Teil 8: Zeiger und Dynamische Speicherverwaltung

Gliederung

Zeiger und Adressen

Zeigerarithmetik

Dynamische Speicherverwaltung

Zeiger und Adressen

Funktion mit mehreren Rückgabewerten?

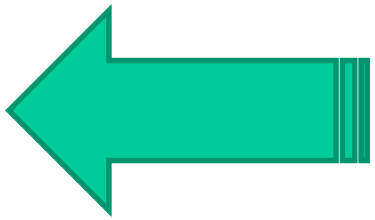
Problem: Eine Funktion **`swap(...)`** soll die Inhalte 2er Variablen vertauschen.
Wie kann dieses Problem gelöst werden?

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

// tausche x und y

...
```

Funktion mit mehreren Rückgabewerten?

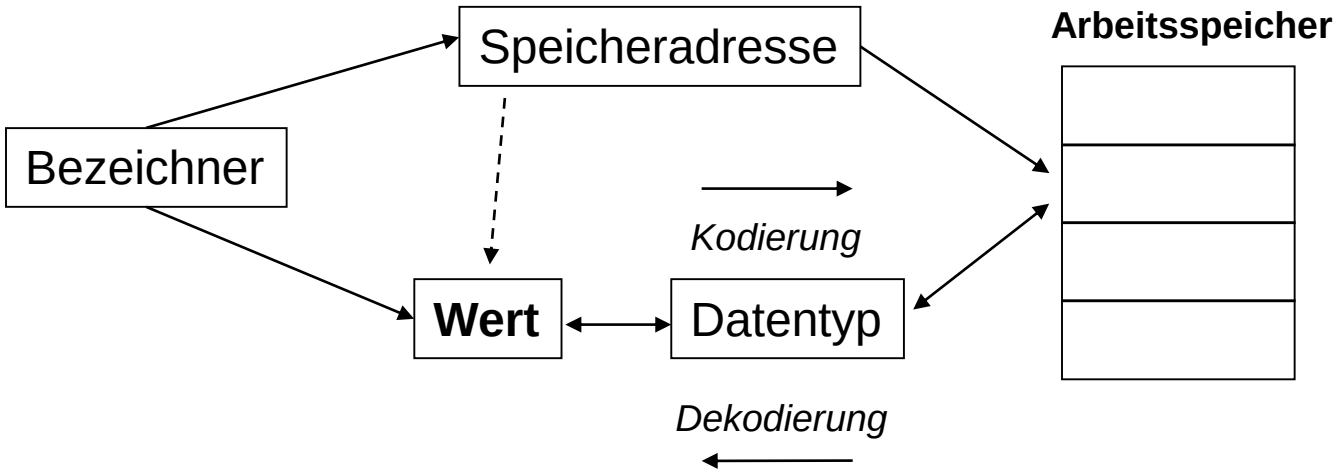
Problem: Eine Funktion **`swap(...)`** soll die Inhalte 2er Variablen vertauschen.
Wie kann dieses Problem gelöst werden?

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

// tausche x und y

...
```

