4. TUTORIUM

Einführung in die Strukturierte Programmierung

1. Übungsbeispiel

1. Übungsbeispiel

- „atomare“ Funktionen
  - Eine Funktion ein Zweck
- Genauestes Einhalten der Spezifikation
  - Programmablauf
  - Keine zusätzlichen Ausgaben
  - Erlaubte Bibliotheken
  - Dateien der Abgabe
  - ➔ Jedes einzelne Wort(!) der Spezifikation mit eurer Abgabe gegenprüfen, ansonsten …. ☹☹☹
- Plagiate
- Fragen zum Übungsbeispiel?

DD

- Keine detaillierten Anforderungen, aber
- was ein DD auf jeden Fall enthalten sollte:
  - Einen Überblick über die Funktionsweise
    - Was tut das Programm
    - Wie ungefähr tut es das
  - Funktionsbeschreibungen
    - Was tut die Funktion
    - Was sind die Eingabeparameter, was ist der Return-Wert
- ➔ Siehe Beispiel-DD im Teachcenter

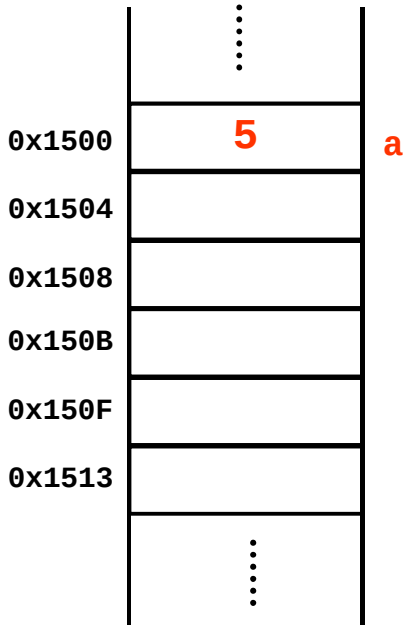
Pointer

Tribute to Matthias Prinz and Georg Kapeller

Pointer

- Wozu?
  - u.a. um in Funktionen nicht nur mit Kopien zu arbeiten

```
int a = 5;
myFunction(a);
```

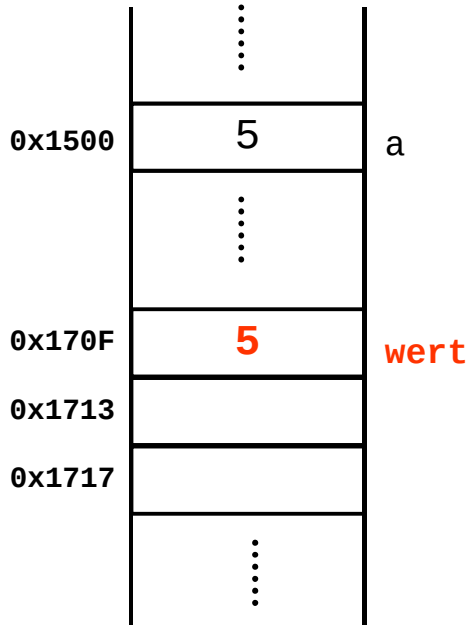


Pointer

- Bei Aufruf einer Funktion wird (bei call by value) der Parameter kopiert

```
int a = 5;
myFunction(a);
...

void myFunction(int wert)
{
  ...
  wert = 7;
}
```

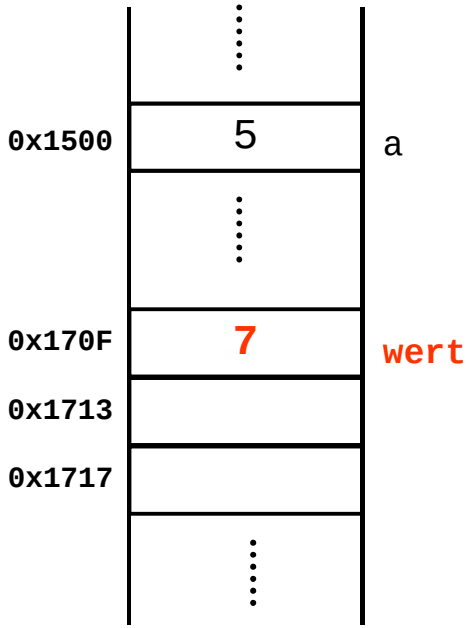


Pointer

- Bei Änderungen wird die Kopie geändert

```
int a = 5;
myFunction(a);
```

```
void myFunction(int wert)
{
  ...
  wert = 7;
}
```

