Praxis der Programmierung

Zeichenketten (Strings), Ein- und Ausgabe

Institut für Informatik und Computational Science
Universität Potsdam

Henning Bordihn

Zeichenketten (Strings)

- String = Zeichenkette
- **konstante** (unveränderliche) Strings in Anführungszeichen definiert
  - bisher als Parameter von `printf`
  - `printf("Ich bin ein String.");`
- *allgemein:* Zeichenkette ist Folge von Character-Werten

$\rightsquigarrow$ kann als Array vom Typ `char` aufgefasst werden

Strings in C

- sind `char`-Arrays mit **Nullzeichen** `'\0'` als Markierung des Stringendes
- Viele String-Funktionen benötigen das Nullzeichen.
  $\rightsquigarrow$ bei Definition des Arrays einplanen!

  ```
  char vorname [6] = {'N', 'a', 'd', 'j', 'a', '\0'};
  ```

- Initialisierung mit konstanter Zeichenkette:

  ```
  char vorname [6] = "Nadja";
  ```

  $\rightsquigarrow$ automatisches Anfügen des Nullzeichens

- *Erinnerung:* Arraybezeichner liefert Pointer auf das erste Element
  $\rightsquigarrow$ alternative Definition: `char *vorname = "Nadja";`
  (ist Pointer auf `char`, nämlich auf den ersten Buchstaben)

Dynamische Strings

- Definition als char-Array: `char vorname [6] = "Nadja";`
- Definition als offenes Array ist möglich: `char str[] = "Hallo";`
- Normale Zugriffsmöglichkeiten wie bei allen Arrays
  ⇝ Überschreiben einzelner Buchstaben im String möglich

⇝ Pointer `str` ist konstant (kann nicht umgesetzt werden)

Statische Strings

- `char`-Pointer mit konstantem String initialisiert: `char *str = "Hallo";`
- Pointer kann auf andere Adresse umgesetzt werden
- Pointer auf *read-only* Speicherblock
  - im statischen Datensegment (Speicherbereich für Daten *neben* dem Stack für globale und statische Variablen) oder
  - direkt im Code-Segment

$\rightsquigarrow$ String darf nur gelesen werden

$\rightsquigarrow$ Veränderung des Strings löst **segmentation fault** aus oder kann zu schweren Fehlern führen (u.U. Änderung am Maschinencode)

Statische Strings benutzen

- Ausgabe des gesamten Strings mit `printf`:
  `printf(str);` oder `printf("... %s ...", str);`

$\rightsquigarrow$ Formatelement `%s` zum Integrieren in formatierte Ausgaben,
Übergabe eines `char`-Pointers (`str`)

Statische Strings benutzen (2)

- Zugriff auf einzelne Zeichen, z.B.:

```
char *str = "Hallo";
char c1 = str[0];   //  == 'H'  (in str nicht veraendern!!!)
char c2 = str[1];   //  == 'a'  (in str nicht veraendern!!!)
char c3 = str[5];   //  == '\0' (in str nicht veraendern!!!)
str = "String";     // erlaubt, da str nicht konstant
```

- Ergebnis:
  - in einem *read-only* Speicherbereich liegen zwei unbenannte char-Arrays (unveränderlich)
  - im Stack liegt ein Pointer `str` auf char (veränderlich)

Dynamische Strings benutzen

- lesender Zugriff wie bei statischen Strings
- außerdem schreibender Zugriff auf einzelne Buchstaben
- Definition z.B. als offene Arrays:

```
char str [] = "Hallo";  // char-Array mit 6 Komponenten
                        // liefert char-Pointer auf das 'H'

str[1] = 'e';           // wandelt den String in "Hello"

str = "String";         // verboten, da Pointer str konstant
str++;                  // verboten, da es Konstante veraendern wuerde
```

Dynamische Strings benutzen (2)

- Einlesen des Strings möglich, z.B. mit `scanf` oder `fgets`:

```
char str2[1024];
printf("Geben Sie Ihren Namen ein!");
fgets(str2, 1024, stdin);   // max. 1023 Zeichen (warum?)
                            // von stdin lesen
                            // und ab Adresse str2 speichern
```

```
char * fgets (char * s, int size, FILE * stream);
```

