Kapitel 09 - Zeiger

Peter Ulbrich

AG Systemsoftware

Veranstaltungswebseite

Einleitung

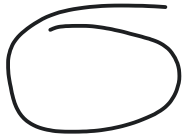
- Aktueller Stand: **Alle** Datenobjekte werden direkt verwendet
- Bei Funktionen: Erstellen einer Kopie
  - Problematisch bei großen `struct`-Typen
  - Außerdem: Wertetausch zweier Parameter so nicht möglich
- Globale Variablen sind ebenfalls problematisch

```c
#include <stdint.h>
#include <stdio.h>
struct Person {
    uint32_t geb_jahr;
    uint32_t geb_monat;
    uint32_t geb_tag;
    char name[64];
};

int main() {
    struct Person alice;
    printf("%lu\n", sizeof(alice));
    return 0;
}
```

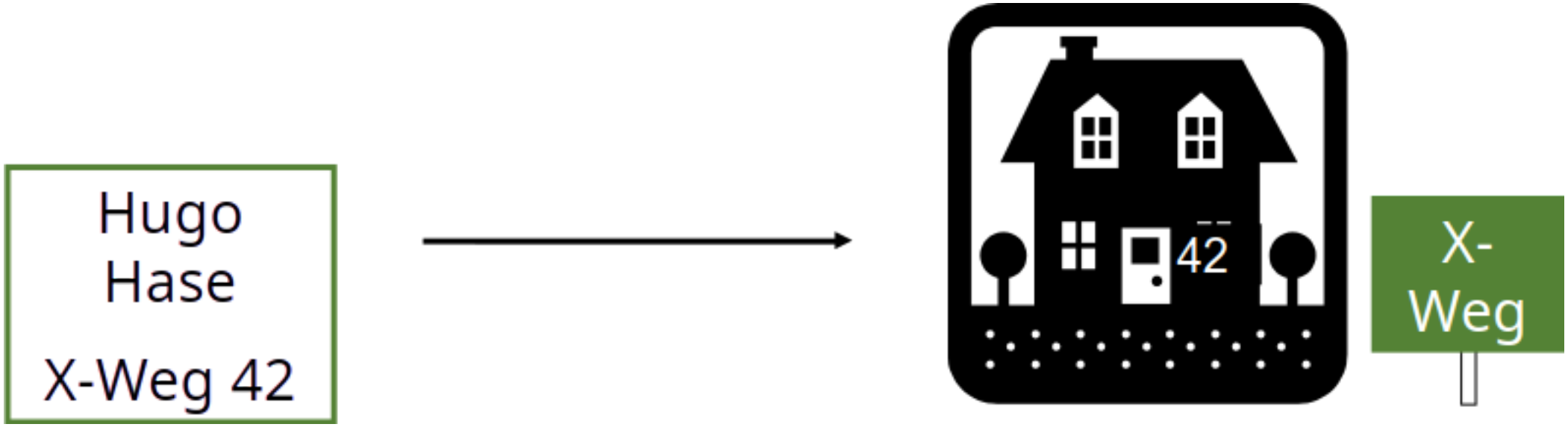
Was tun?

- Die Lösung sind **Zeiger**!



Beispiel – Visitenkarte

„Hugo Hase gibt euch seine Visitenkarte”



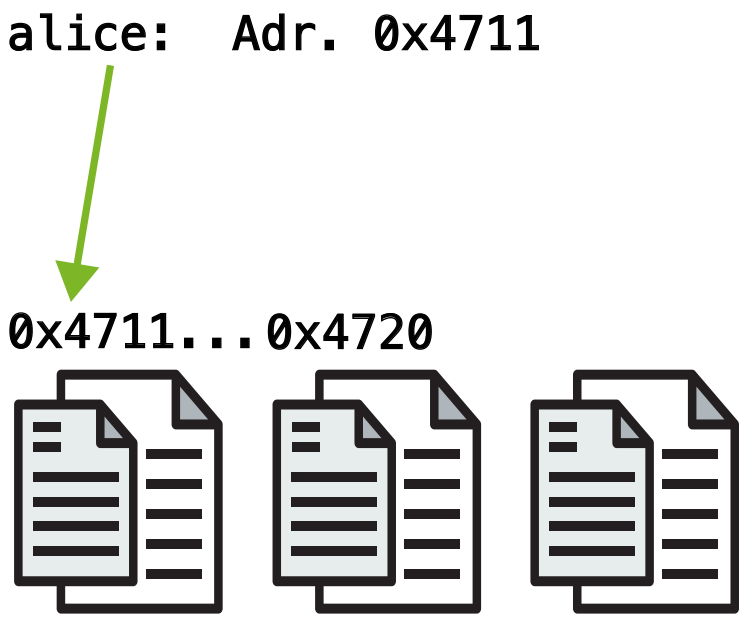
Zeiger: Gibt Adresse an

Adresse: Enthält Haus von Hugo Hase

Zeiger

- Zeiger referenziert Ort im Hauptspeicher
  - referenziert = verweist auf oder zeigt auf
  - Speichert die Adresse
- Wie eine Seitenangabe im Inhaltsverzeichnis

  Eine Zeile mit Seitenverweis anstatt zig Seiten an Inhalt
- Wichtig: Zeiger in C sind ebenfalls Variablen
  - Unterschied zu bisherigen Variablen: *Adresse* (positive Ganzzahl) ist der Wert des Zeigers
- Notation: `int * ptr;` (Zeiger auf einen `int`)



Vorteil von Zeigern

- Größe eines Zeigers ist konstant, aber abhängig von Rechnerarchitektur
- Speichergröße
  - 4 Bytes → 32-Bit-Architektur
  - 8 Bytes → 64 Bit-Architektur
- Adresse, auf die gezeigt wird, kann an beliebiger Stelle stehen
- Bei Funktionen
  - Übergabe von Zeigern anstatt Kopie
  - Arbeitet über Zeiger auf Original

Referenzierung

- Zuweisung einer Adresse ist unkompliziert möglich

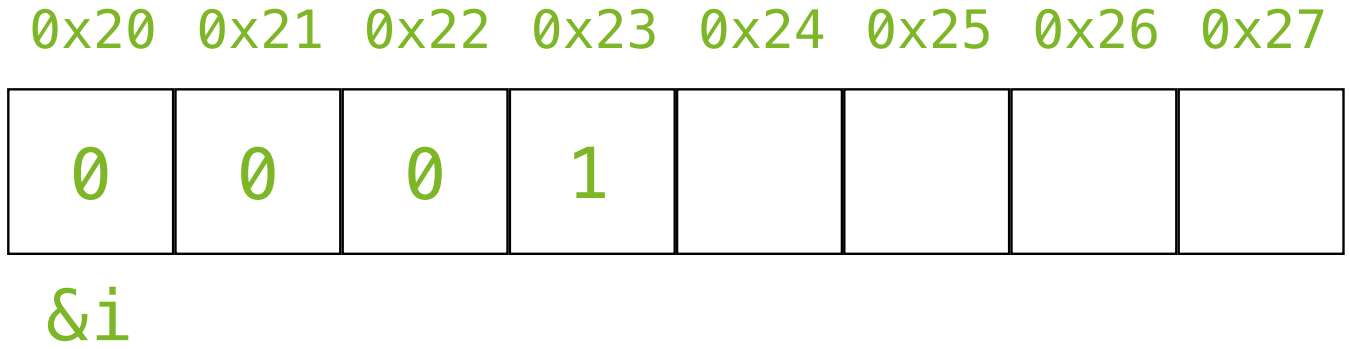
  → Referenzierungsoperator &

- Achtung: Nicht zu verwechseln mit dem binären **UND**!