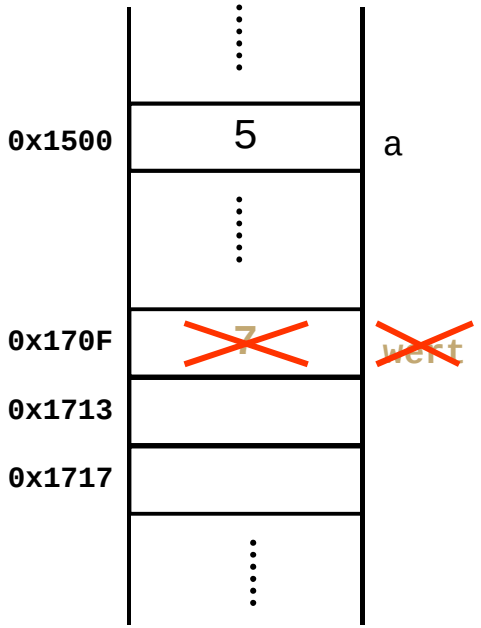


Pointer

- … die nach Ende der Funktion aufhört zu existieren

```
int a = 5;
myFunction(a);
```

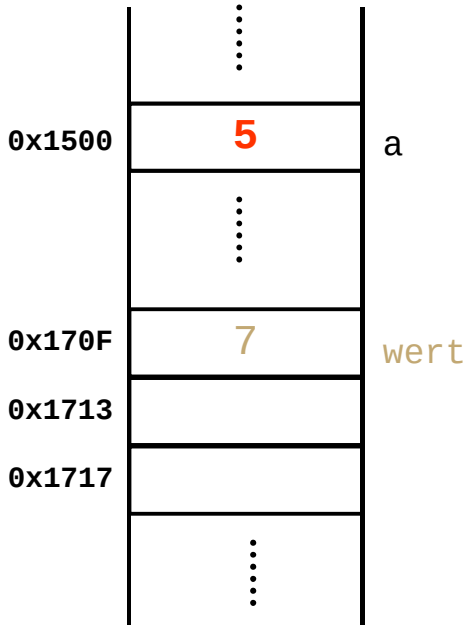
```
void myFunction(int wert)
{
  ...
  wert = 7;
}
```



Pointer

- Und nach dem Aufruf ist alles wie zuvor

```
int a = 5;
myFunction(a);
printf(„%d“,a);
...
```