  - liest `size-1` Zeichen aus `stream`
    oder bis `'\n'` oder **EOF** und speichert sie ab Adresse `s`
  - `'\n'` wird mit gespeichert und `'\0'` angehängt
  - übergibt Pointer `s`

Standardfunktionen zur Eingabe aus `stdio`

- `char * fgets (char * s, int size, FILE * stream);`
  - liest `size-1` Zeichen aus `stream` oder bis `'\n'` oder **EOF** und speichert sie ab Adresse `s`
  - `'\n'` wird mit gespeichert und `'\0'` angehängt
  - übergibt Pointer `s`

- `char * gets (char * s);`
  - liest von `stdin` bis `'\n'` oder **EOF** und speichert ab Adresse `s`
  - ersetzt `'\n'` bzw. **EOF** durch das Nullzeichen
  - übergibt Pointer `s`

- *Warum ist* `gets` *im Vergleich zu* `fgets` *gefährlich?*

Standardfunktionen zur Eingabe aus `stdio` (2)

- `int scanf (const char * format, ...);`
  - Argumente nach dem Formatstring sind **Adressen von Variablen**,
  - Speichern der Werte aus `stdin` auf diesen Adressen
  - Anzahl und Typen der Formatelemente müssen zu den adressierten Variablen passen (sonst Abbruch des Einlesens)
  - Rückgabewert: Anzahl der erfolgreich eingelesenen Werte

- Beispiel:

```
int zahl;

printf ("\nEingabe: ");
scanf("%d", &zahl);

printf ("\nDer Wert %d wurde eingelesen.\n", zahl);
```

- andere Zeichen als Formatelemente im Formatstring möglich:
  - `scanf()` liest und ignoriert diese Zeichen („Wegwerfen“)
  - Whitespace-Zeichen: „Wegwerfen“ einer beliebigen Anzahl dieses Zeichens
  - nichtpassende Zeichen in `stdin` werden zurückgestellt (verbleiben)
  - verhält sich so, bis `'\n'` gelesen wird

```
float t;
printf("Temperatur im Format xx C: ");
scanf("%f C", &t);
t = (9. * t) / 5. + 32.;
printf("\nTemperatur in Fahrenheit: %f F", t);
```

$\rightsquigarrow$ Kein Newline-Zeichen `'\n'` im Formatstring von `scanf()`!

- „bewusstes“ Ignorieren von zum Formatstring passenden Eingaben durch `*` nach `%`: *Lesen ohne zu speichern*

  Datei `daten.txt` enthält
  `Artikel: Tisch Vorrat: 8 Einzelpreis: 290`

  Aufgabe: C-Programm `prog.c` soll Wert des Lagerbestands ermitteln:

```
int anzahl;
int einzelpreis;
scanf("%*s %*s %*s %d %*s %d", &anzahl, &einzelpreis);
printf("\nWert des Lagerbestands: %d", anzahl * einzelpreis);
```

  zu starten mit `prog < daten.txt`

Besonderheiten in der Signatur von `scanf`

```
int scanf (const char * format, ...);
```

- `...` – *Ellipse*
  - muss nach dem letzten expliziten formalen Parameter stehen
  - Anzahl (und Typen) weiterer Parameter offen
  - Beispiel:

```
int ellipse_func (int n, double x, ...);
...
ellipse_func(4, 5.6, "String");   // o.k.
ellipse_func(4, 5.6, 7, 8.9);     // o.k.
ellipse_func(4, 5.6);             // o.k.
ellipse_func(4);                  // Fehler!!!
```

- `const char * format` $\rightsquigarrow$ Array-Elemente (String) konstant
  `char * const format` $\rightsquigarrow$ Pointer konstant

Verwendung des Rückgabewerts von `scanf`

Abfangen von Typfehlern bei Benutzereingaben

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

int main() {
    float n;
    printf("Geben Sie eine Zahl ein: ");
    int status = scanf("%f", &n);
    if (status == 0) {
        printf("Sie haben keine Zahl eingegeben.\n\n");
        exit(EXIT_FAILURE);
    }
    printf("Die Zahl ist %f.\n", n);
    return 0;
}
```

Vermeidung von Überläufen bei Stringeingaben

- Formatelement %ns zum Einlesen eines Strings einer Länge $\leq n$

- längere Strings werden abgeschnitten
  $\rightsquigarrow$ kein Überschreiben von Speicherbereichen außerhalb eines char-Arrays

