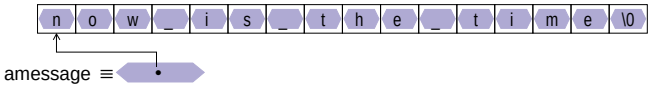
K Ergänzungen zur Einführung in C

K.1 Zeiger, Felder und Zeichenketten

- ➤ Zeichenketten sind Felder von Einzelzeichen (**char**), die in der internen Darstellung durch ein **'\0'**–Zeichen abgeschlossen sind

- wird eine Zeichenkette zur Initialisierung eines char–Feldes verwendet, ist der Feldname ein konstanter Zeiger auf den Anfang der Zeichenkette

```
char amessage[] = "now is the time";
```

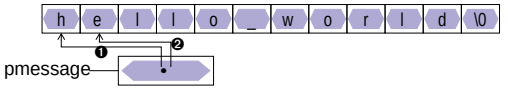


  - ➤ es wird ein Speicherbereich für 16 Bytes reserviert und die Zeichen werden in diesen Speicherbereich hineinkopiert
  - ➤ `amessage` ist ein konstanter Zeiger auf den Anfang des Speicherbereichs und kann nicht verändert werden
  - ➤ der Inhalt des Speicherbereichs kann aber modifiziert werden
    `amessage[0] = 'h';`

K.1 … Zeiger, Felder und Zeichenketten (3)

- wird eine Zeichenkette zur Initialisierung eines **char**–Zeigers verwendet, ist der Zeiger eine Variable, die mit der Anfangsadresse der Zeichenkette initialisiert wird

```
char *pmessage = "hello world";
```



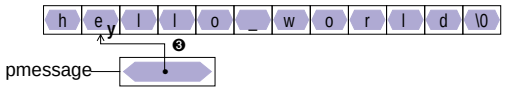

```
pmessage++; ❷
printf("%s", pmessage);  /*gibt "ello world" aus*/
```

  - ➤ es wird ein Speicherbereich für einen Zeiger reserviert (z. B. 4 Byte) und der Compiler legt die Zeichenkette `hello world` an irgendeiner Adresse im Speicher des Programms ab
  - ➤ `pmessage` ist ein variabler Zeiger, der mit dieser Adresse initialisiert wird, aber jederzeit verändert werden darf
    `pmessage++;`

K.1 … Zeiger, Felder und Zeichenketten (4)

- wird eine Zeichenkette zur Initialisierung eines **char**–Zeigers verwendet, ist der Zeiger eine Variable, die mit der Anfangsadresse der Zeichenkette initialisiert wird

```
char *pmessage = "hello world";
```




```
*pmessage = 'y'; ❸
```

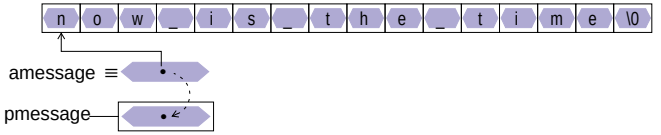
  - ➤ der Speicherbereich von `hello world` darf aber nicht verändert werden
    - – manche Compiler legen solche Zeichenketten in schreibgeschütztem Speicher an
      ➔ Speicherschutzverletzung beim Zugriff
    - – sonst funktioniert der Zugriff obwohl er nicht erlaubt ist
      ➔ Programm funktioniert nur in manchen Umgebungen

K.1 … Zeiger, Felder und Zeichenketten (5)

- die Zuweisung eines **char**–Zeigers oder einer Zeichenkette an einen **char**–Zeiger bewirkt kein Kopieren von Zeichenketten!

```
pmessage = amessage;
```

weist dem Zeiger **pmessage** lediglich die Adresse der Zeichenkette **"now is the time"** zu



- wird eine Zeichenkette als aktueller Parameter an eine Funktion übergeben, erhält diese eine Kopie des Zeigers

K.1 … Zeiger, Felder und Zeichenketten (6)

- Zeichenketten kopieren

```
/* 1. Version */
void strcpy(char s[], t[])
{
    int i=0;
    while ( (s[i] = t[i]) != '\0' )
        i++;
}

/* 2. Version */
void strcpy(char *s, *t)
{
    while ( (*s = *t) != '\0' )
        s++, t++;
}

/* 3. Version */
void strcpy(char *s, *t)
{
    while ( *s++ = *t++ )
        ;
}
```


