Praktikum C-Programmierung 2026

Stack & Pointer-Einführung

Jannek Squar

2026-04-22

Scientific Computing
Universität Hamburg

Universität Hamburg
DER FORSCHUNG | DER LEHRE | DER BILDUNG

Outline

Fortgeschrittene Datenstrukturen

- Arrays
- Enum, Struct, Union

Speicher und Adressen

- Pointer
- Speicher

Table of Contents

Fortgeschrittene Datenstrukturen

- Arrays
- Enum, Struct, Union

Speicher und Adressen

- Pointer
- Speicher

Einführung

- Array bzw. *Vektor*, *Feld* oder *Reihung*
- C unterstützt Arrays beliebiger (aber gleicher!) Datentypen
  - Intern: zusammenhängender Speicherbereich
- Datenzugriff mittels Indexoperator [ ]
  - Indizierung beginnt bei 0
- Ablage im Speicher zeilenweise
  - Beispiel: 2D der Größe 3×3
  - Logische Sicht gegen Speichersicht

0	1	2
3	4	5
6	7	8

wird zu

0	1	2	3	4	5	6	7	8

Array v0

- `Datentyp Arrayname[Anzahl_der_Elemente];`

```
int main (void) {
  int array[10];

  for (int i = 0; i < 10; i++) {
    array[i] = i;
  }

  return 0;
}
```

- Deklaration reserviert automatisch Speicher
  - Mögliche Größe $10 \cdot 4\,\text{B} = 40\,\text{B}$
  - Größe fix, kein *resize*-Mechanisumus

Out-of-bounds

- Zugriff außerhalb reservierten Speicher UB
- Mögliche Folgen:
  - Nichts, Programm läuft weiter
  - Falsche Werte, Programm läuft (erstmal) weiter
  - Absturz (`segmentation fault`)
  - $\rightarrow$ Potentielle Sicherheitslücke

Array v1

```c
void fill_array (int array[]) {
  for (int i = 0; i < 10; i++) {
    array[i] = i;
  }
}

int main (void) {
  int array[10];

  fill_array(array);
  return 0;
}
```

- Arrays können auch als Funktionsparameter übergeben werden

Array v3

```c
int main (void) {
  int array[10];

  for (unsigned int i = 0; i < sizeof(array) / sizeof(*array); i++) {
    array[i] = i;
  }

  return 0;
}
```

- Die Größe eines Arrays kann mit `sizeof` bestimmt werden
  - Allerdings nur an Stellen, an denen der Compiler die Größe kennt
- `sizeof(array)` gibt die Gesamtgröße zurück
  - `sizeof(*array)` die Größe eines Elements

enum v0

```c
#include <stdio.h>

enum {
  ENUM_ZERO,
  ENUM_ONE
};

int main (void) {
  printf("zero: %d\n", ENUM_ZERO);
  printf("one:  %d\n", ENUM_ONE);
  return 0;
}
```

- Mithilfe von enum können Integer-Konstanten eingeführt werden
  - Die Nummerierung startet standardmäßig bei `0`
  - hier: anonymes `enum`
  - Gültig innerhalb des Scopes

enum v1

```c
#include <stdio.h>

enum {
  ENUM_TWO    = 2,
  ENUM_THREE
};

int main (void) {
  printf("two:   %d\n", ENUM_TWO);
  printf("three: %d\n", ENUM_THREE);
  return 0;
}
```

- Der Startindex kann angepasst werden
  - Folgende Einträge werden automatisch um eins erhöht

struct v0

```c
struct foo {
  int bar;
  char baz;
};

int main (void) {
  struct foo a;
  a.bar = 42;
  a.baz = 'a';
  return 0;
}
```

- Ein `struct` ist aus anderen Datentypen zusammengesetzt
  - Auf den Inhalt kann über Variablennamen zugegriffen werden

struct v1

```
1   struct foo {
2     int bar;
3     char baz;
4   };
5
6   int main (void) {
7     struct foo a = { 42, 'a' };
8     struct foo b = { .bar = 42, .baz = 'a' };
9     return 0;
10  }
```

- Die Initialisierung ist mit { } möglich
  - Entweder in der richtigen Reihenfolge oder über Namen

struct v2

```c
#include <stdio.h>

struct foo {
  int bar;
  char baz;
};

int main (void) {
  printf("sizeof: %lu\n", sizeof(struct foo));
  return 0;
}
```

