F.8 sizeof-Operator

- In manchen Fällen ist es notwendig, die Größe (in Byte) einer Variablen oder Struktur zu ermitteln
  - ➤ z. B. zum Anfordern von Speicher für ein Feld (→ malloc)
- Syntax:

sizeof x	liefert die Größe des Objekts x in Bytes
sizeof (*Typ*)	liefert die Größe eines Objekts vom Typ *Typ* in Bytes

- Das Ergebnis ist vom Typ **size_t** (≡ **int**)
  (**#include <stddef.h>**!)
- Beispiel:

```
int a; size_t b;
b = sizeof a;          /* ⇒ b = 2 oder b = 4 */
b = sizeof(double)     /* ⇒ b = 8 */
```


F.9 Felder von Zeigern

- Auch von Zeigern können Felder gebildet werden
- Deklaration

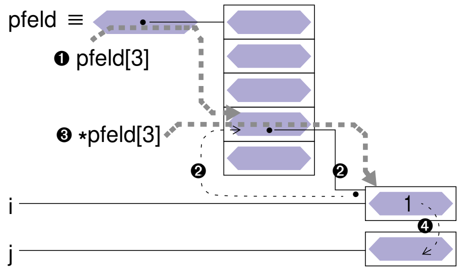
```
int *pfeld[5];
int i = 1
int j;
```

- Zugriffe auf einen Zeiger des Feldes

```
pfeld[3] = &i; ❷
```

- Zugriffe auf das Objekt, auf das ein Zeiger des Feldes verweist

```
j = *pfeld[3]; ❹
```

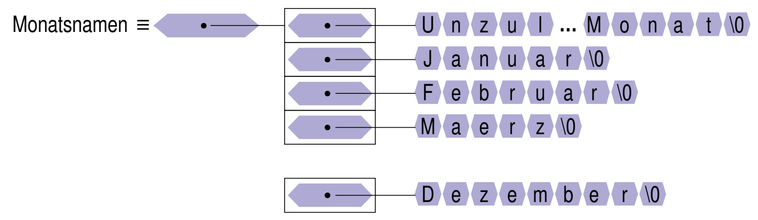


F.9 Felder von Zeigern (2)

- Beispiel: Definition und Initialisierung eines Zeigerfeldes:

```
char *month_name(int n)
{
        static char *Monatsnamen[] = {
                "Unzulaessiger Monat",
                "Januar",
        ...
                "Dezember"
        };

        return ( (n<0 || n>12) ?
                Monatsnamen[0] : Monatsnamen[n] );
}
```



F.10 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

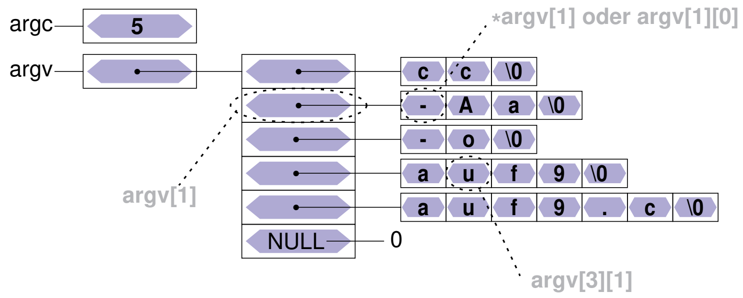
```
int
main (int argc, char **argv)
{
    …
}
```

- 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[]) {
...
```

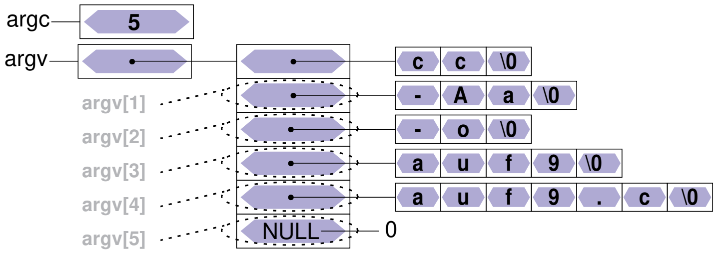


2 Zugriff
Beispiel: Ausgeben aller Argumente (1)

- 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' );
    }
    ...
```