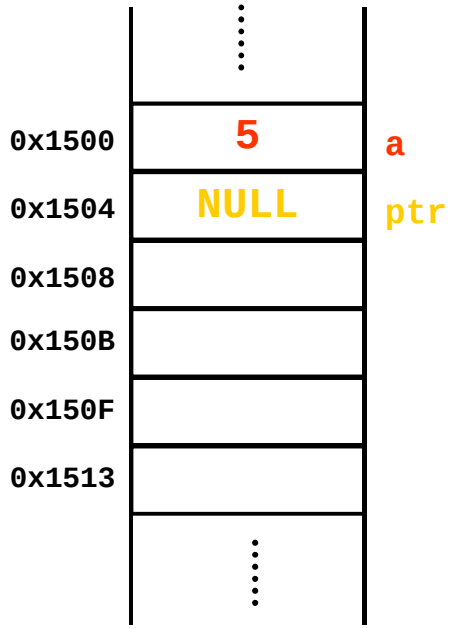


OUTPUT: **5**

Pointer

- Lösung: Call by Reference

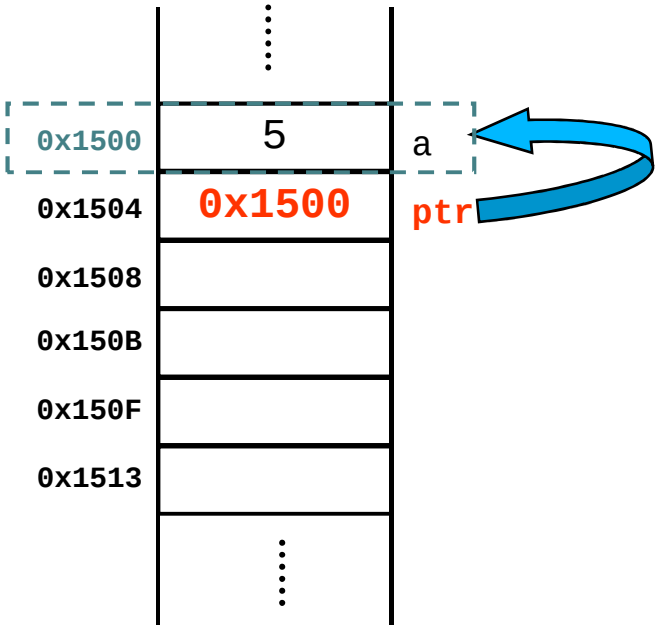
```
int a = 5;
int *ptr = NULL;
ptr = &a;
myFunction(ptr);
```



Pointer

- Ein Pointer wird erstellt

```
int a = 5;
int *ptr = NULL;
ptr = &a;
myFunction(ptr);
```

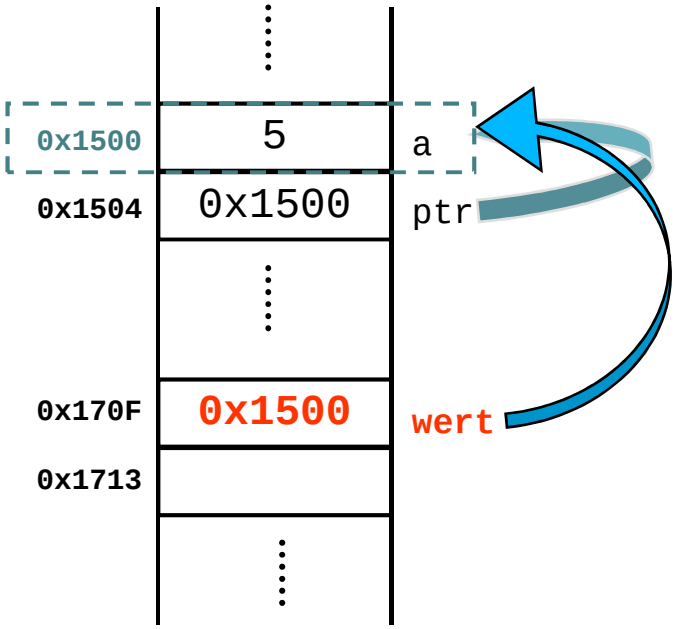


Pointer

- Bei Aufruf wird der Pointer kopiert!

```
int a = 5;
int *ptr = NULL;
ptr = &a;
myFunction(ptr);
```

```
void myFunction(int *wert)
{
  ...
  *wert = 7;
}
```

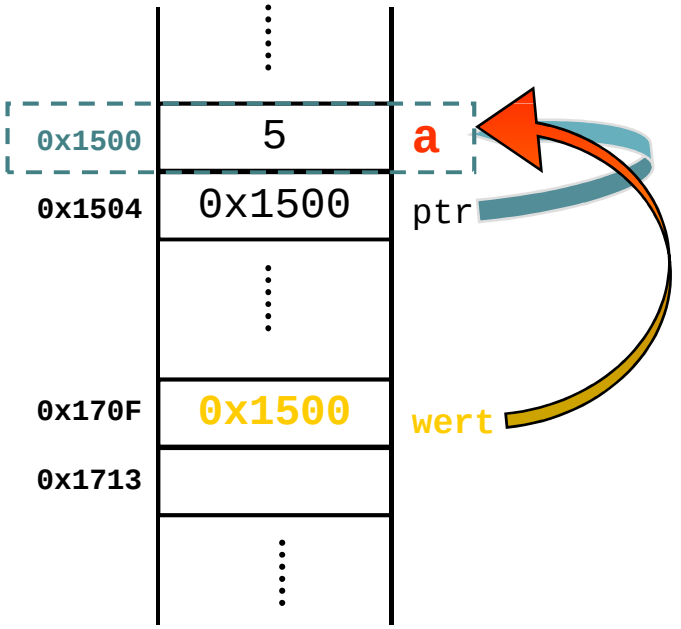


Pointer

- Durch Dereferenzierung kann auf den Wert des Ziels zugegriffen werden

```
int a = 5;
int *ptr = NULL;
ptr = &a;
myFunction(ptr);
```

```
void myFunction(int *wert)
{
  ...
  *wert = 7;
}
```