```
char str[20];
scanf("%19s", str); // bei Eingabe von mehr als 19 Zeichen
                    // verbleiben ueberzaehlige Zeichen
                    // im Buffer von scanf
```

Standardfunktionen zur Eingabe aus `stdio` (3)

- `int getchar ();`
  - liest einzelne Zeichen aus dem Eingabestrom `stdin`
  - liefert ein `unsigned char`, das in `int` konvertiert wird
  - zeilengepuffert (wartet auf `RETURN`)

```
#define LEN 40
...
char str[LEN];
int char_in;
int i = 0;

while(i < LEN && (char_in = getchar()) != '\n')
   str[i++] = (char) char_in;
```

Anwendung: Leeren des Eingabepuffers von `scanf`

Problem: Fehlerhafte Eingabe für `scanf` verbleibt im Eingabepuffer

$\rightsquigarrow$ nächster Aufruf von `scanf` beginnt dort zu lesen

```
int c, status, zahl;
status = scanf("%d", &zahl);
if (status == 0)
   do
      c = getchar();
   while (c != '\n');
```

(*Wichtig für Fehlerbehandlung mit Recovering*)

Standardfunktionen zur Ausgabe aus `stdio`

- `int printf (const char * format, ...);`
- `int puts (const char * s);`
  - schreibt übergebenen String `s` nach `stdout`
  - kopiert das Nullzeichen *nicht* mit
  - fügt ein `'\n'` an
- geben die Länge der ausgegebenen Strings zurück

Übergabe von Strings als Parameter

- **Übergabe** eindimensionaler Arrays an Funktionen:
  formale Parameter als
  - offenes Array oder
  - Pointer auf den Komponententyp
- Anwendung bei Übergabe von Zeichenketten (char-Array)

```
int lengthOfString(char * ar) {
   int i = 0;
   while (ar[i] != '\0') {
      i++;
   }
   return i;
}
```

```
int main() {
   char *s = "Hallo Du!";
   int n = lengthOfString(s);
   ...
}
```

Standardfunktionen zur Stringverarbeitung

- in Header-Datei `<string.h>`:
-

