

Betriebssysteme

Eine Einführung in C

MICHAEL ROITZSCH

20. Oktober 2026

Wieso eine Einführung in C?

- C ist *die* UNIX-Sprache — UNIX, Linux und große Teile des Userlands sind in C geschrieben.
- Im Embedded-Bereich ist C bis heute die dominante Sprache.
- Als Informatik-Studi solltet ihr C mindestens einmal gesehen haben.

Wieso eine Einführung in C?

- C ist *die* UNIX-Sprache — UNIX, Linux und große Teile des Userlands sind in C geschrieben.
- Im Embedded-Bereich ist C bis heute die dominante Sprache.
- Als Informatik-Studi solltet ihr C mindestens einmal gesehen haben.

Im Vergleich

C++ und Rust sind mächtiger — aber auch komplexer, nur an ganz anderen Stellen. C bleibt dagegen überschaubar klein.

Worum es heute geht

- Ziel: C so lesen und schreiben können, wie es für systemnahe Programmierung gebraucht wird — vor allem Pointer und Speicherverwaltung.
- Ihr kennt schon Rust und Python. Vieles überträgt sich direkt, manches ist jetzt die falsche Antwort.

Worum es heute geht

- Ziel: C so lesen und schreiben können, wie es für systemnahe Programmierung gebraucht wird — vor allem Pointer und Speicherverwaltung.
- Ihr kennt schon Rust und Python. Vieles überträgt sich direkt, manches ist jetzt die falsche Antwort.
- Ich werde „macht das nicht“ zu Dingen sagen, die vollkommen legal sind. Das ist Absicht.
- Unterbrecht mich, wenn etwas unklar ist.

Worum es heute geht

- Ziel: C so lesen und schreiben können, wie es für systemnahe Programmierung gebraucht wird — vor allem Pointer und Speicherverwaltung.
- Ihr kennt schon Rust und Python. Vieles überträgt sich direkt, manches ist jetzt die falsche Antwort.
- Ich werde „macht das nicht“ zu Dingen sagen, die vollkommen legal sind. Das ist Absicht.
- Unterbrecht mich, wenn etwas unklar ist.

Kernidee

C vertraut euch. Immer!

Was wir nicht behandeln

- Pointer-Arithmetik jenseits der Grundlagen — Vergleiche, komplexere Allocator-Strukturen.
- Der volle Präprozessor — Conditional Compilation, Token-Pasting, ...
- Unions, Bitfields, variadische Funktionen.
- Mehrdatei-Buildsysteme jenseits eines einzelnen gcc-Aufrufs.
- `fork/exec/wait` und die Semaphore-API — eigene Vorlesungen, die auf diesem Kurs aufbauen.

Das hier ist auch kein Kurs über *Programmieren*. Es ist ein Kurs darüber, wie man das, was ihr schon könnt, in C ausdrückt.

— BLOCK 1 / 4

Vom Quelltext zum Programm

- Hallo, Welt
- Includes und Makros

01



Hallo, Welt

```
1  #include <stdio.h>
3  int main(void)
4  {
5      printf("hallo, welt\n");
6      return 0;
7  }
```

Hallo, Welt

```
1 #include <stdio.h>
3 int main(void)
4 {
5     printf("hallo, welt\n");
6     return 0;
7 }
```

- `#include <stdio.h>` — Zugriff auf `printf`. Mehr dazu gleich.

Hallo, Welt

```
1  #include <stdio.h>
3  int main(void)
4  {
5      printf("hallo, welt\n");
6      return 0;
7  }
```

- `#include <stdio.h>` — Zugriff auf `printf`. Mehr dazu gleich.
- `int main(void)` — der Einstiegspunkt, muss `int` zurückgeben.

Hallo, Welt

```
1  #include <stdio.h>
3  int main(void)
4  {
5      printf("hallo, welt\n");
6      return 0;
7  }
```

- `#include <stdio.h>` — Zugriff auf `printf`. Mehr dazu gleich.
- `int main(void)` — der Einstiegspunkt, muss `int` zurückgeben.
- `return 0;` — Erfolg. Ungleich 0 heißt Fehler.

Compilieren und Ausführen

```
$ gcc -Wall -Wextra -o hello hello.c  
$ ./hello  
hallo, welt
```



- `-Wall -Wextra` — nicht default, aber immer an.
- `-o hello` — Name der ausführbaren Datei; ohne das heißt sie `a.out`.

Compilieren und Ausführen

```
$ gcc -Wall -Wextra -o hello hello.c  
$ ./hello  
hallo, welt
```



- `-Wall -Wextra` — nicht default, aber immer an.
- `-o hello` — Name der ausführbaren Datei; ohne das heißt sie `a.out`.

Best Practice

Compiliert von Anfang an mit `-Wall -Wextra`. Der Compiler findet Fehler, die ihr sonst erst zur Laufzeit seht — oder gar nicht.

Kommandozeilenargumente

Die vollständige Signatur von `main`:

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

- `argc` — Anzahl der Argumente (inklusive Programmname).
- `argv` — die Argumente selbst, als Array von Strings.

Kommandozeilenargumente

Die vollständige Signatur von main:

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

- argc — Anzahl der Argumente (inklusive Programmname).
- argv — die Argumente selbst, als Array von Strings.

```
$ ./programm eingabe.txt ausgabe.txt
```

argv[0]

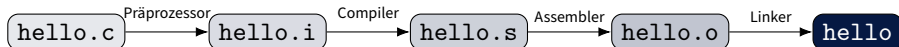
argv[1]

argv[2]

Was `char *argv[]` eigentlich für ein Typ ist, klären wir in Block 2 bei den Pointern.

