

Praxis der Programmierung

Zeichenketten (Strings), Ein- und Ausgabe

**Institut für Informatik und Computational Science
Universität Potsdam**

Henning Bordihn

Einige Folien gehen auf A. Terzibaschian zurück.

Zeichenketten (Strings)

Zeichenketten (Strings)

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

⇒ Array vom Typ `char`

Strings (2)

- sind char-Arrays mit **Nullzeichen** `'\0'` als Markierung des Stringendes
- Viele String-Funktionen benötigen das Nullzeichen.

~> bei Definition des Arrays einplanen!

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

- Initialisierung mit konstanter Zeichenkette:

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

~> automatisches Anfügen des Nullzeichens

- *Erinnerung:* Arrays sind Pointer auf das erste Element

```
char *vorname = "Nadja";
```

(ist Pointer auf den ersten Buchstaben)

Statische *versus* dynamische Strings

- statische Strings
 - char-Pointer mit konstantem String initialisiert: `char *str = "Hallo";`
 - Pointer auf statischen Speicherblock direkt im Binärcode
 - ↪ String darf nur gelesen werden
 - ↪ Veränderung z.B. von `str[500]` kann zu schweren Fehlern führen (u.U. Änderung am Maschinencode)
 - Pointer kann aber auf andere Adresse umgesetzt werden

Statische *versus* dynamische Strings

- statische Strings

- char-Pointer mit konstantem String initialisiert: `char *str = "Hallo";`
- Pointer auf statischen Speicherblock direkt im Binärcode
 - ↪ String darf nur gelesen werden
 - ↪ Veränderung z.B. von `str[500]` kann zu schweren Fehlern führen (u.U. Änderung am Maschinencode)
- Pointer kann aber auf andere Adresse umgesetzt werden

- dynamische Strings

- Definition als char-Array: `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 benutzen

- Zugriff auf einzelne Zeichen, z.B.:

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

- Ausgabe des gesamten Strings mit printf:

```
printf(str);    oder    printf("... %s ...", str);
```

↪ Formatelement %s zum Integrieren in formatierte Ausgaben,
Übergabe eines char-Pointers (str)

Dynamische Strings benutzen

- lesender Zugriff wie bei statischen Strings
- außerdem schreibender Zugriff auf einzelne Buchstaben
- Definition z.B. mit offenen Arrays möglich

```
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 in 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 in `s`
 - `'\n'` wird mit gespeichert und `'\0'` angehängt
 - übergibt Pointer `s`
- `char * gets (char * s);`
 - liest von `stdin` bis `'\n'` oder **EOF** in char-Array `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 unbestimmtes Verhalten)
 - 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 diese Zeichen und ignoriert sie für den Rückgabe-String
 - stellt sie in `stdin` zurück
 - 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);
```

~> 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);
```

- „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!!!
```

