



Programmierung 1

– Pointer & Übergabeverfahren

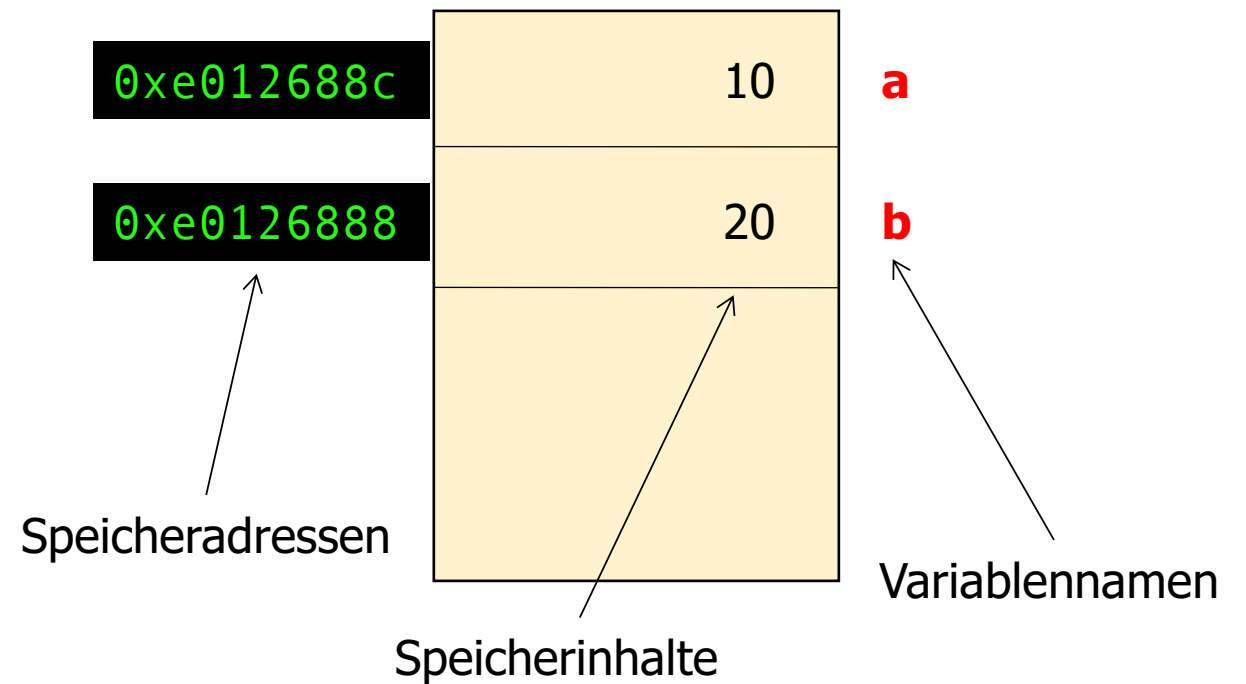


Wo steht was im Speicher?

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

```
int main()  
{  
    int a = 10,  
        b = 20;  
    //some code...  
    return 0;  
}
```

Wie sieht es im Hauptspeicher aus?

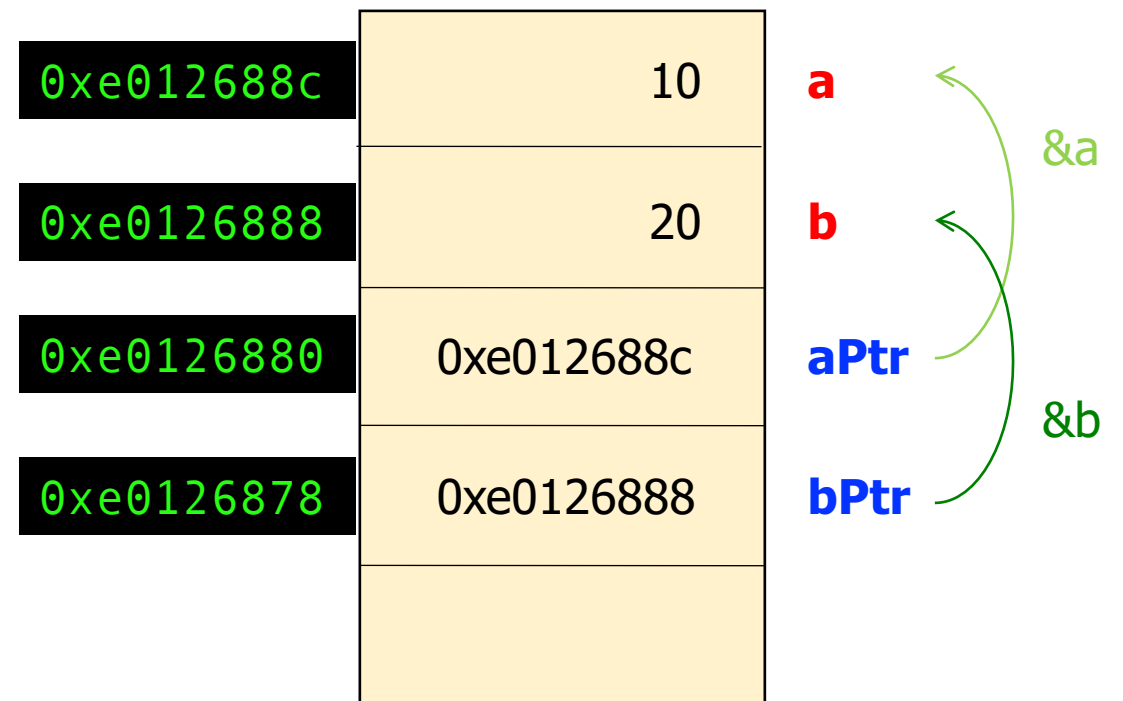


Wo steht was im Speicher?