- Die Größe entspricht nicht immer der Summe der Größe der Komponenten
    - Strukturen werden für effizienten Zugriff mit Padding versehen

```
sizeof: 8
```

struct v2

```c
#include <stdio.h>

struct foo {
  int bar;
  char baz;
};

int main (void) {
  printf("sizeof: %lu\n", sizeof(struct foo));
  return 0;
}
```

- Die Größe entspricht nicht immer der Summe der Größe der Komponenten
  - Strukturen werden für effizienten Zugriff mit Padding versehen

```
sizeof: 8
```

Die Reihenfolge der Komponenten ist wichtig für das Padding

```c
#include <stdio.h>

struct foo {
  char baz0;
  int bar;
  char baz1;
};

int main (void) {
  printf("sizeof: %lu %lu\n", sizeof(char), sizeof(int));
  printf("sizeof: %lu\n", sizeof(struct foo));
  return 0;
}
```

```
sizeof: 1 4
sizeof: 12
```

struct v3

Die Reihenfolge der Komponenten ist wichtig für das Padding

```
1  #include <stdio.h>
2
3  struct foo {
4    char baz0;
5    int bar;
6    char baz1;
7  };
8
9  int main (void) {
10   printf("sizeof: %lu %lu\n", sizeof(char), sizeof(int));
11   printf("sizeof: %lu\n", sizeof(struct foo));
12   return 0;
13 }
```

```
sizeof: 1 4
sizeof: 12
```

struct v3

```c
#include <stdio.h>

struct foo {
  char baz0;
  char baz1;
  int bar;
};

int main (void) {
  printf("sizeof: %lu %lu\n", sizeof(char), sizeof(int));
  printf("sizeof: %lu\n", sizeof(struct foo));
  return 0;
}
```

```
sizeof: 1 4
sizeof: 8
```

Padding plattformabhängig, orientiert sich am größten Datentyp

```c
#include <stdio.h>

struct foo {
  char baz0;
  double bar;
  char baz1;
};

int main (void) {
  printf("sizeof: %lu %lu\n", sizeof(char), sizeof(double));
  printf("sizeof: %lu\n", sizeof(struct foo));
  return 0;
}
```

```
sizeof: 1 8
sizeof: 24
```

struct v3

Padding plattformabhängig, orientiert sich am größten Datentyp

```c
#include <stdio.h>

struct foo {
  char baz0;
  double bar;
  char baz1;
};

int main (void) {
  printf("sizeof: %lu %lu\n", sizeof(char), sizeof(double));
  printf("sizeof: %lu\n", sizeof(struct foo));
  return 0;
}
```

```
sizeof: 1 8
sizeof: 24
```

union v0

```c
#include <stdio.h>
union foo {
  int bar;
  char baz;
};

int main (void) {
  union foo a;
  a.bar = 42;
  a.baz = 'a';
  printf("%d\n", a.bar); // Schrott
  printf("%c\n", a.baz); // a
  return 0;
}
```

- Eine `union` verhält sich ähnlich wie ein `struct`
  - Enthält beliebig viele Komponenten, allerdings ist nur eine davon aktiv

union v1


```c
#include <stdio.h>

union foo {
  int bar;
  char baz;
};

int main (void) {
  printf("sizeof: %lu\n", sizeof(union foo));
  return 0;
}
```

- Eine `union` belegt nur so viel Platz wie ihre größte Komponente

```
sizeof: 4
```

union v1

```c
#include <stdio.h>

union foo {
  int bar;
  char baz;
};

int main (void) {
  printf("sizeof: %lu\n", sizeof(union foo));
  return 0;
}
```

- Eine `union` belegt nur so viel Platz wie ihre größte Komponente

```
sizeof: 4
```

Table of Contents

Fortgeschrittene Datenstrukturen

Arrays

Enum, Struct, Union

Speicher und Adressen

Pointer

Speicher

Was ist ein Pointer?

