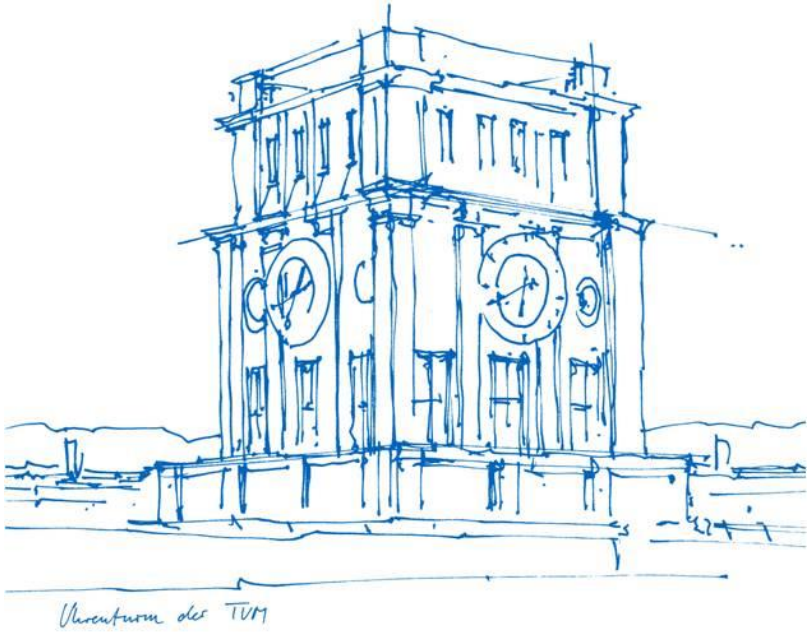
Grundlagenpraktikum: Rechnerarchitektur

SoSe 2024

~ *Danial Arbabi*
danial.arbabi@tum.de



Zulip-Gruppen

Gruppe 29

FR 12:00

https://zulip.in.tum.de/#narrow/stream/2276-GRA-Tutorium---Gruppe-29

Gruppe 32

FR 15:00

https://zulip.in.tum.de/#narrow/stream/2279-GRA-Tutorium---Gruppe-32

Tutoriums-Website

https://home.in.tum.de/~arb/

Disclaimer:
Dies sind keine offiziellen Materialien, somit besteht keine Garantie auf Korrektheit und Vollständigkeit.
Falls euch Fehler auffallen, bitte gerne melden.

Wiederholung

Der C Standard

- C ist standardisiert
  - Wird seit 1990 kontinuierlich weiterentwickelt
  - Dieses Video bezieht sich auf C17 (2017)
- Definiert Anforderungen an konkrete Implementierung des Standards
  - Möglichst rückwärtskompatibel
- Konkrete Implementierung umfasst
  - Compiler
  - Standardbibliothek
  - Betriebssystem
  - und Hardware (Prozessor)
- Unterscheidung von
  - durch den Standard definiertes Verhalten
  - und "implementation-defined behavior"

Grundlegende Datentypen: Integer

Bezeichner	Übliche Größe (LP64)	Garantierte Größe (Standard)
`_Bool`	8 Bit (1 Bit nutzbar)	$\geq$ 1 Bit (1 Bit nutzbar)
`char`	8 Bit	$\geq$ `_Bool` und $\geq$ 8 Bit
`short (int)`	16 Bit	$\geq$ `char` und $\geq$ 16 Bit
`int`	32 Bit	$\geq$ `short` und $\geq$ 16 Bit
`long (int)`	64 Bit	$\geq$ `int` und $\geq$ 32 Bit
`long long (int)`	64 Bit	$\geq$ `long` und $\geq$ 64 Bit

- Größe `char` := 1 Byte

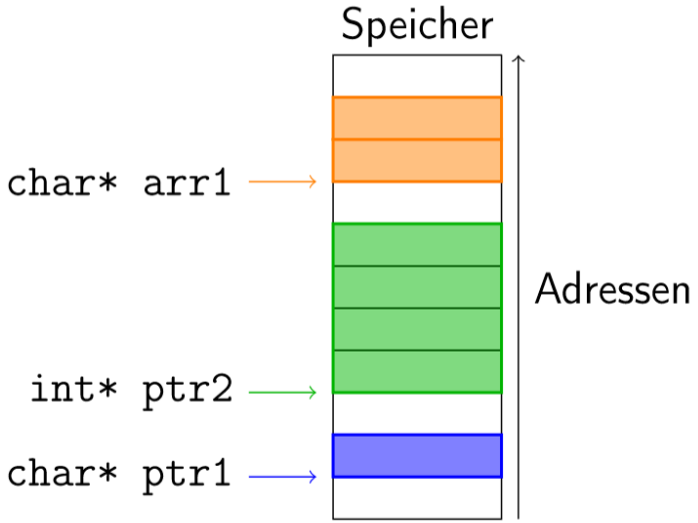
Pointer

Syntax:
<Datentyp>* ptr = <Adresse>;

z.B.:

```c
#include <stdlib.h>

int main(){
    int a = 0;
    int* ptr = &a;          // ptr speichert Adresse von Variable a
    int* ptr = malloc(128); // ptr speichert Heap-Adresse; Achtung: Evtl. Null
}
```