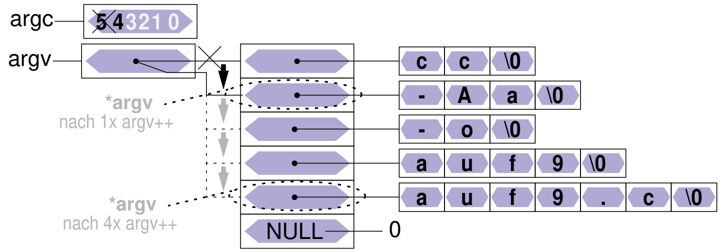
2 Zugriff
Beispiel: Ausgeben aller Argumente (2)

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

2. Version

```
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

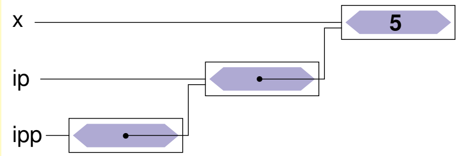


F.11 Zeiger auf Zeiger

- ein Zeiger kann auf eine Variable verweisen, die ihrerseits ein Zeiger ist

```
int x = 5;
int *ip = &x;

int **ipp = &ip;
/* → **ipp = 5 */
```



➤ wird vor allem bei der Parameterübergabe an Funktionen benötigt, wenn ein Zeiger "call bei reference" übergeben werden muss (z. B. swap-Funktion für Zeiger)

F.12 Zeiger auf Funktionen

- Datentyp: Zeiger auf Funktion
  - ◆ Variablendef.: *<Rückgabetyp>* (*<Variablenname>) (<Parameter>);

```
int (*fptr)(int, char*);

int test1(int a, char *s) { printf("1: %d %s\n", a, s); }
int test2(int a, char *s) { printf("2: %d %s\n", a, s); }

fptr = test1;

fptr(42,"hallo");

fptr = test2;

fptr(42,"hallo");
```


F.13 Strukturen

1 Motivation

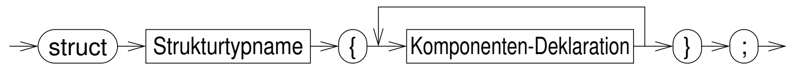
- Felder fassen Daten eines einheitlichen Typs zusammen
  - ➤ ungeeignet für gemeinsame Handhabung von Daten unterschiedlichen Typs
- Beispiel: Daten eines Studenten

– Nachname	`char nachname[25];`
– Vorname	`char vorname[25];`
– Geburtsdatum	`char gebdatum[11];`
– Matrikelnummer	`int matrnr;`
– Übungsgruppennummer	`short gruppe;`
– Schein bestanden	`char best;`

- Möglichkeiten der Repräsentation in einem Programm
  - ◆ einzelne Variablen ➔ sehr umständlich, keine "Abstraktion"
  - ◆ Datenstrukturen

2 Deklaration eines Strukturtyps

- Durch eine Strukturtyp-Deklaration wird dem Compiler der Aufbau einer Datenstruktur unter einem Namen bekanntgemacht
  - ➥ deklariert einen neuen Datentyp (wie `int` oder `float`)
- Syntax



- **Strukturtypname**
  - ◆ beliebiger Bezeichner, kein Schlüsselwort
  - ◆ kann in nachfolgenden Struktur-Definitionen verwendet werden
- **Komponenten-Deklaration**
  - ◆ entspricht normaler Variablen-Definition, aber keine Initialisierung!
  - ◆ in verschiedenen Strukturen dürfen die gleichen Komponentennamen verwendet werden (eigener Namensraum pro Strukturtyp)

2 Deklaration eines Strukturtyps (2)

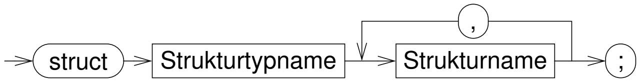
- Beispiele

```
struct student {
    char nachname[25];
    char vorname[25];
    char gebdatum[11];
    int matrnr;
    short gruppe;
    char best;
};
```

```
struct komplex {
    double re;
    double im;
};
```

3 Definition einer Struktur

- Die Definition einer Struktur entspricht einer Variablen-Definition
  - ◆ Name der Struktur zusammen mit dem Datentyp bekanntmachen
  - ◆ Speicherplatz anlegen
- Eine Struktur ist eine Variable, die ihre Komponentenvariablen umfasst
- Syntax



- Beispiele

