U3 3. Übung

- Besprechung Aufgabe 1
- Pointer
- Register und Ports

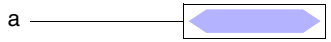
U3-1 Zeiger

1 Einordnung

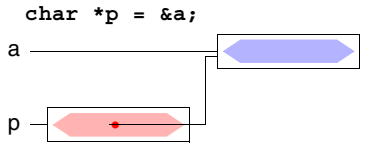
- **Konstante:**

'a' ≡ 0110 0001

- **Variable:**



- **Zeiger-Variable (Pointer):**



2 Definition von Zeigervariablen

- Syntax:

```
Typ *Name ;
```

3 Adressoperatoren

- Adressoperator `&`

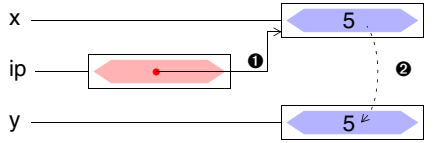
  `&x` Referenz

- Verweisoperator `*`

  `*x` Dereferenzierung

4 Beispiele

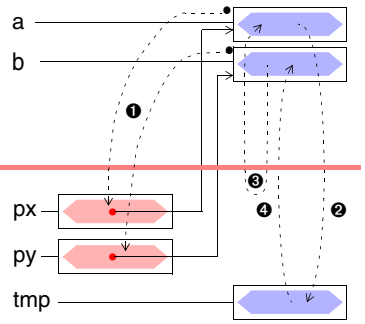
```
int x = 5;

int *ip;

int y;

ip = &x; ❶

y = *ip; ❷
```



5 Zeiger als Funktionsargumente (2)

- Beispiel:

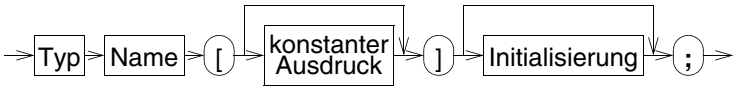
```
void swap (int *, int *);
main(void) {
    int a, b;
    ...
    swap(&a, &b); ❶
}

void swap (int *px, int *py)
{
    int tmp;

    tmp = *px; ❷
    *px = *py; ❸
    *py = tmp; ❹

}
```



U3-2 Felder

1 Eindimensionale Felder

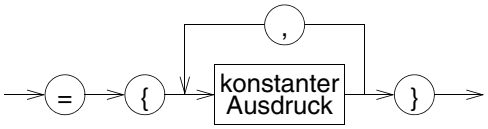
- Gleicher Typ
- Anzahl kann nicht mehr geändert werden
- der Zugriff durch **Index,** beginnend bei Null
- Definition eines Feldes



- Beispiele:

```
int x[5];
double f[20];
```


2 Initialisierung eines Feldes



- Beispiel

```
int prim[4] = {2, 3, 5, 7};
char name[5] = {'O', 't', 't', 'o', '\0'};
```

◆ Nicht initialisierte Elemente werden mit 0 initialisiert.

- Automatische Größe

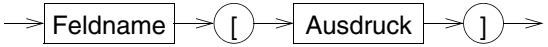
```
int prim[] = {2, 3, 5, 7};
char name[] = {'O', 't', 't', 'o', '\0'};
```

- ***char:*** Initialisierung durch Strings.

```
char name1[5] = "Otto"; // {'O', 't', 't', 'o', '\0'};
char name2[] = "Otto";
```


3 Zugriffe auf Feldelemente

- Indizierung:



wobei: 0 <= Wert(Ausdruck) < Feldgröße

- Beispiele:

```
prim[0] == 2
prim[1] == 3
name[1] == 't'
name[4] == '\0'
```

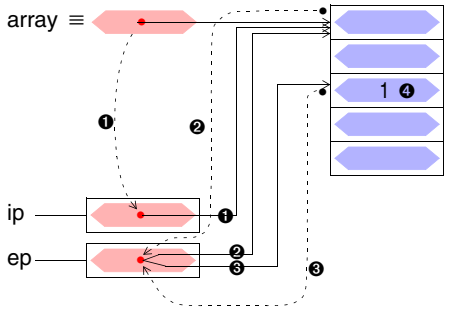
- Beispiel Vektoraddition:

```
float v1[4], v2[4], sum[4];
int i;
...
for ( i=0; i < 4; i++ )
    sum[i] = v1[i] + v2[i];
for ( i=0; i < 4; i++ )
    printf("sum[%d] = %f\n", i, sum[i]);
```