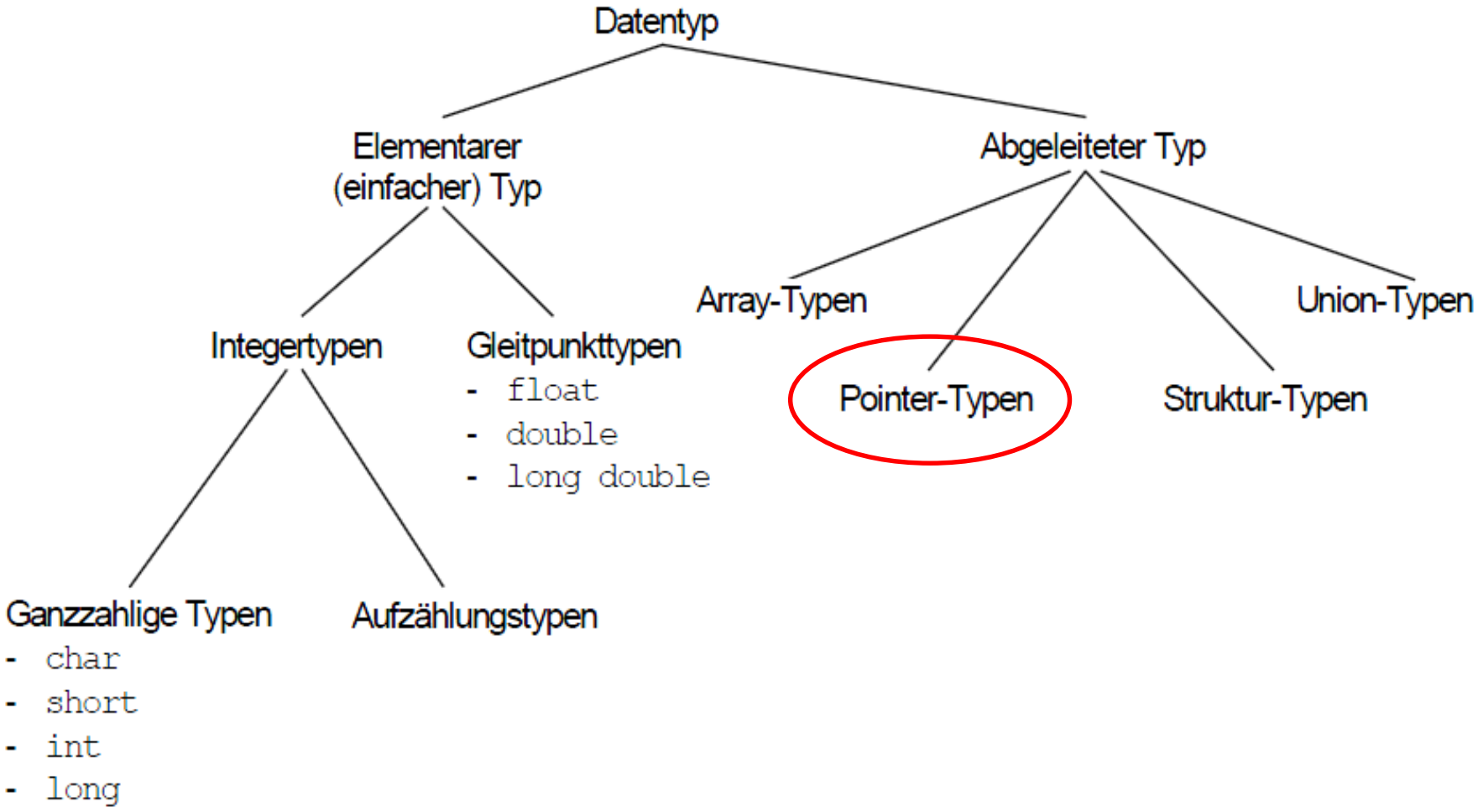


```
int temp;

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

Variablenmodell

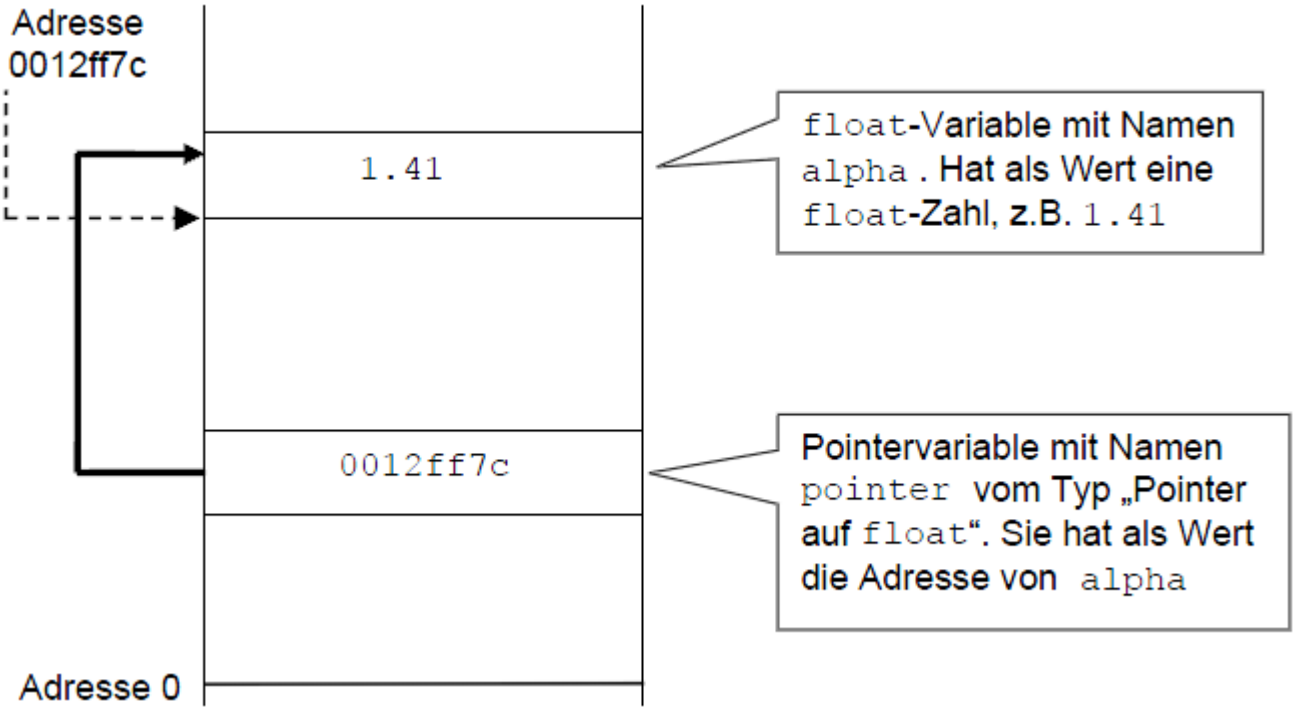


Typ-Klassifikation



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 eins Zeigers

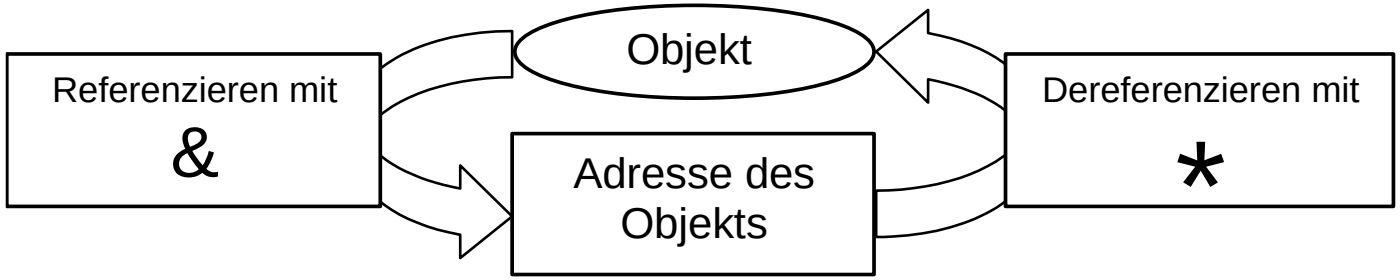
- Zeiger sind typgebunden

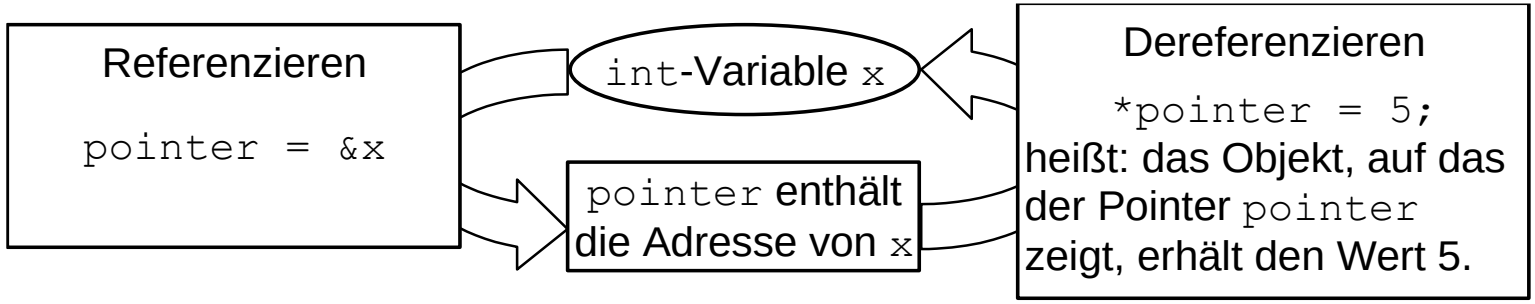
```
int *pi;            // Zeiger auf int