- Beliebter Fehler: Referenzieren wurde vergessen

  → Wert der Variable wird Adresswert

```
int i = 1;          // An Adresse 0x20
int * ptr = &i;     // Wert von ptr: 0x20
```



Referenzierung

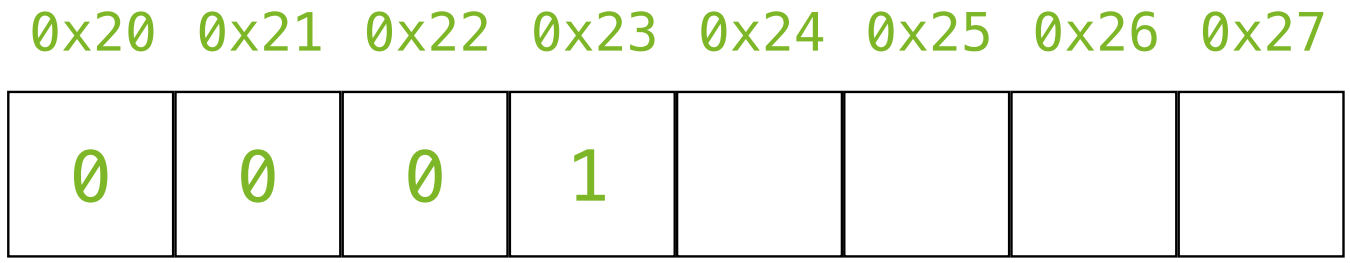
C Run

```c
#include <stdio.h>
int main() {
    int i = 1;
    int * ptr = &i; // Stack-Adresse von i
    printf("int: %p\nptr: %p\n", &i, ptr);
    return 0;
}
```

Dereferenzierung

- Zugriff auf den Wert ist ebenso einfach → Dereferenzierungsoperator *
- Verwechselungsgefahr: Multiplikationsoperator oder Pointer-Stern
- Beliebter Fehler: Dereferenzieren wurde vergessen → Adresse anstatt Wert
- **Wichtig**: Typ einer Zeigervariable bestimmt die Anzahl der gelesenen Bytes ab der Adresse des Zeigers

```
int i = 1;
int * ptr = &i;
*ptr;
```

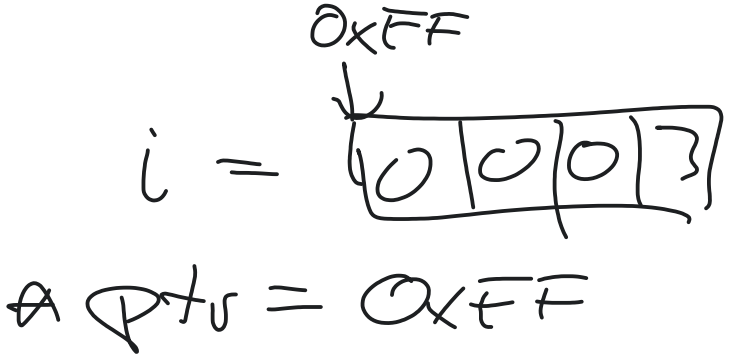


`*ptr == 0x00 00 00 01`

Dereferenzierung

C Run

```c
#include <stdio.h>
int main() {
    int i = 1;
    int * ptr = &i;
    *ptr = 3;
    printf("int: %d\nptr: %d\n", i, *ptr);
    return 0;
}
```



Zeiger als Funktionsparameter

- Mit Zeigern Problem des Wertetauschs (Kap. 4) lösbar
- Bisher: Parameterübergabe als Kopie → Call by Value

```
int add(int sum1, int sum2) { return sum1 + sum2; };
int sum = add(1, 2);
```

- Funktionsparameter können auch Zeiger sein → Call by Reference

```
int add(int * sum1, int * sum2) { return *sum1 + *sum2; };
int sum = add(1, 2);
```

- Deklaration nun mit Zeigern anstatt mit elementaren Variablen

Zeiger als Funktionsparameter

Alt: Wertetausch mit Kopien

(By Value)

C Run ▶

```c
#include <stdio.h>
void swap_values(int one, int two) {
    int temp;
    temp = one;
    one = two;
    two = temp;
}
int main() {
    int a = 1, b = 2;
    swap_values(a, b); // Geht nicht
    printf("a: %d, b: %d\n", a, b);
    return 0;
}
```

Neu: Wertetausch mit Zeigern

(By Reference)

C Run ▶

```c
#include <stdio.h>
void swap_values(int* one, int* two) {
    int temp;
    temp = *one;
    *one = *two;
    *two = temp;
}
int main() {
    int a = 1, b = 2;
    swap_values(&a, &b);
    printf("a: %d, b: %d\n", a, b);
    return 0;
}
```

Wertebereiche

- Zeiger erweitern die Möglichkeiten enorm
- Bringen gleichzeitig neue Probleme
  - Zeigerparameter **müssen** nicht initialisiert sein
  - Funktion kann das nicht erkennen

→ **Massives Problem!**

```c
#pragma GCC diagnostic push
#pragma GCC diagnostic ignored "-Wuninitialized"
#include <stdio.h>
void swap_values(int * one, int * two) {
    int temp;
    temp = *one;
    *one = *two;
    *two = temp;
}
int main() {
    int a = 1, b = 2;
    int *pointer_a = &a;
    int *pointer_b; // Nicht initialisiert
    swap_values(pointer_a, pointer_b);
    printf("a: %d, b: %d\n", a, b);
    return 0;
}
```

Zeigerwerte

Welchen Wert hat dieser Zeiger nach der Deklaration? (64-Bit-Architektur)

C Run ▶

```c
#pragma GCC diagnostic push
#pragma GCC diagnostic ignored "-Wuninitialized"
#include <stdio.h>
int main() {
    int *ptr;
    printf("Adress: %p", ptr);
    return 0;
}
```

A: *Kleinste* Adresse (0x00)

B: *Größte* Adresse ($2^{64} - 1$)

C: Irgendein Wert

Nullzeiger

- „Klassischer“ Nullzeiger (*Null Pointer*) verweist auf Speicheradresse `0x0`
  - Gilt per Konvention als „sicherer“

    → Standardwert für einen nicht initialisierten Zeiger
    - Problem: Muss **immer** händisch zugewiesen werden
    - Programmierer:innen nicht zur Initialisierung oder zur Prüfung verpflichtet
  - Zugriff auf Adresse `0x0` immer noch möglich

    → Absturz bei Zugriff

Sir Tony Hoare (Informatiker)

Bekannt für: Quicksort, Hoare-Kalkül, **Nullreferenz**

„I call it my billion-dollar mistake. […] I couldn’t resist the temptation to put in a null reference, simply because it was so easy to implement. This has led to innumerable errors, vulnerabilities, and system crashes […]“

Quelle: Vortrag InfoQ.com (2009)

Initialisierungsprüfung

- Zeiger können `NULL`/ `nullptr` sein
  - `NULL` häufig als `#define NULL 0` implementiert (*suboptimal*)
  - `nullptr` seit C++11 bzw. C23 (*spezielle Bedeutung*)
  - `nullptr` ist **immer** zu bevorzugen, sofern verfügbar
- Guter Ton: Zeiger *vor* Verwendung überprüfen

  → Defensives Programmieren

```
void foo(int * ptr) {
    if (ptr == nullptr) {
        return;
    }
    else {/* ... */ }
}
```

Probleme von Zeigern

Was aber, wenn der Zeiger nicht initialisiert ist?