U3-3 Zeiger und Felder

- Variable zeigt immer auf das erste Feld
- Wert kann nicht geändert werden
- es gilt:

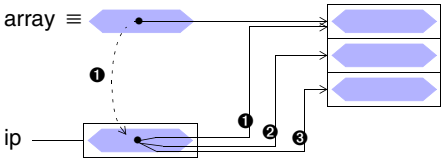
```
int array[5];

int *ip = array; ❶

int *ep;
ep = &array[0]; ❷

ep = &array[2]; ❸

*ep = 1; ❹
```



1 Arithmetik mit Adressen

- `++` -Operator: Inkrement = nächstes Objekt

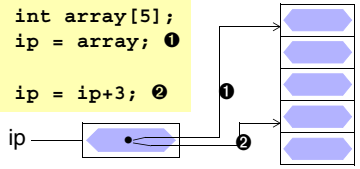
```
int array[3];
int *ip = array; ❶

ip++; ❷
ip++; ❸
```



- `--` -Operator: Dekrement = vorheriges Objekt
- `+, -`

Größe des Objekttyps berücksichtigt!

```
int array[5];
ip = array; ❶

ip = ip+3; ❷
```

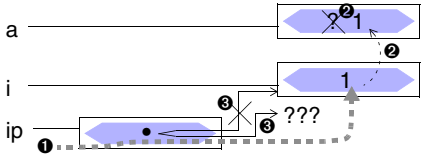


2 Vorrangregeln bei Operatoren

Operatorklasse	Operatoren	Assoziativität
primär	**( ) Funktionsaufruf [ ]**	von links nach rechts
unär	**! ~ ++ -- + – * &**	von rechts nach links
multiplikativ	*** / %**	von links nach rechts
…		

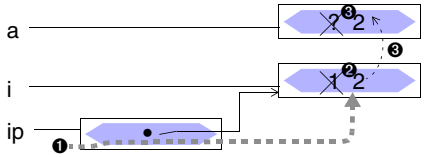
3 Beispiele

```
int a, i, *ip;
i = 1;
ip = &i;

a = *ip++;
(1) a = *ip++;
(2) a = *ip++;
```

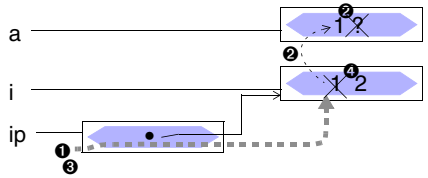


3 Beispiele (2)

```
int a, i, *ip;
i = 1;
ip = &i;

a = ++*ip;
(1) a = ++*ip;
(2) a = ++*ip;
```




```
int a, i, *ip;
i = 1;
ip = &i;

a = (*ip)++;
(1) a = (*ip)++;
(2) a = (*ip)++;
```



4 Zeigerarithmetik und Felder

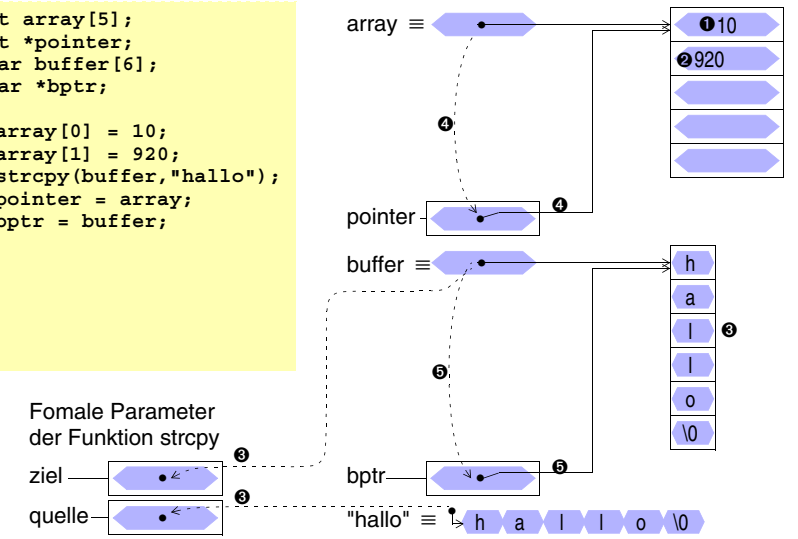
- Ein Feldname ist eine Konstante, für die Adresse des Feldanfangs
  - ➥ Feldname ist ein ganz normaler Zeiger
    - ➤ Operatoren für Zeiger anwendbar (`*`, `[]`)
  - ➥ aber keine Variable ➔ keine Modifikationen erlaubt
    - ➤ keine Zuweisung, kein `++`, `--`, `+=`, …