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

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

~> nächster Aufruf von scanf beginnt dort zu lesen

Anwendung: Leeren des Eingabepuffers von scanf

Problem: Fehlerhafte Eingabe für scanf verbleibt im Eingabepuffer

~> nächster Aufruf von scanf beginnt dort zu lesen

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

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, l = 0;  
    while (ar[i] != '\0') {  
        l++;  
        i++;  
    }  
    return l;  
}
```

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

Standardfunktionen zur Stringverarbeitung

- in Header-Datei `<string.h>`:
- | | |
|--|-------------|
| <code>size_t strlen (const char * s);</code> | Länge |
| <code>char * strcpy (char * dest, const char * src);</code> | Kopieren |
| <code>char * strcat (char * dest, const char * src);</code> | Anhängen |
| <code>int strcmp (const char * s1, const char * s2);</code> | Vergleichen |
| <code>int strncmp (const char * s1, const char * s2,
size_t n);</code> | Vergleichen |

(0 bei Gleichheit, d.h. wenn `*s1 == *s2`)
- Nullzeichen `'\0'` entscheidend für korrektes Arbeiten
- `size_t` vordefinierter Datentyp als Rückgabotyp 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?

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

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

Vergelichen 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

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)

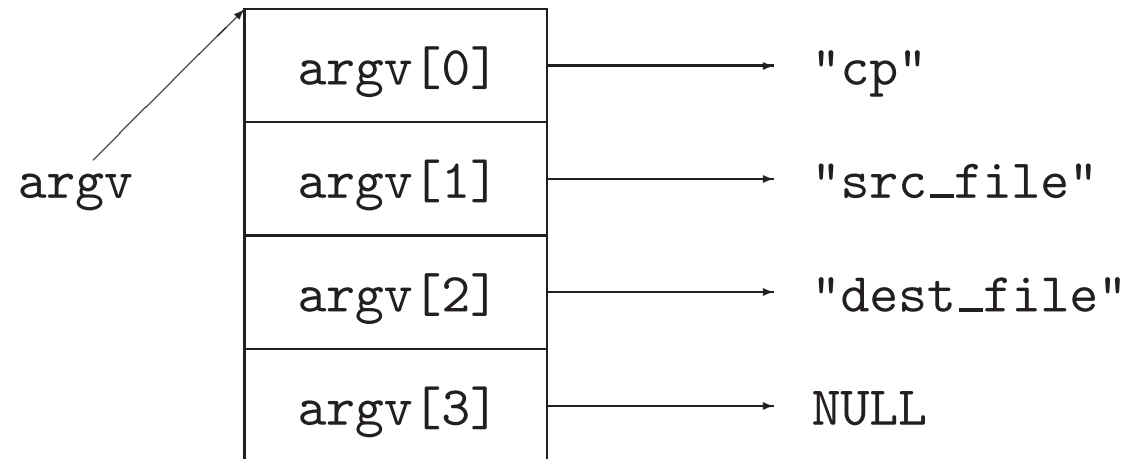
- `void * memcpy(void * dest, const void * src, size_t n);`
kopiert n Bytes aus Speicherplatz src in Speicherplatz dest
~> *Vorsicht bei überlappendem Speicherbereich!*
- `void * memmove(void * dest, const void * src, size_t n);`
wie memcpy, schützt vor Fehlern durch überlappenden Speicherbereich