- Verhalten einer Funktion dokumentieren, einschließlich erwarteter Parameterwerte

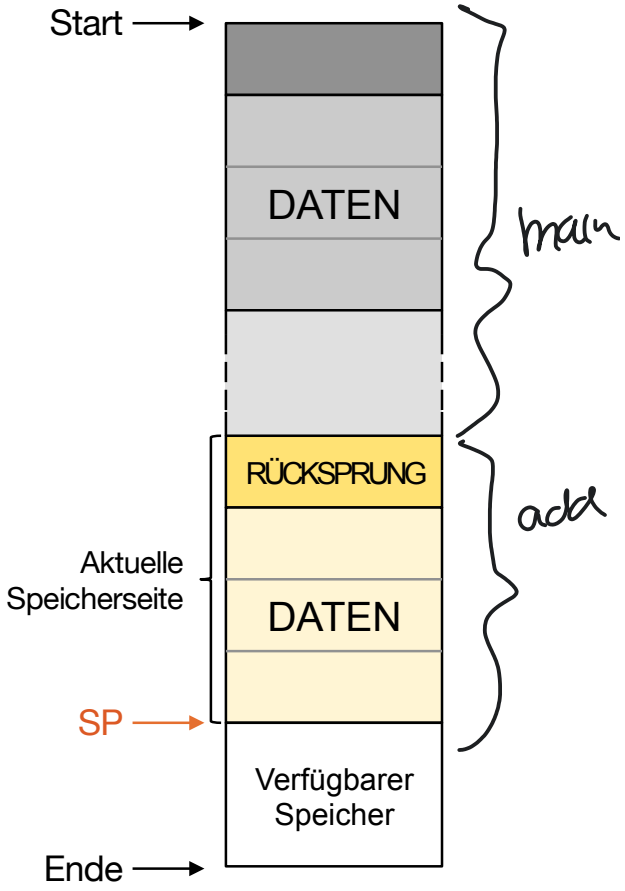
  → Verantwortung auf die Benutzer:in abwälzen

- Externe Regeln erzwingen
  - Programmierregeln i[illegible]sieren“
  - Compiler(-parameter) nutzen → Warnungen weisen darauf hin, z.B. `–Wunitialized`

Aber wie sieht der Speicher aus?

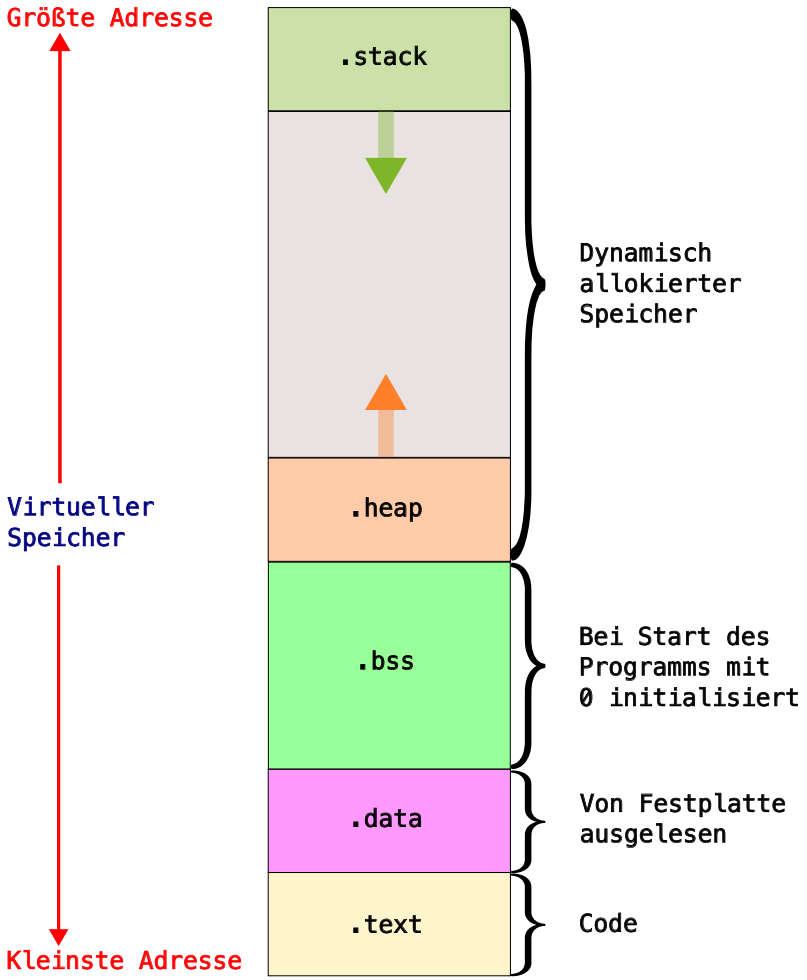
Rückblick: Speicherlayout des Stacks

- Bekannt aus Zusammenhang mit Funktionen
  - Variablen werden bei Deklaration „oben auf den Stapel gelegt"
  - Nach einem Code-Block: „Vom Stapel nehmen"
- Der *Stackpointer* (SP) zeigt stets auf oberstes Element



Speicherlayout - *Stack*

- Stack (`.stack`) ist nur ein Teil des Speichers eines Prozesses
- Weitere wichtige Speichersegmente sind
  block started by symbol
  `.heap`, `.bss`, `.text` und `.data`
  - Gibt aber noch eine Reihe mehr (teils compiler- und plattformabhängig)
  - Beispiel: `.rodata` → enthält Strings



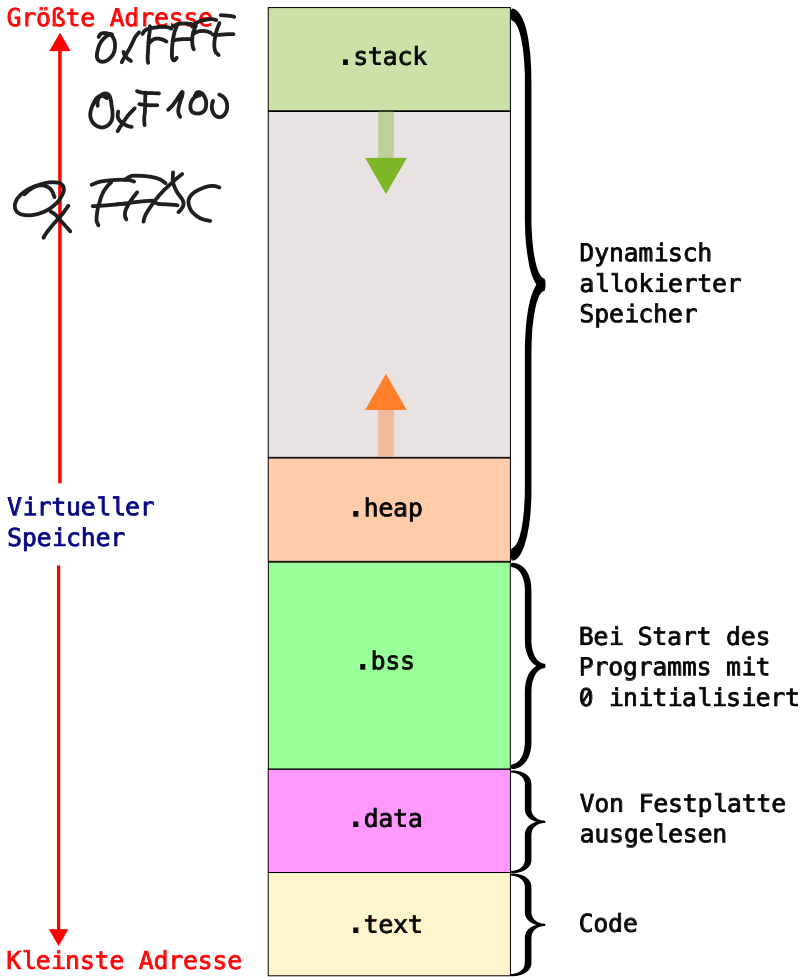
Speicherlayout - *Stack*

- In der Praxis: Beginnt bei größter Adresse

  (→ *Base Adress*)