K.2 Argumente aus der Kommandozeile

- beim Aufruf eines Kommandos können normalerweise Argumente übergeben werden
- der Zugriff auf diese Argumente wird der Funktion ***main( )*** durch zwei Aufrufparameter ermöglicht:

```
int
main (int argc, char *argv[])
{
    …
}
```

oder

```
int
main (int argc, char **argv)
{
    …
}
```

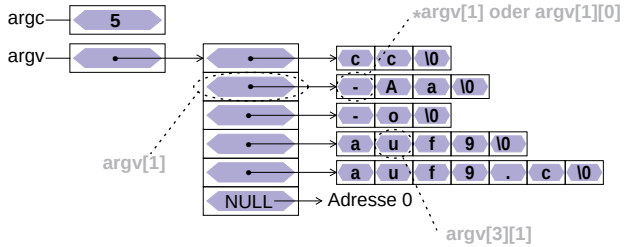
- der Parameter **`argc`** enthält die Anzahl der Argumente, mit denen das Programm aufgerufen wurde
- der Parameter **`argv`** ist ein Feld von Zeiger auf die einzelnen Argumente (Zeichenketten)
- der Kommandoname wird als erstes Argument übergeben (**`argv[0]`**)

1 Datenaufbau

Kommando: **`cc -Aa -o auf9 auf9.c`**

Datei cc.c:

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

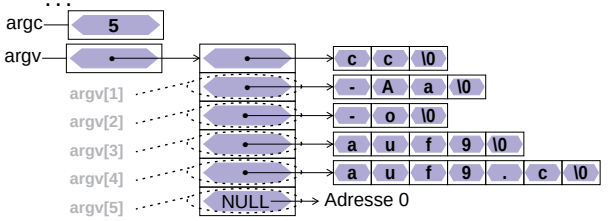


2 Zugriff
Beispiel: Ausgeben aller Argumente (1)

- das folgende Programmstück gibt alle Argumente der Kommandozeile aus (außer dem Kommandonamen)

1. Version

```
int
main (int argc, char *argv[])
{   int i;
    for ( i=1; i<argc; i++) {
        printf("%s%c", argv[i],
            (i < argc-1) ? ' ':'\n' );
    }
    ...
```



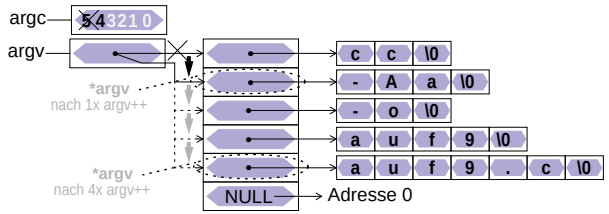
2 Zugriff
Beispiel: Ausgeben aller Argumente (2)

- das folgende Programmstück gibt alle Argumente der Kommandozeile aus (außer dem Kommandonamen)

```
int
main (int argc, char **argv)
{       while (--argc > 0) {
                argv++;
                printf("%s%c", *argv, (argc>1) ? ' ' : '\n' );
        }
        ...
```

linksseitiger Operator: erst dekrementieren, dann while-Bedingung prüfen → Schleife läuft für argc=4,3,2,1

2. Version



K.3 Strukturen

1 Rekursive Strukturen

- Strukturen in Strukturen sind erlaubt — aber
  - die Größe einer Struktur muss vom Compiler ausgerechnet werden können
    - ➤ Problem: eine Struktur enthält sich selbst
  - die Größe eines Zeigers ist bekannt (meist 4 Byte)
    - ➤ eine Struktur kann einen Zeiger auf eine gleichartige Struktur enthalten

```
struct liste {
    struct student stud;
    struct liste *rest;
};
```

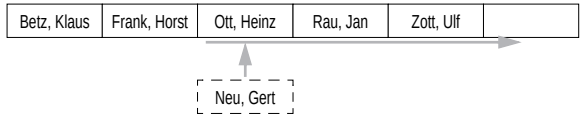
➥ Programmieren rekursiver Datenstrukturen

1 Rekursive Strukturen (2)

- Problem:
  - es sollen beliebig viele Studentendaten eingelesen werden und in sortierter Form im Programm verwaltet werden