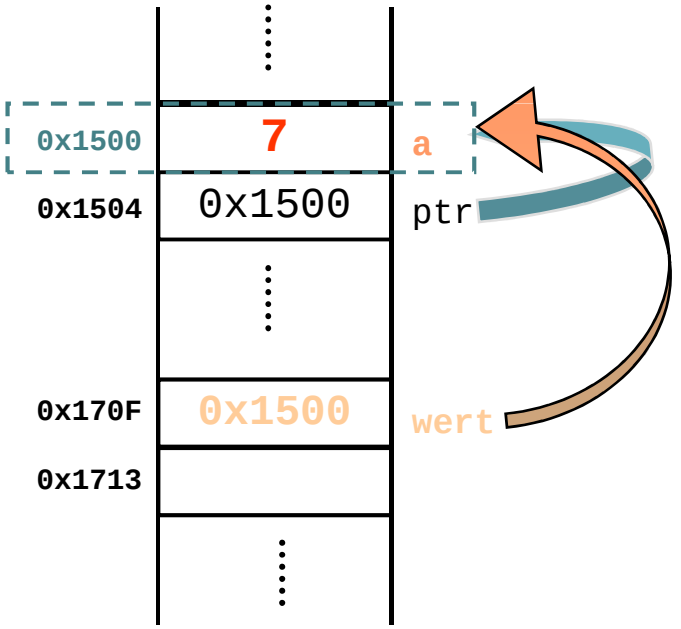


Pointer

- und das Ziel, also das Original, wird verändert

```
int a = 5;
int *ptr = NULL;
ptr = &a;
myFunction(ptr);
```

```
void myFunction(int *wert)
{
  ...
  *wert = 7;
}
```

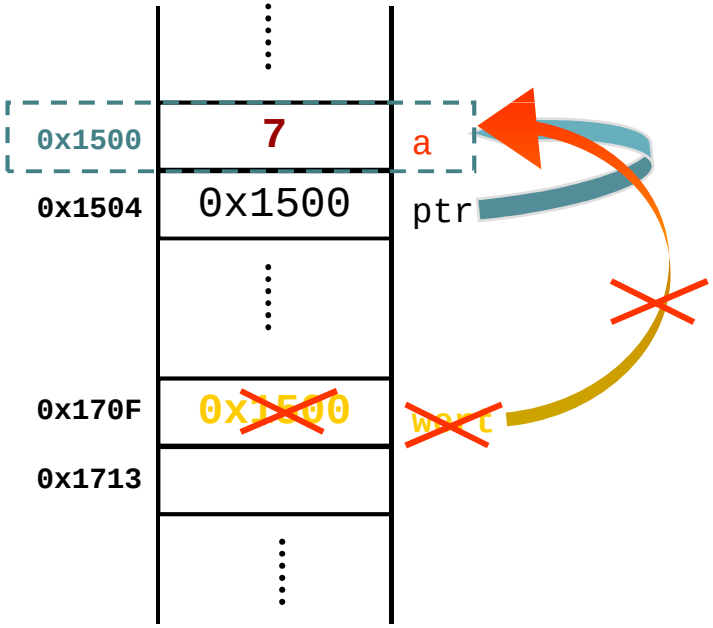


Pointer

- Bei Ende der Funktion, stirbt nur der Zeiger

```
int a = 5;
int *ptr = NULL;
ptr = &a;
myFunction(ptr);
```

