Pointer und Pointer-Arithmetik

Praktikum „C-Programmierung“

Universität Hamburg
DER FORSCHUNG | DER LEHRE | DER BILDUNG

2018-11-19

Wissenschaftliches Rechnen
Fachbereich Informatik
Universität Hamburg

Hintergrund: Speichermodel

Wie funktioniert der Arbeitspeicher (RAM)
aus der Sicht der CPU?

Hintergrund: Speichermodel

```
0xc0ff32f0: ...
0xc0ff3300: 00 00 00 03  c0 ff 33 50 | 00 00 00 00  00 00 00 00  ||  .... ..3J | .... ....
0xc0ff3310: 00 00 00 00  00 00 00 00 | 00 00 00 00  00 00 00 00  ||  .... .... | .... ....
0xc0ff3320: 00 48 65 6c  6c 6f 00 00 | 00 00 00 00  00 00 00 00  ||  .Hel lo.. | .... ....
0xc0ff3330: 00 00 00 00  00 00 00 00 | 00 00 00 00  00 00 00 00  ||  .... .... | .... ....
0xc0ff3340: 00 00 00 00  00 00 00 00 | 00 57 6f 72  6c 64 21 00  ||  .... .... | .Wor ld!.
0xc0ff3350: c0 ff 33 21  c0 ff 33 49 | c0 ff 33 24  00 00 00 00  ||  ..3! ..3I | ..3$ ....
0xc0ff3360: 00 00 00 00  00 00 00 00 | 00 00 00 00  00 00 00 00  ||  .... .... | .... ....
0xc0ff3370: ...
```

Hintergrund: Speichermodel

```
0xc0ff32f0: ...
0xc0ff3300: 00 00 00 03  c0 ff 33 50 | 00 00 00 00  00 00 00 00  ||  .... ..3J | .... ....
0xc0ff3310: 00 00 00 00  00 00 00 00 | 00 00 00 00  00 00 00 00  ||  .... .... | .... ....
0xc0ff3320: 00 48 65 6c  6c 6f 00 00 | 00 00 00 00  00 00 00 00  ||  .Hel lo.. | .... ....
0xc0ff3330: 00 00 00 00  00 00 00 00 | 00 00 00 00  00 00 00 00  ||  .... .... | .... ....
0xc0ff3340: 00 00 00 00  00 00 00 00 | 00 57 6f 72  6c 64 21 00  ||  .... .... | .Wor ld!.
0xc0ff3350: c0 ff 33 21  c0 ff 33 49 | c0 ff 33 24  00 00 00 00  ||  ..3! ..3I | ..3$ ....
0xc0ff3360: 00 00 00 00  00 00 00 00 | 00 00 00 00  00 00 00 00  ||  .... .... | .... ....
0xc0ff3370: ...
```

- `0xc0ff3300` (4 bytes, **int**), Wert: **3**
- `0xc0ff3304` (4 bytes, Pointer), Wert: **`c0ff3350`**
- `0xc0ff3350` (12 bytes, Array mit 3 Pointern), Wert: **`(c0ff3321, c0ff3349, c0ff3324)`**
- `0xc0ff3321` (6 bytes, String), Wert: **”Hello”**
- `0xc0ff3349` (7 bytes, String), Wert: **”World!”**
- `0xc0ff3324` (3 bytes, String), Wert: **”lo”**

Allgemein

Was ist ein Pointer?

- **Pointer sind Speicheradressen**
- *C initialisiert keine Pointer von sich aus*
  - nicht initialisierte Pointer können überallhin zeigen
  - werden sie verwendet, *darf das Programm die Platte formattieren!*

Was ist der Typ eines Pointers?

- Definiert, wie der Speicher an dieser Adresse *interpretiert* wird
  - Falscher Typ, falsche Interpretation!
- Pointer von verschiedenen Typen sind nicht gleich
  - Keine implizite Umwandlung, da sonst Daten falsch interpretiert würden

```
1 int *ip; //ip eine Variable, die die Adresse eines `int` enthällt
2          //Typ: ”pointer to int”
```

Operatoren: address-of und object-of

`&` address-of Operator

- Liefert die Adresse eines Wertes (Variable, Objekt, oder Array)
- Der Typ von `&x` ist ”pointer to <type of x>”

`*` object-of Operator

- Ermöglicht den Zugriff auf ein Objekt durch seine Adresse
- Inverse Operation zu `&`, `*&x` ist immer äquivalent zu `x` in C
- Argument muss ein Pointer sein