➔ Doppel-Pointer, Block 2

Wie ein Programm entsteht



`gcc hello.c -o hello` führt alle vier Schritte in einem Aufruf aus. Jeder Schritt ist auch einzeln ansprechbar:

```
$ gcc -E hello.c -o hello.i      # nur Präprozessor
$ gcc -S hello.i -o hello.s      # bis Assembler
$ gcc -c hello.s -o hello.o      # bis Objektdatei
$ gcc hello.o -o hello           # nur Linken
```



Mehrere Dateien

Jede `.c`-Datei wird für sich *compiliert* — zu einer eigenen Objektdaten. Erst der *Linker* fügt sie zu einer ausführbaren Datei zusammen.

```
$ gcc -c util.c -o util.o  
$ gcc -c main.c -o main.o  
$ gcc util.o main.o -o prog
```



Mehrere Dateien

Jede `.c`-Datei wird für sich *compiliert* — zu einer eigenen Objektdatei. Erst der *Linker* fügt sie zu einer ausführbaren Datei zusammen.

```
$ gcc -c util.c -o util.o  
$ gcc -c main.c -o main.o  
$ gcc util.o main.o -o prog
```



Kernidee

„Compilieren“ prüft eine Datei für sich. „Linken“ prüft, ob am Ende alles zusammenpasst.

Wenn der Linker etwas vermisst

Fehlt eine Objektdaten beim Linken, bleibt die Funktion unauffindbar:

Compiler-Fehler

```
/usr/bin/ld: main.o: in function 'main':  
main.c:(.text+0x12): undefined reference to 'util_init'
```

Wenn der Linker etwas vermisst

Fehlt eine Objektdaten beim Linken, bleibt die Funktion unauffindbar:

Compiler-Fehler

```
/usr/bin/ld: main.o: in function 'main':  
main.c:(.text+0x12): undefined reference to 'util_init'
```

Best Practice

Lest bei einem Linker-Fehler zuerst, *welches* Symbol fehlt. Meist fehlt schlicht eine .o-Datei im Aufruf.

Im Vergleich zu Rust und Python

- Rust: `cargo build` erledigt Präprozessor (Makros), Compiler, Assembler und Linker in einem Schritt.
- Python: kein Compile-Schritt — der Interpreter liest und führt direkt aus.
- C macht jede Stufe sichtbar und einzeln aufrufbar. Das ist unbequemer, aber auch weniger magisch.

Im Vergleich

Rusts und Pythons Werkzeuge nehmen euch diese Stufen ab. Cs Werkzeuge zeigen sie euch — das hilft, sobald etwas schiefgeht.

#include

```
1 | #include <stdio.h>      // Systemheader -- Standardsuchpfad
2 | #include "util.h"       // lokaler Header -- relativ zur eigenen Datei
```

#include

```
1 | #include <stdio.h>      // Systemheader -- Standardsuchpfad
2 | #include "util.h"       // lokaler Header -- relativ zur eigenen Datei
```

Der Präprozessor ersetzt die `#include`-Zeile wörtlich durch den Inhalt der angegebenen Datei — vor dem eigentlichen Compilieren.

Im Vergleich

Anders als Rusts `mod/use` oder Pythons `import`: kein echtes Modulsystem, keine Namespaces — nur Textersatz.

Was in einem Header steht

```
1 // util.h
2 int util_init(void);    // Deklaration -- "das gibt es, so sieht es aus"

3 // util.c
4 #include "util.h"
5 int util_init(void) { /* ... */ return 0; } // Definition -- der Code dazu
```

Was in einem Header steht

```
1 // util.h
2 int util_init(void);    // Deklaration -- "das gibt es, so sieht es aus"

3 // util.c
4 #include "util.h"
5 int util_init(void) { /* ... */ return 0; } // Definition -- der Code dazu
```

- Eine Funktion darf mehrfach *deklariert*, aber nur einmal *definiert* werden.
- Jede .c-Datei, die `util.h` inkludiert, bekommt die Deklaration — nicht den Code.



Funktionen, Block 3

#define — Konstanten

```
1 | #define MAX_LEN 128  
2 | char buffer[MAX_LEN];
```

Der Präprozessor ersetzt `MAX_LEN` überall durch `128` — rein textuell, vor dem Compilieren. Es gibt keinen Typ, keine Sichtbarkeitsregel.

#define — Konstanten

```
1 | #define MAX_LEN 128  
2 | char buffer[MAX_LEN];
```

Der Präprozessor ersetzt `MAX_LEN` überall durch `128` — rein textuell, vor dem Compilieren. Es gibt keinen Typ, keine Sichtbarkeitsregel.

Best Practice

Für echte Konstanten mit Typ: `const int max_len = 128;` oder, in neuerem C, `enum { MAX_LEN = 128 };`

Include Guards

Wird ein Header zweimal inkludiert, kommt sein Inhalt zweimal vor — doppelte Deklarationen, Compilerfehler.

```
1  #ifndef UTIL_H
2  #define UTIL_H

4  int util_init(void);

6  #endif // UTIL_H
```

Include Guards

Wird ein Header zweimal inkludiert, kommt sein Inhalt zweimal vor — doppelte Deklarationen, Compilerfehler.

```
1  #ifndef UTIL_H
2  #define UTIL_H
3
4  int util_init(void);
5
6  #endif // UTIL_H
```

Beim zweiten `#include "util.h"` ist `UTIL_H` schon definiert — der Inhalt zwischen `#ifndef` und `#endif` wird übersprungen.

