C-Kurs 2011: Arrays, Strings, Pointer

Sebastian@Pipping.org

15. September 2011

v3.0.35

Willkommen zum dritten Tag des C-Kurses.

Mein Name ist Sebastian Pipping. Das dritte Jahr spreche ich nun im Rahmen des C-Kurses über Arrays, Strings und Pointer.

Viel von dem, was ich die nächste Stunde vermitteln will, haben Sie gestern bereits gesehen, teilweise auch angewandt. Die nächsten 60 Minuten werde ich sowohl Inhalte wiederholen als auch tiefer einsteigen.

Ganz wichtig zur Vorlesung: Bitte unterbrechen Sie mich, wenn Verständnisfragen auftauchen. Das fördert die Lebendigkeit dieser Vorlesung und trägt dazu bei, dass alle am Ball bleiben, auch wenn es mal etwas theoretischer wird.

Ohne Pointer geht nichts.

Ich möchte beginnen mit dem Statement: „Ohne Pointer geht nichts“, wenigstens nicht in C.

Pointer sind Grundlage für:

- Arrays
- Strings
- Call by Reference
- Komplexe Datentypen

Pointer sind die Grundlage für Arrays, Strings, Call by Reference und komplexe Datentypen.

Themen

- Hello Pointer
- Call by Reference
- Pointer auf Pointer
- Vom Pointer zum Array
- Pointer-Arithmetik
- Strings, Längencodierung, String API
- Const Correctness
- Initialisierung von Strings
- Mehrdimensionale Arrays
- Programm-Argumente: `argc` und `argv`
- Weiterführende Themen
- Zusammenfassung

Und weil in C nichts ohne Pointer geht, werde ich neben Arrays und Strings auch Pointer selbst in Detail beleuchten.

In „Hello Pointer“ möchte ich Pointer an einem Minimalbeispiel vorstellen: Was ist das Konzept? Wie lautet die Syntax?

Danach möchte unter Verwendung von Pointern Call by Reference nachbauen, also Funktionen erlauben, ausgewählten Variablen der Welt um sie herum zu verändern.

Gelegentlich zeigen Pointer auch auf andere Pointer. Dazu werde ich ein Beispiel aus der realen Welt vorstellen.

In „Vom Pointer zum Array“ werden wir die Funktionsweise von Arrays näher kennenlernen.

Damit sind wir direkt beim Thema „Pointer-Arithmetik“ und was sich hinter diesem Begriff verbirgt.

Themen

- Hello Pointer
- Call by Reference
- Pointer auf Pointer
- Vom Pointer zum Array
- Pointer-Arithmetik
- Strings, Längencodierung, String API
- Const Correctness
- Initialisierung von Strings
- Mehrdimensionale Arrays
- Programm-Argumente: `argc` und `argv`
- Weiterführende Themen
- Zusammenfassung

Aufbauend auf unserem Wissen über Arrays werde ich dann Strings als spezielle Arrays vorstellen, wie deren Länge codiert wird und welche Konsequenzen das hat.

Bevor ich die zwei verschiedenen Arten der Initialisierung von Strings vorstellen kann, muss ich in „Const Correctness“ ein paar Grundlagen aufbauen: An ein paar Stellen ist die Frage „konstant oder nicht konstant“ *keine* Geschmacksache.

Auch für die Repräsentation von mehrdimensionalen Arrays gibt es in C zwei Varianten. Ihnen Hilfe für diese Stolperfalle mit auf den Weg zu geben, ist mir wichtig.

Mit dem bis dahin aufgebauten Verständnis sind wir dann in der Lage, die Argumente des Programmaufrufs auszulesen, was ich mit passendem Code demonstrieren werde.

Danach möchte ich mit einer Zusammenfassung und einem Ausblick schließen.

Was ist ein „Pointer“?

Also was ist ein „Pointer“?

„Pointer“ kann meinen eine . . .

A) Adresse
B) Variable, die eine Adresse speichert

„Pointer“ meint entweder eine Adresse — einen Wert — oder eine Variable, die eine Adresse speichert. Diese Mehrdeutigkeit ist gar nichts wirklich Neues: Das Wort „Integer“ verwenden wir ganz selbstverständlich zum einen für ganze Zahlen — sagen wir die 3 — und zum anderen für Variablen, die ganze Zahlen speichern: wir sagen „*i* ist ein Integer“.

„Pointer“ meint also eine Adresse oder eine Variable, die eine Adresse speichert.

Hello Pointer

Hello Pointer - lassen Sie uns dazu etwas Code ansehen.