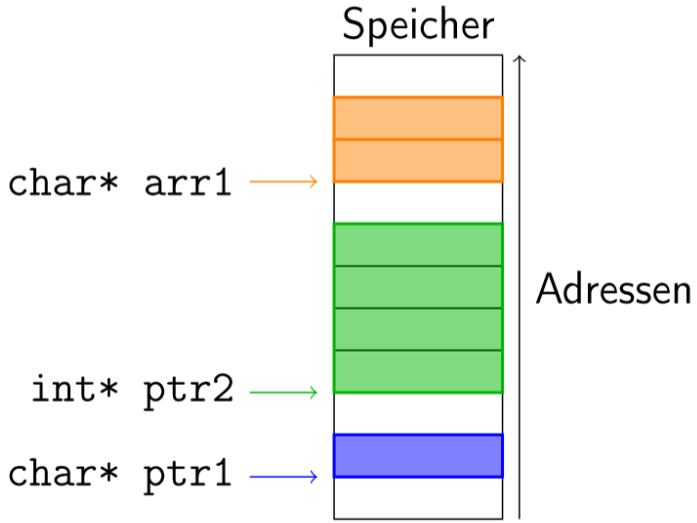


Pointer dereferenzieren

```c
#include <stdlib.h>

int main(){
    int a = 0;
    int* ptr1 = &a;           // ptr speichert Adresse von Variable a
    int* ptr2 = malloc(128); // ptr speichert Heap-Adresse; Achtung: Evtl. Null

    int b = *ptr1;            // ptr1 wird dereferenziert und Wert b zugewiesen
    printf("%d", b);
}
```

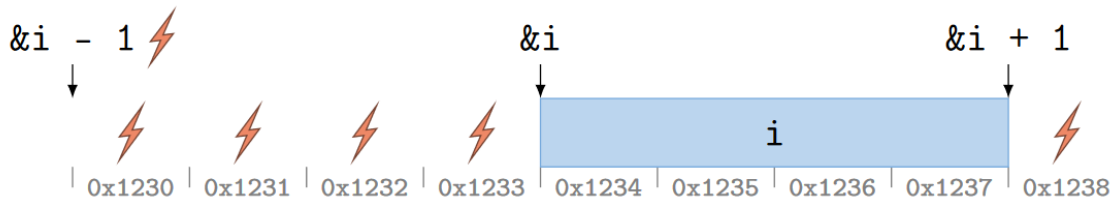


Achtung:

void* gibt keine Information über Datentyp und darf nicht dereferenziert werden!

Pointer-Arithmetik

```
1 // Annahme: sizeof(int) == 4
2 int i = 0;
3 int* i_ptr = &i;  // z.B. 0x1234
4 i_ptr++;          //    -> 0x1238 (= 0x1234 + 4)
5 i_ptr -= 2;       //    -> 0x1230 (= 0x1238 - 8)
6
7 // Achtung: die letzte Operation ist in diesem Fall UB, da der
8 //          resultierende Pointer nicht mehr auf ein Element
9 //          im "Array" zeigt.
```



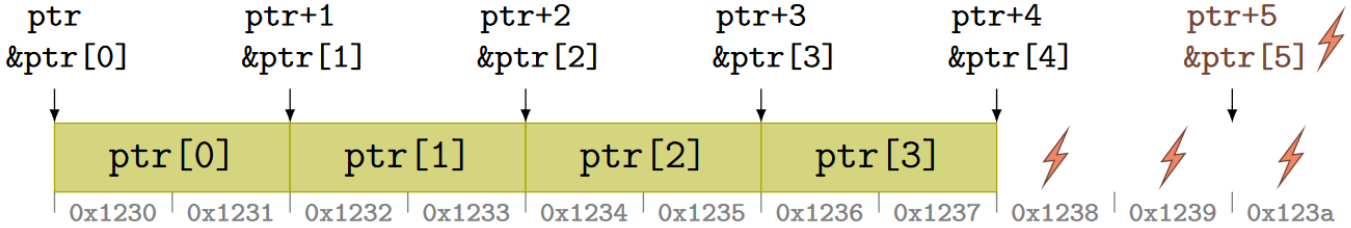
Pointer-Arithmetik: Array-Subscript

- `ptr[0]` ≙ `*ptr`
- `ptr[3]` ≙ `*(ptr + 3)`
- `ptr[`*index*`]` ≙ *index*`[ptr]`
  (Alte Syntax...)