Beispiel[1]

```
1  int ar[20], *ip;
```

- Array und Pointer haben noch keine Verbindung

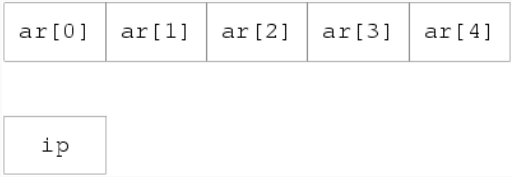

Abbildung 1: Schematische Darstellung

Beispiel[2]

```
1  int ar[20], *ip;
2  ip = &ar[3];
```

- Der Pointer `ip` zeigt auf das Arrayelement `ar[3]`

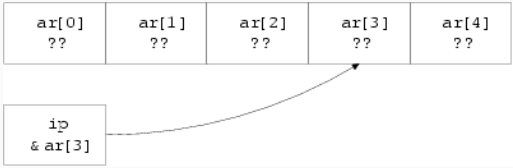

Abbildung 2: Schematische Darstellung

Pass by value

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

void no_change(int i) {
    i = 5;
}

int main(int argc, char** argv) {
    int a = 4;
    printf("before: a = %d\n", a);
    no_change(a);
    printf("after : a = %d\n", a);
    return 0;
}
```

```
before: a = 4
after : a = 4
```

Listing 1: Ausgabe

Pass by reference is explicit in C

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

void change(int* pi) {  //explicit pointer
    *pi = 5;  //explicitly dereferenced
}

int main(int argc, char** argv) {
    int a = 4;
    printf("before: a = %d\n", a);
    change(&a); //explicit taking of address
    printf("after : a = %d\n", a);
    return 0;
}
```

```
before: a = 4
after : a = 5
```

Listing 2: Ausgabe

Pointer to Pointer

- Ist ein Pointer in einer Variablen gespeichert,
  kann die Adresse des Pointers genommen werden
- Das Ergebnis ist ein ”pointer to pointer to ...”
- `int*** threeStarPointer;` ist erlaubt, aber ganz schlechter Stil
- Auf beliebige Datentypen anwendbar

```
1  int main() {
2      int i = 5;
3      int * pi = &i;
4      int ** ppi = &pi;
5
6      printf("%d\n",  i);
7      printf("%d\n",  * pi);
8      printf("%d\n",  ** ppi);
9  }
```

Pointer to Pointer: Beispiel

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

void printBytes(void* data, int byteCount) {
  uint8_t* bytes = data;
  printf("%016zx:", (uintptr_t)data);
  for(int i = 0; i < byteCount; i++) {
    printf(" %02x", bytes[i]);
  }
  printf(" \"");
  for(int i = 0; i < byteCount; i++) {
    printf("%c", bytes[i]);
  }
  printf("\"\n");
}

int main(int argc, char** argv) {
  printf("argc:"); printBytes(&argc, sizeof(argc));
  printf("argv:"); printBytes(&argv, sizeof(argv));
  printf("*argv:"); printBytes(argv, argc*sizeof(*argv));
  for(int i = 0; i < argc; i++) {
    printf("*argv[%d]:", i); printBytes(argv[i], strlen(argv[i]) + 1);
  }
}
```

Pointer to struct

- Pointer können auf alles zeigen, insbesondere auch auf `structs`
- Neuer Operator: `->`
- `a->b` ist äquivalent zu `(*a).b`
  Also: Zugriff auf Member eines `struct` hinter einem Pointer

Pointer to struct: Beispiel

```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);
}
```

void Pointer

Ein `void*` ist eine Adresse *ohne Typ*.

- Alle Pointer können implizit in einen `void*` umgewandelt werden
- Ein `void*` kann implizit in jeden anderen Pointer Typ umgewandelt werden
- Wer das missbraucht, dessen Festplatte ist zum Formattieren freigegeben!

NULL-Pointer

- Zeigt auf kein Objekt und ist somit ungültig
- Wird als `false` ausgewertet
- Dereferenzierung führt *meistens* zu einem Laufzeitfehler
- Signalisiert oft fehlende Argumente oder Fehler

```c
#include <stddef.h>
int main() {
    int *nothing1 = 0;
    int *nothing2 = (void*) 0;
    int *nothing3 = NULL;
    if(nothing1 || nothing2 || nothing3) {
        printf("unreachable printf() call\n");
    }
}
```

Addition und Subtraktion

Mit Pointern kann man rechnen

- Die Einheit der Pointerarithmetik sind immer ganze Objekte
- `intPtr + 1` addiert die Größe eines `int` zur Adresse hinzu
- `doublePtr + 2` addiert die Größe zweier `double` zur Adresse hinzu
- -> mit `void*` kann man nicht rechnen