- es gilt:

```
int array[5]; /* → array ist Konstante für den Wert &array[0] */
int *ip = array;/* ≡ int *ip = &array[0] */
int *ep;

/* Folgende Zuweisungen sind äquivalent */
array[i] = 1;
ip[i] = 1;
*(ip+i) = 1;        /* Vorrang ! */
*(array+i) = 1;

ep = &array[i]; *ep = 1;
ep = array+i; *ep = 1;
```

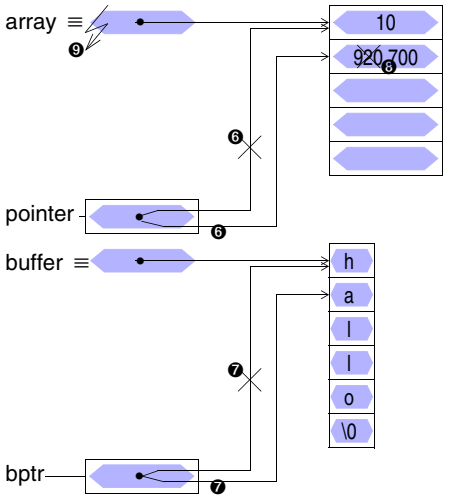

4 Zeigerarithmetik und Felder

```
int array[5];
int *pointer;
char buffer[6];
char *bptr;

❶ array[0] = 10;
❷ array[1] = 920;
❸ strcpy(buffer,"hallo");
❹ pointer = array;
❺ bptr = buffer;
```



4 Zeigerarithmetik und Felder

```
int array[5];
int *pointer;
char buffer[6];
char *bptr;

❶ array[0] = 10;
❷ array[1] = 920;
❸ strcpy(buffer,"hallo");
❹ pointer = array;
❺ bptr = buffer;

❻ pointer++;
❼ bptr++;
❽ *pointer = 700;

❾ array++;
```



5 Vergleichsoperatoren und Adressen

- Vergleich von Zeigern

`<`	kleiner
`<=`	kleiner gleich
`>`	größer
`>=`	größer gleich
`==`	gleich
`!=`	ungleich

U3-4 Eindimensionale Felder als Funktionsparameter

- ganze Felder können in C **nicht *by-value*** übergeben werden
- Verwendung von Zeigern
- bei der Deklaration des formalen Parameters wird die Feldgröße weggelassen
  - ➤ Größe "bekannt"
  - ➤ ggf. Größe mit einem extra Parameter übergeben
  - ➤ Strings enden mit \0

U3-4 Eindimensionale Felder als Funktionsparameter (2)

- Beispiel:

```
int func(int p1, int p2[], int p3);
oder:
int func(int p1, int *p2, int p3);
…
int a, b;
int feld[20];
func(a, feld, b);
```

- die Parameter-Deklarationen `int p2[]` und `int *p2` sind vollkommen äquivalent!

U3-5 Dynamische Speicherverwaltung

- Feldgröße nicht änderbar
- Lösung: `malloc()`
  - ➤ Zeiger auf den Anfang des Speicherbereichs
  - ➤ Zeiger kann danach wie ein Feld verwendet werden (`[]`-Operator)
- `void *malloc(size_t size)`

```
int *feld;
int groesse;
...
feld = (int *) malloc(groesse * sizeof(int));
if (feld == NULL) {
       perror("malloc feld");
       exit(1);
}
for (i=0; i<groesse; i++) { feld[i] = 8; }
...
```

cast-Operator

sizeof-Operator

U3-5 Dynamische Speicherverwaltung (2)

- Speicher muss wieder frei gegeben werden
- `void free(void *ptr)`

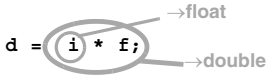
```
double *dfeld;
int groesse;
...
dfeld = (double *) malloc(groesse * sizeof(double));
...
free(dfeld);
```