```
struct student stud1, stud2;
struct komplex c1, c2, c3;
```

- Strukturdeklaration und -definition können auch in einem Schritt vorgenommen werden

4 Zugriff auf Strukturkomponenten

- **.**-Operator
  - ➤ **x.y** ≡ Zugriff auf die Komponente **y** der Struktur **x**
  - ➤ **x.y** verhält sich wie eine normale Variable vom Typ der Strukturkomponenten **y** der Struktur **x**
- Beispiele

```
struct komplex c1, c2, c3;
…
c3.re = c1.re + c2.re;
c3.im = c1.im + c2.im;

struct student stud1;
…
if (stud1.matrnr < 1500000) {
    stud1.best = 'y';
}
```


5 Initialisieren von Strukturen

- Strukturen können — wie Variablen und Felder — bei der Definition initialisiert werden
- Beispiele

```
struct student stud1 = {
    "Meier", "Hans", "24.01.1970", 1533180, 5, 'n'
};

struct komplex c1 = {1.2, 0.8}, c2 = {0.5, 0.33};
```

!!! **Vorsicht**
bei Zugriffen auf eine Struktur werden die Komponenten durch die Komponentennamen identifiziert,
bei der Initialisierung jedoch nur durch die Postion

➥ potentielle Fehlerquelle bei Änderungen der Strukturtyp-Deklaration

6 Strukturen als Funktionsparameter

- Strukturen können wie normale Variablen an Funktionen übergeben werden
  - ◆ Übergabesemantik: **call by value**
    - ➤ Funktion erhält eine Kopie der Struktur
    - ➤ auch wenn die Struktur ein Feld enthält, wird dieses komplett kopiert!
    - !!! Unterschied zur direkten Übergabe eines Feldes
- Strukturen können auch Ergebnis einer Funktion sein
  - ➤ Möglichkeit mehrere Werte im Rückgabeparameter zu transportieren
- Beispiel

```
struct komplex komp_add(struct komplex x, struct komplex y) {
    struct komplex ergebnis;
    ergebnis.re = x.re + y.re;
    ergebnis.im = x.im + y.im;
    return(ergebnis);
}
```

7 Zeiger auf Strukturen

- Konzept analog zu "Zeiger auf Variablen"
  - ➤ Adresse einer Struktur mit &-Operator zu bestimmen
  - ➤ Name eines Feldes von Strukturen = Zeiger auf erste Struktur im Feld
  - ➤ Zeigerarithmetik berücksichtigt Strukturgröße
- Beispiele

```
struct student stud1;
struct student gruppe8[35];
struct student *pstud;
pstud = &stud1;          /* ⇒ pstud → stud1 */
pstud = gruppe8;         /* ⇒ pstud → gruppe8[0] */
pstud++;                 /* ⇒ pstud → gruppe8[1] */
pstud += 12;             /* ⇒ pstud → gruppe8[13] */
```

- Besondere Bedeutung zum Aufbau
  - ➥ rekursiver Strukturen

7 Zeiger auf Strukturen (2)

- Zugriff auf Strukturkomponenten über einen Zeiger
- Bekannte Vorgehensweise
  - ➤ *-Operator liefert die Struktur
  - ➤ .-Operator zum Zugriff auf Komponente
  - ➤ Operatorenvorrang beachten

➥
```
(*pstud).best = 'n';
```
unleserlich!

- Syntaktische Verschönerung
  - ➥ ->-Operator

```
pstud->best = 'n';
```


8 Felder von Strukturen

- Von Strukturen können — wie von normale Datentypen — Felder gebildet werden
- Beispiel

```
struct student gruppe8[35];
int i;
for (i=0; i<35; i++) {
    printf("Nachname %d. Stud.: ", i);
    scanf("%s", gruppe8[i].nachname);
    …
    gruppe8[i].gruppe = 8;

    if (gruppe8[i].matrnr < 1500000) {
        gruppe8[i].best = 'y';
    } else {
        gruppe8[i].best = 'n';
    }
}
```


9 Strukturen in Strukturen

- Die Komponenten einer Struktur können wieder Strukturen sein
- Beispiel

