '<ihre@mail.adresse>'
```

- Damit „kennt“ Git Sie

Git – step by step: Neues Projekt

- Neues Verzeichnis, z.B. „Introprog-ProgKurs“

```
mkdir Introprog-ProgKurs
cd Introprog-ProgKurs
git init
```

  Damit haben Sie ein neues Projekt „Introprog-ProgKurs“ angelegt

- Eine neue Datei zum Projekt hinzufügen
  - Sie schreiben zuerst ihren Sourcecode, z.B. „myprog.c“
  - Danach müssen Sie Git sagen, dass es diese Datei verwalten soll (1x)

```
git add myprog.c
```

- Nun soll eine Version (bestimmen Sie selbst) im Repository gespeichert werden (damit ist diese Version „gesichert“)

```
git commit -m "<Nachricht>"
```

  Die Nachricht bestimmen Sie selbst, z.B. „Erste Version von myprog.c“

Git – step by step: Status

- Ein Projekt hat einen Status

```
git status
On branch main
nothing to commit, working tree clean
```

  Damit sagt Git Ihnen, Sie haben die aktuellen Versionen aller Dateien

- Eine Datei wird lokal geändert

```
git status
On branch main
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        modified:   myprog.c
no changes added to commit (use "git add" and/or "git commit -a")
```

- Mit `git restore myprog.c` könnte die alte Version wieder hergestellt werden oder Sie entscheiden, dass die Änderung gesichert werden soll, dann `git commit -m "<Nachricht>"` verwenden

Git – step by step: Neue Abgabe

- In Introprog ist jede Abgabe ein „Branch“ (Verzweigung bzw. Variante)
- Neuen Branch erstellen und in den Branch wechseln

```
git branch Abgabe1
git switch -c Abgabe1
```

-c: „create“

Damit haben Sie einen neuen Branch „Abgabe1“ angelegt und haben Git mitgeteilt, dass Sie ab jetzt an dieser Variante weiterarbeiten

- Nun können Sie neue Dateien zu diesem Branch hinzufügen oder existierende verändern
  - Sie schreiben neuen Sourcecode, z.B. „neuesprog.c“ und verändern „myprog.c“
  - Danach müssen Sie Git sagen, dass es nun auch „neuesprog.c“ verwalten soll (1x)

```
git add neuesprog.c
```

- Nun soll eine Version dieser Verzweigung im Repository gesichert werden

```
git commit –m "<Nachricht>"
```

Die Nachricht bestimmen Sie selbst, z.B. „Neue Version von myprog.c und neuer Code in neuesprog.c“

Git – step by step: Nächste Abgabe

- Sie sind jetzt mit Abgabe1 fertig und möchten nun an Abgabe2 zu arbeiten
  - **ACHTUNG:** Jede Abgabe soll ein eigener Branch sein!

- In welchem Branch bin ich?

```
git status
On branch Abgabe1
nothing to commit, working tree clean
```

Sie sind noch in Branch „Abgabe1“!

Git Repositories, die vor dem 01.10.2020 erstellt wurden, haben häufig stattdessen einen master Branch.

- Zuerst zurück in den „main“-Branch, dann neuen Branch „Abgabe2“ erzeugen und in „Abgabe2“ wechseln, sonst würde „Abgabe1“ kopiert werden.