U3-6 Explizite Typumwandlung — Cast-Operator

- C kann automatisch umwandeln (vgl. Abschnitt D.5.10)

Beispiel:

```
int i = 5;
float f = 0.2;
double d;
```



- Aber nicht immer (so wie man will)
  - ◆ Syntax:

(Typ) Variable

Beispiele:

```
(int) a
(float) b
```

```
(int *) a
(char *) a
```

  - ◆ Beispiel:

```
feld = (int *) malloc(groesse * sizeof(int));
```

malloc liefert Ergebnis vom Typ (void *)

cast-Operator macht daraus den Typ (int *)

U3-7 sizeof-Operator

- Bestimmung der Größe einer Variablen / Stuktur
- Syntax:

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

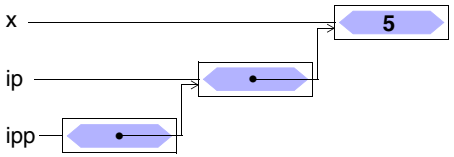
- Das Ergebnis ist vom Typ `size_t` (≡ `int`) (`#include <stddef.h>`!)
- Beispiel:

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


U3-8 Zeiger auf Zeiger

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

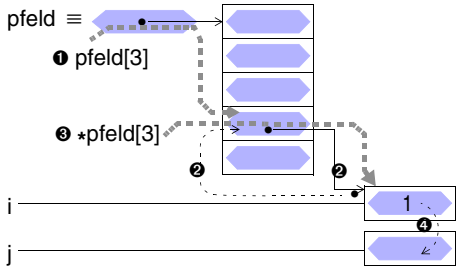
```
int x = 5;
int *ip = &x;

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



U3-9 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
❶
```

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

```
j = *pfeld[3]; ❹
      ❶
❹     ❸
```

U3-10 Zeiger auf Funktionen

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

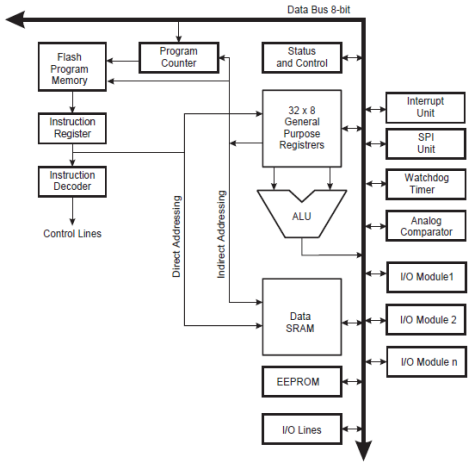
```
int (*fptr)(int, char*);

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

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

fptr = test1;

fptr(42,"hallo"); // "1: 42 hallo\n"

fptr = test2;

fptr(42,"hallo"); // "2: hallo 42\n"
```


U3-11 Register beim AVR-µC

- Beim AVR-µC sind die Register
  - in den Speicher eingebettet
  - am Anfang des Adressbereichs angeordnet
- Adressen sind der Dokumentation zu entnehmen
- vollständige Dokumentation für "unseren" Mikrokontroller ATMega32:
  `http://www4.informatik.uni-erlangen.de/Lehre/WS09/V_SPIC/Uebung/doc/mega32.pdf`
- Für die Aufgaben benötigte Register sind auf den Folien erwähnt
- Die Bibliothek (avr-libc), die wir verwenden, definiert bereits sinnvolle Makros für alle Register des AVR µC
  (`#include <avr/io.h>`)

1 Funktionsweise von Registern

- Register sind über den Daten-Bus angebunden
- Jedes Register hat eine vorgegebene Speicheradresse. Es kann wir "normaler" Speicher gelesen und geschrieben werden



2 Makros für Register-Zugriffe

- Makros mit aussagekräftigen Namen können den Umgang mit Registern deutlich vereinfachen
- Beispiel:
  - Makro für Register an Adresse 0x3b (PORTA beim ATmega32):

```
#define PORTA (*(volatile uint8_t *)0x3b)
```

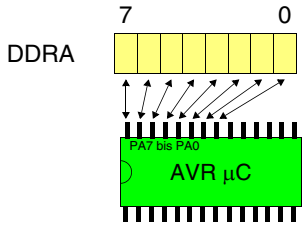
  - Verwenden dieses Registers:

```
volatile uint8_t *portPtr = &PORTA;
PORTA = 0;              /* schreibender Zugriff */
...
if (PORTA == 0x04)      /* lesender Zugriff */
    PORTA &= ~4;        /* lesender und schreibender Zugriff */
*portPtr |= 1;          /* Zugriff über Zeiger */
```

- Das `volatile`-Schlüsselwort wird später erläutert, im Moment ist es bei sämtlichen Zugriffen auf Hardwareregister zu verwenden.