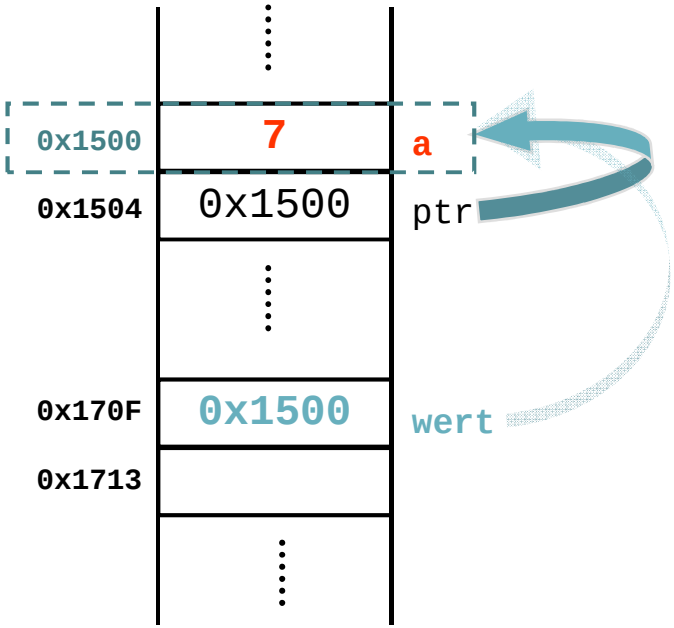
```
void myFunction(int *wert)
{
  ...
  *wert = 7;
}
```



Pointer

- Und nach dem Aufruf ist nichts mehr wie zuvor

```
int a = 5;
int* ptr = NULL;
ptr = &a;
myFunction(ptr);
printf(„%d“,a);
...
```



OUTPUT: 7

Pointer - Fehlerquellen

- Achtung Nullpointer!
- Achtung Zeiger auf Speichermüll!
- Pointerarithmetik
- Siehe Folien ESP Vorlesung 4:
  - 8 / 9
  - 13 / 14

Arrays

Tribute to Matthias Prinz and Georg Kapeller

Arrays

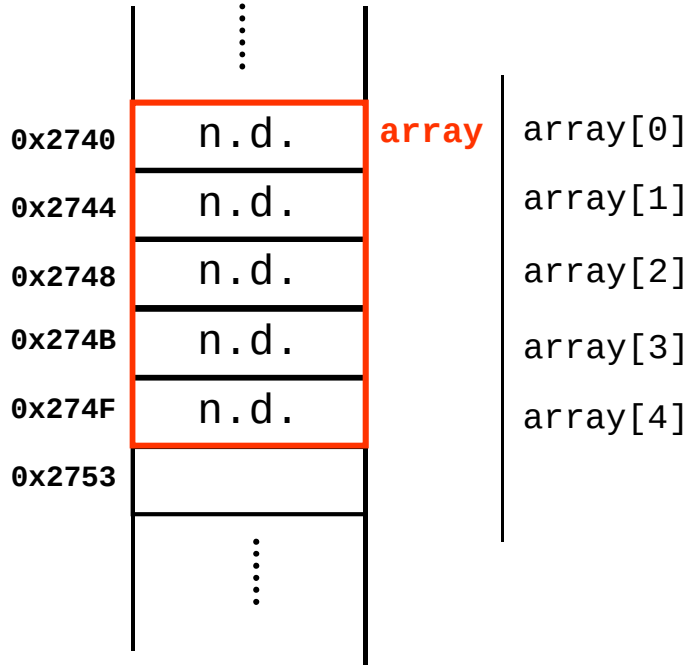
- Ein Array ist ein hintereinander liegendes Feld von Daten im Speicher

```
int array[5];
```

array befindet sich an der Speicherstelle 0x2740

Zugriff auf 2. Element:
array[1]
oder:
***(array + 1)**

Beide Notationen liefern den Wert an der Stelle 0x2744



Arrays - Hinweise

- Menge von Daten gleichen Typs (Berechnung der Adresse durch Pointerarithmetik oder durch den []-Operator)
- Initialisieren!
- Länge wird bei Zugriff über Index nicht überprüft!
- off by one
- Strings mit abschließendem `'\0'`