```
struct name {
    char nachname[25];
    char vorname[25];
};

struct student {                    struct prof {
    struct name name;                   struct name pname;
    char gebdatum[11];                  char gebdatum[11];
    int matrnr;                         int lehrstuhlnr;
    short gruppe;                   }
    char best;
}

struct student stud1;
strcpy(stud1.name.nachname, "Meier");
if (stud1.name.nachname[0] == 'M') {
    …
```

10 Rekursive Strukturen

- Strukturen in Strukturen sind erlaubt — aber
  - ◆ die Größe einer Struktur muss vom Compiler ausgerechnet werden können
    - ➤ Problem: eine Struktur enthält sich selbst

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

falsch!

  - ◆ 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;
};
```

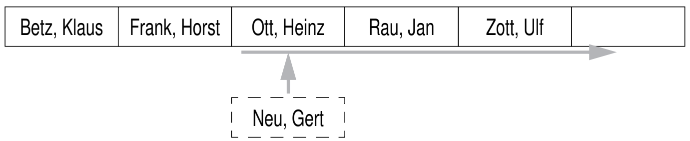
➥ Programmieren rekursiver Datenstrukturen

10 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

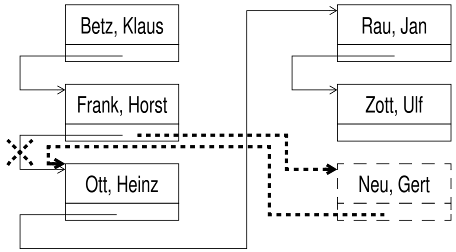


10 Rekursive Strukturen (3)

- 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



10 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));
}
```

F.14 Verbundstrukturen — Unions

- In einer Struktur liegen die einzelnen Komponenten hintereinander, in einem Verbund liegen sie übereinander
  - ➤ die gleichen Speicherzellen können unterschiedlich angesprochen werden
  - ➤ Beispiel: ein int-Wert (4 Byte) und die einzelnen Bytes des int-Werts

```
union intbytes {
    int ivalue;
    char bvalue[4];
} u1;
...
u1.ivalue = 259000;
printf("Wert=%d, Byte0=%d, Byte1=%d, Byte2=%d, Byte3=%d\n",
   u1.ivalue, u1.bvalue[0], u1.bvalue[1], u1.bvalue[2], u1.bvalue[3]);
```

- Einsatz nur in sehr speziellen Fällen sinnvoll
  konkretes Wissen über die Speicherorganisation unbedingt erforderlich!

F.15 Bitfelder

- Bitfelder sind Strukturkomponenten bei denen die Zahl der für die Speicherung verwendeten Bits festgelegt werden kann.
- Anwendung z. B. um auf einzelnen Bits eines Registers zuzugreifen

```
struct cregister {
    unsigned int protection : 1;
    unsigned int interrupt_mask : 3;
    unsigned int enable_read : 1;
    unsigned int enable_write : 1;
    unsigned int pedding : 2;
    unsigned int address : 8;
};
```


F.15 Bitfelder (2)

- Struktur mit Bitfeld kann ihrerseits Teil einer Union sein
  - ➤ Zugriff auf Register als Ganzes und auf die einzelnen Bits

```
union cregister {
    unsigned short all;
    struct bits {
        unsigned int protection : 1;
        unsigned int interrupt_mask : 3;
        unsigned int enable_read : 1;
        unsigned int enable_write : 1;
        unsigned int pedding : 2;
        unsigned int address : 8;
    };
};
```

- Adresse und Aufbau eines Registers steht üblicherweise in der Hardwarebeschreibung.

F.15 Bitfelder (3)

- Beispiel:
  - ➤ Adresse auf Register anlegen,
    Registerinhalt sichern
    Bits verändern
    Registerinhalt wieder herstellen

```
union cregister *creg;
unsigned short oldvale;
creg = (union cregister *)0x2400; /* Addr. aus Manual */
oldvalue = creg->all;             /* Wert sichern */
creg->bits.protection = 0;
creg->bits.enable_read = 1;
creg->bits.address = 0x40;
...
creg->all = oldvalue;             /* Wert restaurieren */
```