double *pd;         // Zeiger auf double
```

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

Referenzieren und Dereferenzieren

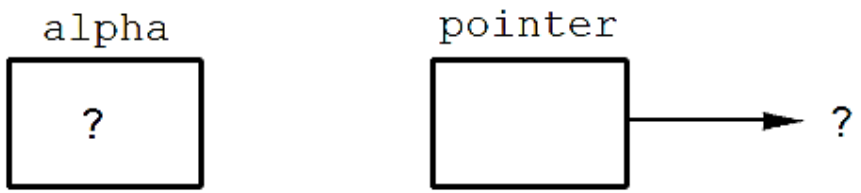
- **Referenzieren**: Adressoperator **&** liefert **Adresse** eines Objekts
- **Dereferenzieren**: Dereferenzierungsoperator ***** liefert **Inhalt** an der Adresse



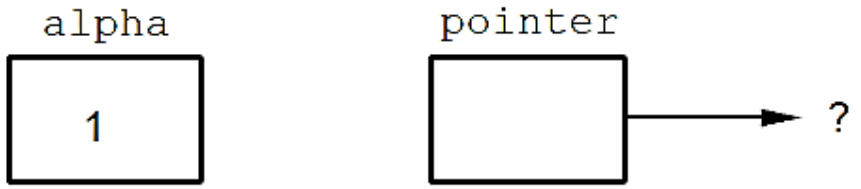


Adresszuweisung an einen Zeiger I

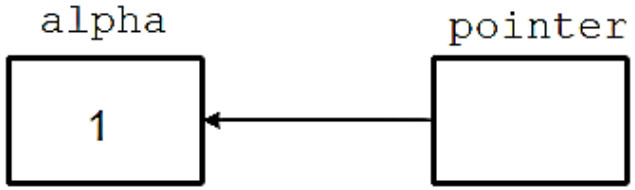
```
int alpha;
int * pointer;
```



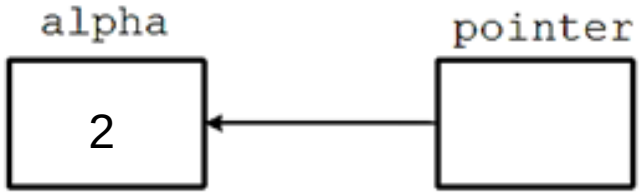