```
git switch main
git switch -c Abgabe2
git status
On branch Abgabe2
nothing to commit, working tree
```

Kein „-c“ !!!

- Nun kann es mit Abgabe2 weitergehen!
  - **ACHTUNG:** „neuesprog.c“ existiert nur in „Abgabe1“, aber nicht in „Abgabe2“, d.h. es ist „verschwunden“.
  - **Nicht verwirren lassen:** Sobald Sie in Branch „Abgabe1“ wechseln, ist „neuesprog.c“ wieder da ☺

Git – step by step: Verteiltes Repo

- Bis jetzt war alles lokal in Ihrem Verzeichnis
- Der nächste Schritt: Repository (Repo) von einem Server laden („clonen“)
  - `git clone https://git.tu-berlin.de/intoprog-ws25/<TUB-Account>`
  - Damit wird ein Verzeichnis angelegt und der aktuelle Zustand vom remote repository auf den Rechner übertragen.
- Der Rest ist gleich wie zuvor :
  - Ins angelegte Verzeichnis wechseln
  - Mit `git switch …` Branches anlegen und wechseln
  - Programmieren und alle neuen Dateien mit `git add …` hinzufügen
  - Mit `git commit …` sichern (wie zuvor)
- **Fast gleich** ☺ ⇒ **nächste Folie!**

Git – step by step: Verteiltes Repo

- **Fast gleich ☺ !!!**
  - `git commit …` **sichert nur lokal** auf Ihrem Rechner
  - `git push` sichert danach **in das Repository auf dem Server**
    - Beim **1. Mal** in einem **neuen Branch:**
      `git push --set-upstream origin <neuer_branch>`
    - **Jedes weitere Mal** für diesen Branch:
      `git push`
  - `git commit` immer vor `git push` machen

Git – step by step: Verteiltes Repo

Technische Universität Berlin

Zusammengefasst:

1. `git clone https://git.tu-berlin.de/introprog-ws24/<TUB-Account>`
2. `cd <TUB-Account>`
3. `git switch -c <abgabeX>`
4. `... <abgabeX.c>` editieren
5. `git add <abgabeX.c>`
6. `git commit –m "<Nachricht>"`
7. `git push --set-upstream origin <abgabeX>`
8. `... <abgabeX.c>` editieren
9. `git commit –m "<Nachricht>"`
10. `git push`
11. Ab Schritt 8 wiederholen, bis `<abgabeX.c>` fertig ist

Open Distributed Sytems
ODS

Ausblick

VL 0 „Organisation und Inhalt“: Ablauf der Vorlesung, Termine
VL 1 „Hello World“: „Lebenswichtiges“, Programablauf, Programmierablauf, Kompilierung und Ausführung von Programmen
VL 2 „Die ersten Schritte“: Erstes C-Programm, Elementare C-Strukturen, Datentypen, Operatoren, Schleifen
VL 3 „Kontrollstrukturen & Funktionen“: Syntax, Semantik, bedingte Anweisungen, Blöcke, Sichtbarkeit
VL 4 „Rekursive Funktionen & Bibliotheken“: rekursive Funktionsaufrufe, Modularisierung
VL 5 „Typen“: Einfache und strukturierte Datentypen, Wertebereiche, Typendefinition
VL 6 „Speicher und Adressen“: Speicher, Pointer, Funktionsaufrufe „call by value“ vs. „call by reference“
VL 7 „Speicher und Arrays“: Speicher, Arrays, mehrdimensionale Arrays, Arrays und Pointer
VL 8 „Dynamische Speicherverwaltung“: Speicherallokation, Fehlerbehandlung, Rückgabewerte, Arrays/Pointer/Adressen
VL 9 „Strings, Kanäle, Git“: Strings und Arrays, Zeichensätze, Stringlänge, Ein- und Ausgabe, Arbeiten mit git
VL 10 „Debugging und Stack“: Fehlverhalten/Bugs, Fehlersuche Strategien und Werkzeuge

Slides für Interessierte

Stringfunktionen aus `<string.h>`

Technische Universität Berlin

Stringfunktionen aus der Standardbibliothek werden eingebunden mit `#include <string.h>`

- `int strlen(char *s)`
  - Liefert Länge des String s ohne das `'\0'`
- `char *strncpy(char *dst, char *src, size_t n)`
  - Kopiert String `src` nach String `dst`
  - Alle Zeichen bis zum terminierenden `'\0'`, aber maximal `n` Chars
    Achtung: `dst` muss groß genug sein (sonst Buffer Overflow)
  - Best Practice: Länge von `dst` als Parameter `n` übergeben
  - Alte Version (unsicher): `char *strcpy(char *dst, char *src)`

Stringfunktionen aus `<string.h>`

- `int strncmp(char *s1, char *s2, size_t n)`
  - Vergleicht zeichenweise `s1` und `s2` bis zum ersten `'\0'`
    - liefert `0` bei Gleichheit
    - liefert ansonsten, ob `s1` „größer“ als `s2` ist (z.B. ‘a‘ > ‘2‘)
  - vergleicht maximal `n` Zeichen
  - Alte Version (unsicher): `int strcmp(char *s1, char *s2)`

- `int strncat(char *s1, char *s2, size_t n)`
  - Fügt `s2` an `s1` an (siehe `man strncat` für Details)

- `char* strtok_r(char *str, char *sep, char **lasts);`
  - Splittet `str` an Vorkommen von `sep` auf (Siehe `man strtok_r` für Details)

Stringfunktionen aus `<stdlib.h>`

C-Standardbibliothek bietet viele Stringfunktionen – Darunter Konvertierungsfunktionen

- `(int) strtol(char *s, char **next, int base)`
  `int atoi(char *s)`
  wandelt String `s` in `int` um
- `long strtol(char *s, char **next, int base)`
  `long atol(char *s)`
  wandelt String `s` in `long` um
- `strtod(char *s, char **next)`
  `double atof(char *s)`
  wandelt String `s` in `double` um

`printf()`: Sonderzeichen / Maskierung

Technische Universität Berlin

Wichtige Sonderzeichen

```
\n Newline, Zeilenumbruch
\r Carriage-Return, Wagenrücklauf
\t Tabulator
\0 NUL - Endezeichen im String
```

Maskierung (Escaping) von reservierten Zeichen