Hello Pointer

```
#include <stdio.h>

int main() {
  int i = 3;
  int * p = &i;

  printf(“i == %d, p == %p\n”, i, p);
  *p = 4;
  printf(“i == %d, p == %p\n”, i, p);
  p = 5;
  printf(“i == %d, p == %p\n”, i, p);

  return 0;
}
```


Beginnen wir mit einem Gerüst, dass Ihnen inzwischen sehr bekannt vorkommen sollte.

Wir definieren einen Integer *i* — bisher nichts Neues.

Als Nächstes definieren wir einen Pointer *p*, der auf *i* zeigt. Der &-Operator liefert hierbei die Adresse — den Speicherort — von *i*. *p* ist vom Typ „int Stern“: ein Pointer auf einen `int`.

Danach schreiben wir den Wert 4 nach `*p`. Der *-Operator folgt der Adresse in *p* - wir schreiben also effektiv in den Speicher von *i*!

p selbst ist auch schreibbar — hier schreiben wir den Wert 5.

Als Letztes fügen wir noch ein paar printf()s ein, um die Werte von *i* und *p* auszugeben.

Lassen Sie uns das einmal ausführen ...

Ausgabe

```
# gcc hello_pointer.c -o hello_pointer
# ./hello_pointer
i == 3, p == 0xbfa3ad8c
                          | *p = 4;
i == 4, p == 0xbfa3ad8c
                          | p = 5;
i == 4, p == 0x5
```


Erstmal kompilieren wir das.

Dann ausführen ...

Integer *i* ist 3 — so wie wir es initialisiert hatten — und Pointer *p* zeigt auf *i*, das an Adresse „bfa und so weiter“ liegt. Nebenbei: Diese Adresse kann bei jeden Aufruf verschieden sein.

Zur Erinnerung: Dann kam `*p = 4;`.

Weil der *-Operator der Adresse in *p* folgt, landet die 4 in *i*. Der Pointer *p* selbst bleibt unverändert.

Dann kam `p = 5;`.

Diesmal wird *p* selbst verändert: *p* zeigt jetzt auf Adresse 5.

Lassen Sie mich das kurz zusammenfassen ...

Zusammenfassung 1/2

- Operator & liefert eine Adresse
- Operator * folgt einer Adresse
  (er *dereferenziert*)
- & und * sind komplementär; es gilt:
  `*(&x) = x = &(*x)`

Operator & liefert eine Adresse.

Operator * folgt einer Adresse. Man sagt auch „er dereferenziert“.

Besonders für die Mathematiker unter Ihnen: Diese beiden Operatoren sind komplementär. Wenn ich nach der Adresse von x frage und dann dieser Adresse folge, arbeite ich wieder mit x selbst — und anders herum.

Zusammenfassung 2/2

Pointer zeigen auf typisierte Daten:

`int *` $\neq$ `char *`

Pointer zeigen auf typisierte Daten. Es gibt einen Unterschied zwischen Pointern auf `int`s und Pointern auf `char`s. Diese Unterscheidung ist die Grundlage für Pointer-Arithmetik — dazu später mehr.

Zuvor möchte ich Ihnen zeigen, wie man mithilfe von Pointern Call by Reference in C nachbaut.

Call by Reference

Zur Erinnerung: Call by Reference bedeutet, einer Funktion zu erlauben, ausgewählten Variablen der Welt um sie herum zu verändern.

Call by Reference

```
#include <stdio.h>

void by_value    (int   x) {   x += 3;  }
void by_reference(int * x) {  *x += 4;  }

int main() {
  int i = 0;

  by_value(i);
  printf("i == %d\n", i);
  by_reference(&i);
  printf("i == %d\n", i);

  return 0;
}
```


Fangen wir an mit einem bekannten Gerüst: mit einer `main`-Funktion und einem Integer *i*.

Übergeben wir *i* nun an eine Funktion `by_value` und eine Funktion `by_reference`.

`by_value` verändert Paramater *x*, einer Kopie von *i*.

`by_reference` dagegen schreibt in den Speicher hinter der kopierten Adresse von `&i` und verändert damit den Wert von *i*.

Welchen Wert hat *i* also am Ende von `main()`?

<PAUSE>

Fügen wir noch ein paar `printf()`s ein und lassen es laufen ...

Ausgabe

```
# gcc call_by_reference.c \
      -o call_by_reference
# ./call_by_reference
i == 0
i == 4
```


Kompilieren, ausführen, *i* ist 0 und dann 4.