— BLOCK 2 / 4

Typen, Pointer, Speicher

- Basistypen
- Pointer-Grundlagen
- Speicher: Stack, Heap, Doppel-Pointer

02



Grundtypen

- `char`, `short`, `int`, `long`, `long long` — jeweils `signed` oder `unsigned`.
- `float`, `double` — Gleitkommazahlen.
- `void` — „kein Typ“.

Grundtypen

- `char`, `short`, `int`, `long`, `long long` — jeweils `signed` oder `unsigned`.
- `float`, `double` — Gleitkommazahlen.
- `void` — „kein Typ“.

Der C-Standard garantiert nur *Mindestbreiten* — `int` ist „mindestens 16 Bit, üblicherweise 32“. Kein festes Format wie Rusts `i32` oder Pythons beliebig genaue `int`.

Grundtypen

- `char`, `short`, `int`, `long`, `long long` — jeweils `signed` oder `unsigned`.
- `float`, `double` — Gleitkommazahlen.
- `void` — „kein Typ“.

Der C-Standard garantiert nur *Mindestbreiten* — `int` ist „mindestens 16 Bit, üblicherweise 32“. Kein festes Format wie Rusts `i32` oder Python's beliebig genaue `int`.

Im Vergleich

Rusts Typen sind ein Versprechen, das der Compiler prüft. Cs Typen sind ein Versprechen, dem der Compiler meistens einfach vertraut.

sizeof

`sizeof` liefert die Größe eines Typs (oder eines Werts) in Byte — berechnet zur Compile-Zeit, nicht zur Laufzeit.

sizeof

`sizeof` liefert die Größe eines Typs (oder eines Werts) in Byte — berechnet zur Compile-Zeit, nicht zur Laufzeit.

```
1 | printf("%zu\n", sizeof(int));  
2 | printf("%zu\n", sizeof(long));
```

sizeof

`sizeof` liefert die Größe eines Typs (oder eines Werts) in Byte — berechnet zur Compile-Zeit, nicht zur Laufzeit.

```
1 | printf("%zu\n", sizeof(int));  
2 | printf("%zu\n", sizeof(long));
```

```
$ ./sizes  
4  
8
```



sizeof

`sizeof` liefert die Größe eines Typs (oder eines Werts) in Byte — berechnet zur Compile-Zeit, nicht zur Laufzeit.

```
1 | printf("%zu\n", sizeof(int));  
2 | printf("%zu\n", sizeof(long));
```

```
$ ./sizes  
4  
8
```



Gebraucht wird `sizeof` vor allem bei dynamischer Allokation, z. B. `malloc(n * sizeof(int))` für ein Array mit `n` Elementen. ➔ `malloc`, gleich

Unsigned-Overflow

```
1 | unsigned int n = 0;  
2 | n = n - 1;           // kein Fehler, kein Panic  
3 | printf("%u\n", n);  // 4294967295
```

Unsigned-Overflow

```
1 unsigned int n = 0;  
2 n = n - 1;           // kein Fehler, kein Panic  
3 printf("%u\n", n);   // 4294967295
```

Stolperfalle

Unsigned-Underflow läuft klaglos um

Anders als Rusts Debug-Build (Panic bei Overflow) oder Python's `int` (beliebig genau): C rechnet modulo der Typbreite weiter — lautlos.

Signed-Overflow

```
1 | #include <limits.h>
2 | int x = INT_MAX;
3 | x = x + 1;           // kein Wrap-around -- Undefined Behaviour
```

Signed-Overflow

```
1 | #include <limits.h>
2 | int x = INT_MAX;
3 | x = x + 1;           // kein Wrap-around -- Undefined Behaviour
```

Stolperfalle

Undefined Behaviour heißt: alles ist erlaubt

Anders als beim unsigned Wrap-around gibt es hier kein „definiertes“ Ergebnis. Der Compiler darf warnen, auf der Zielplattform umlaufen lassen, abstürzen — oder beim Optimieren schlicht annehmen, dass $x + 1$ nie überläuft, und Code auf dieser Annahme wegoptimieren.

Feste Breiten und Bool

```
1  #include <stdint.h>
2  int32_t  id;          // exakt 32 Bit, signed
3  uint8_t  flag;        // exakt 8 Bit, unsigned

5  #include <stdbool.h>
6  bool done = false;    // C kennt nativ keinen Bool-Typ
```

Feste Breiten und Bool

```
1  #include <stdint.h>
2  int32_t  id;          // exakt 32 Bit, signed
3  uint8_t  flag;        // exakt 8 Bit, unsigned

5  #include <stdbool.h>
6  bool done = false;    // C kennt nativ keinen Bool-Typ
```

`stdint.h` macht Typen mit fester, garantierter Breite verfügbar — auf die könnt ihr euch verlassen. `stdbool.h` erweitert C um `bool` mit `true/false`.