- **Pointer speichern Speicheradressen**
- nicht initialisierte Pointer können überall hinzeigen
- Pointer-Typ: Interpretation des Speichers an dieser Adresse
    - Falscher Typ, falsche Interpretation!
- Verschiedene Pointer Typen $\rightarrow$ unterschiedliche Interpretation
    - Meist keine implizite Umwandlung, da sonst Daten falsch interpretiert würden

Verwendung

```c
int i = 5;
int *pi = &i; /* Adressoperator */
int n = *pi; /* Dereferenzierungsoperator */
```

- Adressoperator (&): virtuelle Adresse der Variable
- Dereferenzierungsoperator (*): Speicherinhalt
  - Verwechslungsgefahr: `int *y` dereferenziert nicht, Datentyp heißt `int *`
- Nicht initialisierter Zeiger beinhaltet zufällige Adresse
- Null-Pointer `NULL`
  - Initialisierung
  - Fehler

Pointers Store **One** Address Only

```
char  c = 3;
short s = 5;
int   i = 9;

char*  pc = &c;
short* ps = &s;
int*   pi = &i;
```

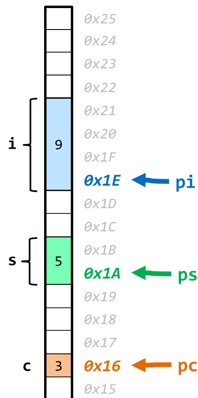

Abbildung 1: Trennung Adresse und Inhalt[3]

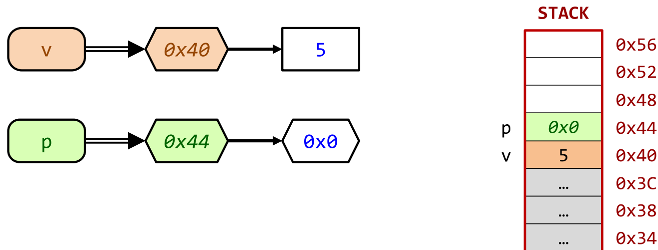

Abbildung 2: Trennung Adresse und Inhalt[3]

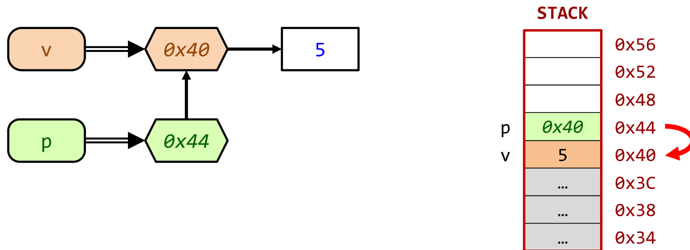

Abbildung 3: Trennung Adresse und Inhalt[3]

Pointer auf Pointer

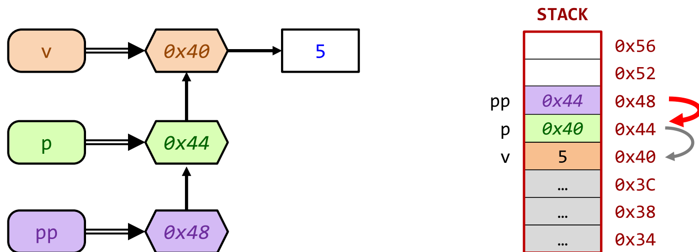

Abbildung 4: Trennung Adresse und Inhalt[3]

Sind Arrays Pointer?

- Arrays und Pointer sind **nicht** das Gleiche![1]
- Aber: Array-Namen werden als Zeiger auf das erste Array-Element angesehen
  - *Array zerfällt u.U in einen Pointer*

Beispiel:

```
1 int element[8] = { 1, 2, 4, 8, 16, 32, 64, 128 };
2 int *ptr;
3
4 ptr = element;
```

Gleichwertig:

```
int main(int argc, char **argv)  {...}   ≡   int main(int argc, char *argv[]) {...}

                     ptr = element;   ≡   ptr = &element[0];
```

[1] *Call-by-value* vs *Call-by-reference*

Pointer auf struct: Beispiel

- Pointer können auf alles zeigen, insbesondere auch auf `structs`
- Neuer Operator: `->`
- `a->b` ist äquivalent zu `(*a).b`

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

struct Point {
  double x;
  double y;
};

int main(int argc, char** argv) {
  struct Point p = {1.0, 2.5};
  struct Point * pp = &p;
  pp->x = 10;
  (*pp).y = 15;
  printf("(%f, %f) = (%f, %f)\n", p.x, p.y, (*pp).x, pp->y);
}
```

Adressenausgabe mit printf()