- „Stack wächst nach unten"
- Neue Variablen werden **unterhalb** der Startadresse angelegt
- Randbemerkung: Abstand/Differenz zu einer Startadresse → *Offset*

add 0xFFAA, 0xFFAC



Speicherlayout - *Heap*

- Grenzt typischerweise direkt an

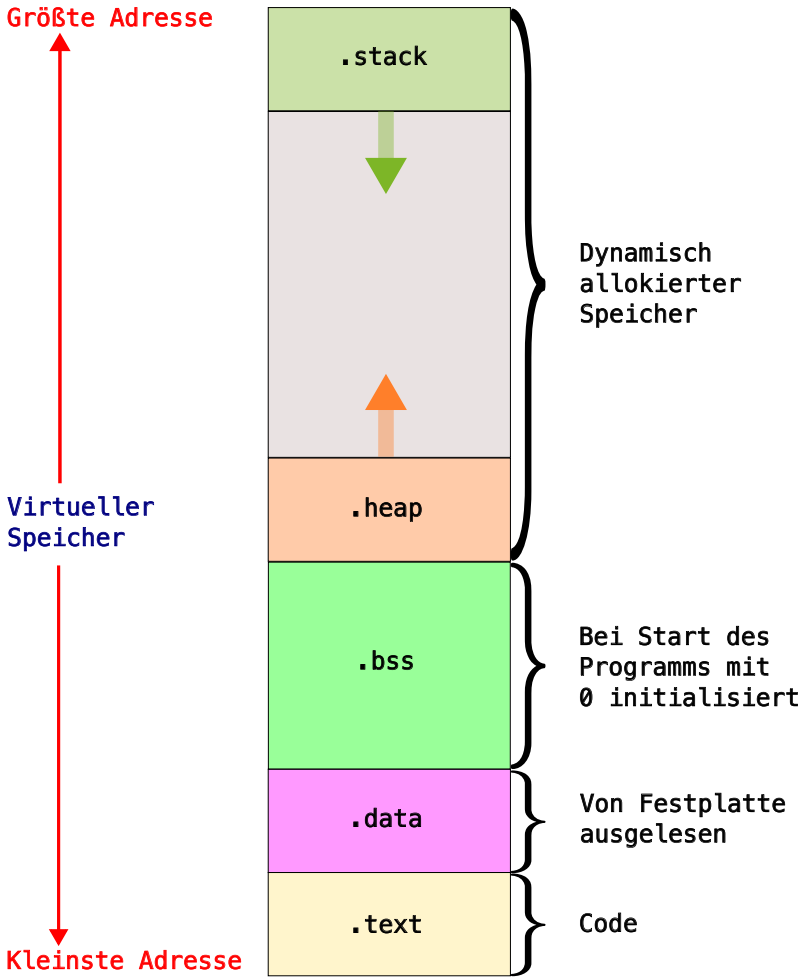
  `.data`, `.bss` und `.text`

  - Ausdehnung von diesen Segmenten weg
  - „Heap wächst nach oben“

- Speicherort für dynamisch allozierte Objekte

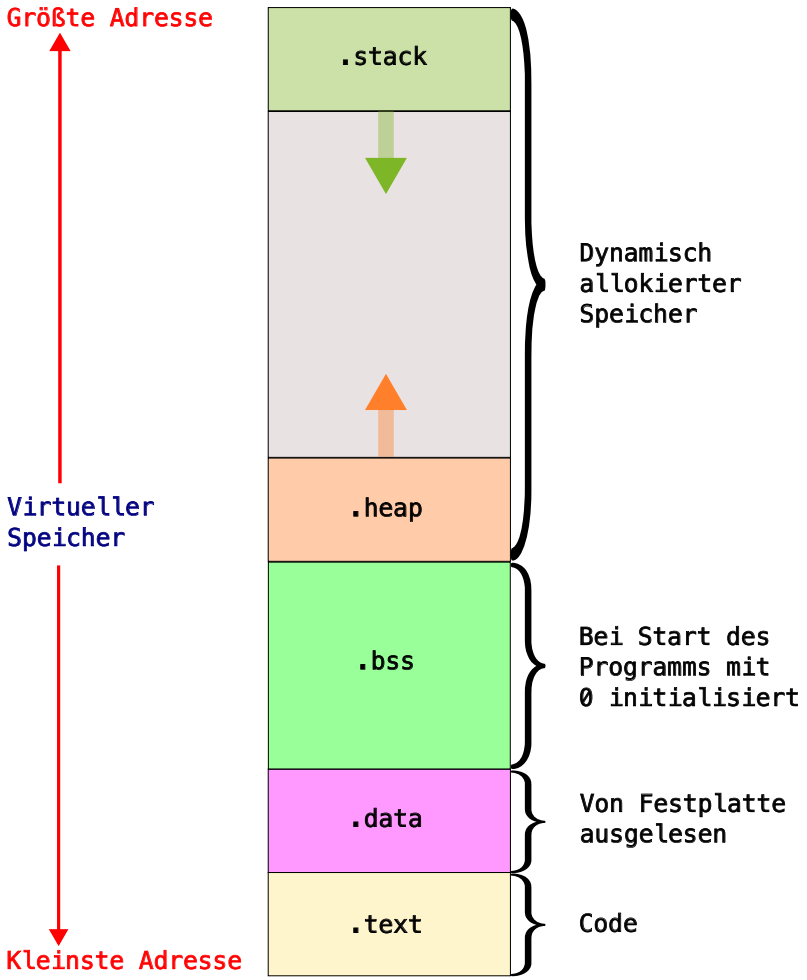
  → `malloc()`

- *Stack* und *Heap* werden bei Programmstart erzeugt



Speicherlayout - Daten

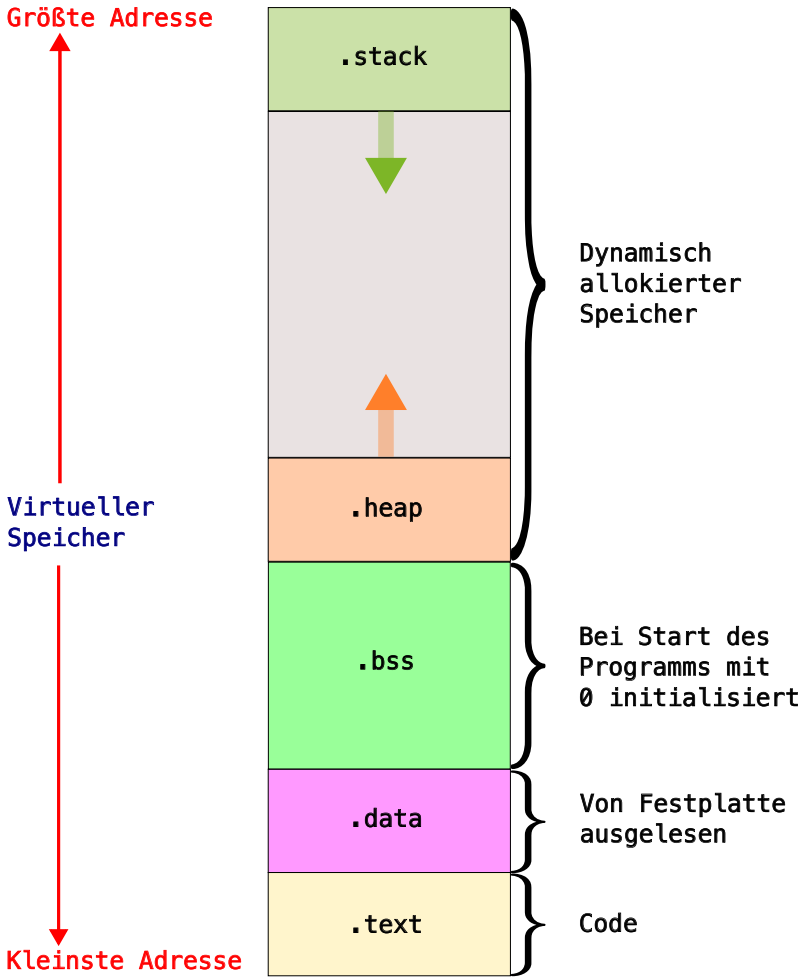
- `.data` und `.bss` sind Teil des Datensegments (globale und statische Variablen)
- `.data` enthält initialisierte Variablen
  - Beispiel: `static int i = 1;`
- `.bss` enthält nicht-initialisierte Variablen
  - Beispiel: `static int j;`
  - Werden automatisch jeweils mit `0` initialisiert
- `.bss` = *Block Started By Symbol*