Feste Breiten und Bool

```
1  #include <stdint.h>
2  int32_t  id;          // exakt 32 Bit, signed
3  uint8_t  flag;        // exakt 8 Bit, unsigned

5  #include <stdbool.h>
6  bool done = false;    // C kennt nativ keinen Bool-Typ
```

`stdint.h` macht Typen mit fester, garantierter Breite verfügbar — auf die könnt ihr euch verlassen. `stdbool.h` erweitert C um `bool` mit `true/false`.

Best Practice

Seit C23 (`-std=gnu23`) sind `bool` sowie `true/false` echte Schlüsselwörter — der Include wird optional.

Adressen und Pointer

```
1 | int  n = 42;  
2 | int *p = &n;      // p zeigt auf n  
3 | *p = 7;           // n ist jetzt 7 -- mittels p geschrieben
```

Adressen und Pointer

```
1 | int n = 42;  
2 | int *p = &n;    // p zeigt auf n  
3 | *p = 7;         // n ist jetzt 7 -- mittels p geschrieben
```



Adressen und Pointer

```
1 | int n = 42;  
2 | int *p = &n;      // p zeigt auf n  
3 | *p = 7;           // n ist jetzt 7 -- mittels p geschrieben
```



- `&n` — die Adresse von `n`.
- `int *p` — `p` ist „ein Pointer auf `int`“.

Adressen und Pointer

```
1 | int n = 42;  
2 | int *p = &n;      // p zeigt auf n  
3 | *p = 7;         // n ist jetzt 7 -- mittels p geschrieben
```



- `&n` — die Adresse von `n`.
- `int *p` — `p` ist „ein Pointer auf `int`“.
- `*p` — Dereferenzieren: „das, worauf `p` zeigt“.

NULL

```
1 | int *p = NULL;    // zeigt bewusst auf nichts
2 | *p = 7;           // Absturz
```

NULL

```
1 | int *p = NULL;    // zeigt bewusst auf nichts
2 | *p = 7;           // Absturz
```

Compiler-Fehler

Segmentation fault (core dumped)

Im Vergleich

Eine Rust-`&T` ist ein Versprechen; ein C-Pointer ist nur eine Adresse. Sie kann falsch, veraltet oder NULL sein, und niemand sagt es euch, bis es crasht.

Arrays

```
1 | int values[4] = {10, 20, 30, 40};  
2 | values[2] = 99;           // genau wie Indizierung in Python/Rust
```

Arrays

```
1 | int values[4] = {10, 20, 30, 40};  
2 | values[2] = 99;           // genau wie Indizierung in Python/Rust
```

Aber: feste Größe, zur Compile-Zeit bekannt, **keine** Bounds-Checks.

```
3 | values[10] = 1;    // kompiliert. liest/schreibt irgendwohin.
```



kein
Bounds-Check

Im Vergleich

Rust-Slices und Python-Listen prüfen den Index und werfen Panic/Exception. C prüft nichts.

Pointer-Arithmetik

```
1 | int values[4] = {10, 20, 30, 40};  
2 | int *p = values;    // ein Array "zerfaellt" zu einem Pointer aufs erste Element  
3 | int x = *(p + 2);    // == values[2] == 30
```

Pointer-Arithmetik

```
1 | int values[4] = {10, 20, 30, 40};  
2 | int *p = values;      // ein Array "zerfaellt" zu einem Pointer aufs erste Element  
3 | int x = *(p + 2);     // == values[2] == 30
```

Stolperfalle

$p + 1$ heißt nicht „ein Byte weiter“

$p + 1$ rückt um `sizeof(*p)` Byte vor, nicht um 1 Byte. Bei `int *p` sind das 4 Byte. Das ist die Grundlage für jedes Navigieren durch zusammenhängenden Speicher.

➔ Speicher, gleich

Strings

Ein C-String ist ein `char`-Array, NUL-terminiert (`'\0'` markiert das Ende):

```
1 | char name[6] = "alice";    // 'a','l','i','c','e','\0' -- 6 Byte
```

Strings

Ein C-String ist ein `char`-Array, NUL-terminiert (`'\0'` markiert das Ende):

```
1 | char name[6] = "alice";    // 'a','l','i','c','e','\0' -- 6 Byte
2 | char buf[16];
3 | gets(buf);                // liest beliebig lang, egal wie gross buf ist
4 | char buf[16];
5 | fgets(buf, sizeof(buf), stdin); // liest höchstens 15 Zeichen + '\0'
```

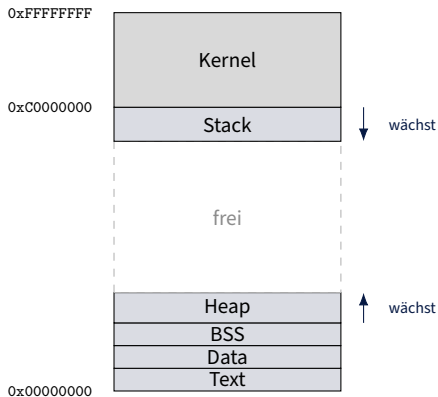


Buffer Overflow

Im Vergleich

Kein `std::string`, kein Python-`str` — ihr verwaltet die Größe selbst.