```
1 int arr[5], *ptr = &arr[2];
2 assert(ptr - 2 == &arr[0]);
3 assert(ptr + 2 == &arr[4]);
4 assert((intptr_t)(ptr + 1) == (intptr_t)ptr + sizeof(*ptr));
5 assert((intptr_t)(ptr + 2) == (intptr_t)ptr + 2*sizeof(*ptr));
```

Differenz von Pointern

Die Differenz gibt die Anzahl der Elemente zwischen den Pointern zurück

```
1  ip1 = &ar[5];
2  ip2 = &ar[7];
3  assert(ip2 - ip1 == 2);
```

Addition und Subtraktion

Regeln:

- `ptr + integer -> ptr`
- `integer + ptr -> ptr`
- `ptr + ptr` ist verboten
- `ptr - integer -> ptr`
- `ptr - ptr -> integer`
- `integer - ptr` ist verboten

Inkrement und Dekrement `+=`, `-=`, `++` und `--` verhalten sich entsprechend

Vergleich von Zeigern

- Für den Zeigervergleich können die folgende Operatoren verwendet werden
  - `<`, `>`, `<=`, `>=`, `==`, `!=`
- Die Operatoren vergleichen Speicheradressen

Beispiel: Arrays and Pointers

```
1  for(ip = &ar[0]; ip < &ar[20]; ip++)
2      *ip = 0;
```

```
1  for(i = 0; i < 20; i++)
2      ar[i] = 0;
```

- Der Standard sichert zu, dass die Adresse `&ar[20]` benutzt werden kann, auch wenn dieses Element nicht exisiert
- Es darf aber nicht auf `arr[20]` zugegriffen werden, sonst darf das Program die Festplatte formattieren

Anwendungen von Pointern: Rückgabeargumente

Wenn eine Funktion mehrere Werte zurückgeben muss, werden meistens Pointer verwendet:

```
1 char* answer(int* out_answer) {
2     if(out_answer) *out_answer = 42;
3     return "Die Antwort";
4 }
```

Anwendungen von Pointern: Dynamischer Speicher

`malloc()` liefert einen Pointer zurück,
über den man auf den alloziierten Speicher zugreift

-> Basis für objektorientierte Programmierung

Anwendungen von Pointern: Datenstrukturen

Bäume und verkettete Listen:

Die Knoten enthalten Pointer auf andere Knoten,
oder `NULL` Pointer um das Ende der Kette zu signalisieren

Zusammenfassung

- Pointer sind Speicheradressen
  - Typ des Pointers definiert, was hinter der Adresse erwartet wird
  - Pointer können auch auf Pointer zeigen
- `NULL`-Pointer ist besonders
  - Signalisiert das Fehlen eines Wertes
- **`void`**`*` ist besonders
  - Adresse ohne Typ, wird als Black-Box verwendet
- Mit der Pointerarithmetik kann man relativ zu einer Speicheradresse in einem Array navigieren
- Die Speicheradressen sind vergleichbar

Beispiel

```
1 void double_swap ( double * p0 , double * p1 ) {
2     double tmp = * p0 ;
3     * p0 = * p1 ;
4     * p1 = tmp ;
5 }
```

Anwendungen von Pointern: Arrayzugriff

Fun-Fact:
`a[b]` ist definiert als `*(a + b)`,
Arrayzugriffe sind also *immer* Pointerarithmetik

Darum ist auch `2["foo"]` legal und liefert den `char 'o'` zurück...

Anwendungen von Pointern: Arrayzugriff

Fun-Fact:
`a[b]` ist definiert als `*(a + b)`,
Arrayzugriffe sind also *immer* Pointerarithmetik

Darum ist auch `2["foo"]` legal und liefert den `char 'o'` zurück...
(Addition ist kommutativ, also `2["foo"] == "foo"[2]`)

Pointer in anderen Sprachen

Jede ernsthafte Programmiersprache verwendet Pointer:
C, C++, Java, Javascript, Python, Ruby, Lisp, Scheme, etc.

Ausnahmen: Prolog(?), Bash, Brainfuck

Wenn eine Sprache behauptet, keine Pointer zu haben,
dann versteckt sie sie meistens nur.
C ist einfach nur die ehrlichste Sprache...