```
size_t strlen (const char * s);                          Länge
char * strcpy (char * dest, const char * src);           Kopieren
char * strcat (char * dest, const char * src);           Anhängen
int strcmp (const char * s1, const char * s2);           Vergleichen
int strncmp (const char * s1, const char * s2,
                                        size_t n);       Vergleichen
```

  (0 bei Gleichheit)
- Nullzeichen `'\0'` entscheidend für korrektes Arbeiten
- `size_t` vordefinierter Datentyp als Rückgabetyp des `sizeof`-Operator (ist meist `unsigned int` oder `unsigned long`)

Warum Stringfunktionen wie `strcpy` ?

- Aufgabe: Kopieren von String `src` in String `dest`
- naives Herangehen: `dest = src;`

  $\rightsquigarrow$ Was passiert?
- Übergabe des Pointers

  $\rightsquigarrow$ Jede Änderung an `dest` auch in `src` und umgekehrt
- `strcpy` ändert keinen Pointer,
  sondern kopiert den Inhalt von `src` an die Stelle `dest`

  $\rightsquigarrow$ Verdopplung des Strings im Speicher

Vergleichen mit `strcmp` und `strncmp`

- `int strcmp (const char * s1, const char * s2)`
  - zeichenweiser Vergleich bis Unterschied oder `'\0'`
  - Rückgabewert ist
    - `<` `0` wenn erster String lexikographisch kleiner
    - `>` `0` wenn erster String lexikographisch größer
    - `0` bei Gleichheit

- `int strncmp (const char * s1, const char * s2, size_t n)`
  - wie `strcmp` mit zusätzlichem Abbruchkriterium
  - Abbruch, wenn Unterschied, `'\0'` oder `n` Zeichen verglichen

Funktionen zur Speicherbearbeitung

- ähnliche Funktionen zu den Stringfunktionen für beliebige Speicherobjekte
- Funktionsbezeichner beginnen mit `mem` statt mit `str`
  (z.B. `memcpy`, `memcmp` etc.)
- formale Parameter `void *` statt `char *`
- verarbeiten die übergebenen Speicherobjekte byteweise
- keine Prüfung/Verwendung des `'\0'`-Zeichens
- haben Anzahl der zu bearbeitenden Bytes als weiteren Parameter
- `#include <string.h>`

Funktionen zur Speicherbearbeitung (2)

- `void * memcpy(void * dest, const void * src, size_t n);`
  kopiert `n` Bytes aus Speicherplatz `scr` in Speicherplatz `dest`
  $\rightsquigarrow$ *Vorsicht bei überlappendem Speicherbereich!*

- `void * memmove(void * dest, const void * src, size_t n);`
  wie `memcpy`, schützt vor Fehlern durch überlappenden Speicherbereich
  $\rightsquigarrow$ kopiert zuächst in Zwischenpuffer, bevor auf `dest` geschrieben wird

- `int memcmp(const void * s1, const void * s2, size_t n);`
  byteweiser Vergleich, bis Unterschied oder `n` Bytes verglichen

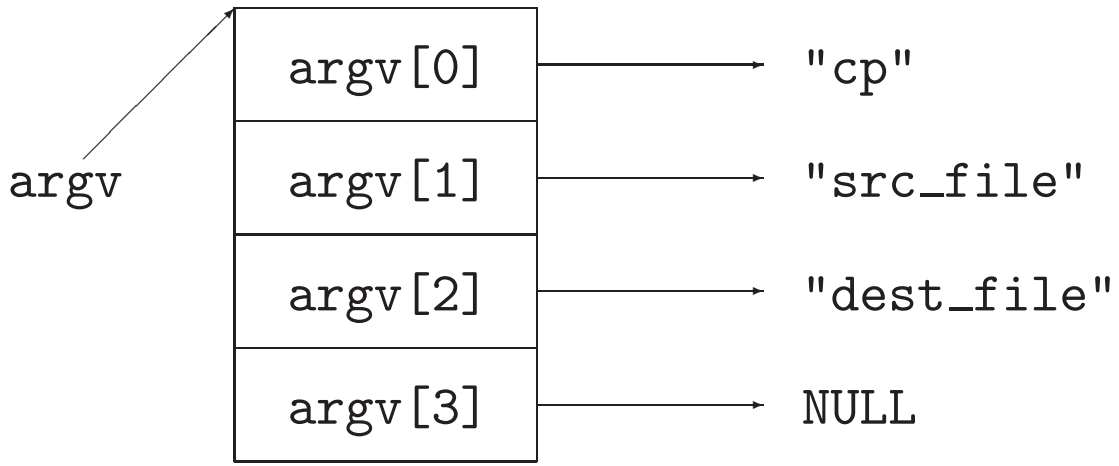
Funktionen zur Speicherbearbeitung (3)

- `void * memchr(const void * s, int c, size_t n);`
  durchsucht die ersten `n` Bytes des Speicherobjekts an `s` nach dem Wert `c` (interpretiert als `unsigned char`)
  $\rightsquigarrow$ gibt Pointer auf das erste Vorkommen von `c` oder `NULL` zurück

- `void * memset(void * s, int c, size_t n);`
  setzt die `n` Bytes ab Adresse `s` auf `c` (konvertiert in `unsigned char`)

Parameterübergabe beim Programmaufruf

- Beispiel: `cp src_file dest_file`
- zwei Varianten der `main`-Funktion:
  - `int main()` — parameterlos
  - `int main(int argc, char * argv[])` — zwei Parameter
- `argc` (**arg**ument **c**ounter): Anzahl der Argumente
- `argv` (**arg**ument **v**ector): Vektor (Array) der Argumente
  Argumente sind Strings $\rightsquigarrow$ Array von `char`-Arrays
  $\rightsquigarrow$ Array von Pointern auf `char`
- erstes Element von `argv` (`argv[0]`): Programmname ($\Longrightarrow$ `argc` ≥ 1)

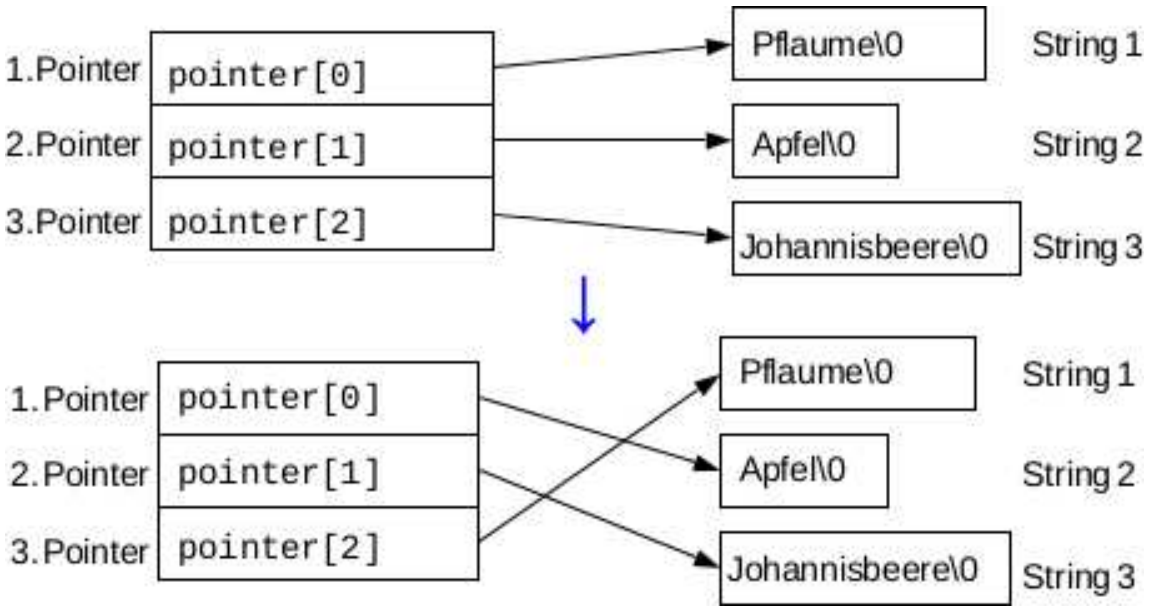


- Übergabe von Zahlen: Typumwandlung String $\longrightarrow$ Zahltyp erforderlich
- Standardfunktionen aus <stdlib.h>