Mehrdimensionale Arrays

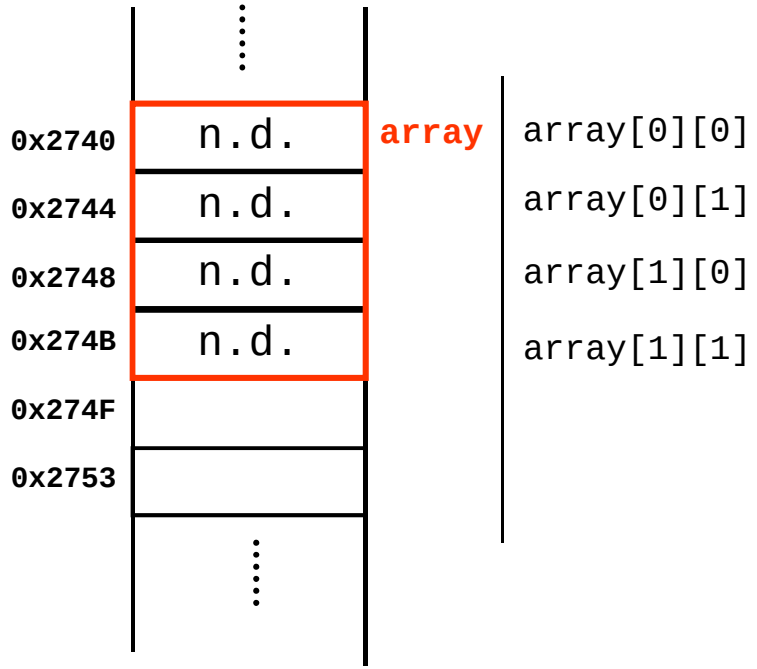
- Auch mehrdimensionale Arrays liegen sequentiell im Speicher:

```
int array[2][2];
```

array befindet sich an der Speicherstelle 0x2740

Zugriff auf 2. Element der 2. Dimension:
array[1][1]
oder:
***(*(array + 1) + 1)**

Beide Notationen liefern den Wert an der Stelle 0x274B



Dynamischer Speicher

```
#include <stdlib.h>

void *malloc (size_t size);
void  free   (void *base_ptr);
void *realloc(void *base_ptr,
              size_t size);

Beispiele: dyn_better.c
```

Dynamischer Speicher - Hinweise

- Neuer Speicherblock ist uninitialisiert
- Freigeben nicht vergessen
- Pointer auf freigegebenen Speicher (Speichermüll!)
- Nach ..alloc() immer auf NULL überprüfen (Speicher voll?)
- realloc(): tmp pointer verwenden
- realloc(): vorsicht in funktionen
- VALGRIND VERWENDEN!!

Testen/Fehlersuche

Tribute to Matthias Prinz and Georg Kapeller

Valgrind

- Auch bei Cygwin dabei
- Auf pluto.tugraz.at installiert
- Bei gcc mit –g kompilieren

- Aufruf
  `valgrind --tool=memcheck --leak-check=yes ./name`

- Beispiel: dyn_errors.c

- Valgrind Tutorial:
  http://www.student.tugraz.at/weinberger/esp/valgrind

fstack-protector

- Nicht bei Cygwin enthalten
- Auf pluto.tugraz.at installiert
- Aufruf

```
gcc -Wall -g -fstack-protector-all -o name name.c
```

- Beispiel am Pluto: stack_smash_test.c
  © by Georg „Smash It“ Kapeller

Fehlersuche

- Programm durchdenken,
  an einem Beispiel: tut es auch wirklich das, was es tun soll

- Debug-Ausgaben mit printf(...)
  Welche Werte haben die Variablen zu bestimmten Zeitpunkten im Programm

- Valgrind
  Von großem Vorteil bei zufälligem Verhalten
  Wichtig: Valgrind-Fehler sind immer(!) Fehler im Programm (auch wenn alles funktioniert) – also ernstnehmen!

- Wenn garnichts hilft: Debugger
  GDB/DDD, VC – ist bei Programmen dieser Komplexität aber normalerweise nicht notwendig! Meist umständlich in der Handhabe...

Fehlersucheprophylaxe

- Macht euch in regelmäßigen Abständen Sicherheitskopien von funktionierenden Versionen!!!
  (weil oft passiert es, dass nach geringfügigen Veränderung auf einmal garnichts mehr geht und ihr auch nicht mehr wisst, was ihr verändert habt)

- Nützlich: http://svn.tugraz.at

Best Practice

- Programm in sinnvolle Funktionen aufteilen
- Sinnvolle Kommentare schreiben
- Variablen sprechende Namen geben
- allocs auf NULL prüfen
- Realloc mit Temp-Pointer
- Pointer nach free() auf NULL setzen
- Variablen initialisieren
- Condingstandard beachten

I/O Redirection

```
./program < input.txt

./program > output.txt

./program 1>output.txt 2>error.log

./program < input.txt > output.txt
```

Bis zum nächsten Mal