Valide Pointer zeigen immer auf...

- ...ein Objekt (z.B. eine Variable),
- ...eine Stelle in einem Array, oder
- ...an das Ende eines Arrays
  (dann nicht dereferenzierbar)

Andernfalls: Verhalten undefiniert!



Pointer-Casts

- Explizite Typumwandlung
- Neuer Datentyp darf keine strengeres Alignment fordern
- Derenferenzierung von umgewandelten Pointern: *undefined behavior*!
  (Einzige Ausnahme: `char`-Pointer)
- Explizite Casts daher vermeiden

```
1 int* i_ptr = /* ... */;
2 char* c_ptr = (char*) i_ptr;      // Zugriff möglich
3 short* s_ptr = (short*) i_ptr;    // Zugriff undefined behavior
4 long* l_ptr = (long*) i_ptr;      // Cast undefined behavior, da
5                                   // long strengere Alignment-
6                                   // anforderungen hat als int!
```

Makros

```
1 #define NUMBER 42      // Ersetze NUMBER durch  42
2 #define MYNUM  2 + 3   // Ersetze MYNUM  durch  2 + 3
3
4 int a = NUMBER;       // = 42
5 int b = MYNUM * 2;    // = 2 + 3 * 2 = 8 (nicht (!) 10)
6
7 #undef MYNUM
```

#include Direktiven

```
1 #include <system_header.h> // Copy-paste Inhalte von
2                            // system_header.h  an diese Stelle
3
4 #include "local_header.h"  // Copy-paste Inhalte von
5                            // local_header.h  an diese Stelle
```

Header-Files

Achtung:
Hier beim Include Anführungszeichen statt Krokodils-Klammern, wegen eigener Header-File

foo.h:

```c
void foo(void);
```

foo.c:

```c
#include "foo.h"

void foo(void) {
    ...
}
```

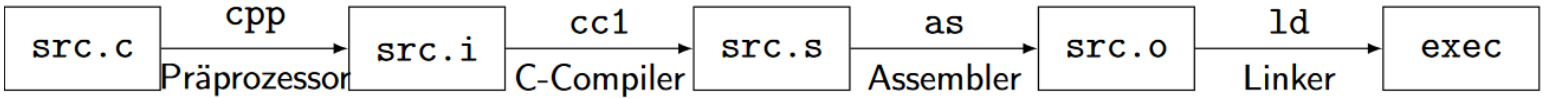
main.c:

```c
#include "foo.h"

int main(void) {
    foo();
    return 0;
}
```

Präprozessor

- Vor dem Kompilieren: *Preprocessing*
- Auflösen von Makros
- Kombination mehrerer Dateien



Aufgaben

Aufgabe
T2.1 Speicherbereiche

Speicherbereiche

Aufgabe 1.1

Betrachten Sie den Abschnitt Sections und lokalisieren die vier oben abgebildeten Abschnitte (.text, .rodata, .data, .bss). Können Sie aus den gezeigten Informationen Rückschlüsse auf den Inhalt ziehen?

```
objdump -wh a.out

a.out:     file format elf64-x86-64

Sections:
Idx Name          Size      Flags
  0 .interp       001c      CONTENTS, ALLOC, LOAD, READONLY, DATA
 14 .text         011f      CONTENTS, ALLOC, LOAD, READONLY, CODE
 15 .fini         0009      CONTENTS, ALLOC, LOAD, READONLY, CODE
 16 .rodata       0011      CONTENTS, ALLOC, LOAD, READONLY, DATA
 19 .init_array   0008      CONTENTS, ALLOC, LOAD, DATA
 20 .fini_array   0008      CONTENTS, ALLOC, LOAD, DATA
 21 .dynamic      01e0      CONTENTS, ALLOC, LOAD, DATA
 22 .got          0028      CONTENTS, ALLOC, LOAD, DATA
 23 .got.plt      0020      CONTENTS, ALLOC, LOAD, DATA
 24 .data         0014      CONTENTS, ALLOC, LOAD, DATA
 25 .bss          0004      ALLOC
 26 .comment      001e      CONTENTS, READONLY
```

Flags

Die einzelnen Flags bedeuten:

- **CONTENTS**: *Die Section hat Inhalte in der Datei (sonst leer).*
- **ALLOC**: *Die Section benötigt Speicher.*
- **LOAD**: *Die Section wird in den Speicher geladen (wenn* **CONTENTS** *vorhanden sind, diese, ansonsten wird mit 0 initialisiert).*
- **READONLY**: *Die Section ist nicht beschreibbar.*
- **CODE**: *Die Section enthält Code und muss ausführbar sein.*
- **DATA**: *Die Section enthält Daten.*