Lösung 1: Feld

- ➤ wie groß machen? — und was, wenn es nicht reicht?
- ➤ Einsortieren =
  richtige Position suchen + Rest nach oben verschieben + eintragen

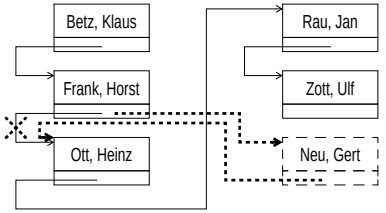


1 Rekursive Strukturen (3)

- Problem:
  - es sollen beliebig viele Studentendaten eingelesen werden und in sortierter Form im Programm verwaltet werden

Lösung 2: verkettete Liste von dynamisch angeforderten Strukturen

- ➤ Speicher für jeden Eintrag mit malloc() anfordern
- ➤ Einsortieren =
  richtige Position suchen + zwei Zeiger setzen



1 Rekursive Strukturen (4)

- Realisierung von Lösung 2 (Skizze):

```
struct eintrag {
       struct student stud;
       struct eintrag *naechster;
};
struct eintrag leer = { {"","}, NULL};        /* Leeres Listenelement */
struct eintrag *stud_liste;                   /* Zeiger auf Listen-Anfang */
struct eintrag *akt_eintrag;                  /* aktuell bearbeiteter Eintrag */
struct eintrag *einfuege_pos;                 /* Einfuegeposition */

int student_lesen(struct student *);
struct eintrag *suche_pos(struct eintrag *liste, struct eintrag *element);
/* erstes Listen-Element anfordern */
akt_eintrag = (struct eintrag *)malloc(sizeof (struct eintrag));
stud_liste = &leer;          /* Listenanfang auf leeres Element setzen (vermeidet später
                                /* eine Sonderbehandlung für Listenanfang) */

while (student_lesen(&akt_eintrag->stud) != EOF ) {
       /* Eintrag, hinter dem einzufügen ist suchen */
       einfuege_pos = suche_pos(stud_liste, akt_eintrag);
       /* akt_eintrag einfügen */
       akt_eintrag->naechster = einfuege_pos->naechster;
       einfuege_pos->naechster = akt_eintrag;
       /* nächstes Listen-Element anfordern */
       akt_eintrag = (struct eintrag *)malloc(sizeof (struct eintrag));
}
```


K.4 Ergänzungen zum Präprozessor

1 Parametrierte Makros

- Präprozessor-Makros können mit Parametern versehen werden, die in den Ersatztext eingebaut werden
  - ➤ entspricht "optisch" einem Funktionsaufruf
  - ➤ Unterschied: Makro wird zur Übersetzungszeit expandiert, Funktionsaufruf erfolgt zur Laufzeit
  - ➤ parametrierte Makros sind damit ähnlich zu inline-Funktionen (wie z. B. in C++)
  - ➤ ein Makro wird durch die **`#define`**–Anweisung definiert
- Syntax:

```
#define Makroname(Par1, Par2, ...) Ersatztext
```

  - ➤ Vorkommen von *Par1*, *Par2*, etc. im Ersatztext werden entsprechend eingesetzt

K.4 Ergänzungen zum Präprozessor (2)

- Beispiele:

```
#define max(a, b) ((a) < (b) ? (b) : (a))
...
x = max (n+m, y+z);
```

```
#define numeric(c) ((c) >= '0' && (c) <= '9')
...
c = getchar();
if numeric(c) { ...
```

  - ➤ im Ersatztext Parameter unbedingt in ( ) klammern (sonst evtl. Probleme mit Operator-Vorrang im expandierten Text!)
  - ➤ einige Sonderregeln im Umgang mit Zeichenketten (Kernighan und Ritchie schreiben dazu: *The details ... are described more precisely in the ANSI standard ... Some of the new rules ... are bizarre.*)

K.4 Ergänzungen zum Präprozessor (3)

2 Deaktivieren von Makros

- Makros wirken ab der Zeile in der sie definiert wurden bis zum Ende der Datei - können aber auch wieder deaktiviert werden
- Syntax:

```
#undef Makroname
```

  - ➤ ab dieser Zeile unterbleibt Expansion des Makros