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

```
int main()  
{  
    int a = 10,  
        b = 20;  
    int *aPtr = &a,  
        *bPtr = &b;  
    //some code...  
}
```

Wie sieht es nun im Hauptspeicher aus?



Pointer bzw. Zeiger



- Können Speicheradressen von Variablen eines geg. Typs speichern
 - Soll z.B. Adresse einer Integer-Variablen gespeichert werden, ist ein Zeiger vom Typ **int*** nötig (für Float-Variable wäre es analog **float***)
 - Bsp.: `int n = 42, *nP = &n;`
 - Steht * bei Variablendeklaration vor Variablenname, handelt es sich um Deklaration eines Zeigers auf diesen Datentyp
- Wichtige Symbole: * und &
 - Adressoperator & liefert numerische Speicheradresse einer Variablen
 - Inhaltsoperator * liefert Speicherinhalt der Adresse, auf die Pointer zeigt
 - Bsp.: `printf("Wert von n: %d\n", *nP); //42`
 - Zugreifen auf Daten an dieser Speicheradresse heißt *Dereferenzieren* eines Pointers

Speicheradressen ausgeben



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

```
int main()  
{
```

```
    int a = 10, b = 20;  
    int *aPtr = &a, *bPtr = &b;
```

```
    // Speicherinhalte von a und b anzeigen
```

```
    printf("Werte:          %14d \t %14d\n", *aPtr, *bPtr); // a, b
```

```
    // Speicheradressen von a und b
```

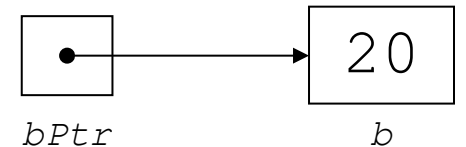
```
    printf("Adressen:      %p \t %p\n", aPtr, bPtr);
```

```
    // Speicheradressen von aPtr und bPtr
```

```
    printf("Pointeradr.: %p \t %p\n", &aPtr, &bPtr);
```

```
}
```

```
Werte:          10          20  
Adressen:      0x7ffee012688c 0x7ffee0126888  
Pointeradr.: 0x7ffee0126880 0x7ffee0126878
```



Zeiger-Beispiel

- Einzelschrittansicht: Zugriff auf Variable mittels Pointer

```
int main(void)
{
    int i;
    int *ip;
    ...

    i = 5;
    ip = &i;
    *ip = *ip + *ip;
    ...
}
```

Hauptspeicher

1000		
1002	10	
1004	1002	
1006		
1008		
1010		
1012		
1014		

Übung

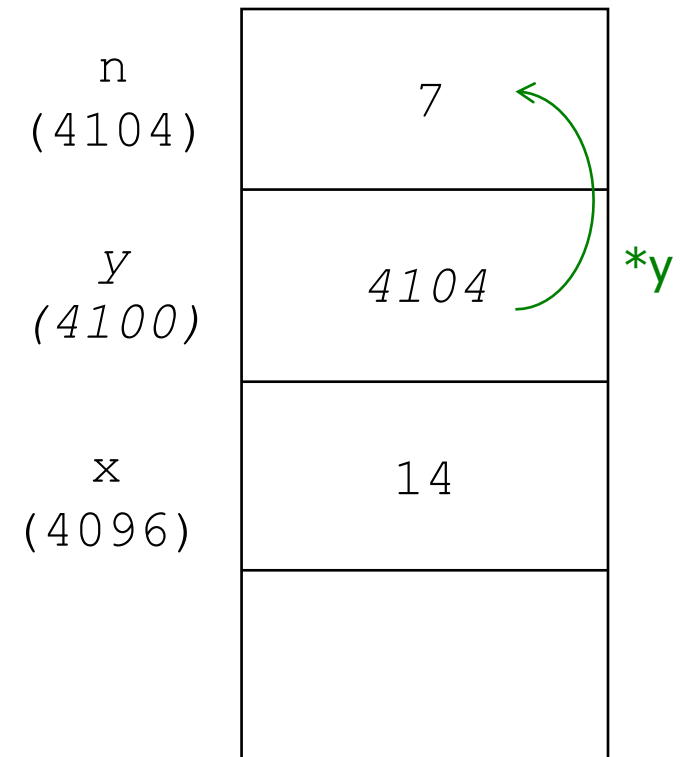
- Welcher Wert wird ausgegeben?

```
int  n = 7;  
int  *y = &n;  
int  x = *y * 2;  
printf("%d\n", x);
```

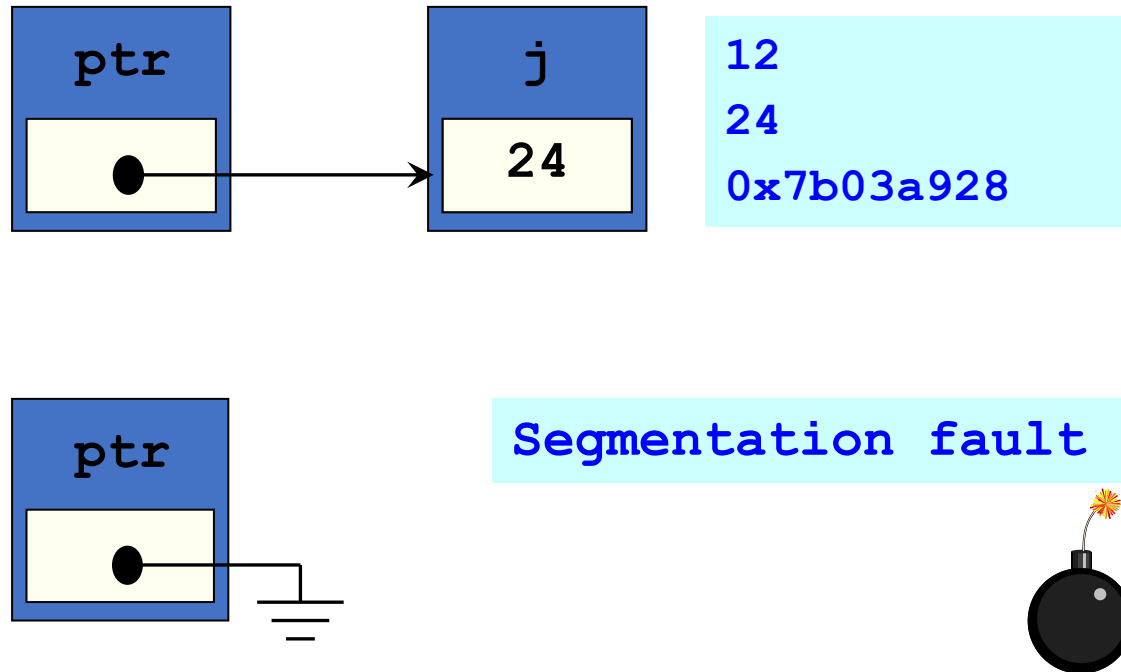
// x = 7 * 2 = 14

```
int  sum, var = 11;  
int  *varptr = &var;  
*varptr += 4;  
sum = var + *varptr;  
printf("%d\n", sum);
```