```
\'   einfaches Anführungszeichen '
\"   doppeltes Anführungszeichen "
%%   Prozentzeichen %
\\   Backslash \
```

`snprintf()`: Konvertierung von Strings

```
int snprintf(char *s, int n, char *fmt, …)
```

- Formatierte Ausgabe wie printf(), jedoch in String `s`
- Maximal `n` Zeichen lang
- Achtung: `s` muss groß genug sein (sonst Buffer Overflow)
- Best Practice: Länge von `s` als Parameter `n` übergeben

Kommandozeilenparameter und Rückgabewert

Kommandozeilenparameter

Technische Universität Berlin

Jedes C-Programm startet mit der Funktion main()

```
int main(int argc, char *argv[], char *envp[])
```

- `int argc`: Anzahl von Kommandozeilenparametern
- `char *argv[]`: Kommandozeilenparameter
  - Array von Strings, `argv[argc] == NULL`
  - `argv[0]` enthält den Namen des Programms

- `char *envp[]`: Umgebungsvariaben (seltener benutzt)
  - Array von Strings, NULL terminiert `envp[] == NULL`

Rückgabewert

Jedes C-Programm startet mit der Funktion main()

```
int main (int argc, char *argv[], char *envp[])
```

- Rückgabewert wird vom Betriebssystem ausgewertet
  - Konvention: Wert 0 bedeutet Programm zeigt keinen Fehler an
  - Konvention: Werte != 0 bedeuten Programm hat Fehler erkannt
  - Werte über 128 kommen vom Betriebssystem (siehe „`man signal`“)

Kommandozeilenparameter

Technische Universität Berlin

Beispiel:

```
#include <stdio.h>
int main(int argc,
         char *argv[] // entspricht char **argv
) {
   for(int i = 0; i < argc; i++) {
         printf("%02d: %s\n", i, argv[i]);
   }
   return(0);
}
```

Technische Universität Berlin

Eingabe

Open Distributed Sytems
ODS

Zeilenweise Eingabe: `getline`

Technische Universität Berlin

```
size_t getline(char** lp, size_t* lz, FILE* fp)
```

liest eine Zeile von `fp` und speichert sie in `lp`

- `lp` muss der Anfang eines mit `malloc/calloc` allozierten Speicherbereiches oder `NULL` sein.
- Die Größe von `lp` muss in `lz` stehen.
- Der Rückgabewert ist die
  - Anzahl der chars incl. `'\n'`
  - `-1` bei Fehler / `EOF` („End Of File“ – Ende der Datei / Eingabe)
- Wenn die Zeile nicht in `lp` passt, wird der Speicherbereich automatisch mit `realloc` vergrößert.

Zeilenweise Eingabe: getline

```
ssize_t getline(char** lp, size_t* lz, FILE* fp)
```

Beispiel:

Benötigt ggf. Compiler-Parameter: `-D_DEFAULT_SOURCE`

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

int main(int argc, char **argv) {

int lines = 0;
int chars = 0;
char* buf = NULL;
size_t buflen = 0;
```

```
while (getline(&buf, &buflen, stdin) > 0){
   lines++;
   chars+=strlen(buf);
}

printf("Read %2d lines, %3d chars\n",
       lines, chars);

if (buf != NULL) { free(buf); }
return(0);
}
```

Formatierte Eingabe: `scanf`

Technische Universität Berlin

`int scanf(fmt, …)`

`scanf()` liest von `stdin` (üblicherweise Tastatur) und versucht die Eingabe unter Kontrolle des Formatstrings `fmt` auf die Parameter abzubilden

- Der Formatstring `fmt` ist eine Zeichenkette mit Leerzeichen
- Die Parameter dürfen nicht fehlen (für jeden im Formatstring einer!)
- Die Parameter müssen den selben Typ haben, wie im Formatstring `fmt` angegeben

Beispiele:

```
int a, b; scanf("%d %d", &a, &b);
float x; scanf("%f", &x);
char a; scanf("%c", &a);
```

Formatierte Eingabe: scanf

Technische Universität Berlin

`int scanf(fmt, …)`

Der Rückgabewert von scanf ist

- Wenn mindestens eine Eingabe erfolgreich:
  Anzahl der Einträge, die erfolgreich gelesen wurden
- Wenn nicht erfolgreich: -1 (EOF == End Of File)

RETURN VALUE von ***`scanf()`***
These functions return the number of input items successfully matched and assigned, which can be fewer than provided for, or even zero in the event of an early matching failure.

The value EOF is returned if the end of input is reached before either the first successful conversion or a matching failure occurs. EOF is also returned if a read error occurs, in which case the error indicator for the stream (see ferror(3)) is set, and errno is set indicate the error.