Speicherlayout eines Prozesses



Text — Code

Data — globale/statische, initialisiert

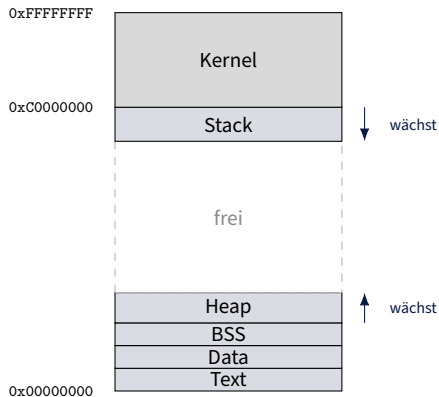
BSS — globale/statische, uninitialisiert

Heap — dynamisch (`malloc`)

Stack — Aufrufe, Locals

Kernel — Kernel-Code und -Daten,
unsichtbar

Speicherlayout eines Prozesses



Text — Code

Data — globale/statiche, initialisiert

BSS — globale/statiche, uninitialisiert

Heap — dynamisch (`malloc`)

Stack — Aufrufe, Locals

Kernel — Kernel-Code und -Daten, unsichtbar

Kernidee

Heap und Stack wachsen aufeinander zu, durch denselben freien Adressraum.

Speicher: Stack und Heap

Zwei Bereiche daraus schauen wir uns jetzt genauer an:

Stack

wächst/schrumpft
automatisch mit
Funktionsaufrufen

Heap

explizit verwaltet —
ihr allokiert, ihr gebt frei

Speicher: Stack und Heap

Zwei Bereiche daraus schauen wir uns jetzt genauer an:

Stack

wächst/schrumpft
automatisch mit
Funktionsaufrufen

Heap

explizit verwaltet —
ihr allokiert, ihr gebt frei

- Lokale Variablen (`int n;`) leben auf dem Stack — sie verschwinden, wenn die Funktion zurückkehrt.
- Heap-Speicher lebt, bis ihr ihn explizit freigibt — oder das Programm endet.

Im Vergleich

Rust entscheidet das meist automatisch (Box, Ownership). Python legt praktisch alles auf einen verwalteten Heap. In C entscheidet ihr — und verwaltet.

malloc **und** free

```
1  #include <stdlib.h>

3  int *p = malloc(sizeof(int));    // Heap-Speicher fuer einen int
4  if (p == NULL) { /* Allokation fehlgeschlagen */ }

6  *p = 42;
7  free(p);                        // genau einmal, wenn p nicht mehr gebraucht wird
```

malloc **und** free

```
1  #include <stdlib.h>

3  int *p = malloc(sizeof(int));    // Heap-Speicher fuer einen int
4  if (p == NULL) { /* Allokation fehlgeschlagen */ }

6  *p = 42;
7  free(p);                        // genau einmal, wenn p nicht mehr gebraucht wird
```

- malloc gibt den Speicher *nicht* initialisiert zurück.
- malloc kann fehlschlagen — dann liefert es NULL.

Im Vergleich

Rusts Drop räumt automatisch auf, Pythons GC auch. C hat beides nicht. Ihr seid der Garbage Collector.

Leak und Double-Free

```
1 | int *p = malloc(sizeof(int));  
2 | return;                // p geht verloren, der Speicher bleibt belegt
```

**Leak**

Leak und Double-Free

```
1 | int *p = malloc(sizeof(int));  
2 | return;                      // p geht verloren, der Speicher bleibt belegt  
  
3 | free(p);  
4 | free(p);                     // zweites free -- Absturz oder schlimmer
```

**Leak****Double-Free**

Stolperfalle

Jede Allokation genau ein `free`

Kein `free`: Leak, der Speicher wird nie mehr frei. Zweimal `free`: undefiniertes Verhalten, oft ein Crash — manchmal erst viel später.

Dangling Pointer

```
1 | int *p = malloc(sizeof(int));  
2 | free(p);  
3 | *p = 7;           // p ist jetzt ein "dangling pointer" -- der Speicher ist frei
```

Dangling Pointer

```
1 | int *p = malloc(sizeof(int));  
2 | free(p);  
3 | *p = 7;           // p ist jetzt ein "dangling pointer" -- der Speicher ist frei
```

Stolperfalle

Use-after-Free

`free` gibt den Speicher frei, löscht aber nicht den Wert von `p`. `p` zeigt weiter auf eine Adresse, die jetzt für etwas anderes vergeben werden kann. Der Compiler warnt davor nicht.

void* — der typlose Pointer

```
1 void *malloc(size_t size);    // die echte Signatur  
  
3 int      *ip = malloc(sizeof(int));    // kein Cast nötig in C  
4 struct point *pp = malloc(sizeof(struct point));
```

void* — der typlose Pointer

```
1 void *malloc(size_t size);    // die echte Signatur  
  
3 int      *ip = malloc(sizeof(int));    // kein Cast nötig in C  
4 struct point *pp = malloc(sizeof(struct point));
```

`void*` ist eine Adresse ohne bekannten Typ — deswegen kann `malloc` denselben Code für jeden Zieltyp verwenden. Ihr gebt dem Compiler den Typ, indem ihr das Ergebnis einer passenden Pointer-Variable zuweist.