Speicherlayout - *Text*

- `.text` beinhaltet Programmcode (→ Maschinencode)
- **Schreibgeschützt**, nur Lesen ist erlaubt

  (im Gegensatz zu `.data` und `.bss`)
- Grund: Programm darf sich nicht selber modifizieren
- Bei Programmstart: BS kopiert `.text`, `.bss` und `.data` aus Programmdatei in Speicher

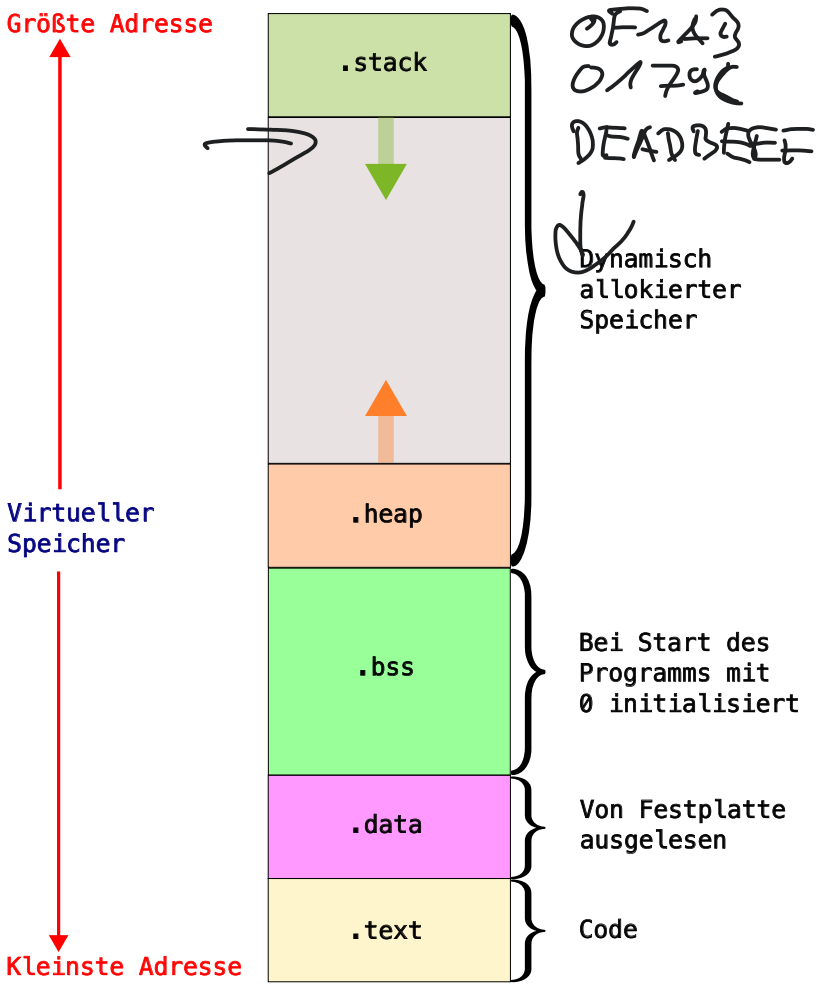


Speicherüberlauf (*Overflow*)

- „Überlaufen“ von Stapel und Halde einfach möglich
  - *Heap/Stack Overflow*
  - Hochproblematisch, Erkennung ist schwer
- Für Heap: Nicht ohne weiteres erkennbar

  (meist mit externen Werkzeugen)
- Für Stack: Bekannte Werte auf Stack schreiben

  (→ *Canaries*)
  - Regelmäßige Überprüfung, ob Werte korrekt
  - Entdeckung zur Laufzeit falls Bereich überschrieben wird (sowohl unabsichtlich als auch bösartig)
  - Überschreiben nennt sich auch *Stack Smashing*



Beispiel - *Overflow* mit Eingabe

- Simples Beispiel für *Stack Smashing*

```c
#include <string.h>

int main(int argc, char* argv[]) {
    char buf[4];
    strcpy(buf, argv[0]);
    printf("%s", buf);
    return 0;
}
```

- Aufruf mit `./stack_smash Hi` ✅
- Aber: Aufruf mit `./stack_smash Hello` ❌

```
*** stack smashing detected ***: terminated
    Aborted (core dumped)
```

Beispiel - *Overflow* ohne Eingabe

- Simples Beispiel für *Stack Smashing* **ohne** Eingabe

C Run ▶

```c
#include <stdio.h>

int main(int argc, char* argv[]) {
    char buf[4];
    buf[0] = 'H';
    buf[1] = 'e';
    buf[2] = 'l';
    buf[3] = 'l';
    buf[4] = 'o';
    buf[5] = '!';
    buf[6] = '\0';
    printf("%s\n", buf);
    return 0;
}
```

Einschub: `errno`

- Im letzten Kapitel bereits angesprochen:
  - `malloc()` signalisiert Fehler über Rückgabewert
  - Tritt ein Fehler auf, gibt `malloc()` NULL zurück
- Außerdem: „C teilt den Grund mit"
- Globale Variable namens `errno`
- `errno` schauen wir uns jetzt genauer an

Variable `errno`

- Im Header `errno.h` zu finden
- Tritt Fehler auf, wird `errno` gesetzt
- Beispiel: `errno == EEXIST;` → „File exists“
- Fehler„werte” sind Makros
- Liste von möglichen Werten für GCC hier verlinkt
- Konvertierung zu lesbarem Text: `strerror(errno);`
- Für Speichermangel: Makro `ENOMEM` (steht für *Error: No Memory*)

Beispiel – errno

C Run ▶

```c
#include <stdio.h>
#include <errno.h>
#include <string.h>

int main() {
    const char * file_path = "example.txt";
    // Öffne Datei zum Lesen
    FILE * f_handle = fopen(file_path, "r");
    if (f_handle == NULL) {
        printf("Fehler: %s\n", strerror(errno));
    } else {
        fclose(f_handle); // Aufräumen
    }
    return 0;
}
```

Einsatz von `errno`

- Programmierer:in kann `errno` ignorieren
- **Aber:** Einzige Möglichkeit, Fehler zu detektieren
- Beispiel: `malloc()` ohne Fehlerbehandlung
  - Schreibzugriff auf unzulässige Adresse
  - Betriebssystem erkennt dies
  - BS schreitet ein → Absturz 💥
- Verwendung von `errno` wird ausführlich dokumentiert