```
alpha = 1;
```




```
pointer = &alpha;
```

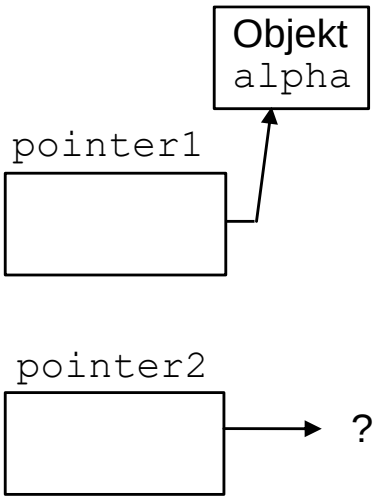



```
*pointer = 2;
```

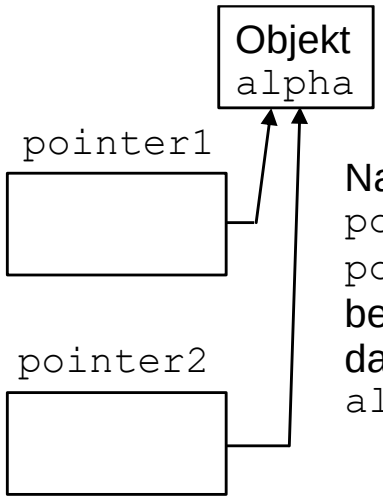


Adresszuweisung an einen Zeiger II

```
...
pointer1 = &alpha

pointer2 = pointer1
...
```



Vor der Zuweisung `pointer2 = pointer1` enthält `pointer2` irgendeine Adresse

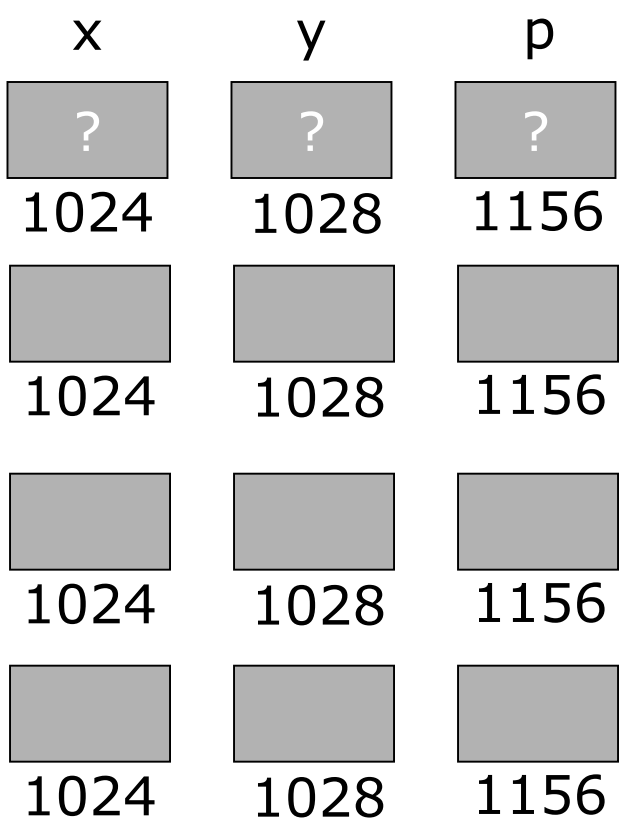


Nach der Zuweisung `pointer2 = pointer1` zeigen beide Pointer auf dasselbe Objekt `alpha`

Beispiel 1: Zeigeroperationen

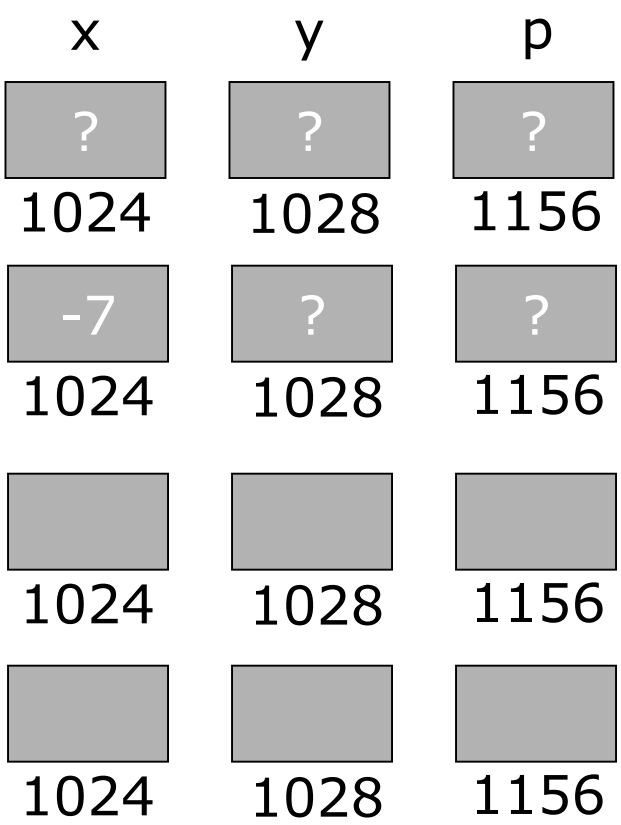
≙ Speicherzelle an Adresse 1024