Pointer auf Pointer

Auch Pointer auf Pointer, manchmal „Doppelpointer“ oder „Pointer-Pointer“ genannt, werden regelmäßig gebraucht.

Zum Beispiel für ...

Anwendungen

- Mehrdimensionale Arrays
- Call by Reference *von* Pointern

... mehrdimensionale Arrays oder wenn Pointer selbst mittels Call by Reference verändert werden sollen.

Eine Funktion, die Sie selbst verwenden werden, macht Gebrauch davon: ...

man strtod

```
STRTOD(3)   Linux Programmer's Manual   STRTOD(3)

NAME
   strtod, strtof, strtold - convert ASCII string
                          to floating-point number

SYNOPSIS
   #include <stdlib.h>

   double strtod(const char *nptr, char **endptr);
   float strtof(const char *nptr, char **endptr);

...
```


... die Funktion `strtod()`.

Hier sehen Sie einen Ausschnitt aus der Manpage von `strtod()`.

Unten rechts sehen Sie einen Pointer auf einen Pointer. `strtod()` arbeitet mit Call by Reference. In einer unserer Übungsaufgaben können Sie genau mit dieser Funktion herumexperimentieren.

Wirklich schwer sind Doppelpointer nicht; außer einem weiteren Stern erwartet Sie auch syntaktisch nichts Neues.

Pointer auf Pointer

```
int i = 3;
int * p = &i;
int ** pp = &p;
```

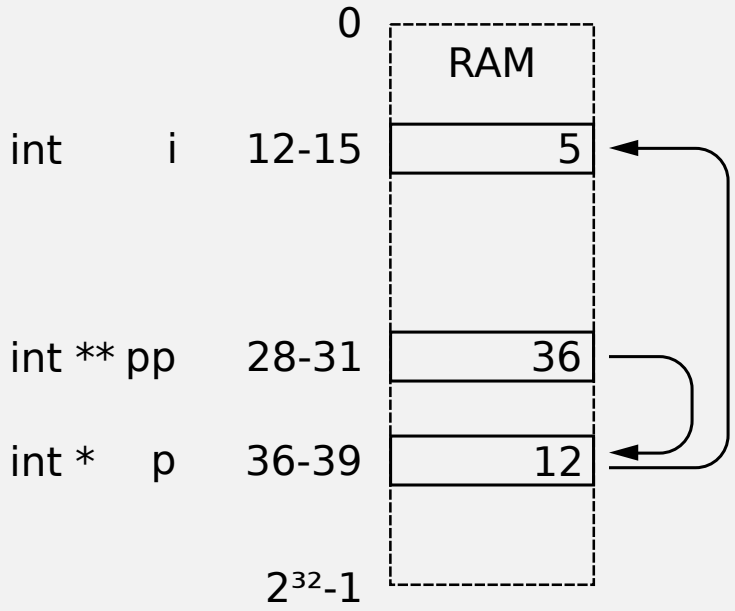

Das kennen Sie schon: Pointer *p* wird mit der Adresse von *i* initialisiert.

p ist ein einfacher Pointer.

Hier wird nun Doppelpointer *pp* mit der Adresse von *p* initialisiert: Wir setzen also eine weitere Indirektionsschicht oben drauf.

Vielleicht sollten wir uns das auch mal im RAM ansehen.

Variablen im RAM



Gibt es Fragen bis hierher?

Vom Pointer zum Array

In C hängen Arrays und Pointer zusammen.

Schauen wir uns an, wie dieser Zusammenhang genau aussieht.

Vom Pointer zum Array

```
#include <stdio.h>

void dump(const int * data, int count) {
  int i = 0;
  for (; i < count; i++) {
    printf("Field %d: %d\n", i + 1, *(data + i));
  }
}

int main() {


  return 0;
}
```


Dieses Gerüst kennen Sie schon.

Hier definiere ich eine Funktion `dump`, die Daten ausgeben soll. Die Daten sind Integer, die im Speicher direkt hintereinanderliegen: ein Array von `int`s.

Von diesen `int`s gibt es `count` viele, der erste von ihnen liegt an Adresse `data`.

Was ich jetzt machen kann, ist, mit einer simplen `for`-Schleife über die Daten iterieren und jedes Datum einzeln ausgeben.

An den jeweiligen Wert gelangen wir, indem wir zu der Basisadresse `data` den Index *i* addieren und dann der resultierenden Adresse folgen.

Sind Sie einverstanden, dass das so funktioniert?

