Teil 5: Zeiger

Gliederung

Mehrdimensionale Felder

Zeiger und Adressen

Zeigerarithmetik

Mehrdimensionale Felder

Speicherabbild

```
int alpha[5];
```

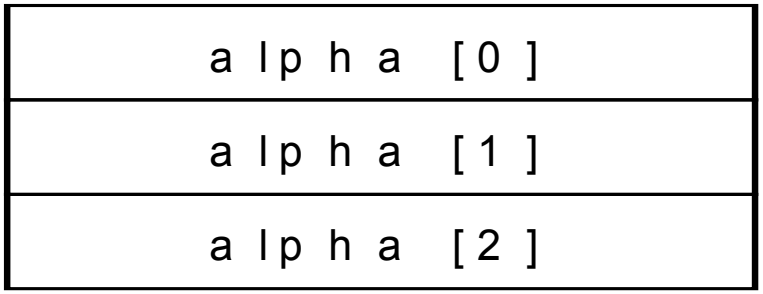
alpha[0] int	alpha[1] int	alpha[2] int	alpha[3] int	alpha[4] int
?	?	?	?	?

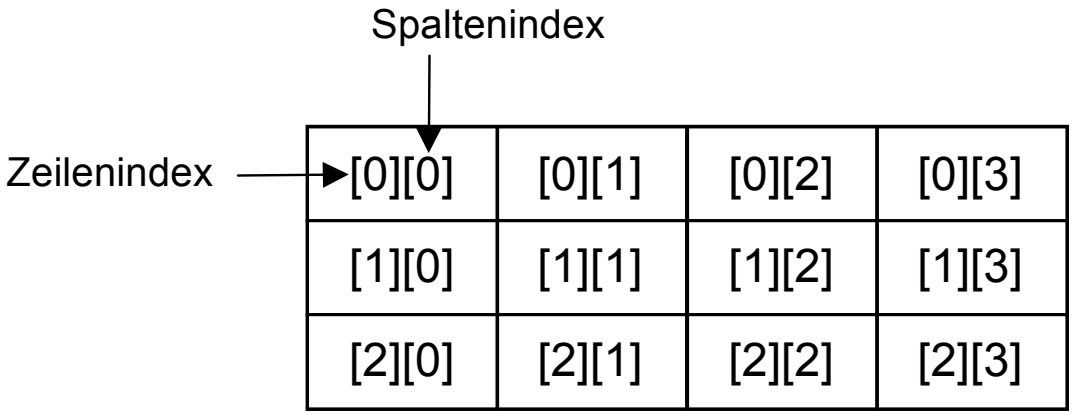
```
for (int i = 0; i < 5; i++)
    alpha[i] = i * i;
```

alpha[0] int	alpha[1] int	alpha[2] int	alpha[3] int	alpha[4] int
0	1	4	9	16

Mehrdimensionale Felder

```
int alpha[3][4];
```





Allgemein kann ein n-dimensionales Array als ein eindimensionales Array, dessen Komponenten (n-1)-dimensionale Arrays sind, interpretiert werden.

Mehrdimensionale Felder: Initialisierung

```
int alpha[3][4] = {
                     {1, 3, 5, 7},
                     {2, 4, 6, 8},
                     {3, 5, 7, 9},
                  };
```

1	3	5	7
2	4	6	8
3	5	7	9

Felder als Parameter (1)

```
#define SIZE 10
void ausgabe(int[], int);

int main()
{
    int meinFeld[SIZE] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};

    ausgabe(meinFeld, SIZE);

    return 0;
}

void ausgabe(int feld[], int laenge)
{
    for (int i = 0; i < laenge; i++)
        printf("%d\n", feld[i]);
}
```

Felder als Parameter (1)

```
#define SIZE 10
void ausgabe(int[], int);

int main()
{
    int meinFeld[SIZE] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};

    ausgabe(meinFeld, SIZE);

    return 0;
}

void ausgabe(int feld[], int laenge)
{
    for (int i = 0; i < laenge; i++)
        printf("%d\n", feld[i]);
}
```

call by reference – call by value

```
#define SIZE 10
void erhoehe(int feld[], int laenge);

int main()
{
    int meinFeld[SIZE] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};

    ausgabe(meinFeld, SIZE);
    erhoehe(meinFeld, SIZE);
    ausgabe(meinFeld, SIZE);

    return 0;
}

void erhoehe(int feld[], int laenge)
{
    int i;

    for (i = 0; i < laenge; i++)
        feld[i] = feld[i] + 1;
}
```

Mehrdimensionale Felder als Parameter

```
#define ZEILEN 3
#define SPALTEN 4
void ausgabeMatrix(int[][SPALTEN], int);

int main()
{
    int matrix[ZEILEN][SPALTEN] = { { 1,  2,  3,  4 },
                                    { 5,  6,  7,  8 },
                                    { 9, 10, 11, 12 } };

    ausgabeMatrix(matrix, ZEILEN);

    return 0;
}

void ausgabeMatrix(int matrix[][SPALTEN], int zeilen)
{
    int i, j;

    for (i = 0; i < zeilen; i++ )
        for (j = 0; j < SPALTEN; j++)
            printf("%d ", matrix[i][j];
}
```

Mehrdimensionale Felder: Strings

- Einzelner String:

```
char wort[] = "Hallo";
```

- Mehrere Strings in einem Feld:

```
char woerter[6][15] = { "eins", "zehn", "hundert", "tausend",
                        "zehntausend", "hunderttausend" };
```

```
printf("%s", woerter[0]);
printf("%c", woerter[2][1]);
```

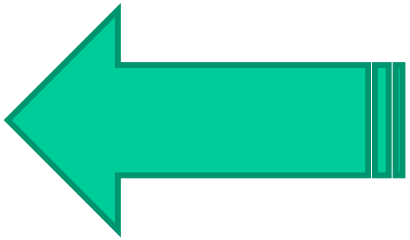
Zeiger und Adressen

Funktion mit mehreren Rückgabewerten?