Doppel-Pointer: T**

```
1 | int n = 42;  
2 | int *p = &n;           // p zeigt auf n  
3 | int **pp = &p;        // pp zeigt auf p
```

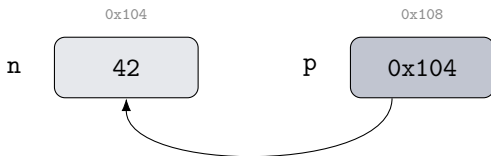
Doppel-Pointer: T**

```
1 | int n = 42;  
2 | int *p = &n;           // p zeigt auf n  
3 | int **pp = &p;         // pp zeigt auf p
```



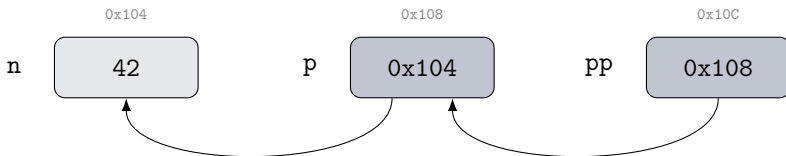
Doppel-Pointer: T**

```
1 | int n = 42;  
2 | int *p = &n;           // p zeigt auf n  
3 | int **pp = &p;         // pp zeigt auf p
```



Doppel-Pointer: T**

```
1 | int n = 42;  
2 | int *p = &n;           // p zeigt auf n  
3 | int **pp = &p;        // pp zeigt auf p
```



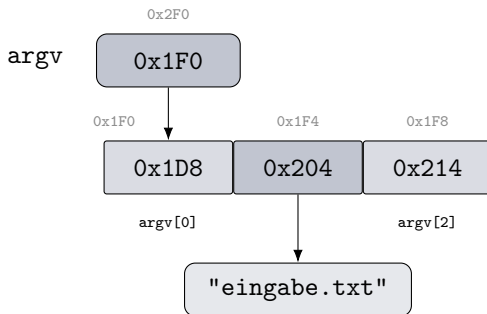
Gebraucht, wenn eine Funktion den Pointer selbst verändern soll — nicht nur das, worauf er zeigt (z. B. eine Freelist, die ein eigener Allocator intern pflegt).

Ein zweites Beispiel: argv

```
1 | int main(int argc, char *argv[]) // char *argv[] ist char **argv
2 | {
3 |     printf("%s\n", argv[1]);      // das erste Kommandozeilenargument
4 | }
```

Ein zweites Beispiel: argv

```
1 int main(int argc, char *argv[]) // char *argv[] ist char **argv
2 {
3     printf("%s\n", argv[1]);      // das erste Kommandozeilenargument
4 }
```



— BLOCK 3 / 4

Programme schreiben

- Funktionen
- Structs
- Kontrollfluss
- stdio und stdlib

03



Funktionen

```
1 | int add(int a, int b)           // Definition: Signatur + Koerper
2 | {
3 |     return a + b;
4 | }
```

Funktionen

```
1 | int add(int a, int b)      // Definition: Signatur + Koerper
2 | {
3 |     return a + b;
4 | }
```

Deklaration (Prototyp) und Definition dürfen getrennt sein — genau das nutzt ihr in Headern. ➔ Includes, Block 1

```
5 | int add(int a, int b);    // Deklaration -- steht typischerweise im Header
```

Call-by-Value

```
1 void increment(int n) { n = n + 1; }  
  
3 int x = 5;  
4 increment(x);  
5 printf("%d\n", x);    // 5 -- increment hat nur seine Kopie veraendert
```

Call-by-Value

```
1 void increment(int n) { n = n + 1; }  
  
3 int x = 5;  
4 increment(x);  
5 printf("%d\n", x);    // 5 -- increment hat nur seine Kopie veraendert
```

Im Vergleich

Jedes Argument wird kopiert. Genau wie in Rust (ohne `&mut`) und Python bei unveränderlichen Typen — nur dass C *keine* Alternative ohne Pointer anbietet.

Einen Pointer übergeben

```
1 void increment(int *n) { *n = *n + 1; }  
  
3 int x = 5;  
4 increment(&x);  
5 printf("%d\n", x);      // 6 -- increment hat ueber den Pointer geschrieben
```

Einen Pointer übergeben

```
1 void increment(int *n) { *n = *n + 1; }  
  
3 int x = 5;  
4 increment(&x);  
5 printf("%d\n", x);    // 6 -- increment hat ueber den Pointer geschrieben
```

Das ist der einzige Weg in C, den Aufrufer etwas verändern zu lassen — und genau das Muster, das `scanf(&n)` später benutzt. ➡ `stdio`, später in Block 3

Was C nicht hat

- Keine Default-Argumente — jeder Aufruf nennt alle Parameter.
- Kein Overloading — ein Funktionsname, eine Signatur.

Was C nicht hat

- Keine Default-Argumente — jeder Aufruf nennt alle Parameter.
- Kein Overloading — ein Funktionsname, eine Signatur.

```
1 | void nothing_in(void);           // void heisst hier "keine Parameter"  
2 | void nothing_out(int x) { }     // void heisst hier "kein Rueckgabewert"
```

Best Practice

Schreibt immer den vollen Prototyp, inklusive `void` für „keine Parameter“.
`int f()` ohne `void` ist in C etwas anderes, als die meisten erwarten.

struct

Ein `struct` bündelt Werte unterschiedlichen Typs unter einem Namen.

```
1 struct point {  
2     int x;  
3     int y;  
4 };
```

struct

Ein `struct` bündelt Werte unterschiedlichen Typs unter einem Namen.

```
1 struct point {  
2     int x;  
3     int y;  
4 };
```

Direkt

```
5 struct point p = {1, 2};  
6 p.x = 3;
```

Über Pointer

```
7 struct point *pp = &p;  
8 pp->x = 3;
```

Structs als Parameter

```
1 | void move(struct point p) { p.x += 1; }    // p ist eine Kopie -- call by value
```