// sum = 15 + 15 = 30



Null-Pointer



```
int main()
{
    int j = 12;
    int *ptr = &j;
    printf("%d\n", *ptr);
    j = 24;
    printf("%d\n", *ptr);
    printf("%p\n", ptr);
    ptr = NULL;
    printf("%d\n", *ptr); //crash
}
```

- NULL bezeichnet ungültige (noch nicht initialisierte) Speicheradresse
 - Vorteil: Null-Pointer kann nicht (evtl. fälschlich) dereferenziert werden

Pointer-Syntax bei Strukturen

- Strukturen fassen Objekte verschiedener Datentypen zu einer Einheit zusammen

- Bisher:

```
struct student {  
    char name[80];  
    unsigned matnr;  
};
```

```
struct student studi, *studP;  
  
strcpy(studi.name, "Angela");  
studi.matnr = 123456;
```

- Bei Zeigern aber erstmal sehr umständlich:

- Achtung: '*' hat niedrigere Präzedenz als '.'
*studP.matnr bedeutet *(studP.matnr)
Was nicht kompiliert...

```
studP = &studi;  
(*studP).matnr = 987654;
```

- Liegt Strukturvariable als Zeiger vor, erfolgt Zugriff auf Komponenten i.d.R. stattdessen mit Pfeiloperator: ->

```
studP->matnr = 987654;  
strcpy(studP->name, "Olaf");
```

Kommandozeile



- Hauptprogramm kann auch Argumente haben

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

- **argument count**: Anzahl der Kommandozeilen-Argumente eines Programms
- **argument vector**: Feld von Zeigern auf Strings, welche die Argumente enthalten
 - Array speichert nur Pointer auf Zeichenketten, die irgendwo anders im Speicher liegen
- Bei Aufruf von Konsole aus kann Programm mit Parametern versorgt werden
 - Ermöglicht so z.B. Batch-Verarbeitung
 - Zugriff auf erstes Argument mit `argv[1]`
 - Feld `argv` hat feste Größe `argc`, aber Strings können hier beliebig groß sein
 - Programmname selbst steht in `argv[0]`

```
Yvonne@PCYJUNG ~/Eigene Dateien/Devel/Avalon/apps/aopt
$ aopt -i test.wrl -x test.x3d
LOG      Avalon   Init: 24/449, v2.2.0 build: R-0 Sep 29 2014 Windows x86_64
WARNING  Avalon   Incomplete DocStatus: NodeTypes: 416/417 Fields: 5381/5387
=====
License found: full functionality and commercial use granted.
=====
Command Line: c:\Users\Yvonne\Documents\Devel\Avalon\Build\bin64\RelWithDebInfo\
aopt.exe -i test.wrl -x test.x3d
LOG      Avalon   Read time: 0.002000
=====
Call: writeX3D with 1 param
Write raw-data to test.x3d as model/x3d+xml
WARNING  Avalon   Avalon::exitSystem() call and node/obj left: 0/2333
```

Exkurs: Untypisierte Zeiger

- `void*`
 - Zeiger auf `void` bezeichnet Zeiger, der jeden anderen Zeiger ersetzen kann, ohne dass Information verloren geht
 - Jeder Zeiger kann in `void*` verwandelt werden
 - Dient als Platzhalter für beliebige Zeiger
- Beispiel (mit Type Cast)

```
int a = 11;  
char *cA = NULL;  
void *ptr;
```

```
ptr = (void*)&a;  
ptr = (void*)cA;
```





Parameterübergabeverfahren

Call-by-Value vs. Call-by-Reference

Parameterübergabe (1)



- Bei Funktionsaufruf werden alle Argumente von links nach rechts ausgewertet und an die formalen Parameter gebunden
 - Die übergebenen (Variablen-) Werte werden dabei jeweils kopiert
 - Diese Art der Parameterübergabe nennt sich daher *Call-by-Value*
 - Vorteil: Änderungen lokaler Variablen haben keine Wirkung nach außen – keine Seiteneffekte
 - Problem: Manchmal möchte man an Funktion übergebene Argumente dauerhaft verändern, oder man möchte, insbes. bei großen `struct`-Variablen, aufwendiges Kopieren vermeiden
- Wird in aufgerufener Funktion eine `return`-Anweisung ausgeführt, wird der entsprechende Ausdruck zurückgegeben
 - In aufrufender Funktion wird der Aufruf durch den Rückgabewert ersetzt
 - Problem: Was ist, wenn Funktion aber zwei oder mehr Werte zurückliefern soll?