Problem: Eine Funktion **`swap(...)`** soll die Inhalte 2er Variablen vertauschen.
Wie kann dieses Problem gelöst werden?

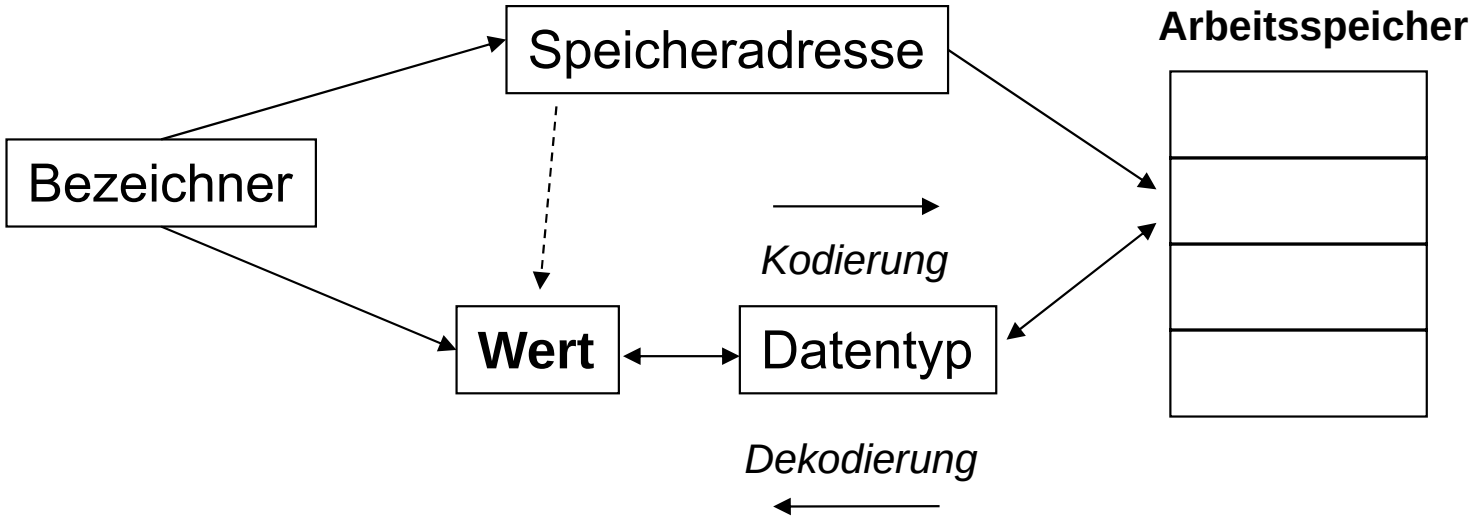
Beispiel:

```
int x = 1, y = 2;

// tausche x und y

...
```

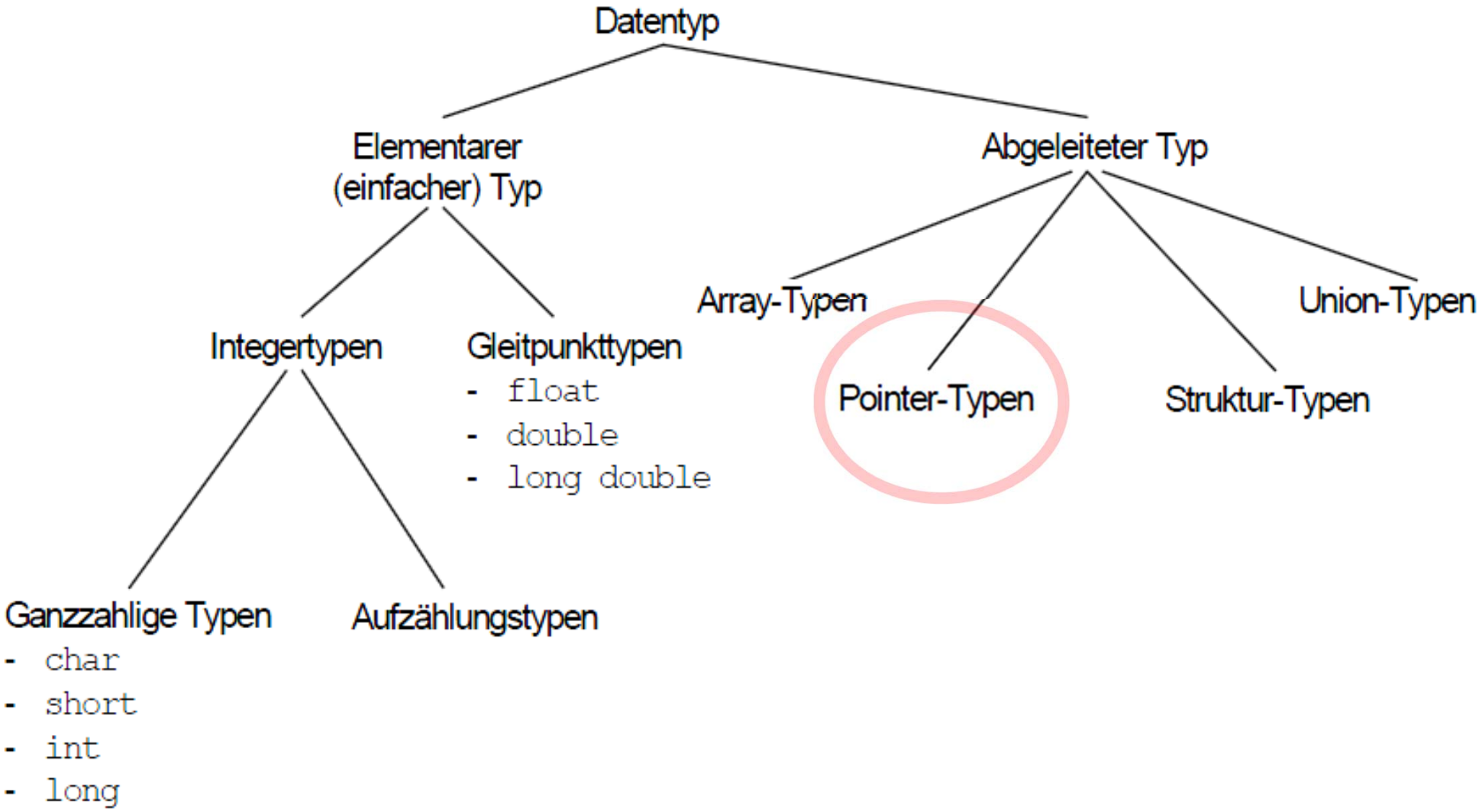


```
int temp;

temp = x;
x = y;
y = temp;
```