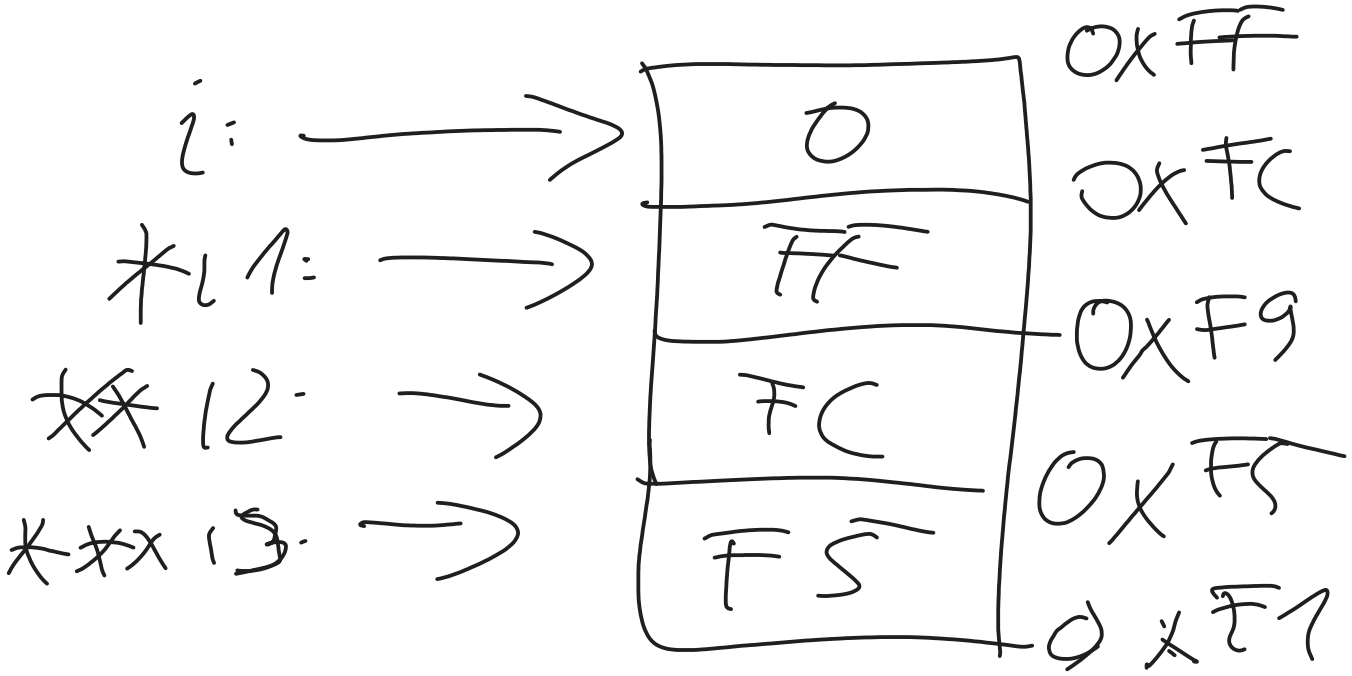
  → siehe Dokumentation → Lest sie! 😎
- Programmiert bitte defensiv → behandelt möglichst viele Fehlerfälle
- C++ kennt bessere Wege (→ später mehr)

Zeiger auf Zeiger

- Das Konzept erlaubt im Prinzip unbeschränkte Indirektion
- Zeiger auf Zeiger sind problemlos erlaubt

```
int i = 0;
int * i1 = &i;       // Zeiger auf int
int ** i2 = &i1;     // Zeiger auf Speicher (interpretiere als int *)
int *** i3 = &i2;    // Zeiger auf Speicher (interpretiere als int **)
/* ... */
```

- Wichtig ist hier das **genaue** Beachten der Typen bei der Zuweisung
- Aus technischer Sicht: Alle Zeiger verweisen auf eine Speicherstelle
  - Beispiel: `i3 = (int ***) &i; // Prinzipiell erlaubt (aber meist sinnfrei)`
- Korrekte Interpretation der Speicherstelle ist Aufgabe der Programmierer:in



=

***i3 → **i2 → *i1 → i = 0

Zeiger auf Zeiger

- Zeiger mit mehrfacher Indirektion erfordern viel Aufmerksamkeit

  → **sehr** großes Fehlerpotential

- Tipp: Bei Zuweisungen/Zugriffen mentale Checkliste durchgehen

  1. „Welche Typen haben die Variablen?"
     - `int **i, *j, k = 1;`
  2. „Passen die Typen schon, wenn ich die so verwende?

     Oder muss ich noch passend referenzieren/dereferenzieren?"
     - `j = &k; i = &j;`
  3. „Welchen Wert brauche ich gerade? Was muss ich dafür tun?"
     - Beispiel: Zugriff auf zugrundeliegenden `int` k
     - `*(*i);`

Arrays

- Zeiger werden auch in Arrays verwendet
- Genauer gesagt: Arrays sind Pointer mit zusätzlicher Dereferenzierung und Längeninformation
- Beispiel: `char arr[4];`
- Arraybezeichner `arr` ist Zeiger auf den Beginn des Speicherblocks

  (≙ erster Eintrag des Arrays `arr[0]`)
- Arrayindex gibt Offset von der Startadresse an
- Bei Umwandlung von Array zu Pointer entsteht Informationsverlust (Länge)

  → Wird auch *Pointer Decay* genannt (*Herabstufung zu Zeiger*)

Zeigerarithmetik - Mit Zeigern „rechnen“

- Offset-Veränderung um n entspricht Byte-Veränderung von

  n x `sizeof(Datentyp)`

C Run ▶

```c
#include <stdio.h>

int main() {
    double d = 1.0;
    double * ptr = &d;

    printf("%p\n", ptr);

    ptr += 1;                // ptr + 8 Bytes
    printf("%p\n", ptr);

    char * c = (char*)"c";
    printf("%ld\n", (c+1) - c);
}
```

Beispiel - Zeigerarithmetik

C Run

```c
#include <stdio.h>

int main() {
    int * ptr;
    int array[4] = {1, 2, 3, 4};

    ptr = &array[0];          // array[0]
    ptr += 1;                 // ptr == &array[1]
    ++ptr;                    // ptr == &array[2]
    ptr--;                    // ptr == &array[1]
    printf("%d\n", *ptr);
    ptr += 10;                // GEFAHR

    return 0;
}
```

Notation von Zeigerarithmetik

- Folgende Notationen sind identisch:

C Run ▶

```c
#include <stdio.h>

int main() {
    int arr[4] = {1, 2, 3, 4};
    int *ptr = arr;

    int i = arr[2];             // Standard, bereits bekannt
    int j = *(arr + 2);         // Umwandlung des Compilers von arr[2]
    int k = *(&(arr[0]) + 2);
    int l = ptr[2];  = *(ptr+2)  // Arraynotation geht auch mit Zeigern
    int m = 2[arr];             // *(2+arr) == *(arr+2)

    printf("i: %d\nj: %d\nk: %d\nl: %d\nm: %d\n", i, j, k, l, m);
    return 0;
}
```

- Array-Notation erspart zusätzliches Tippen (Dereferenzierungsoperator)

Arithmetik - Das Tor zur Hölle

- 🚨 Achtung: Zeigerarithmetik wird nicht überprüft

  → Berechnung von **beliebigen** Adressen möglich

- Bei Reinterpretierung der Speichers (*Type Casting*) entstehen leicht Fehler

```
int i = 1;
char arr [3] = "ca";
int *k = 0x20;
++k;
```

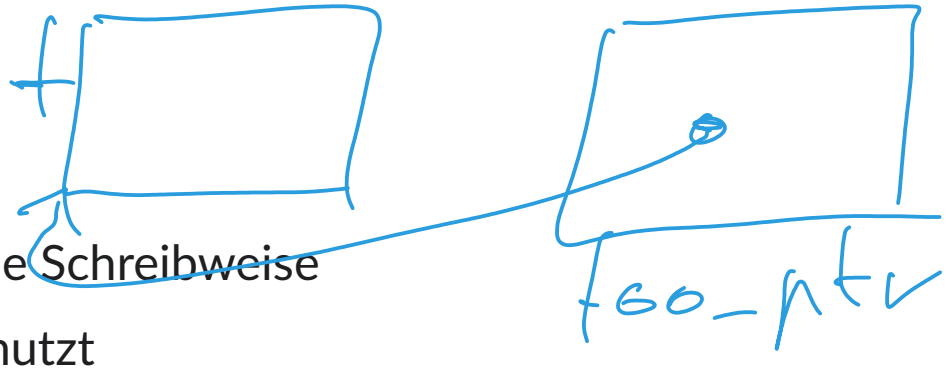
Pointer-Arithmetik nur sehr behutsam und sparsam benutzen

