Praktikum C-Programmierung 2026

Heap

Jannek Squar

2026-05-06

Scientific Computing
Universität Hamburg

Universität Hamburg
DER FORSCHUNG | DER LEHRE | DER BILDUNG

Outline

Table of Contents

Rechnen mit Pointern

- Addition/Subtraktion immer auf Größe ganzer Datentypen
- Basis `sizeof(var)` → Automatische Bestimmung der „Schritt“-Größe
- `a[b]` ⇔ `*(a+b)`
- `ptr + a` ≡ `(char*)ptr + a*sizeof(*ptr)`

Pointer Arithmetik Beispiel

Pointers Store **One** Address Only

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

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

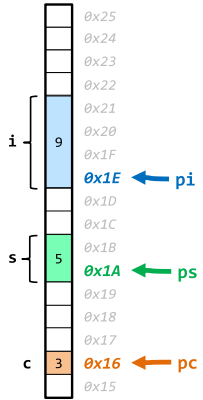


Abbildung 1: Grundlage[4]

Pointer Arithmetic: Increment by 1

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

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

pc++;
ps++;
pi++;
```

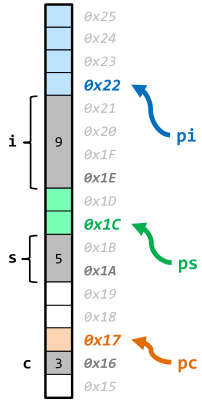


Abbildung 2: Addition[4]

Pointer Arithmetic: Decrement by 1

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

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

pc--;
ps--;
pi--;
```

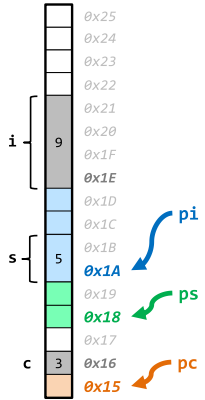


Abbildung 3: Subtraktion[4]

Pointer Arithmetic

here: `sizeof(int) = 4 * sizeof(char)`

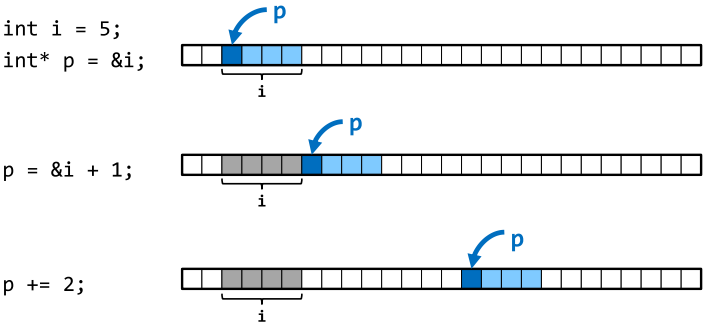
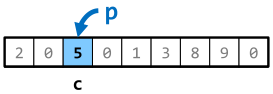


Abbildung 4: Sprünge[4]

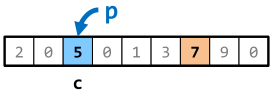
The Subscript Operator [ ]

p[n] = access value at (address in pointer + **n**)

```
char c = 5;
char* p = &c;
```




```
*(p + 4) = 7;
```




```
p[4] = 2;
```

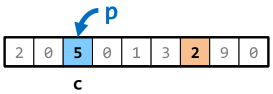


Abbildung 5: Index-Operator[4]

Code-Beispiele

```c
#include <assert.h>
int main() {
  int arr[5] = {0};
  int *ptr = &arr[2];
  assert( ptr - 2 == &arr[0] );
  assert( ptr + 2 == &arr[4] );
  assert( *(ptr + 1) == arr[3] );
  assert( ptr + 1 == (int *)( (char *)ptr + sizeof(*ptr) ));
  assert( ptr + 2 == (int *)( (char *)ptr + 2*sizeof(*ptr) ));
  return 0;
}
```

Die Differenz gibt die Anzahl der Elemente zwischen den Pointern zurück

```c
int *ip1 = &arr[1];
int *ip2 = &arr[4];
assert(ip2 - ip1 == 3);
```

Addition und Subtraktion

Regeln:

- `ptr + integer -> ptr`
- `integer + ptr -> ptr`
- `ptr - integer -> ptr`
- `ptr - ptr     -> integer`

`+=`, `-=`, `++` und `--` entsprechend

Beispiel: Arrays und Pointer

- Vergleich von Zeiger-Adressen mit <, >, <=, >=, ==, !=
- Adresse `&arr[5]` existiert, aber Inhalt darf nicht benutzt werden

```c
for(int *p = &arr[0]; p < &arr[5]; p++) {
  *p = 0;
 }
```

```c
for(int i = 0; i < 5; i++) {
  arr[i] = 0;
 }
```

Übergabe an Funktionen

Wie genau wird ein Parameter an eine Funktion übergeben?

- Call-by-Value: Übergabe einer Kopie
- Call-by-Reference: Übergabe einer Referenz
  - Aber: Der eigentliche Pointer wird auch nur kopiert!

C benutzt Call-by-value, Zeiger ermöglichen aber Call-by-Reference

Beispiel: Call-by-Value

```c
#include <stdio.h>

void increment(int x) {
    x = x + 1;
    printf("Innen: %d (%p)\n", x, &x);
}

int main(void) {
    int a = 5;
    increment(a);
    printf("Außen: %d (%p)\n", a, &a);
    return 0;
}
```

```
$ ./call_by_value.x
Innen: 6  (0x7ffe5503750c)
Außen: 5 (0x7ffe5503752c)
```

Beispiel: Call-by-Reference

```c
#include <stdio.h>

void increment(int *x) {
    *x = *x + 1;
    printf("Innen: %d (%p)\n", *x, &*x);
}

int main(void) {
    int a = 5;
    increment(&a);
    printf("Außen: %d (%p)\n", a, &a);
    return 0;
}
```

```
$ ./call_by_reference.x
Innen: 6  (0x7ffcbe587cac)
Außen: 6 (0x7ffcbe587cac)
```

Beispiel: Call-by-Value II

```c
#include <stdio.h>
void assign(int *ptr, int *target) {
    ptr = target;
    printf("Innen: %d\n",*ptr);
}

int main(void) {
    int a = 10, b = 20;
    int *ptr = &a;

    assign(ptr, &b);
    printf("Außen: %d\n",*ptr);
    return 0;
}
```

```
$ ./call_by_value_pointer.x
Innen: 20
Außen: 10
```

Beispiel: Call-by-Reference II

```c
#include <stdio.h>
void assign(int **ptr, int *target) {
    *ptr = target;
    printf("Innen: %d\n",**ptr);
}

int main(void) {
    int a = 10, b = 20;
    int *ptr = &a;

    assign(&ptr, &b);
    printf("Außen: %d\n",*ptr);
    return 0;
}
```

```
$ ./call_by_reference_pointer.x
Innen: 20
Außen: 20
```

Table of Contents

Rückblick: Speicherlayout

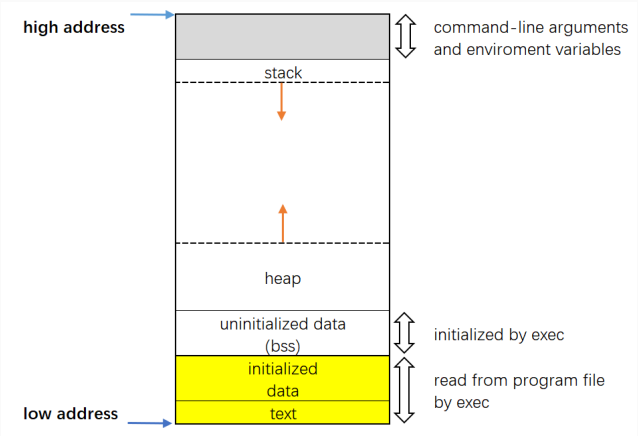


Abbildung 6: Speicherlayout [1]

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

Stack vs. Heap [2]

Stack

- Sehr schneller Zugriff
- Keine explizite Speicher-Freigabe
- Beschränkte Stackgröße
- Fixe Variablengröße
- Effiziente Speicherverwaltung
  - Keine Fragmentierung

Heap

- Langsamere Speicherverwaltung
- Manuelle Speicherverwaltung
- „Unbegrenzter“ Speicher
- Flexible Speichergröße
- Potentielle Fragmentierung