Parameterübergabe (2)

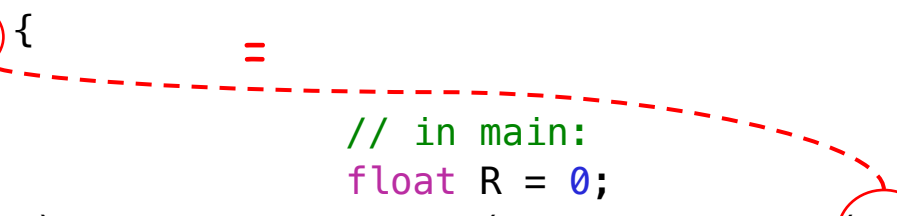


- Ansatz: Wir orientieren uns daran, wie Arrays übergeben werden
 - Bei Feldern wird die Anfangsadresse eines Feldes an Funktion übergeben
 - Dadurch arbeitet Funktion auf demselben Speicherbereich wie aufrufende Funktion, Änderungen am Feld bleiben nach Beenden der Funktion bestehen
 - Da statt konkretem Wert eine Referenz auf einen Speicherbereich übergeben wird, spricht man hier auch von *Call-by-Reference*
- Lösung: Verwendung von Pointer-Variablen
 - Pointer speichern Adressen von Variablen – werden diese übergeben, kann Speicherinhalt der Variablen auch in Funktion modifiziert werden
 - Variablen aus aufrufender Funktion sind damit über Pointer veränderbar
 - Syntaktisch: Inhaltsoperator * bei formalem Parameter, Adressoperator & bei Argument

Call-by-Pointer

- Bsp. 1: Einlesen des Widerstands R mit Fehlerprüfung
 - Funktion soll 2 Werte liefern: R und Fehlerzustand

```
bool holeWiderstand(float *r) {  
    char buf[80];  
    printf("R eingeben: ");  
    fgets(buf, 80, stdin);  
    int i = sscanf(buf, "%f", r);  
    return i == 1 && *r >= 0;  
}  
  
// in main:  
float R = 0;  
while (!holeWiderstand(&R))  
    ;
```



"Call-by-Reference"

- Bsp. 2: Vertauschen zweier Zahlen (kein Rückgabewert nötig)

```
void swap(int *a, int *b) {  
    int tmp = *a;  
    *a = *b;  
    *b = tmp;  
}  
  
// in main:  
int x = 17, y = 4;  
swap(&x, &y);  
printf("x=%d, y=%d\n", x, y);
```

Call-by-Pointer



- Bsp. 3: Einfache Berechnung als Prozedur statt Funktion

- Ergebnis liegt im Speicherbereich der Original-Variablen, Rückgabewert nicht erforderlich
- Aufruf: `summiere(9, &erg);` //Ergebnis liegt danach in `erg` vor

```
void summiere(int n, int *sum)
{
    *sum = 0;
    for (int i = 1; i <= n; i++)
        *sum += i;
}
```

← Übergabe der Speicheradresse eines Integerwerts an Parameter `sum`

← Arbeiten direkt auf Speicherplatz des übergebenen Integerwerts `erg`

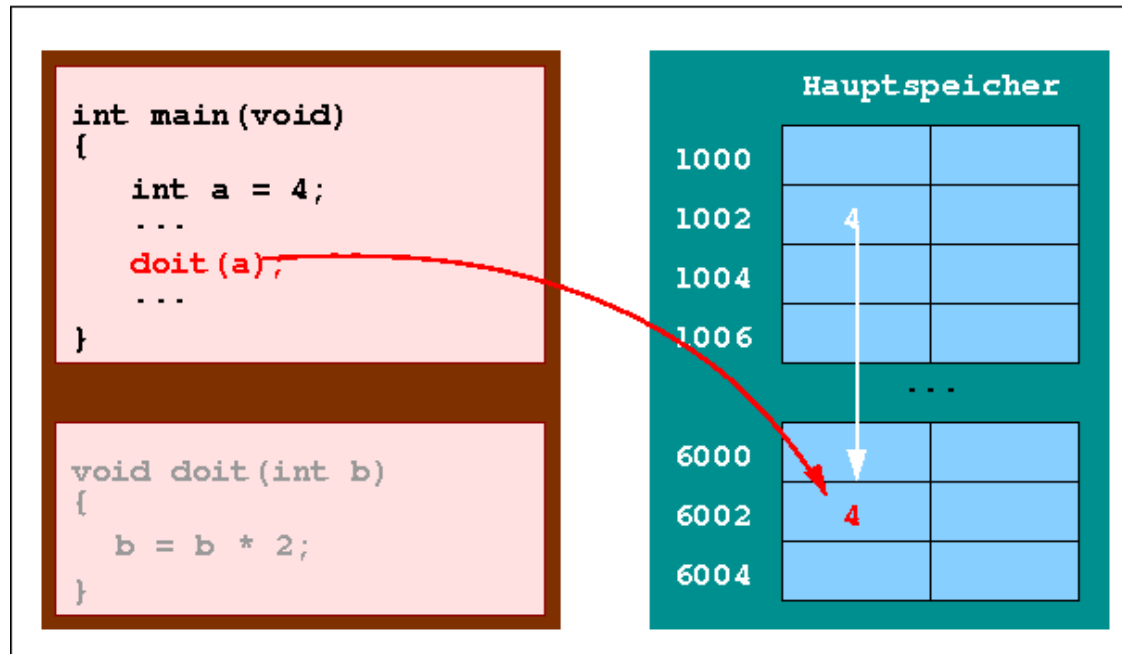
- Übergabeverfahren in Literatur fälschlich oft Call-by-Reference genannt

- C kennt de facto nur Call-by-Value, da bei Parameterübergabe immer nur kopiert wird
- Aber: kopierte Werte können Adressen von Variablen aus aufrufender Funktion sein 😊

Vergleich (1)