U3-12 I/O-Ports des AVR-µC

- Jeder I/O-Port des AVR-µC wird durch drei 8-bit Register gesteuert:
  - Datenrichtungsregister (`DDRx` = data direction register)
  - Datenregister (`PORTx`)
  - Port Eingabe Register (`PINx` = port input register, nur-lesbar)
- Jedem Anschluss-Pin ist ein Bit in jedem der 3 Register zugeordnet
  - ➤ Beispiel: DDR von Port A:



1 I/O-Port-Register

- **`DDRx`**: hier konfiguriert man einen Pin *i* von Port *x* als Ein- oder Ausgang
  - Bit *i* = 1→ Pin *i* als **Ausgang** verwenden
  - Bit *i* = 0→ Pin *i* als **Eingang** verwenden
- **`PORTx`**: Auswirkung abhängig von `DDRx`:
  - ist Pin *i* als **Ausgang** konfiguriert, so steuert Bit *i* im `PORTx` Register ob am Pin *i* ein high- oder ein low-Pegel erzeugt werden soll
    - Bit *i* = 1→ high-Pegel an Pin *i*
    - Bit *i* = 0→ low-Pegel an Pin *i*
  - ist Pin *i* als **Eingang** konfiguriert, so kann man einen internen pull-up-Widerstand aktivieren
    - Bit *i* = 1→ pull-up-Widerstand an Pin *i* (Pegel wird auf high gezogen)
    - Bit *i* = 0→ Pin *i* als tri-state konfiguriert
- **`PINx`**: Bit *i* gibt den aktuellen Wert des Pin *i* von Port *x* an (nur lesbar)

2 Beispiel: Initialisierung eines Ports

- Pin 3 von Port B (PB3) als Ausgang konfigurieren und auf V_{CC} schalten:

```
DDRB |= 0x08;  /* PB3 als Ausgang nutzen... */
PORTB |= 0x08; /* ...und auf 1 (=high) setzen */
```

- Pin 0 von Port D (PD0) als Eingang nutzen, pull-up-Widerstand aktivieren und prüfen ob ein low-Pegel anliegt:

```
DDRD &= ~0x01; /* PD0 als Eingang nutzen... */
PORTD|= 0x01;  /*...und den pull-up-Widerstand aktivieren*/

if ( (PIND & 0x01) == 0) { /* den Zustand auslesen */
    /* ein low Pegel liegt an, der Taster ist gedrückt */
}
```

- Die Initialisierung der Hardware wird in der Regel **einmalig** zum Programmstart durchgeführt

U3-13 Lebensdauer von Variablen

- Die Lebensdauer einer Variablen bestimmt, wie lange der Speicherplatz für die Variable aufgehoben wird
- Zwei Arten
  - statische (**`static`**) Variablen
    - ➤ Speicherplatz bleibt für die gesamte Programmausführungszeit reserviert
  - dynamische (**`automatic`**) Variablen
    - ➤ Speicherplatz wird bei Betreten eines Blocks reserviert und danach wieder freigegeben

U3-13 Lebensdauer von Variablen (2)

auto-Variablen

- Alle lokalen Variablen sind automatic-Variablen
  - ➤ der Speicher wird bei Betreten des Blocks / der Funktion reserviert und bei Verlassen wieder freigegeben
    - ➥ der Wert einer lokalen Variablen ist beim nächsten Betreten des Blocks nicht mehr sicher verfügbar!
- Lokale auto-Variablen können durch beliebige Ausdrücke initialisiert werden
  - ➤ die Initialisierung wird bei jedem Eintritt in den Block wiederholt

!!! **wird eine auto-Variable nicht initialisiert, ist ihr Wert vor der ersten Zuweisung undefiniert** (= irgendwas)