```
int x, y, *p;

x = -7;

p = &x;

y = *p;
```

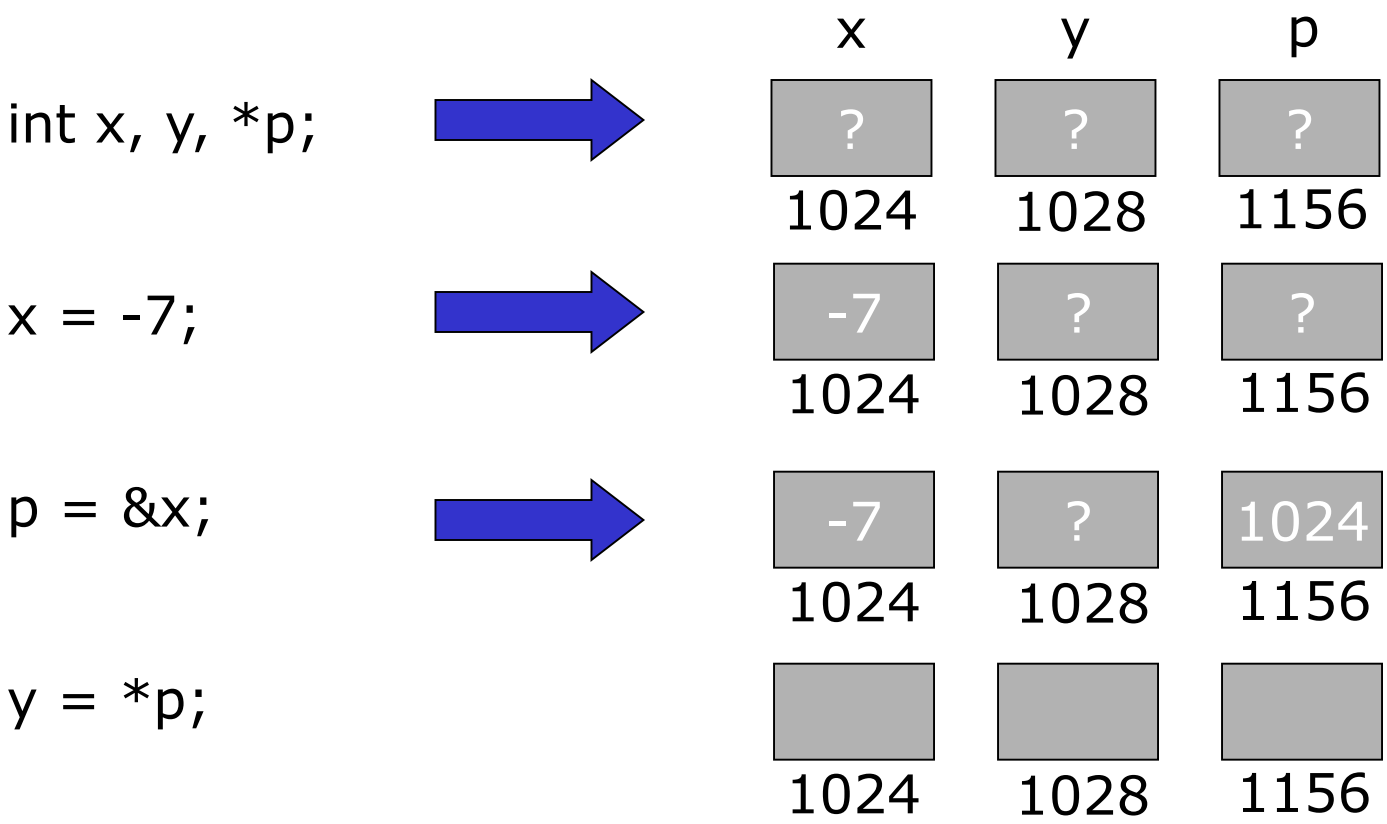


Beispiel 1: Zeigeroperationen

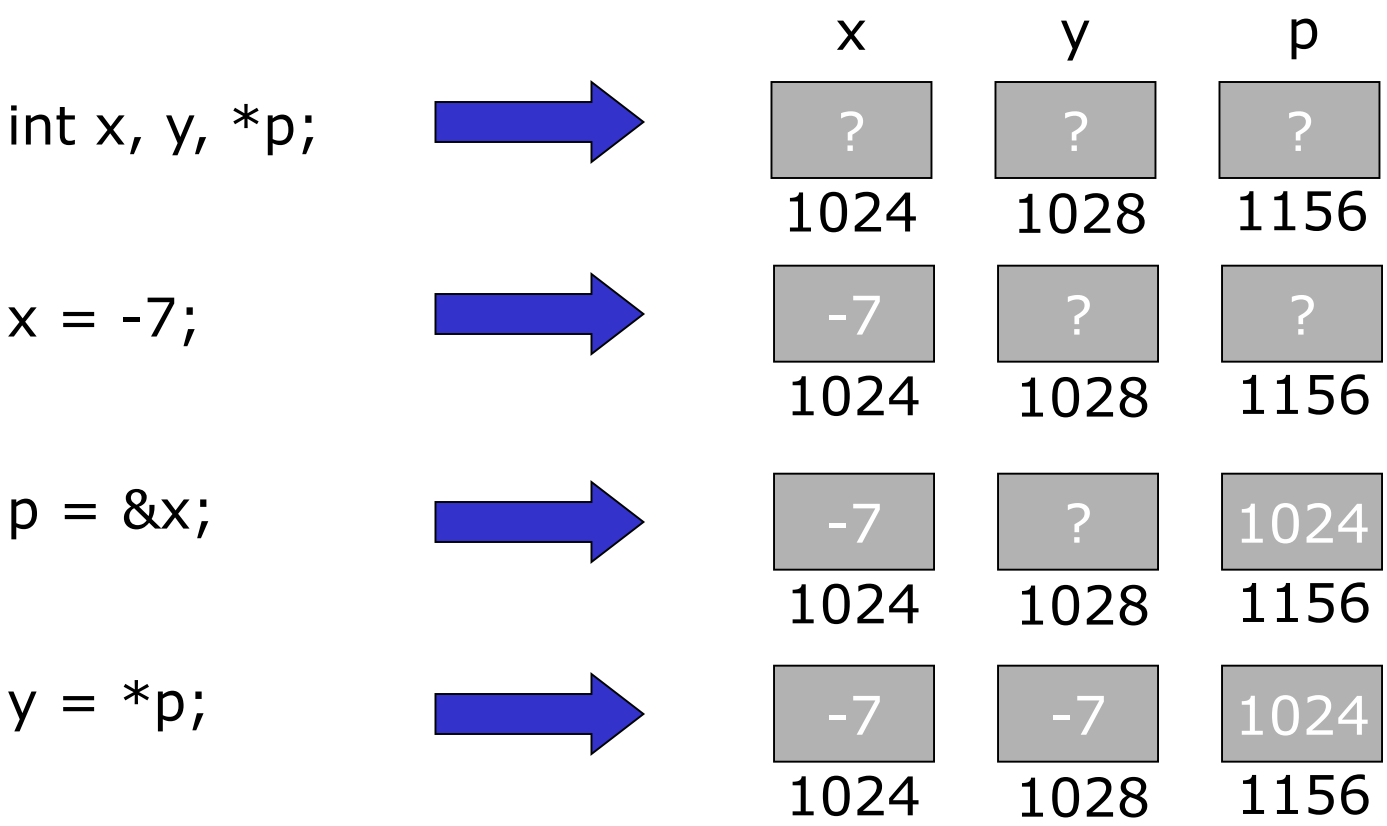