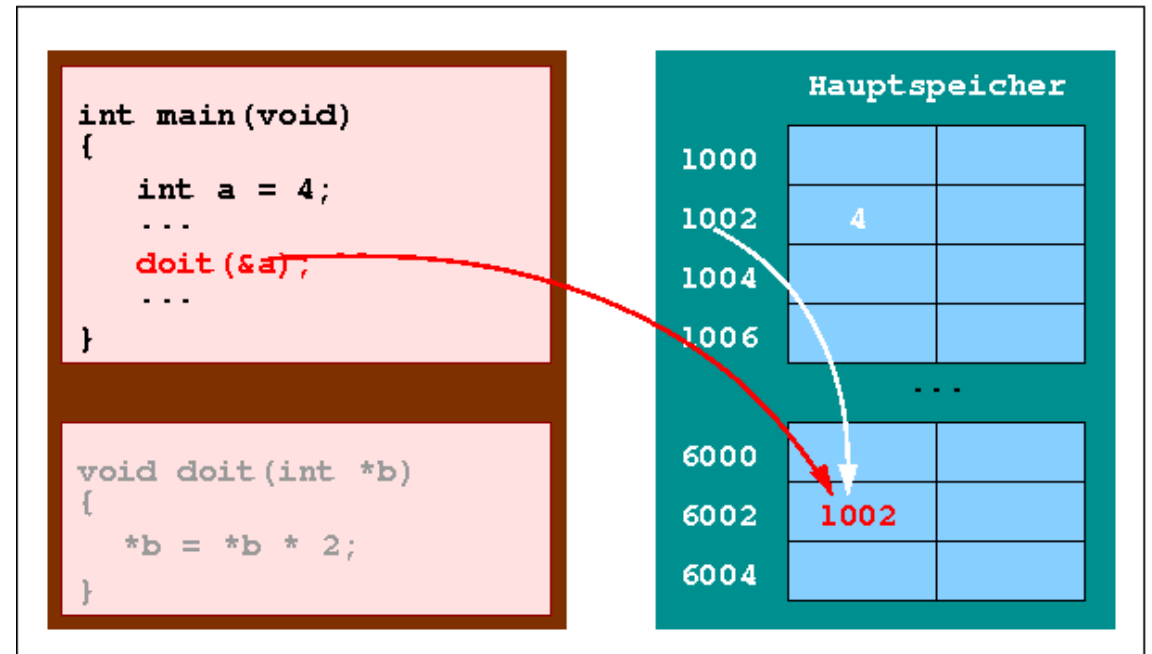
Call-by-Value

- Kopie des Wertes der Variablen



Call-by-Reference (bzw. Pointer)

- Kopie der Adresse der Variablen



Vergleich (2)



Call-by-Value

- Wert des Arguments wird formalem Parameter vor Ausführungsbeginn zugewiesen
 - Kopie des Wertes wird übergeben
 - Veränderung des formalen Parameters innerhalb der Funktion hat *keine* Wirkung auf aufrufende Funktion
- Keine Seiteneffekte

Call-by-Reference

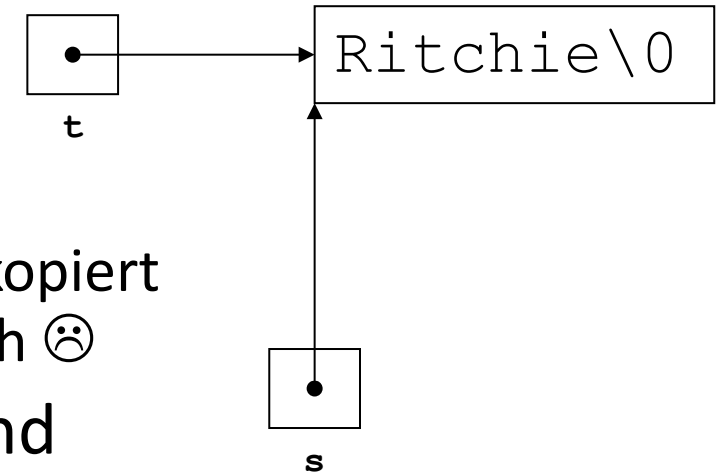
- Argument „verschmilzt“ mit formalem Parameter (nutzt gleiche Speicherstelle)
 - Variable wird direkt *referenziert*
 - Alle Veränderungen des formalen Parameters sind auch nach Funktionsausführung *nach außen sichtbar*
- Seiteneffekte möglich!

Strings kopieren revisited

- Implementierungsvorschlag (nach Kernighan & Ritchie)

```
void strcpy(char *s, char *t) {  
    //weist zuerst zu, prüft dann  
    while ( (*s++ = *t++) )  
        ; //erhöht nach Anweisung  
}
```

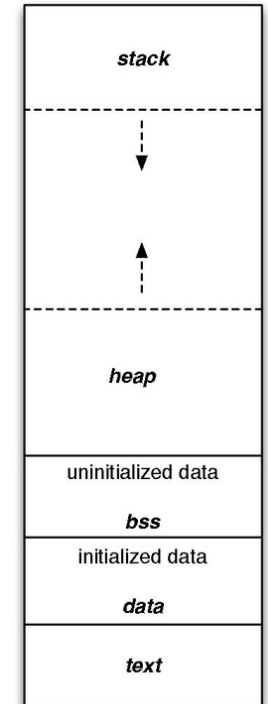
- Achtung: Es wird nicht überprüft, wie viele Zeichen kopiert werden, Zugriff auf fremden Speicherbereich möglich ☹
- Wenn **s** und **t** Zeiger (z.B. auf Zeichenketten) sind
 - Dann kann man **t** nicht kopieren mittels `s = t;`
 - Setzt nur Zeiger **s** auf Zeiger **t**, kopiert nur Zeiger