```c
int i = 5;
int *pi = &i;

printf("  i = %d\n", i);
printf("*pi = %d\n", *pi); /* Inhalt */
printf(" pi = %p\n", pi); /* Adresse */
```

Ausgabe:

```
  i = 5
*pi = 5
 pi = 0x7ffc6045c61c
```

void

- unvollständiger Datentyp, signalisiert Abwesenheit eines Typs
  - `void func(void);`
- void-Pointer `void *`
  - Verwendung als Black-Box
  - generisch, darf nicht dereferenziert werden
- Typecast:
  - `void *vp = (void*) p; // Typecast zu void*`
  - `int *y = (int*) vp; // zurück zu int*`

Virtueller Adressraum

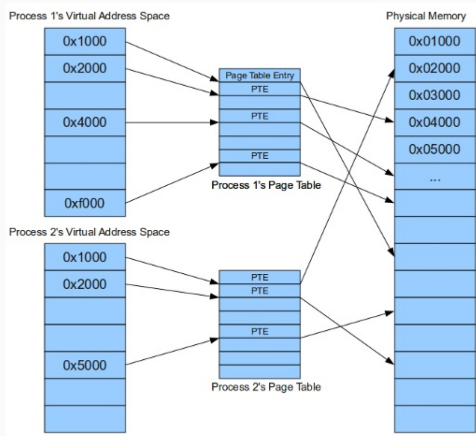

Abbildung 5: Memory mapping [1]

- OS verwaltet Speicher
- physikalischer Speicher → kontinuierlicher, virtueller Adressraum

Speicherlayout von C-Programmen

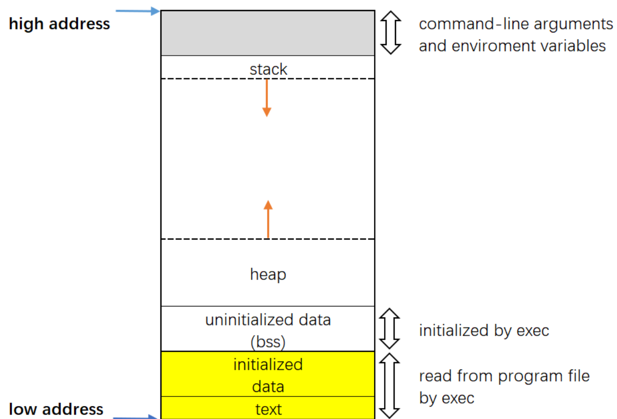

Abbildung 6: Speicherlayout [2]

- Segment Text
  - Code
- Segment data
  - Globale Variablen
  - Vorinitialisiert
- Segment BSS
  - Globale Variablen
  - Null-initialisiert
- Segment Heap
- Segment Stack

Beispiel

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

int g_var = 42;

int main() {
    int a = 42;
    int *pa = &a;
    int *b = malloc(10*sizeof(int));

    printf("&g_var: %p\n", &g_var);
    printf("&a:     %p\n", &a);
    printf("pa:     %p\n", pa);
    printf("&pa:    %p\n", &pa);
    printf("b:      %p\n", b);
    printf("&b:     %p\n", &b);
    return 0;
}
```

Ausgabe:

```
&g_var: 0x403014
&a:     0x7fff0d1a01cc
pa:     0x7fff0d1a01cc
&pa:    0x7fff0d1a01c0
b:      0x1ae5a310
&b:     0x7fff0d1a01b8
```

Beispiel

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

int g_var = 42;

int main() {
    int a = 42;
    int *pa = &a;
    int *b = malloc(10*sizeof(int));

    printf("&g_var: %p\n", &g_var);
    printf("&a:     %p\n", &a);
    printf("pa:     %p\n", pa);
    printf("&pa:    %p\n", &pa);
    printf("b:      %p\n", b);
    printf("&b:     %p\n", &b);
    return 0;
}
```

Ausgabe:

```
&g_var: 0x403014
&a:     0x7fff0d1a01cc
pa:     0x7fff0d1a01cc
&pa:    0x7fff0d1a01c0
b:      0x1ae5a310
&b:     0x7fff0d1a01b8
```

References

[1] Buffer Overflows and You
https://turkeyland.net/projects/overflow/intro.php

[2] Memory Layout of C Programs
https://xvirt.ink/2018/11/16/memory-layout/

[3] Pointer Arithmetic
https://hackingcpp.com/cpp/lang/pointer_arithmetic