```
int x, y, *p;
x = -7;
p = &x;
y = *p;
```



Beispiel 1: Zeigeroperationen



Beispiel 1: Zeigeroperationen



Beispiel 2: Zeigeroperationen

Was ist an den folgenden Anweisungen a bis e falsch?

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

a) `*p = 3.14159;`

b) `p = &3.14159;`

c) `*x = 5.4;`

d) `x = &y;`

e) `p = *y;`

Funktion mit mehreren Rückgabewerten?

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

```
void swap(_______________)
{

  // TODO: Do the magic here!

}

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

  // tausche die Inhalte von x und y
  swap(_______________);

  return 0;
}
```

Warum Zeiger?

- call by reference
  - 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

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

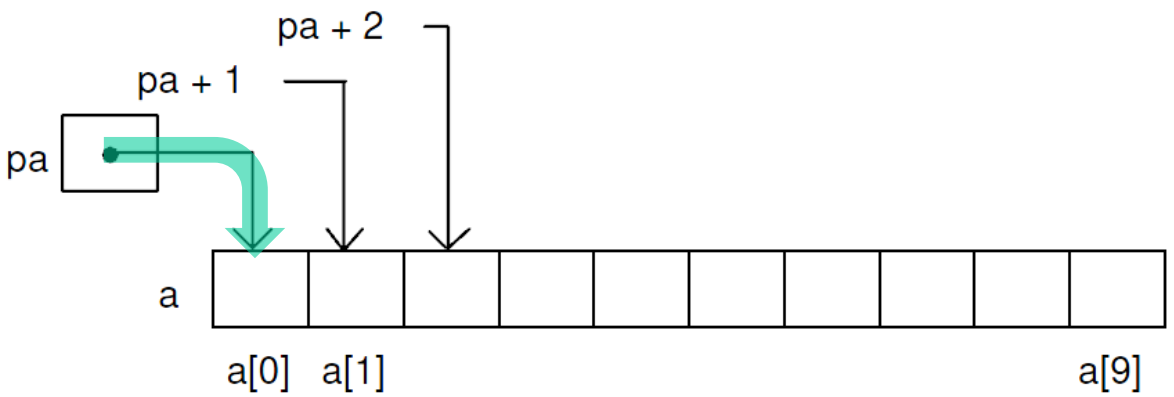
  Ein Zugriff `a[i]`
  wird vom Compiler als `*(a + i)` behandelt.

→ 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 strcopy(char destination[], const char source[]);

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

- sei `a` ein beliebiges Feld, dann ist

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

  `&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 feld[5]
long *zeiger;

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

Zeigerarithmetik

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