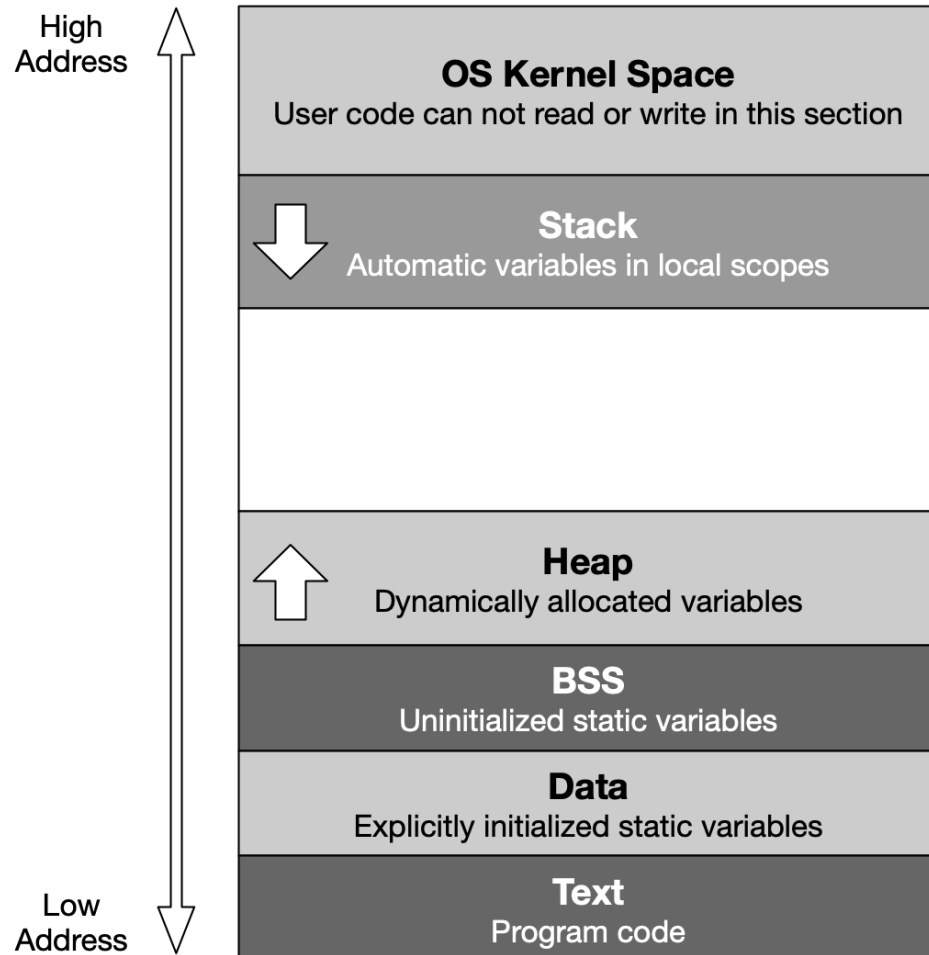


Speicherverwaltung

Stack vs. Heap



Prozess-Speichermodell



- Stack beinhaltet den sog. Call Stack
 - Eigener Speicherbereich für jeden Funktionsaufruf (sog. Stack Frame)
 - Lokale Variablen werden hier angelegt
 - Lebenszeit endet mit Block-/Funktionsende
- Heap (dynamischer Speicher)
 - Zuständig für dynamisch zur Laufzeit vom Entwickler angeforderten Speicher
 - Lebenszeit endet erst, wenn Speicherblock explizit wieder freigegeben wird
- Data- und BSS-Segment
 - Hält globale bzw. statische Variablen

Dynamischer Speicher



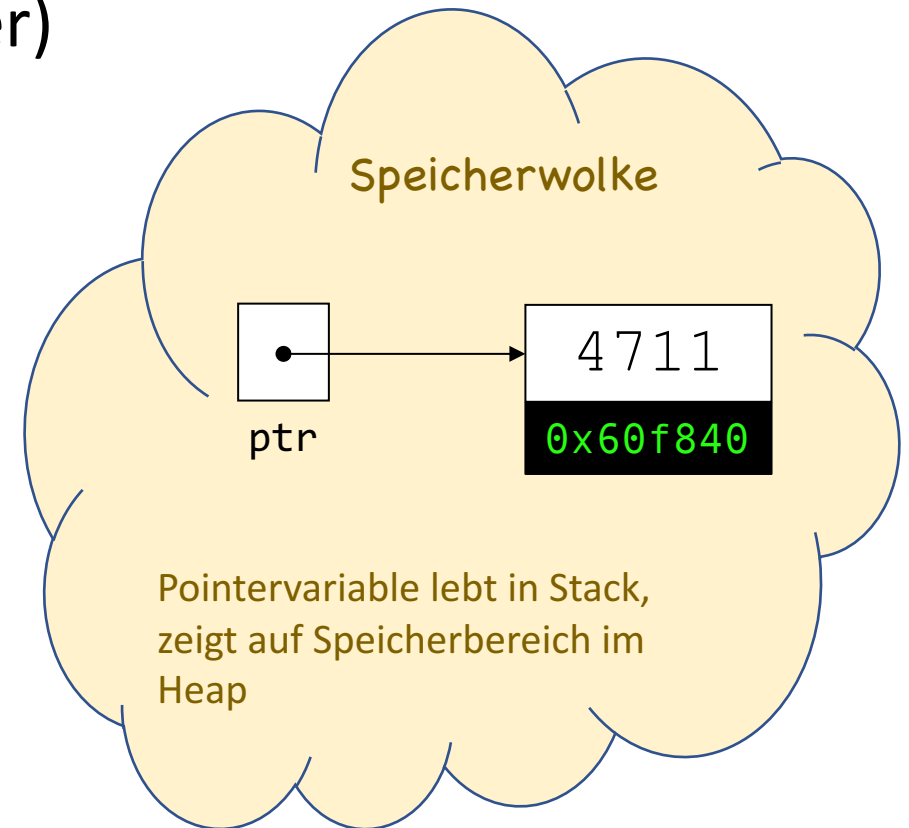
- Speicher anfordern (im Bsp. für einen Integer)

```
int *ptr = (int*)malloc(sizeof(int));

if (ptr != NULL) {
    *ptr = 1024;    // Integerwert setzen
}
else {
    printf("Kein Speicher verfuegbar!\n");
}
```