Variablenmodell

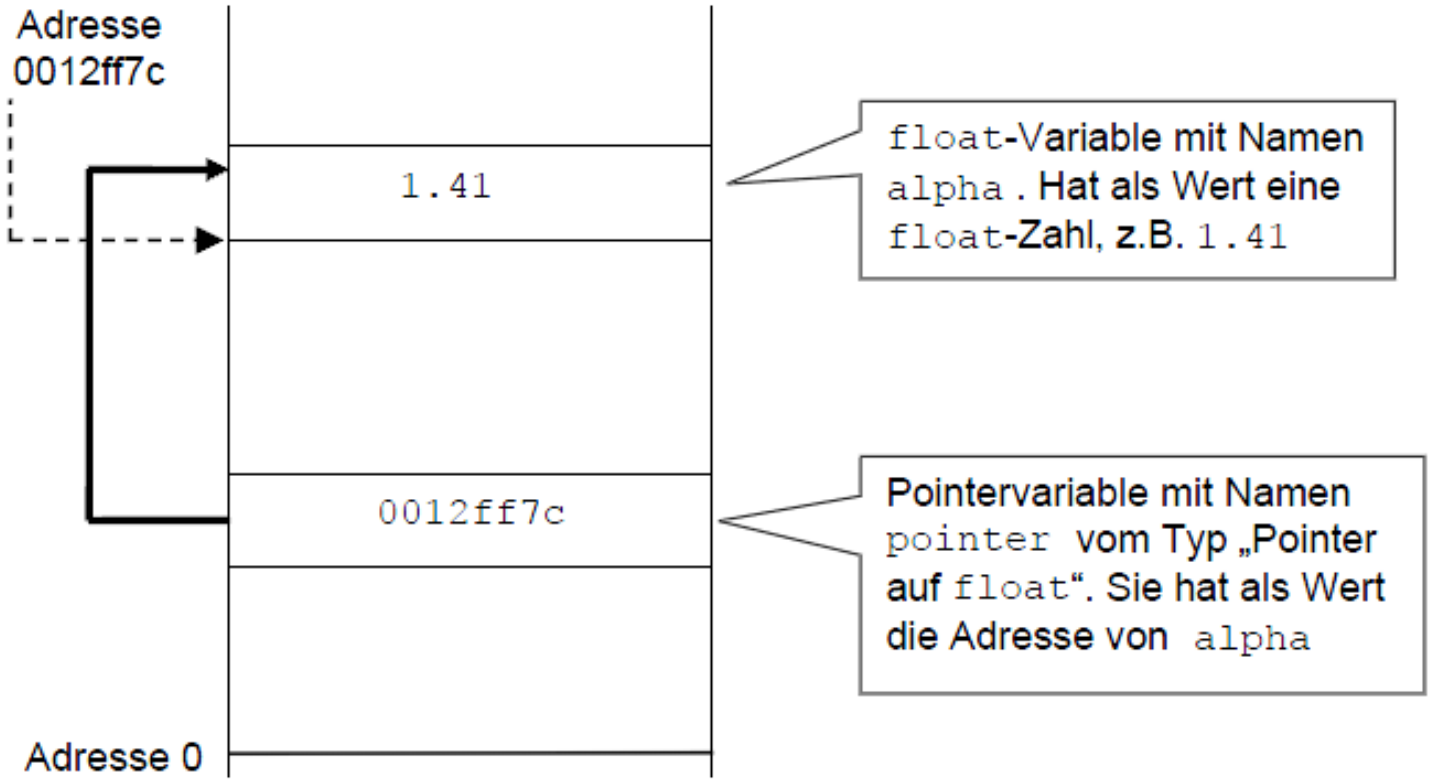


Typ-Klassifikation



Was ist ein Zeiger?