Structs als Parameter

```
1 | void move(struct point p) { p.x += 1; }    // p ist eine Kopie -- call by value
```

Stolperfälle

Große Structs werden ganz kopiert

Jeder Aufruf kopiert das komplette Struct. Bei großen Structs: einen Pointer übergeben (`struct point *p`), wie im Abschnitt „Funktionen“.

if, while, for

```
1  if (x > 0) { ... } else { ... }  
  
3  while (x > 0) { x--; }  
4  do { x--; } while (x > 0);  
  
6  for (int i = 0; i < 10; i++) { ... }
```

if, while, for

```
1  if (x > 0) { ... } else { ... }  
  
3  while (x > 0) { x--; }  
4  do { x--; } while (x > 0);  
  
6  for (int i = 0; i < 10; i++) { ... }
```

Syntaktisch fast identisch zu Rust und Python — `do-while` ist der einzige Neuzugang: der Rumpf läuft mindestens einmal, die Bedingung steht am Ende.

Wahr und Falsch

```
1 | if (42)      { ... } // wahr -- jeder Wert ungleich 0 ist wahr
2 | if (0)       { ... } // falsch
3 | if (p)       { ... } // wahr, wenn p nicht NULL ist
```

Wahr und Falsch

```
1 | if (42)      { ... } // wahr -- jeder Wert ungleich 0 ist wahr
2 | if (0)       { ... } // falsch
3 | if (p)       { ... } // wahr, wenn p nicht NULL ist
```

Im Vergleich

Kein `bool` zwingend erforderlich — anders als Rust, das nur echte `bool`-Ausdrücke in `if` zulässt.

switch/case

```
1 switch (level) {  
2     case 0: printf("debug\n");  
3     case 1: printf("info\n");    // faellt durch, wenn level == 0 ist  
4     default: printf("unknown\n");  
5 }
```



fehlendes break

Ohne `break` läuft die Ausführung in den nächsten `case` weiter — *by design*.

switch/case

```
1 switch (level) {  
2     case 0: printf("debug\n"); break;  
3     case 1: printf("info\n"); break;  
4     default: printf("unknown\n");  
5 }
```

Ohne `break` läuft die Ausführung in den nächsten `case` weiter — *by design*.

switch/case

```
1 switch (level) {  
2     case 0: printf("debug\n"); break;  
3     case 1: printf("info\n"); break;  
4     default: printf("unknown\n");  
5 }
```

Ohne `break` läuft die Ausführung in den nächsten `case` weiter — *by design*.

Stolperfalle

Rusts `match` ist exhaustive, Cs `switch` ist beides nicht

Rusts `match` fällt nie durch und erzwingt alle Fälle beim Compilieren. Cs `switch` tut weder das eine noch das andere.

printf

```
1 | int n = 42;  
2 | printf("Wert: %d, Adresse: %p\n", n, &n);
```

printf

```
1 | int n = 42;  
2 | printf("Wert: %d, Adresse: %p\n", n, &n);
```

```
$ ./demo
```

```
Wert: 42, Adresse: 0x7ffee2a1b05c
```



printf

```
1 | int n = 42;  
2 | printf("Wert: %d, Adresse: %p\n", n, &n);
```

```
$ ./demo
```

```
Wert: 42, Adresse: 0x7ffee2a1b05c
```



Zeichen	Typ	Zeichen	Typ
%d	int	%s	String
%u	unsigned	%p	Pointer
%f	double	%zu	size_t

Format-String und Argument müssen zusammenpassen

```
1 | double x = 3.5;  
2 | printf("%d\n", x);    // undefiniertes Verhalten -- kompiliert trotzdem
```



**Format-
Mismatch**

Format-String und Argument müssen zusammenpassen

```
1 | double x = 3.5;  
2 | printf("%d\n", x);    // undefiniertes Verhalten -- kompiliert trotzdem
```

 **Format-Mismatch**

Mit `-Wall` warnt der Compiler bei literalen Format-Strings:

Compiler-Fehler

```
warning: format '%d' expects argument of type 'int',  
        but argument 2 has type 'double' [-Wformat=]
```

Anders als Rusts `format!` oder Python's `f-Strings`: ohne `-Wall` prüft nichts, ob Format und Argument zusammenpassen.

scanf

```
1 | int n;  
2 | scanf("%d", &n);    // &n -- scanf braucht die Adresse, um zu schreiben
```

scanf

```
1 | int n;  
2 | scanf("%d", &n);    // &n -- scanf braucht die Adresse, um zu schreiben
```

Genau das Muster von vorhin: ein Pointer übergeben, damit die Funktion eure Variable verändern kann. ➔ Pointer an Funktionen übergeben, Block 3

Die Format-String-Syntax (%d, %s, %f, ...) ist dieselbe wie bei printf.

scanf-Rückgabewert prüfen

```
1 | int n;  
2 | scanf("%d", &n);           // Eingabe war "abc"? n bleibt undefiniert.  
3 | printf("%d\n", n);
```

**ungeprüft**

scanf-Rückgabewert prüfen

```
1 int n;  
2 scanf("%d", &n);           // Eingabe war "abc"? n bleibt undefiniert.  
3 printf("%d\n", n);
```

**ungeprüft**

```
4 int n;  
5 if (scanf("%d", &n) != 1) {  
6     fprintf(stderr, "ungueltige Eingabe\n");  
7 }
```

Best Practice

scanf gibt zurück, wie viele Werte erfolgreich gelesen wurden. Prüft das — niemand sonst tut es für euch.