- Speicher freigeben (wenn nicht mehr nötig)

```
if (ptr != NULL) {
    free(ptr);
    ptr = NULL;    // Definiert auf 0 setzen
}
```



Stack vs. Heap

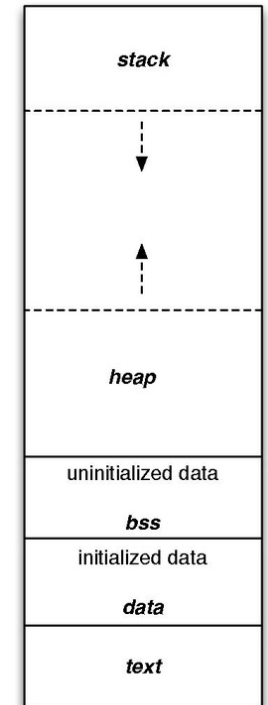
- Stack-Allokation (im Bsp. Integervariable sowie Array)

```
int i, intArr[3];  
intArr[0] = 4711;
```

- Heap-Allokation (im Bsp. Speicherblock für 5 Integers)

```
int *intPtr = (int*)malloc(5 * sizeof(int));  
if (intPtr) {  
    intPtr[0] = 4711;  
    // some more code...  
    free(intPtr);  
}
```

(Optional) Type Cast auf *int**, da Rückgabetyt von `malloc()` vom Typ *void**



- Wenn Lebensdauer von `intPtr` endet, bleibt Speicherblock in Heap erhalten

Speicherverwaltung in C

- Geschieht mit Hilfe von Bibliotheksfunktionen

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

- Speicherblock erst anfordern mit Funktion *malloc()*

```
void* malloc(unsigned long size);
```

- Ziemlich low-level: benötigt als Argument gewünschte Größe des Speicherbereichs in Byte; liefert Anfangsadresse eines (zusammenhängenden) Speicherblocks im Heap
- Bsp.: dynamisch Platz für zehn Short-Werte allozieren (reservieren)

```
short *ptr = (short*)malloc(10 * sizeof(short));
```

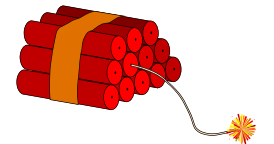
- Kein Unterschied zwischen Einzelwerten und "Feldern"!

- Speicherblock wieder freigeben mit Funktion *free()*

```
void free(void *ptr);
```


Speicherverwaltung in C

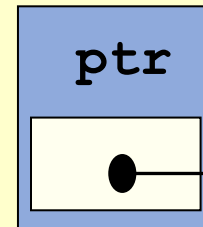
- Die Funktionen *malloc()* und *free()* werden verwendet, um dynamisch (Heap-) Speicher anzufordern bzw. freizugeben
 - Heap ist zusätzlicher Speicherbereich, den System für dynamisch vom Programmierer angeforderten Speicher bereitstellt
- *malloc()* allokiert Speicherblock gewünschter Größe u. gibt Startadresse zurück
 - Im Fehlerfall wird Null-Pointer zurückgegeben
 - Deshalb danach erst testen, ob Operation erfolgreich war, bevor Pointer dereferenziert wird
 - Dereferenzieren eines Pointers vor *malloc()* oder nach *free()* führt zu Absturz
- Speicher (inkl. Heap) hat nur endliche Größe
 - Speicherblock, der mit *malloc()* geholt wurde, muss mit *free()* freigegeben werden, sonst kann Speicher nicht mehr genutzt werden (→ Memory Leak)
 - Argument von *free()* ist Pointer, der mit *malloc()* initialisiert wurde



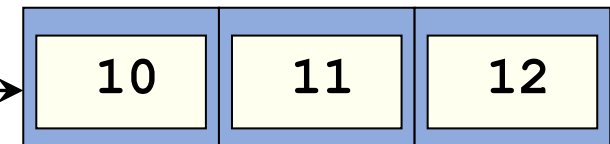
Pointer auf Speicherblock

- Zeiger auf mehrere zusammenhängende Speicherzellen im Heap

```
#include <stdlib.h>
int main() {
    int *ptr = (int*)malloc(3 * sizeof(int));
    if (ptr) {
        ptr[0] = 10;
        ptr[1] = 11;
        ptr[2] = 12;
        // some code...
        free(ptr);
    }
}
```



Pointervariablen enthalten
(bzw. zeigen) auf Adressen

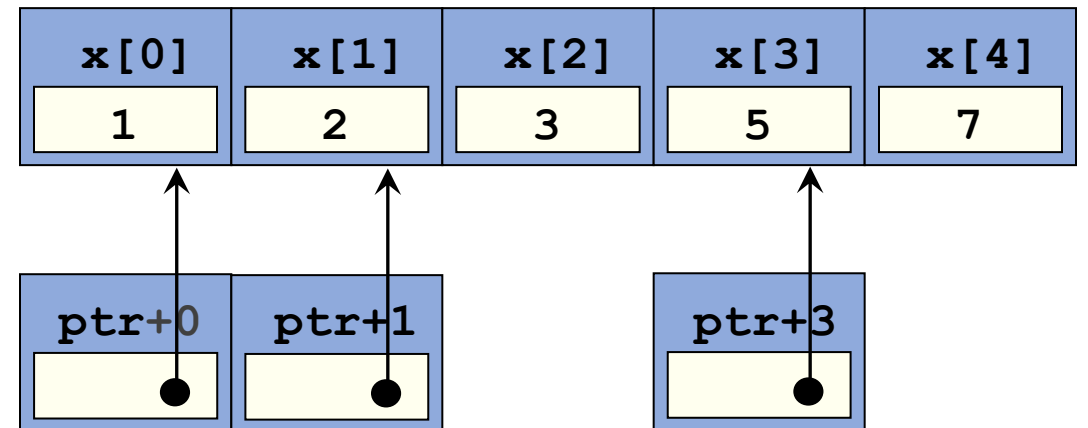
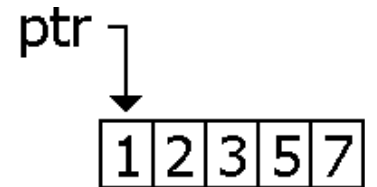