Ein **Zeiger (Pointer)** ist eine Variable, welche die **Adresse einer im Speicher befindlichen *Variablen* oder *Funktion*** aufnehmen kann. Damit verweist eine Zeigervariable mit ihrem Variablenwert auf die jeweilige Adresse.



Deklaration / Definition eines Zeigers

- Zeiger sind typgebunden

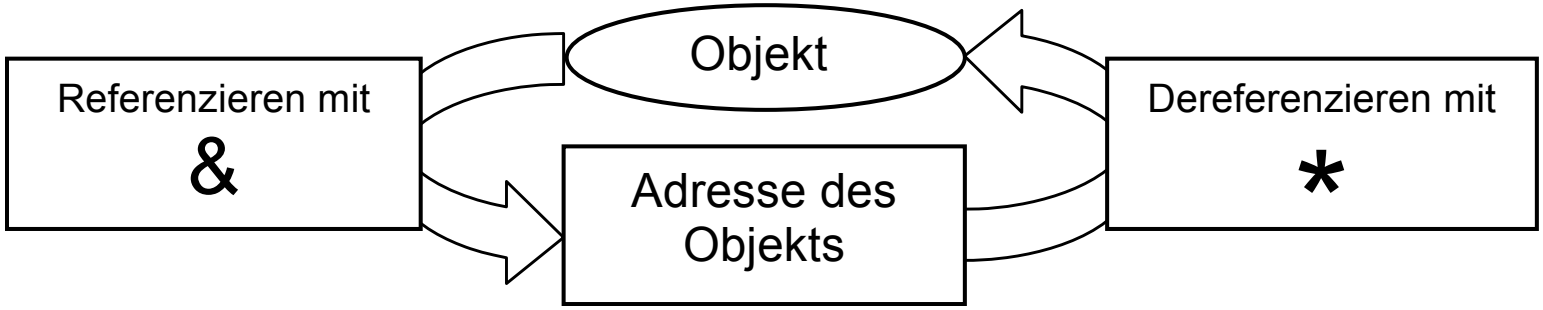
```
int *pointerInt;              // Zeiger auf int

double *pointerDouble;        // Zeiger auf double
```

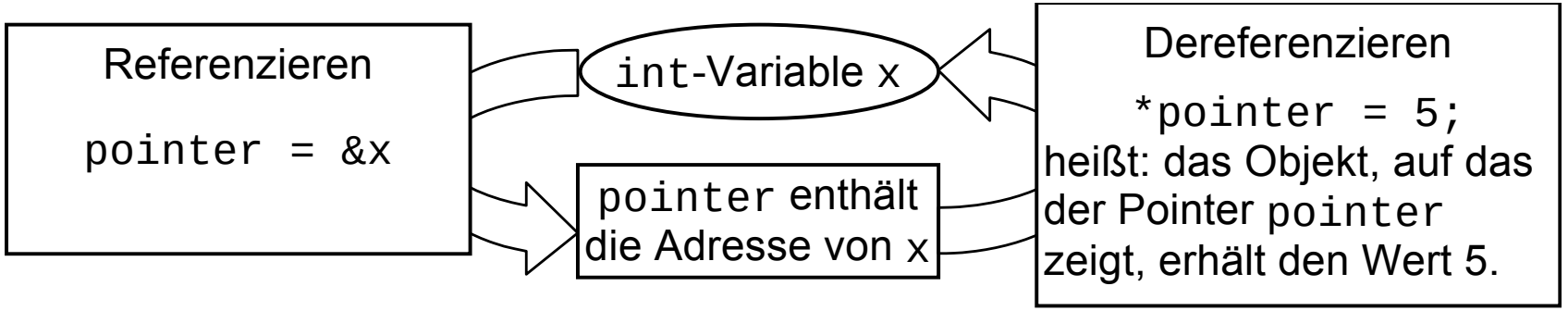
- Besonderheiten bei kombinierten Deklarationen

Deklaration	Entspricht
`int * pointer, alpha;`	`int * pointer;` `int alpha;`
`int * pointer1, * pointer2;`	`int * pointer1; int * pointer2;`

Referenzieren und Dereferenzieren

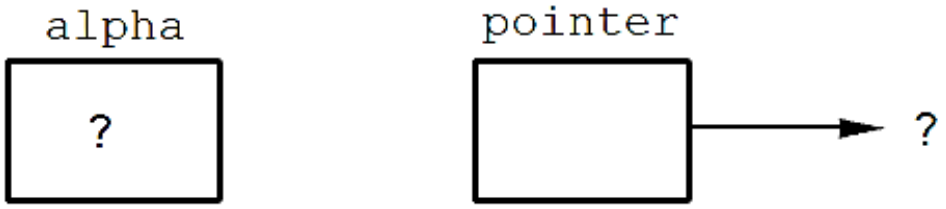
- **Referenzieren**: Adressoperator **&** liefert **Adresse** eines Objekts
- **Dereferenzieren**: Dereferenzierungsoperator ***** liefert **Inhalt** an der Adresse