Dateien: fopen, fprintf

```
1 FILE *f = fopen("out.txt", "w");  
2 if (f == NULL) { /* Fehlerbehandlung */ }  
  
4 fprintf(f, "Ergebnis: %d\n", 42);  
5 fclose(f);
```

Dateien: fopen, fprintf

```
1 FILE *f = fopen("out.txt", "w");  
2 if (f == NULL) { /* Fehlerbehandlung */ }  
  
4 fprintf(f, "Ergebnis: %d\n", 42);  
5 fclose(f);
```

fopen gibt NULL zurück, wenn etwas schiefgeht — Datei existiert nicht, keine Rechte, ...
Immer prüfen, bevor f benutzt wird.

Dateien lesen

```
1 FILE *f = fopen("out.txt", "r");  
2 int value;  
3 fscanf(f, "Ergebnis: %d", &value);  
4 fclose(f);
```

Dateien lesen

```
1 FILE *f = fopen("out.txt", "r");  
2 int value;  
3 fscanf(f, "Ergebnis: %d", &value);  
4 fclose(f);
```

```
$ ./write_then_read  
Ergebnis: 42
```



Best Practice

`fopen`, `fclose` — genau wie bei `malloc/free`: jede erfolgreiche Öffnung braucht genau ein Schließen.

— BLOCK 4 / 4

Abschluss

— Abschluss

04



Die Kurzversion

- Ein Pointer ist nur eine Adresse — nichts prüft, ob sie gültig ist.
- Heap-Speicher verwaltet ihr selbst: jedes `malloc` bekommt genau ein `free`.
- `switch` fällt durch, Arrays kennen ihre Größe nicht.
- Ein Pointer als Parameter ist auch die Art, wie Funktionen etwas an den Aufrufer zurückgeben.

Kernidee

C vertraut euch — und das heißt: die Kontrolle (und die Verantwortung) liegt bei euch.

Was wir übersprungen haben

- Pointer-Arithmetik jenseits der Grundlagen — Vergleiche, Differenzen zwischen Pointern.
- Der Rest des Präprozessors — Conditional Compilation, Token-Pasting.
- Unions, Bitfields, variadische Funktionen.
- Mehrdatei-Buildsysteme jenseits eines einzelnen gcc-Aufrufs.

Schlagt das bei Bedarf nach — ihr wisst jetzt, wonach ihr suchen müsst.

Wo ihr weiterlest

- **cppreference.com** — auch für C eine gute Referenz, trotz des Namens.
- **Modern C**, Jens Gustedt — frei online, von den Grundlagen bis zu modernen C-Idiomen.
- Die **Man-Pages** — `man 3 printf`, `man 3 malloc`, `man 2 open`.

Und wenn der Compiler meckert: lest zuerst den *ersten* Fehler, behebt den, compiliert neu. Der Rest sind meist dessen Folgefehler.

Fragen?

— ZUSATZMATERIAL

Backup

Vertiefendes und Optionales für Rückfragen.

Exkurs: Funktionsartige Makros

```
1 | #define SQUARE(x) x * x  
2 | int y = SQUARE(1 + 2);
```

// expandiert zu $1 + 2 * 1 + 2 = 5$, nicht 9 ohne Klammern



Exkurs: Funktionsartige Makros

```
1 | #define SQUARE(x) x * x
2 | int y = SQUARE(1 + 2);      // expandiert zu 1 + 2 * 1 + 2 = 5, nicht 9 ohne Klammern

3 | #define SQUARE(x) ((x) * (x))
4 | int y = SQUARE(1 + 2);      // ((1 + 2) * (1 + 2)) = 9
```



Stolperfalle

Makros sind Textersatz, keine Funktionen

Jedes Argument und der ganze Ausdruck gehören in Klammern. Und: ein Argument, das mehrfach im Makro vorkommt, wird auch mehrfach ausgewertet.

enum

```
1 | enum color { RED, GREEN, BLUE };    // benannte Integer, Start bei 0
3 | enum color c = GREEN;                // c ist 1
```

enum

```
1 | enum color { RED, GREEN, BLUE };    // benannte Integer, Start bei 0
3 | enum color c = GREEN;                // c ist 1
```

RED, GREEN, BLUE sind letztlich `int`-Konstanten — 0, 1, 2.

enum

```
1 | enum color { RED, GREEN, BLUE };    // benannte Integer, Start bei 0
3 | enum color c = GREEN;                // c ist 1
```

RED, GREEN, BLUE sind letztlich `int`-Konstanten — 0, 1, 2.

```
4 | enum color c = 42;    // kompiliert -- 42 ist kein benannter Wert
```



C-enum

enum

```
1 | enum color { RED, GREEN, BLUE };    // benannte Integer, Start bei 0
3 | enum color c = GREEN;                // c ist 1
```

RED, GREEN, BLUE sind letztlich `int`-Konstanten — 0, 1, 2.

```
4 | enum color c = 42;    // kompiliert -- 42 ist kein benannter Wert
```



C-enum

Keine Exhaustiveness-Prüfung im `switch`, keine assoziierten Daten — anders als Rust-Enums (Tagged Unions) oder Pythons `Enum`-Klasse.

Kernidee

Ein C-enum ist ein `int` mit Namensschild.