Dynamisch allokiertes Feld

Pointer und Arrays

- Felder in C sind konstante Zeiger auf zusammenhängende Speicherblöcke
 - Feldname ohne Index repräsentiert Anfangsadresse des Feldes, ist somit unveränderlicher (konstanter) Zeiger auf erstes Feldelement
 - Jede Operation, die man durch Indizierung von Feldelementen ausdrücken kann, lässt sich auch mit Zeigern realisieren

```
int main()
{
    int x[5];
    x[2] = 3;
    x[4] = 7;
    int *ptr = x;
    *(ptr+0) = 1; // x[0] = 1
    *(ptr+1) = 2; // x[1] = 2
    ptr[3] = 5;   // x[3] = 5
}
```

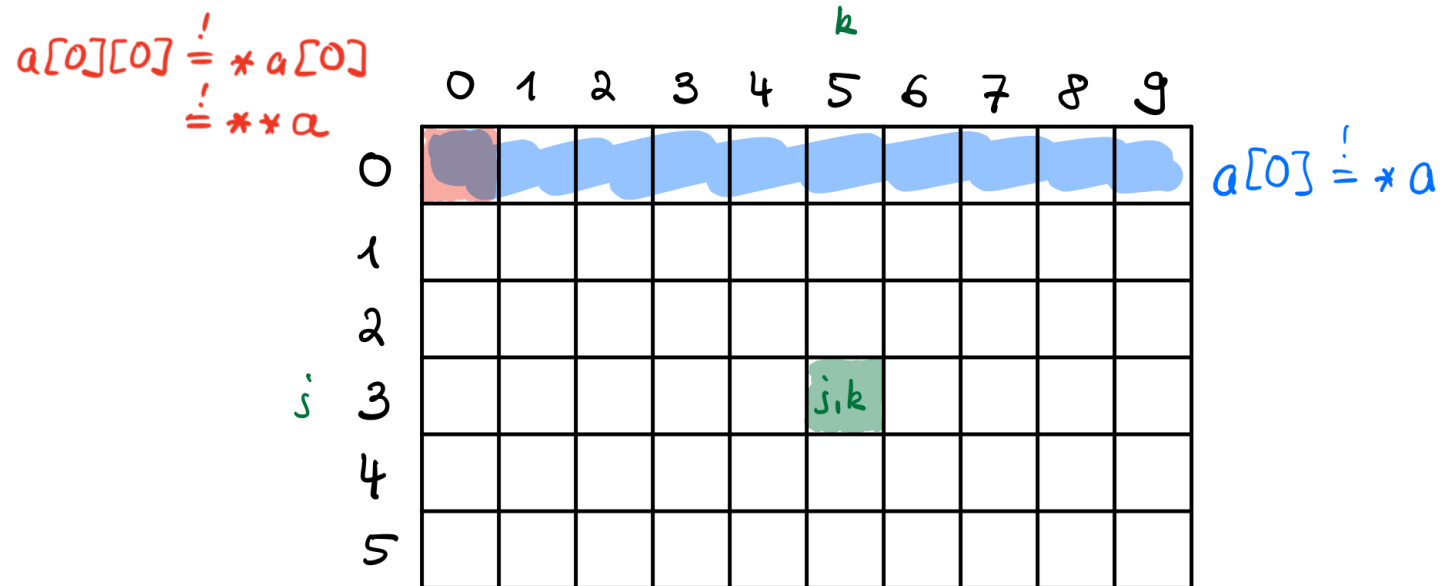


Zeigerarithmetik



- Ist Rechnen mit Zeigern und Adressen
- Nur Addition bzw. Subtraktion mit Integerwerten ist erlaubt (z.B. $p++$)
 - Zeiger p zeigt damit auf nächstes Speicherobjekt des geg. Datentyps (entspricht $p + 1$)
 - Achtung: Felder sind Adresskonstanten, daher bei Feldern Änderung der Anfangsadresse nicht möglich (also z.B. kein Inkrement/Dekrement)
- Ergebnis typabhängig: $p++$ bedeutet, dass Größe eines Speicherobjekts des gegebenen Typs auf Adresse addiert wird
 - n Elemente entsprechen $n * \text{sizeof}(<\text{Datentyp}>) \text{ Byte}$
 - Ausdruck $p + n$ bedeutet also, dass n -mal die Datentypgröße auf die in p gespeicherte Adresse addiert wird (analog für $p - n$)
 - Damit bezeichnet $p + n$ das n -te Objekt nach dem Speicherobjekt, auf das p gerade zeigt

Zweidimensionale Felder



`int a[6][10];` Allgemein: $a[j][k] = *(*(a + j) + k)$

Zweidimensionales Feld auch über eindimensionales darstellbar:

`int a[6 * 10];`

Zugriff auf Stelle (j, k) : $a[10 * j + k]$

Annotations for the formula $a[10 * j + k]$:

- 10 : Anz. Spalten (Number of columns)
- j : akt. Zeile (active row)
- k : akt. Spalte (active column)



Vielen Dank!

Noch Fragen?