```
double atof(const char * nptr);   ascii to float
int atoi(const char * nptr);      ascii to int
long atol(const char * nptr);     ascii to long
```

Arrays von Pointern

- z.B. in `int main(int argc, char * argv[])`
- erlaubt z.B. Sortieren von Strings ohne Kopieraktionen



Arrays von Pointern *versus* mehrdimensionale Arrays

- mehrdimensionale Arrays: Anzahl der Elemente für jede Dimension fest:

  `int matrix [6][10];` $\rightsquigarrow$ Array mit 60 `int`-Werten

- Array von Pointern: nur die Anzahl der Pointer ist fest:

  `int * pointer_array [6]` $\rightsquigarrow$ 6 Pointer auf `int`

  $\rightsquigarrow$ „zweite Dimension“ nicht festgelegt

- häufigste Anwendung für Datentyp `char`

  $\rightsquigarrow$ Array von Strings unterschiedlicher Länge

Pointer auf Pointer als formale Parameter

- `char * stringArray[]` ausdrückbar als `char * * stringArray`

- beim Aufruf: Übergabe eines Stringarrays
  ⇝ Übergabe der Adresse des ersten Strings im Array
  ⇝ Übergabe der Adresse des ersten Zeichens der ersten Komponente

- z.B. Ausgabe aller Strings in einem Array `ar` mit 36 Strings als Text:

```
void textausgabe(char * * stringArray, int anzahl) {
      int i;
      for (i = 0; i < anzahl; i++)
            printf("%s ", stringArray[i]);
}
```

Aufruf: `textausgabe(&ar[0],36);`

Pointer auf Pointer als formale Parameter (2)

Nach Übergabe von `&ar[0]` an `char * * stringArray`:

- `*stringArray` ist Pointer `ar[0]` (Pointer auf `char`)
- `**stringArray` ist das erste Zeichen des Strings in `ar[0]`
- `stringArray++` verschiebt `stringArray` auf `ar[1]`

```
void textausgabe(char * * stringArray, int anzahl) {
      while(anzahl-- > 0)
            printf("%s ", *stringArray++);  // dereferenzieren,
}                                           // dann Nebeneffekt (Postfix)
```