<PAUSE>

Vom Pointer zum Array

```
#include <stdio.h>

void dump(const int * data, int count) {
  int i = 0;
  for (; i < count; i++) {
    printf("Field %d: %d\n", i + 1, data[i]);
  }
}

int main() {
  int const primes[] = {2, 3, 5, 7, 11};
  dump(primes, 5);
  return 0;
}
```


Wenn Sie jetzt denken, dass diese Syntax unnötig hässlich ist: Sie haben Recht: Dafür gibt es eine Kurzschreibweise. Diese Syntax kennen Sie vermutlich bereits von Java.

Lassen Sie uns nun die Funktion mit Daten füttern.

Hier definieren wir ein Array, das die 5 kleinsten Primzahlen enthält und rufen `dump()` damit auf.

Die Länge des Arrays müssen wir explizit übergeben, weil die Funktion `dump()` einem Pointer allein nicht ansehen kann, wie lang das Array dahinter ist: die Länge von Arrays muss mitherumgeschleppt werden!

Vom Pointer zum Array

```
#include <stdio.h>

void dump(const int * data, int count) {
  int i = 0;
  for (; i < count; i++) {
    printf("Field %d: %d\n", i + 1, data[i]);
  }
}

int main() {
  int const primes[] = {2, 3, 5, 7, 11, 13, 17};
  dump(primes, sizeof(primes) / sizeof(int));
  return 0;
}
```


Dass wir explizit sagen müssen, dass unser Array 5 Einträge lang ist, ist unflexibel und fehleranfällig.

Vermeiden lässt sich das hier mithilfe des `sizeof`-Operators. `sizeof` erkennt zur Compilezeit, wieviele Byte Speicher sein Argument belegt. Auf einer 32-Bit-x86-Machine belegt das Array `primes` 20 Byte, ein `int` belegt 4 Byte. 20 geteilt durch 4 sind 5.

Durch diesen Trick könnten wir nun zwei weitere Primzahlen an `primes` anhängen, ohne den Aufruf von `dump()` anpassen zu müssen.

Lassen Sie mich das eben zusammenfassen ...

Zusammenfassung (1/5)

Array-Zugriff via $[k]$ ähnlich Java

Der Array-Zugriff funktioniert mit eckigen Klammern und einem Index k wie auch in Java.

Zusammenfassung (2/5)

Array = Pointer auf das erste Element

Ein Array ist auch ein Pointer auf sein erstes Element. Mit dieser Adresse können wir auch direkt arbeiten, konkret: damit rechnen. Mehr dazu später.

Zusammenfassung (3/5)

$$a[k] = *(a + k)$$

Die Syntax mit den eckigen Klammern ist eine syntaktische Abkürzung. Eigentlich wird die Adresse des ersten Elements gelesen, der Index k addiert und dann der resultierenden Adresse gefolgt.

Array-Zugriff ist also auch dereferenzieren!

Zusammenfassung (4/5)

`sizeof(`*`variable`*`)`
=
Von *`variable`* belegter Speicher in Byte

Der Operator `sizeof` liefert die Anzahl Byte, die eine Variable im Speicher belegen wird.

Diese Auswertung geschieht zur Compilezeit.

Zusammenfassung (5/5)

`sizeof(`*type*`)`

=

Von *type*-Instanzen belegter Speicher in Byte

Des Weiteren kann `sizeof` auch die Größe von Typen bestimmen: wieviele Byte Speicher wird Typ `xy` belegen.

`sizeof` kann noch ein bisschen mehr, aber das soll uns hier reichen.

Pointer-Arithmetik

Kommen wir nun zu Pointer-Arithmetik, die wir vorhin bereits mehrfach verwendet haben.

Mit Pointern kann man rechnen, daher Arithmetik.

Beim Array-Zugriff passiert das automatisch. Aber was passiert da eigentlich genau?

Pointer-Arithmetik

```
int numbers[3];
char text[] = "software libre";
```

Es gilt:

1. `numbers[i] = *(numbers + i)`
2. `text[j] = *(text + j)`
3. `sizeof(char)` $\neq$ `sizeof(int)`

Für Operatoren +/- auf Pointern folgt daher:
Sprungweite variiert mit dem Typen!

Angenommen wir haben zwei Arrays: `numbers` und `text`. Eines von beiden ist auch ein String, aber das stört nicht weiter.

Ein paar Dinge müssen dann gelten:

1. Der Array-Zugriff auf Array `numbers` gehorcht dieser Formel; das hatte ich ja vorhin eingeführt.
2. Auch der Array-Zugriff auf Array `text` gehorcht dieser Formel.
3. `char` und `int` sind meist nicht gleich groß.