0x20	0x21	0x22	0x23	0x24	0x25	0x26	0x27
0	0	0	1	'c'	'a'	'\0'	

→ Bei Dereferenzierung wird String ab 0x24 als `int` interpretiert

Zeiger auf Datenstrukturen (`struct`)

- Zunächst einmal: Gelten die gleichen Regeln wie für alle anderen Datentypen
- Kleine Unterschiede bei Notation
  - Initialisierung/Deklaration ist prinzipiell gleich
  - Bei Zugriff gibt es wie bei Array-Indizes eine eigene Schreibweise
  - Statt Punktoperator (`.`) wird Pfeiloperator (`->`) genutzt




```
struct Foo { int i; };
struct Foo f = { 1 };
struct Foo * k = &f;
k->i = 2;        // -> ist Kurzform ...

(*k).i = 42;     // ... für dieses Konstrukt
```

*f

Beispiel - Zeiger auf Datenstrukturen

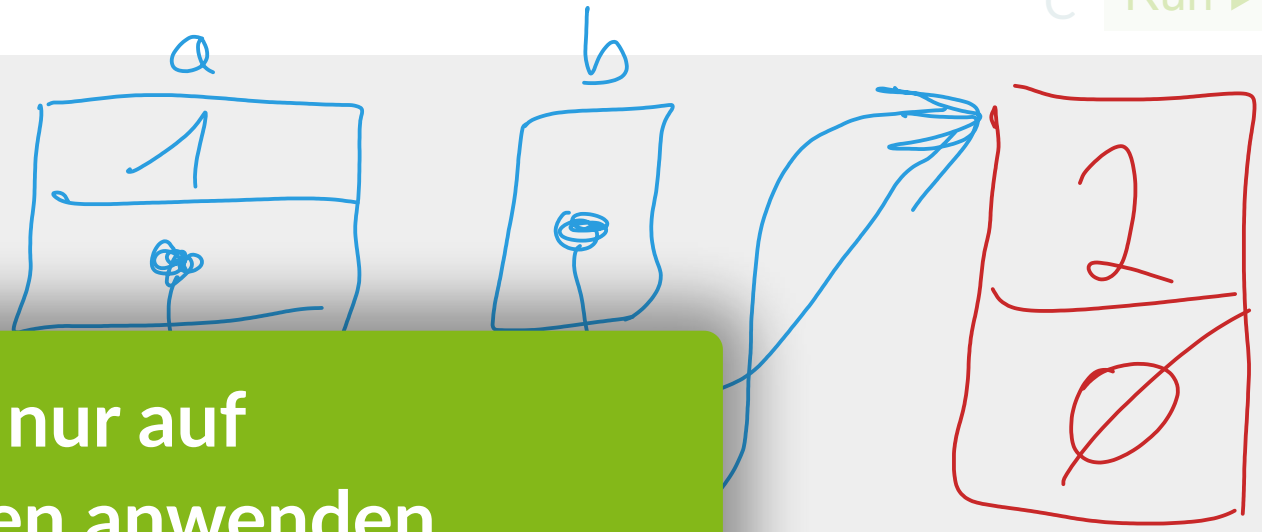
C Run ▶

```
#include <stdio.h>
#include <stdlib.h>
struct A { int i; struct A * ptr; };

int main() {
    struct A a = { 1, n
    struct A *b = (stru

    b->i = 2;
    b->ptr = nullptr;
    a.ptr = b;

    printf("A: %d\nB: %d\n", a.i, a.ptr->i); // Pfeil -> nur für Zeiger
    free(a.ptr);
    printf("A: %d\nB: %d\n", a.i, a.ptr->i); // UB: Use after free()
}
```



Pfeiloperator nur auf Zeigervariablen anwenden (beliebter Fehler)

Ergänzung zur Datenstrukturen

- Bisher: Datenstrukturen dürfen sich selbst nicht enthalten

```
struct Foo { struct Bar b; };
struct Bar { struct Foo f; }; // Error
```

- Aber: Zeiger auf sich selbst sind möglich

```
struct Foo { Foo * f; };
```

- Beachtet: Es wird nur der Zeiger gespeichert
- Größe des Zeigers vorab bekannt → Größe der Datenstruktur bekannt
- Klassisches Beispiel: Single Linked List

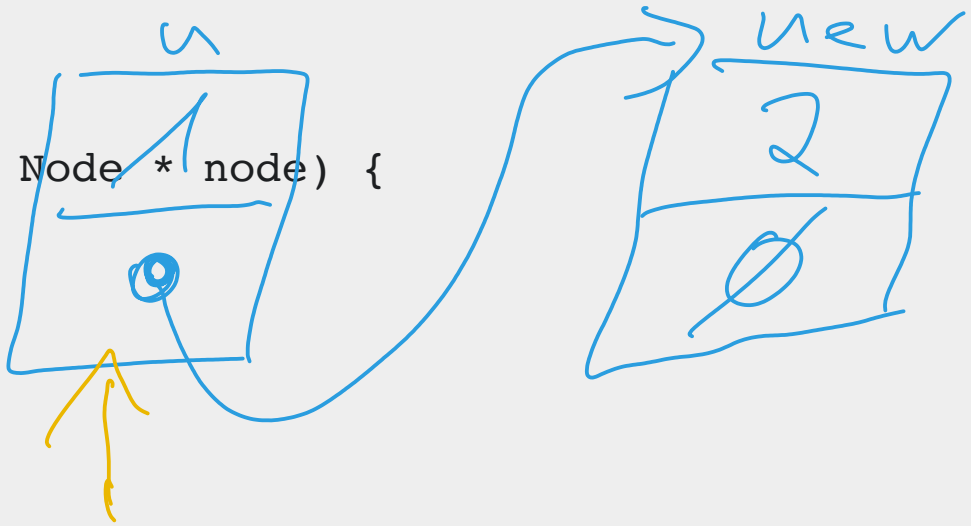
Beispiel - *Single Linked List*

C Run ▶

```c
#include <stdio.h>
struct Node {
    int i;
    struct Node * next;
};

void add_node(struct Node * root, struct Node * node) {
    if (root == nullptr) { return; }
    struct Node * current = root;
    while (current->next != nullptr) {
        current = current->next;
    }
    current->next = node;
}

int main() {
    struct Node n = { 1, nullptr };
    struct Node m = { 2 }; // m next mit 0 initialisiert
```



Einschub: Speicherausrichtung

- Frage: Welche Größe wird ausgegeben?