Stack vs. Heap [2]

Stack

- Sehr schneller Zugriff
- Keine explizite Speicher-Freigabe
- Beschränkte Stackgröße
- Fixe Variablengröße
- Effiziente Speicherverwaltung
  - Keine Fragmentierung

Heap

- Langsamere Speicherverwaltung
- Manuelle Speicherverwaltung
- „Unbegrenzter“ Speicher
- Flexible Speichergröße
- Potentielle Fragmentierung

Funktionen zur Speicherverwaltung

`malloc`	Reserviert Speicher, keine Initialisierung
`calloc`	Reserviert Speicher, Initialisierung mit 0
`realloc`	Ändert Größe reservierten Speichers
`free`	Freigabe des Speichers

Funktionen sind definiert im Header `stdlib.h`

Speicherreservierung mit malloc()

```c
void *malloc(size_t size);
```

- reserviert `size` Bytes
- Rückgabewert:
  - Erfolg: Zeiger auf reservierten Speicher
  - Fehler: `NULL`
- Keine Speicher-Initialisierung

Beispiel malloc()

```
1 int *ptr1 = malloc(10 * sizeof(int));
2 int *ptr2 = malloc(10 * sizeof(*ptr2));
```

Expliziten Type-Cast in C vermeiden:

```
1 int *ptr3 = (int *)malloc(10 * sizeof(*ptr3));
2 int *ptr4 = malloc(10 * sizeof(*ptr4));
```

- `void *` wird automatisch gecastet (DRY)
- alte C-Versionen nehmen `int malloc()` an, wenn `stdlib.h` fehlt
- *Hinweis*: C++ würde expliziten Type-Cast verlangen!

Speicherreservierung mit calloc()

```c
void *calloc(size_t num, size_t size);
```

- reserviert num Elemente mit der Größe size
- Rückgabewert:
  - Erfolg: Zeiger auf reservierten Speicher
  - Fehler: NULL
- Initialisiert den Speicher mit Nullen
- Abweichende Signatur von malloc

Speicherreservierung mit realloc()

```c
void *realloc(void *ptr, size_t size);
```

- Verändert Größe des Speicherbereichs `ptr` auf neue Größe mit `size` Bytes
- verändert nicht den Inhalt im Speicherbereich innerhalb von $[0, min(\texttt{old\_size}, \texttt{new\_size})]$
- Keine Initialisierung von zusätzlichem Speicher
- Rückgabewert:
  - Erfolg: Zeiger auf Speicher
  - Fehler: `NULL`

Potentieller Memory-Leak: `ptr = realloc(ptr, new_size);`

```c
int *tmp = realloc(ptr, new_size);
if (tmp != NULL)
    ptr = tmp;
```

Speicherfreigabe mit free()

```c
void free(void *ptr);
```

- Freigabe Speicher, auf den `ptr*` zeigt
  - Ausdruck muss auf allokierten Speicher zeigen
  - Ausdruck muss auf Anfang zeigen
  - Alias-Pointer auf freigegebenden Bereich müssen nicht erneut freigegeben werden
- UB bei wiederholtem Aufruf
- NOP, wenn `ptr == NULL`

```c
int *ptr = malloc(sizeof(*ptr) * 10);
int *ptr2 = ptr + 1;
free(ptr2-1); // kein free(ptr) mehr nötig
```

Beispiel: Array vs Pointer I

Stack

```c
size_t size = 5;
int a[size];

for (int i = 0; i < size; ++i)
{
 a[i] = i * i;
}

```

Heap

```c
size_t size = 5;
int *p = malloc(size * sizeof(*p));

for (int i = 0; i < size; ++i)
{
 p[i] = i * i;
}
free(p);
```

Beispiel: Array vs Pointer II

Erinnerung: Arrays als Funktionsargument „zerfallen“ zu Zeigern

```c
void print_array(int *a, size_t size, char* comment) {
 printf("%s\n", comment);
 for (int i = 0; i < size; ++i) {
  printf("%d ", a[i]);
 }
 printf("\n");
}
```

Stack

```c
int a_s[size];
print_array(a_s, size, "Stack");