Sections

Hieraus ergibt sich dann die Bedeutung der Sections:

- *`.text`: Programmcode, der Bereich ist* **read-only** *und* **executable** *und kommt aus der Binary.*
- *`.rodata`: Konstante initialisierte globale Variablen, Strings, usw..*
- *`.data`: Initialisierte globale Variablen, i.d.R.* **read-write** *(aber nicht* **executable***) und kommt ebenfalls aus der Binary.*
- *`.bss`: Globale Variablen, die mit 0 initialisiert werden und daher keinen Speicherplatz in der kompilierten Datei benötigen. Ansonsten wie `.data`.*

Heap vs. Stack

Aufgabe 1.2
Was ist der Unterschied zwischen den Speicherbereichen Heap und Stack und welchen Verwendungszweck haben diese?

- **Heap:** *Hier können jederzeit Speicherbereiche alloziiert werden und zu jedem Zeitpunkt wieder freigegeben werden. Jede Allokation geschieht (in C) nur bei einem Aufruf von* `malloc`/`calloc`.
- **Stack:** *Der Stack nimmt lokale Variablen auf und arbeitet nach dem* **LIFO-Prinzip**; *Allokationen werden mit Ende der Funktion wieder freigegeben. (Wieso nämlich?) Die Größe des Stack ist begrenzt (i.d.R 8–16 MiB) und daher nur für kleine Allokationen (wenige kiB) geeignet.*

Heap vs. Stack

Aufgabe 1.3

Betrachten Sie folgendes C-Programm. In welchen der Speicherbereichen werden die einzelnen Variablen alloziiert (.text, .rodata, .data, .bss, Heap, Stack)? Verwenden Sie auch die man-Pages von malloc und alloca.

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

int v0 = 6;
int v1;
const int v2[4] = {1, 3, 3, 7};

int main(int argc, char** argv) {
  int v3 = 5;
  int v4[v3];
  int* v5 = malloc(v3 * sizeof(int));
  if (v5 == NULL) abort(); // Wichtig!!
  int* v6 = alloca(v3 * sizeof(int));
  // ...
}
```

v0	v1	v2	v3	v4	v5	v6
.data	*.bss*	*.rodata*	*Stack*	*Stack*	*Heap*	*Stack*

Aufgabe
T2.2 Speicherzugriff und Pointerarithmetik in C

C Source	Erläuterung
`int v = 5;`	Definition einer normalen Variable
`int* a;`	Deklaration eines Pointers der auf ein (mehrere) `int` zeigen kann
`a = &v;`	Der Pointer wird auf die Adresse von `v` gesetzt
`int q = *a;`	Der Pointer wird *dereferenziert*, d.h. der Wert aus dem Speicher geladen
`*a = 2;`	Der Wert im Speicher auf den `a` zeigt wird auf 2 gesetzt

C Source	Wert der Variable
`int array[4] = { 10, 20, 30, 40 };`	`0xfffe1000` *(beispielhaft!)*
`int val1 = array[0];` `int val2 = array[1];`	*10* *20*

`int val3 = *array;`
`int val4 = (*array)+1;`
`int val5 = *(array+1);`
	10
	11
	20

`int* ptr1 = array;` `int val6 = *ptr1;`	*0xfffe1000* *10*
`int* ptr2 = &array[2];` `int val7 = *ptr2;`	*0xfffe1008* *30*
`int* ptr3 = array + 3;` `int val8 = *ptr3;`	*0xfffe100c* *40*

Ein Array ist lediglich ein Pointer auf einen Speicherbereich – die Länge des Speicherbereiches muss bekannt sein, es gibt keinen automatischen Schutz vor out-of-bounds Zugriffen (sog.* Buffer Overflows*).

Aufgabe
T2.3 Kopieren eines Strings

	`strcpy`	`strncpy`	`stpncpy`	`strlcpy`	`memccpy`
Standardisiert?	*C*	*C*	*POSIX*	*– (BSD)*	*POSIX*
Buffer-Länge spezifizierbar?	—	*Ja*	*Ja*	*Ja*	*Ja*
Rückgabe von String-Ende?	—	—	*Ja*	*Ja*	*Ja*
Schreibt immer NUL-Byte?	*Ja*	—	—	*Ja*	—
Schreibt nur notwendige Bytes?	*Ja*	—	—	*Ja*	*Ja*

Die Funktion `strlcat` ist zwar funktional optimal, aber nicht sehr weit verbreitet und daher nicht standardisiert. Die Funktion `memccpy` ist zur Aufnahme in den C23-Standard vorgesehen.