~> kopiert zunächst 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

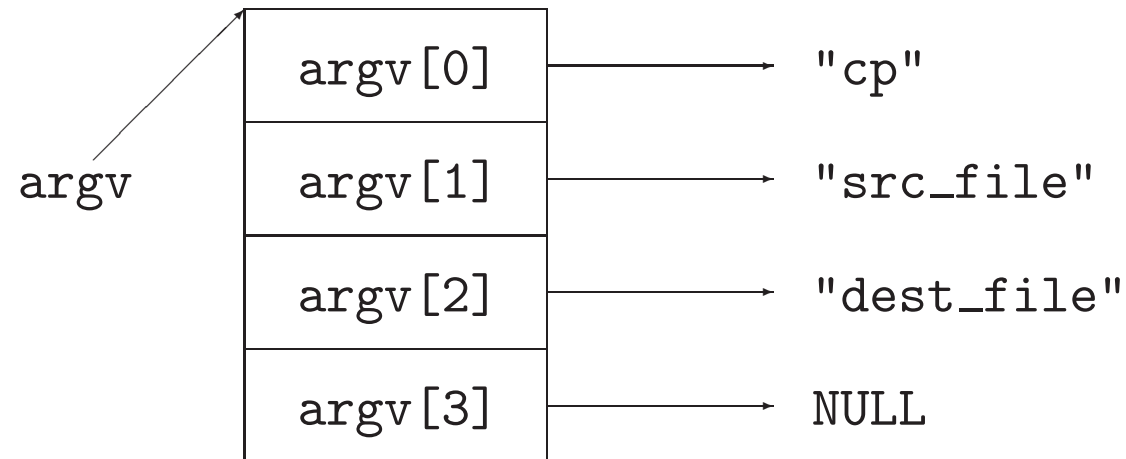
Funktionen zur Speicherbearbeitung (3)

- `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`)
~> 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` (**a**rgument **c**ounter): Anzahl der Argumente
- `argv` (**a**rgument **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





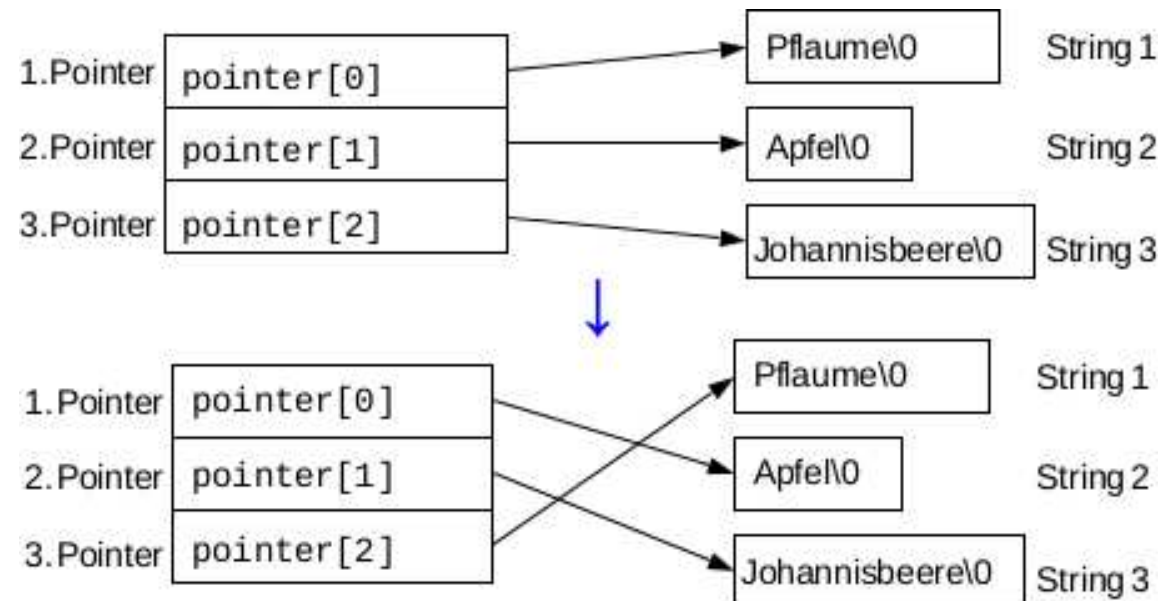
- Übergabe von Zahlen: Typumwandlung String \longrightarrow Zahltyp erforderlich

- Standardfunktionen aus `<stdlib.h>`

<code>double atof(const char * nptr);</code>	ascii to float
<code>int atoi(const char * nptr);</code>	ascii to int
<code>int atol(const char * nptr);</code>	ascii to long

Arrays von Pointern

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



Pointer auf Pointer als formale Parameter

- `char * stringArray[]` gleichwertig mit `char * * stringArray`
- beim Aufruf: Übergabe eines Stringarrays
 - ~> Übergabe der Adresse des ersten Strings im Array
 - ~> Übergabe der Adresse des ersten Zeichens der ersten Komponente

Pointer auf Pointer als formale Parameter

- `char * stringArray[]` gleichwertig mit `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]`
- `**stringArray` ist das erste Zeichen des Strings in `ar[0]`
- `*(*stringArray)++` liefert das erste Zeichen in `ar[0]`
und setzt `*stringArray` auf das nächste Zeichen des Strings in `ar[0]`
- `stringArray++` verschiebt `*stringArray` auf `ar[1]`

Pointer auf Pointer als formale Parameter (2)

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

- `*stringArray` ist Pointer `ar[0]`
- `**stringArray` ist das erste Zeichen des Strings in `ar[0]`
- `*(*stringArray)++` liefert das erste Zeichen in `ar[0]`
und setzt `*stringArray` auf das nächste 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
```

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