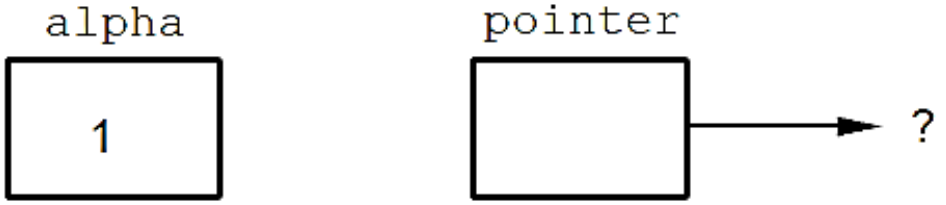


Adresszuweisung an einen Zeiger I

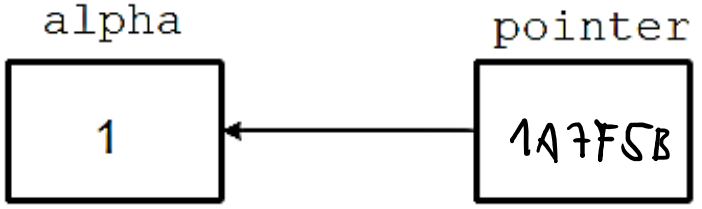
```
int alpha;
int * pointer;
```




```
alpha = 1;
```