C Run ▶

```c
#include <stdio.h>

struct Example {
    char foo[3]; // 3 Bytes
    int bar;     // 4 Bytes
    bool baz;    // 1 Byte
};

int main() {
    struct Example ex = { "ab", 4, true };
    printf("Size: %ld, Alignment: %ld\n", sizeof(ex), alignof(struct Example));
}
```

Einschub: Speicherausrichtung

- Speicher ist typischerweise an festen Grenzen *ausgerichtet*
  - Lesen und Schreiben erfolgt typischerweise in mehrere Speicherzellen gleichzeitig
  - Ausrichtung (*Alignment*) macht dies möglichst effizient
- Einzelne Elemente kleiner als Alignment-Wert

  → Einfügen von „Polsterung“ (*Padding Bytes*)
- Alignment-Raster abfragen mit `alignof()`

Einschub: Speicherausrichtung

- Zwei Möglichkeiten, Padding zu eliminieren (→ spart Speicherplatz)
  1. Mittels *Packing*: ✅
     - `__attribute__((packed))`
     - Beinträchtigt ggf. Performance (→ ≥2 Zugriffe notwendig)
     - Nur bei bestimmten Szenarien sinnvoll (Beschreiben einer Speicherzelle)
  2. Durch clevere Anordnung der Elemente
     - Kleine Elemente im „Schatten" von großen

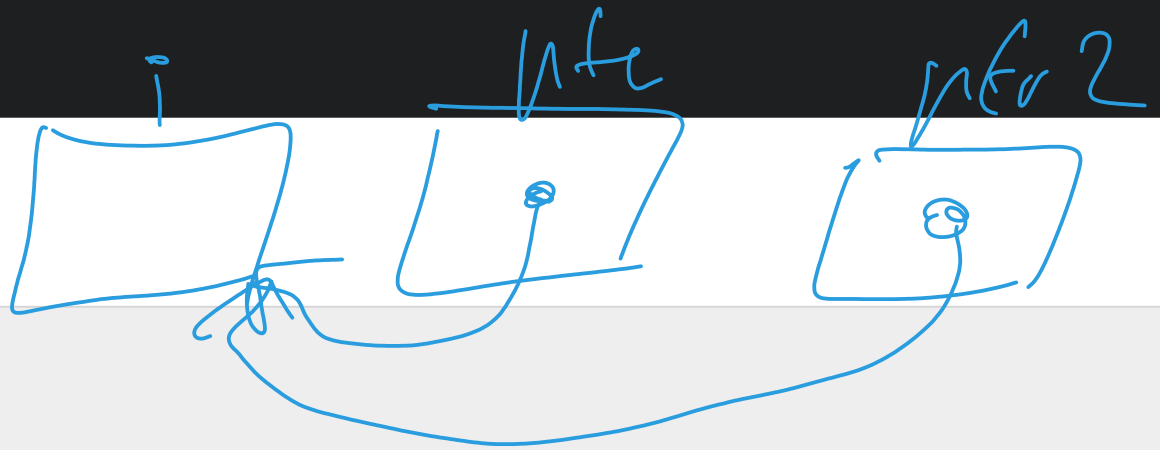
C Run ▶

```c
#include <stdio.h>

struct Example {
    char foo[3];
    bool baz; // Vertauscht mit bar
    int bar;
};

int main() {
    struct Example ex = { "ab",
        true, 4 };
    printf("Size: %ld", sizeof(ex));
}
```

Type Punning



- Was passiert hier?

```
int i = 1;
int * ptr1 = &i;
double * ptr2 = (double *) ptr1;
```

- Konvertierung eines Zeigertyps in einen anderen Zeigertyp
- Uminterpretierung zu anderem Typ → *Type Punning*
- Funktional ähnlich wie `union`, aber viel gefährlicher🚨
- Programmier:in trägt alleinige Verantwortung für Korrektheit

Beispiel - *Type Punning*

C Run

```c
#include <limits.h>
#include <stdio.h>
struct A { int i; };
struct B { double d; struct A* a; };

int main() {
    struct A a = { 127 };
    struct B* b = (struct B*)&(a);
    printf("A: %d\n", a.i);
    printf("B: %f\n", b->d);
}
```

Beispiel - Zugriff auf Floating-Point-Bits

C Run

```c
#include <stdio.h>

void print_float_bits(float f) {
    int * i = (int *) &f;
    for (unsigned int index = 0; index < sizeof(f) * 8; ++index) {
        int bit = ((1 << index) & *i) > 0;
        printf("%c", bit == 1 ? '1' : '0');
    }
}

int main() {
    print_float_bits(3.5f);
    return 0;
}
```

0011 0111

Strongly typed, weakly enforced

- Das letzte Beispiel zeigt: C hat Probleme!
- Einerseits: Typangabe bei Deklaration immer erforderlich
- Andererseits: Umgehung des Typsysstems **sehr** einfach möglich
  - Der Compiler lässt das zu; warnt höchstens
  - Deswegen: Programmier:in trägt Verantwortung für Korrektheit
- Wird häufig „stark typisiert, aber mit schwacher Umsetzung" bezeichnet (*strongly typed, weakly enforced*)
- C++ ist hier deutlich strenger, aber deswegen auch komplizierter

Callbacks

- Callbacks sind Funktionen als Aufrufparameter
- Werden anschließend (oder zu gegebener Zeit) *aufgerufen* (→ Name)
- **Sehr nützlich**: Erlauben Definition einer generischen Programmierschnittstelle
- In C sind alle Funktionen Zeiger!
- Beispiel:

```
int add (int x, int y);
int sub (int x, int y);

int call_func(int (*func)(int, int), int int1, int int2) {
    return func(int1, int2);
}
```

Callbacks

C Run

```c
#include <stdio.h>
int add (int x, int y) { return x + y; };
int sub (int x, int y) { return x - y; };
int bigger_num (int x, int y) { return (x > y) ? x : y; }
int call_func(int (*func)(int, int), int int1, int int2) {
    return func(int1, int2);
}

int main() {
    printf("%d\n", call_func(add, 1, 2));
    printf("%d\n", call_func(sub, 1, 2));
    printf("%d\n", call_func(bigger_num, 1, 2));
}
```

Exkurs: `volatile`

- Grundannahme: Nur das Programm modifziert Speicher
- Folge: Compiler darf Zugriffe entfernen und tut das!
  - Siehe Compiler-Optionen –O1 bis –O3
  - Bringt Performance
- Problem: Annahme gilt nicht für Hardware
  - Geräte ändern **asynchron** ihren Zustand
  - Zustandänderung über Register im Speicher lesbar
- Lösung: Markiere Speicherzelle als `volatile`: `volatile int * j;`
- Bei **jedem** Zugriff auf Variable wird Wert aus dem Arbeitsspeicher geladen

Exkurs: `const`

- Analog zu `volatile` gibt es Schlüsselwort `const`
- Markiert Variable als unveränderlich
- Variable mit `const` kann **einmal*** initialisiert werden, danach unveränderbar
- Bereits implizit bekannt und benutzt: String-Literale haben den Typ
  - Beispiel: `"Hello World"; // Typ: const char *`
- Compiler warnt bei Modifikation 😀

Zeiger mit `volatile` und `const`

- Bei Zeigern ist bei Verwendung von `const` etwas Vorsicht geboten🚨
- Einerseits: Zeiger auf als `const` definierte Typen
  - Beispiel: `const int * i;`
- Andererseits: **Konstante** Zeiger auf (variable) Typen
  - Beispiel: `int * const j;`
- Und schließlich konstante Zeiger auf konstante Typen
  - Beispiel: `const int * const k;`

Zusammenfassung

- Bisher: Verbergen des eigentlichen Speicherzugriffs - `int j = 42;`
- Jetzt: Zeiger (oder das Tor zur Hölle 😈) - `int *k = 0x4711;`
- Zeiger sind ...
  - selbst Variablen, deren Größe sich nach der Architektur richtet

    und **nicht** nach dem Datentyp, auf den sie zeigen
  - können **beliebige** Adressen enthalten und greifen dort auf Speicher zu 🚨
  - veränderbar: Zeigerarithmetik
- 🚨 Zeigerarithmetik 🚨
  - **Beliebiges** Erhöhen oder Erniedrigen der Zeigervariablen verändert die Adresse → `k = k + 12;`
  - **Wichtig:** Bei der Adressberechung spielt der Datentyp, auf den gezeigt wird, eine Rolle:

$$k = k + n * sizeof(*k)$$

Kapitel 09 - Zeiger