```

Heap

```c
int *a_h = malloc(size * sizeof(*a_h));
print_array(a_h, size, "Heap");
free(a_h);
```

Beispiel: Fehlerbehandlung

Prüfung notwendig, ob Speicher allokiert werden konnte

```c
int *ptr = (int *) malloc(10 * sizeof(*ptr));

if (ptr == NULL) {
 /* Error handling */
} else {
 /* Allocation succeeded. Do something. */
 free(ptr);
}
```

Beispiel: Eigene Speicherverwaltung

Speicher-Allokation in Funktion

```c
int *func_a(void) {
 int *x = (int *) malloc(25 *
 ↪  sizeof(*x));
 return x;
}

void func_b() {
 int *pi = func_a();
 /* do something with pi */
 free(pi);
}
```

Vorteil:

- Verbirgt potentielles Book-Keeping

Nachteil:

- Nutzer für Freigabe verantwortlich
- Nicht offensichtlich
- Evtl. eigene Freigabefunktion

Häufige Fehler

Manuelle Speicherverwaltung ist fehleranfällig [3]

- Speicherlecks
- Zugriff auf nicht reservierten Speicher
- Nutzung nach `free()`
- Freigabe von nicht reserviertem Speicher
- Mehrfacher Aufruf von `free()`
- Verlorene Pointer

Speicherlecks

```c
int memory_leak() {
 int *ptr = malloc(sizeof(*ptr));
 return 0;
}
```

- Zeiger nach Funktionsende verloren
- Speicher bleibt bis Programmende reserviert
  - Hier egal
  - Große Memory-Leaks weniger egal
  - Große Memory-Leaks in Schleife noch weniger egal

Zugriff auf nicht reservierten Speicher

Nutzung von nicht allokiertem Speicher

```c
int *func_a(void) {
 int x[25];
 return x;
}

void func_b() {
 int *pi = func_a();
 *(pi + 1) = 5;
 free(pi); // UB
}
```

Nutzung nach free

Speicherzugriff nach Aufruf von `free()`

```c
int *ptr = malloc(sizeof (int));
free(ptr);
*ptr = 7; // UB
```

Freigabe von nicht dynamisch reservierten Speicher

Freigabe von nicht durch `malloc`, `calloc` or `realloc` allokierten Speicher

```c
char *msg = "Default message";
int tbl[100];
free(msg);
free(tbl); // UB
```

Multiple free

Mehrfache Speicherfreigabe

```
1
2  void func_a(int *g) {
3   printf("%d", *g);
4   free(g);
5  }
6
7  void func_b() {
8   int *p;
9   p = (int *)malloc(10 * sizeof(int));
10  func_a(p);
11  free(p); // UB
12 }
13
```

Verlorene Pointer

Falsche Freigabe-Reihenfolge von geschachtelten Pointern

```c
#include <stdlib.h>
int main() {
  int **ptr = malloc(sizeof(int *) * 10);
  for (int i = 0; i < 10; ++i)  {
    ptr[i] = malloc(sizeof(int) * 10);
  }
  free(ptr);
  for (int i = 0; i < 10; ++i)  {
    free(ptr[i]);
  }
}
```

```
$ ./free-order-wrong.x
free(): invalid pointer
[1]    94422 IOT instruction (core dumped)  ./free-order-wrong.x
```

Verlorene Pointer (korrigiert)

Freigabe von „innen“ nach „außen“

```c
#include <stdlib.h>
int main() {
  int **ptr = malloc(sizeof(int *) * 10);
  for (int i = 0; i < 10; ++i)  {
    ptr[i] = malloc(sizeof(int) * 10);
  }

  // innen
  for (int i = 0; i < 10; ++i)  {
    free(ptr[i]);
  }

  // außen
  free(ptr);
}
```

Behandelte Themen

- Fortsetzung Pointer
  - Pointer-Arithmetik
  - Funktionsparameter
- Heap
  - Einleitung
  - Funktionen
  - Typische Fehler

References

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

[2] 7. Memory : Stack vs Heap https://gribblelab.org/teaching/CBootCamp/7_Memory_Stack_vs_Heap.html

[3] C Programming/stdlib.h/malloc https://en.wikibooks.org/wiki/C_Programming/stdlib.h/malloc#Common_errors

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