```
long feld[5]
long *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 identisch mit feld[4] = 123;
```

Zeigerarithmetik

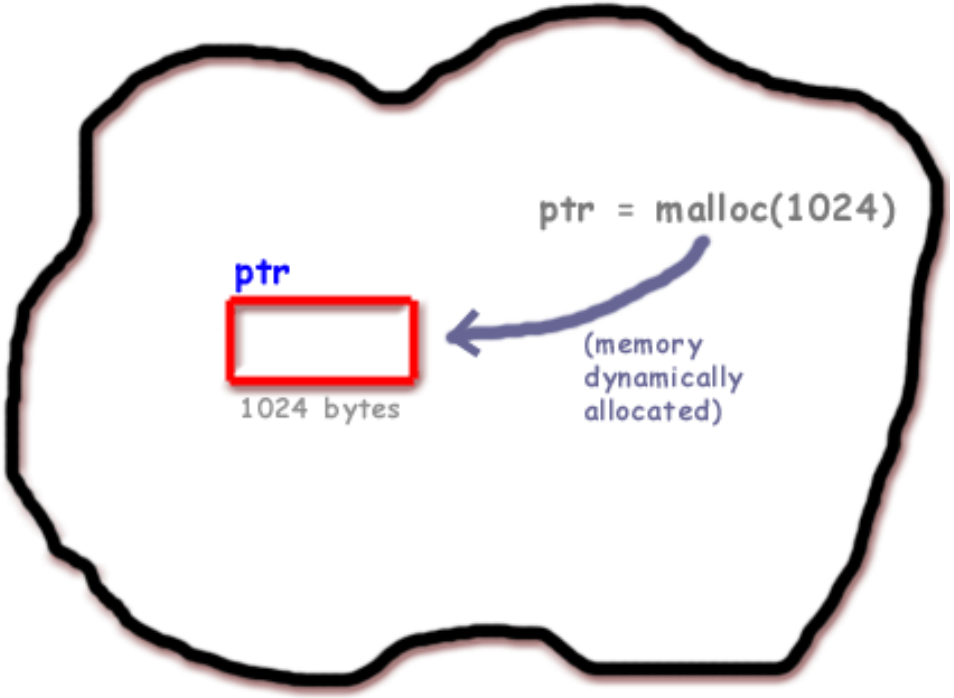
Fall 1: Zeiger +/- int-Wert -> Zeiger

```
long feld[5]
long *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 zeichenkette[])
{
  char *anf, *end;

  ...


}
```

Dynamische Speicherverwaltung



Dynamische Speicherverwaltung

Problem: Speicherbedarf ist während verschiedener Programmausführungen unterschiedlich

Lösung: Variablen (insb. Arrays) maximal dimensionieren:

```
char eingabetext[1000];
```

→ Nachteile?

Variable Feldgröße in lokalen Arrays nutzen **(ab C99)**

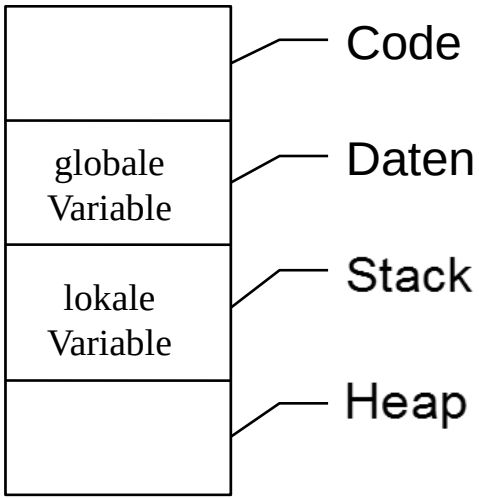
Statische vs. dynamische Variablen

- statische Variable (inkl. lokal, global)