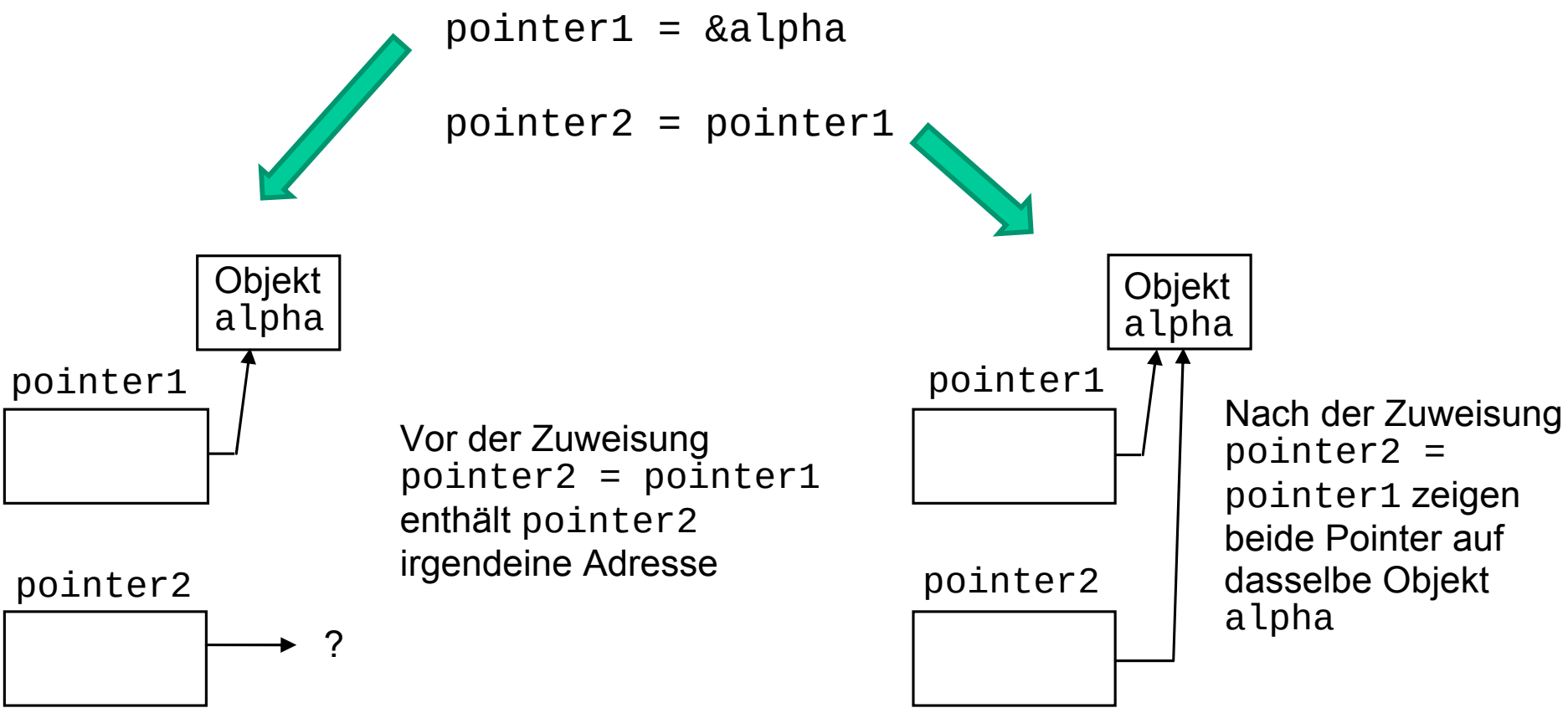



```
pointer = &alpha;
```



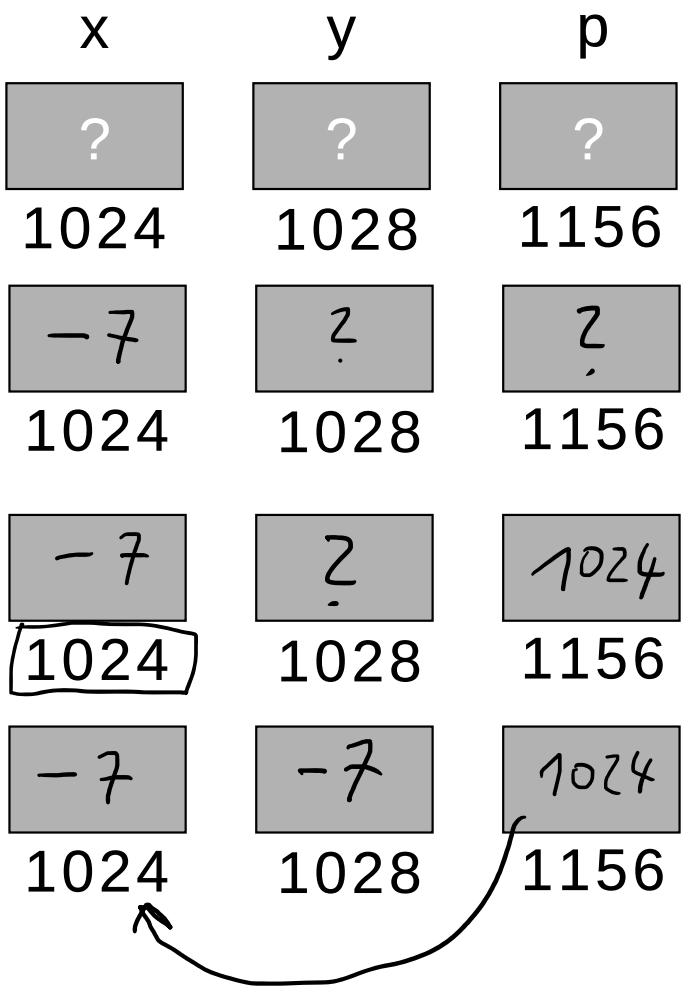

```
*pointer = 2;
```

Adresszuweisung an einen Zeiger II



Beispiel 1: Zeigeroperationen


```
int x, y, *p;
```



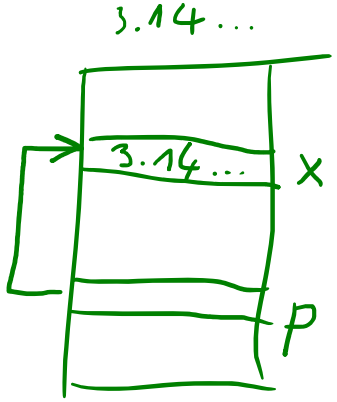

```
→ x = -7;
```



```
→ p = &x;
```



```
y = *p;
```



Beispiel 2: Zeigeroperationen

Was ist an den folgenden Anweisungen a bis e falsch?

```
float x, y, *p;
```

a) 1. → `p = &x;` 2. → `*p = 3.14159;` // Dereferenzierung einer ungültigen Adresse

b) `p = &3.14159;` // Literale haben keine Adresse

c) `*x = 5.4;` // Dereferenzierung eines Nicht-Zeigers

d) `x = &y;` // Adresse in Nicht-Zeiger ablegen

e) `p = *y;` // Dereferenzierung eines Nicht-Zeigers
// Zuweisung einer Nicht-Adresse an Zeiger

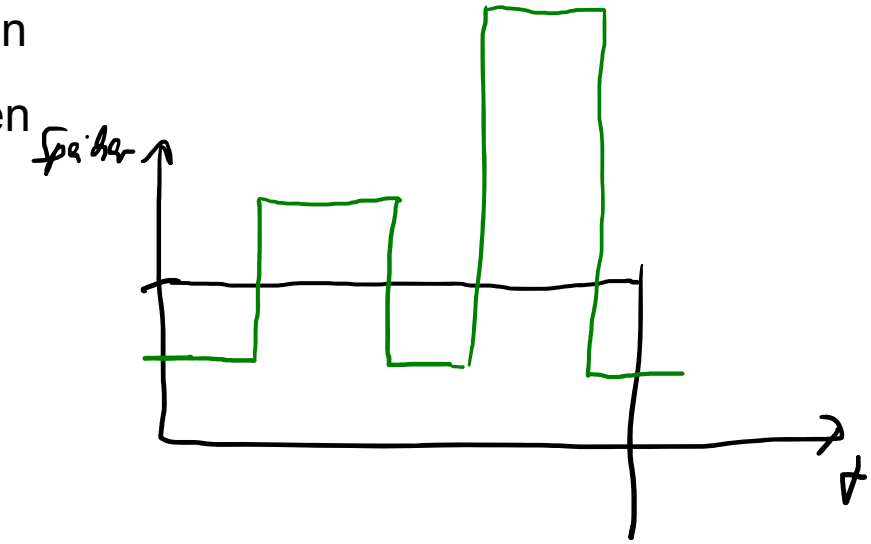
Funktion mit mehreren Rückgabewerten?

Problem: Eine Funktion **`swap(...)`** soll die Inhalte 2er Variablen vertauschen.
Wie kann dieses Problem gelöst werden?

```
void swap( int* , int* );

int main()
{
  int x = 1, y = 2;

  // tausche die Inhalte von x und y
  swap( &x , &y );        // call by reference

  return 0;
}

void swap( int *a , int *b )
{
  int temp;
  temp = *a;
  *a = *b;
  *b = temp;
}

// TODO: Do the magic here!
```

Warum Zeiger?

- "call by reference" (vs. "call by value")
  - mehrere Rückgabewerte bei Funktionsaufrufen
  - Beschleunigung von Funktionsaufrufen (warum?)
- erlauben direkte Speicher-Manipulationen
- dynamisch Speicher anfordern / freigeben
- Umgang mit Arrays und Strings
- **Nachteil**: Hohe Gefahr von Fehlern



Zeigerarithmetik

Dualität von Zeigern und Feldern I

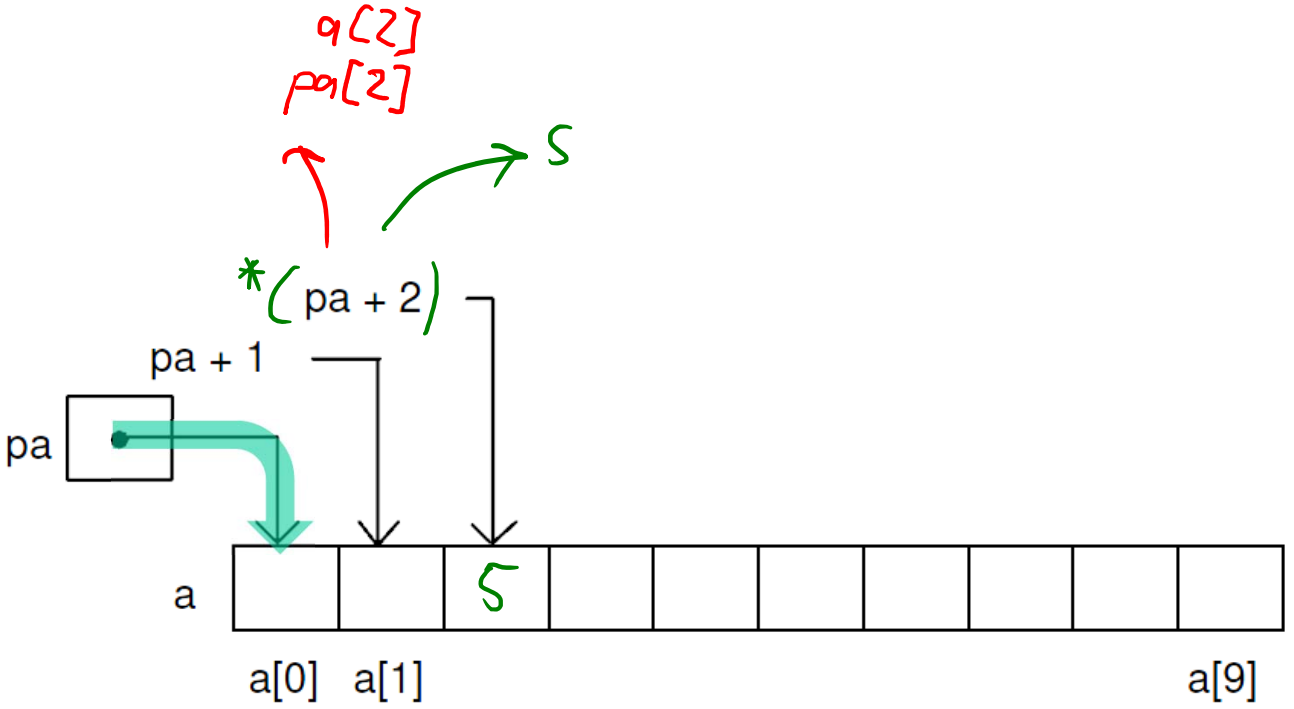
- Der Bezeichner eines Arrays referenziert (wie ein Zeiger) eine Adresse:

  Ein Zugriff `a[i]`
  wird vom Compiler als `*(a + i)` behandelt.

→ Elemente eines Arrays können über Zeiger referenziert werden:

- Beispiel:

```
int a[10];
int *pa;

pa = a;
```



Dualität von Zeigern und Feldern II

- Als Funktionsparameter werden Arrays und Zeiger äquivalent behandelt:

```
void strcpy(char destination[], const char source[]);

void strcpy(char *destination, const char *source);
```

- Sei a ein beliebiges Feld, dann ist

Wertebene `a[i]` identisch zu `*(a + i)`

Adressebene `&a[i]` identisch zu `a + i`

- Sei pa eine beliebige Zeigervariable, dann ist

`pa[i]` identisch zu `*(pa + i)`

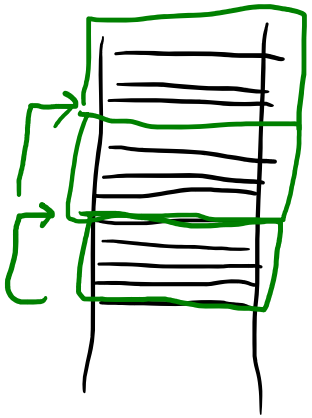
`&pa[i]` identisch zu `pa + i`

Zeigerarithmetik

Fall 1: Zeiger +/- int-Wert -> Zeiger

```
long int feld[5]
long int *zeiger;

zeiger = feld;
zeiger++;
zeiger += 3;
*zeiger = 123;
```

Zeigerarithmetik



Fall 1: Zeiger +/- int-Wert -> Zeiger

```
long int feld[5]
long int *zeiger;

zeiger = feld;              // Zeiger verweist auf Feld-Anfang
zeiger++;                   // Zeiger verweist nun auf 2. Element
zeiger += 3;                // Zeiger verweist nun auf 5. Element
*zeiger = 123;              // ist nun identisch mit feld[4] = 123;
```

Zeigerarithmetik

Fall 1: Zeiger +/- int-Wert -> Zeiger

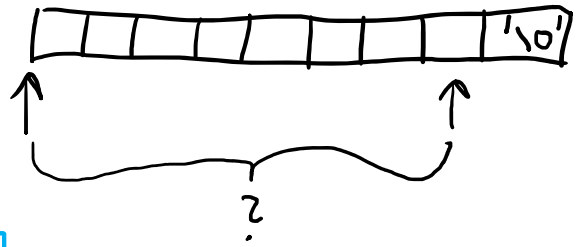
```
long int feld[5]
long int *zeiger;

zeiger = feld;              // Zeiger verweist auf Feld-Anfang
zeiger++;                   // Zeiger verweist nun auf 2. Element
zeiger += 3;                // Zeiger verweist nun auf 5. Element
*zeiger = 123;              // ist nun identisch mit feld[4] = 123;
```