```
int x;
```

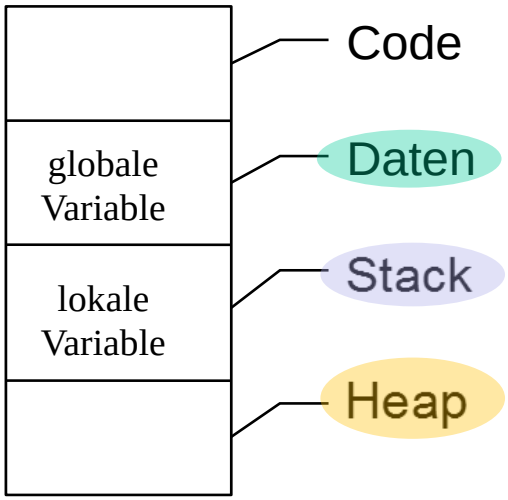
Verfügbarkeit und Ablage/Speichersegment je nach Speicherklasse (auto, register, static, extern)

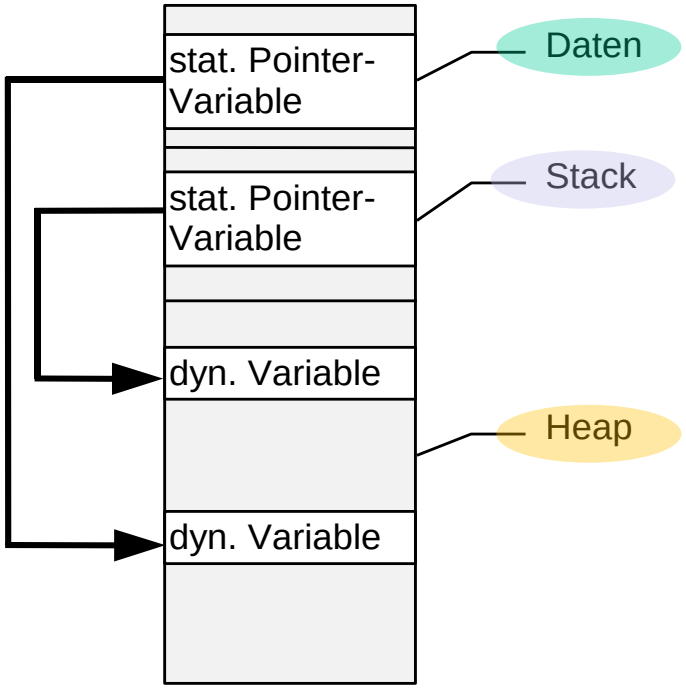
- Speichersegmente



Statische vs. dynamische Variablen

- dynamische Variable hat keinen Bezeichner, keine Vereinbarung
- Anlegen erfolgt bei Bedarf auf dem **Heap**
- Zugriff durch **Zeiger auf Bereich im Heap**





Dynamische Speicherverwaltung

Problem:	Speicherbedarf ist während verschiedener Programmausführungen unterschiedlich
Lösung **bisher:**	Variablen (insb. Arrays) maximal dimensionieren → Nachteil: "verschenkter" Speicher

Dynamische Speicherverwaltung

Problem:	Speicherbedarf ist während verschiedener Programmausführungen unterschiedlich
Lösung **bisher:**	Variablen (insb. Arrays) maximal dimensionieren → Nachteil: "verschenkter" Speicher
Lösung **neu:**	Speicherbedarf dynamisch, d.h. zur Programmlaufzeit festlegen

stdlib.h bietet Funktionen zum Speicheranforderung und -freigabe:

```
void* malloc(int Groesse);
void* calloc(int AnzahlElemente, int ElementGroesse);
void* realloc(void *memBlock, int Groesse);

void free(void *memBlock);
```

Speicherreservierung mit malloc()

```
#include <stdio.h>
#include <stdlib.h>

int main()
{
   int *pMem;

   pMem = malloc(sizeof (int));
   if (pMem != NULL)
   {
      *pMem = 3;
      printf("pointer auf int-Zahl mit Wert: %d\n", *pMem);
      free(pMem);
   }
   else
      printf("Speicheranforderung fehlgeschlagen.\n");

   return 0;
}
```

Speicherreservierung mit realloc() I

```
#include <stdio.h>
#include <stdlib.h>

int main()
{
   int *pMem;

   pMem = malloc(sizeof (int));
   if (pMem == NULL)
   {
      printf("Speicheranforderung fehlgeschlagen.\n");
      return -1;
   }

   *pMem = 3;
   printf("Nach malloc():\n");
   printf("pointer auf int-Zahl mit Wert: %d\n", *pMem);

   ...
```

Speicherreservierung mit realloc() II

```
    ...

    pMem = realloc ((void *)pMem, 2 * (sizeof(int)));
    if (pMem == NULL)
    {
       printf ("Speicheranforderung fehlgeschlagen.\n");
       return -1;
    }

    pMem[0] = 4;
    pMem[1] = 6;
    printf("Nach realloc():\n");
    printf("1. Element von pMem hat den Wert: %d\n", pMem[0]);
    printf("2. Element von pMem hat den Wert: %d\n", pMem[1]);
    free(pMem);

    return 0;
}
```

Speicherreservierung für ein Feld

```
#include <stdio.h>
#include <stdlib.h>

int main()
{
   int *pArray = NULL;
   int i, anzahl = 0;

   printf "Geben Sie die Anzahl der Feld-Elemente ein: ");
   scanf("%d", &anzahl);
   pArray = malloc(anzahl * sizeof(int));
   if (pArray == NULL)
      return -1;

   // Nutzung des reservierten Speicherbereichs
   for (i = 0; i < anzahl; i++)
      pArray[i] = i * i;

   ...
   free(pArray);
   return 0;
}
```

malloc & free: Typische Fehler

	Fehler	Folge
1)	kein Speicher mehr frei	
2)	Freigabe einer falschen Adresse	u.U. Absturz
3)	Freigabe bereits freigegeben Speichers	u.U. Absturz
4)	Freigabe eines regulären Feldes / einer regulären Variablen mit free()	u.U. Absturz
5)	Freigabe eines nicht-initialisierten Speichers	u.U. Absturz
6)	Zugriff auf ungültigen Speicher vor der Speicheranforderung	u.U. Absturz
7)	Zugriff auf bereits freigegebenen Speicher	u.U. Absturz
8)	Zugriff auf Speicher mit ungültigen Indizes	u.U. Absturz
9)	Verlust des Speichers durch Überschreiben des Zeigers	"memory leak"
10)	Verlust des Zeigers durch Rücksprung aus einer Funktion	"memory leak"
11)	Verlust des Zeigers bei Rückgabe (return-value verworfen)	"memory leak"