Fall 2: Zeiger +/- Zeiger -> int-Wert

```
int strlen(char str[])
{
  char *start, *end;
  int i=0;
  while (str[i] != '\0')
    i++;

  return i ;
}
```

// Berechnet Länge eines Strings

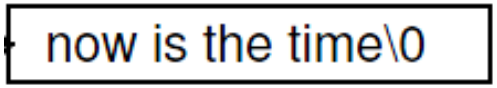
```
start = str;
end = str;
while (*end != '\0')
  end++;
return end-start;
```



Strings und String-Konstanten

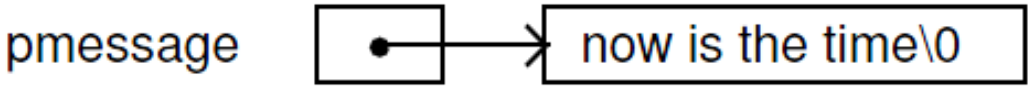
- Zeiger ist konstant, **String ist variabel**:

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



- Zeiger ist variabel, **String ist konstant**:

```
char *pmessage = "now is the time";
```



Beispiel 3: Parameter der main()-Funktion

- Funktionskopf der main()-Funktion ohne Parameter:

```
int main();        bzw.        int main(void);
```

- Kommandozeilenparameter am Beispiel des Shell-Kommandos "cp":

```
copy file_a.doc file_b.doc
```

`copy`	Name der Anwendung / des Programms
`file_a.doc`	1. Kommandozeilenparameter
`file_b.doc`	2. Kommandozeilenparameter

→ Verarbeitung der Parameter erfordert modifizierte `main()`-Funktion:

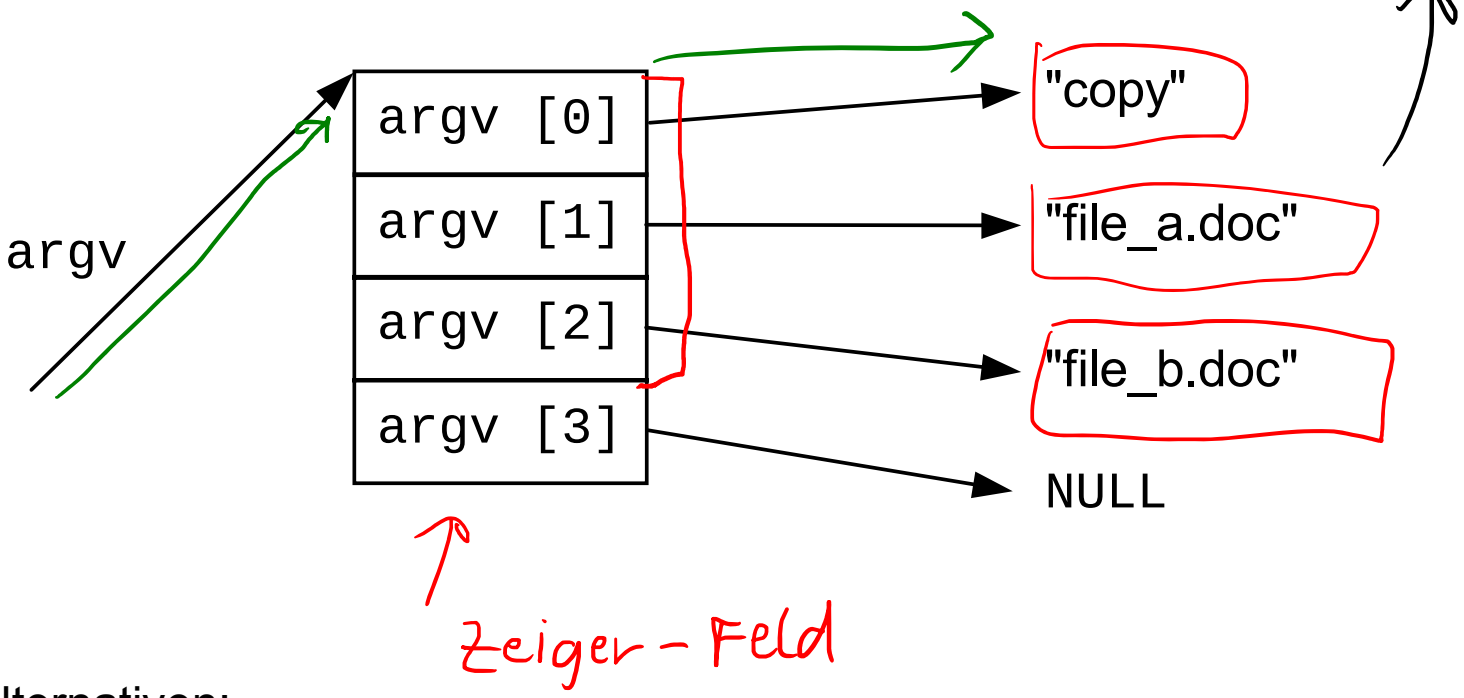
```
int main(int argc, char *argv[]);
```

`argc`	("argument count"), Anzahl der Kommandozeilenparameter + 1
`argv`	("argument value"), referenzieren der Reihe nach den Programmnamen, sowie die Kommandozeilenparameter

Beispiel 3: Parameter der main()-Funktion

- Beispiel:

```
C:\>copy file_a.doc file_b.doc
```



- Alternativen:

```
char *argv[]        oder   char **argv
```