Gibt es da nicht ein Problem - kann das funktionieren? Ich geb die Frage mal an Sie weiter: Kann das funktionieren? <PAUSE>

Für die Operatoren Plus und Minus auf Pointern folgt: Die Sprungweite variiert mit dem Typen. Auf einer 32-Bit-Machine springt `numbers + 1` vier Byte wegen `sizeof(int)` $= 4$, aber `text + 1` springt nur ein Byte weiter wegen `sizeof(char)` $= 1$. Das ist der Grund, warum `int *` und `char *` nicht das Gleiche sind. Der Compiler braucht diese Information für Pointer-Arithmetik.

Strings

Kommen wir nun zu ganz speziellen Arrays, zu Strings.

Längencodierung

Pascal Strings

`"ABC"` ⟶ `{ 3, 65, 66, 67 }`

C Strings

`"ABC"` ⟶ `{ 65, 66, 67, 0 }`

Strings sind in C spezielle Arrays. Das besondere an ihnen ist, dass Sie Ihre Länge verraten.

In der programmiersprache Pascal sieht der String „ABC“ so aus: 4 Byte, ein Byte für die Länge — 3 — am Anfang gefolgt von den ASCII-Werten für die Großbuchstaben „A“, „B“ und „C“.

In C wird das anders gelöst. Aus „ABC“ macht der Compiler die ASCII-Werte von „A“, „B“ und „C“ *gefolgt* von Null, dem Nullterminator. Diese Null markiert das Ende des Strings.

Frage an Sie: Was fällt Ihnen für diese Varianten an Vor- und Nachteilen ein? <PAUSE>

Strings

```
"ABC"    ⟶ { 65, 66, 67, 0 }
"012"    ⟶ { 48, 49, 50, 0 }
"012\0"  ⟶ { 48, 49, 50, 0, 0 }
```

```
#include <string.h>
...
strlen("ABC\0\0");
= 3
```


Aus „ABC“ wird also das Array: 65, 66, 67, 0.

Dass die Null das Ende des Strings markiert heißt aber nicht, dass wir die Zahl Null nicht in einen String schreiben dürfen:

Dieser String - „012“ - wird ersetzt durch 48, 49, 50, 0. Der ASCII-Wert des *Zeichens* „0“ ist 48.

Künstlich den ASCII-Wert Null einfügen können wir auch: Aus 0, 1, 2, Backslash 0 wird 48, 49, 50, 0 und nochmal 0.

Die Länge eines Strings müssen wir nicht per Hand berechnen. Das macht die Funktion `strlen()` aus `string.h` für uns.

`strlen` von „ABC\0\0“ liefert den Wert 3.

my_strlen

```
int my_strlen(const char * str) {
  char const * const begin = str;
  while (*str) {
    str++;
  }
  return (str - begin);
}
```


Ich möchte nun mit Ihnen ein eigenes `strlen()` programmieren.

Ich habe unsere Funktion `my_strlen` genannt. Sie arbeitet auf einem `char`-Pointer „str“. Zu dem `const` kommen wir später.

Zunächst merken wir uns den Wert von str in einer eigenen Variable `begin`. Wir werden str verändern — quasi mit `str` über den String laufen — und würden den ursprünglichen Wert sonst verlieren.

Nun laufen wir in einer Schleife solange im String vorwärts, wie das aktuelle Zeichen nicht 0 ist, also nicht das Ende des Strings markiert.

Nach dieser Schleife zeigt `str` genau auf den Terminator. Was müssen wir nun zurückgeben um die Länge des Strings zu erhalten?

<PAUSE>

Richtig: `str - begin`. Auch das ist wieder Pointer-Arithmetik.

String API

Die String-API möchte ich nur kurz anreißen.

String API (Auszug)

- Vergleich
  - `strcmp`, `strncmp`, `strcasecmp`
- Suche
  - `strstr`, `strchr`
- Formatierung, Verkettung
  - `snprintf`
- Auswertung, Analyse
  - `strtol`, `sscanf`
- Clonen
  - `strdup`

Für die elementaren Operationen auf Strings sind Funktionen in der C-Standard-Library vorhanden: Vergleich, Suche, Formatierung, Clonen und mehr.

Manpages

```
# man string
# man string.h
# man stdio.h
# man stdlib.h
```


Die Liste der verfügbaren Funktionen und deren Anwendung ist in den Manpages `string`, `string.h`, `stdio.h` und `stdlib.h` beschrieben.

Bitte vermeiden:

- `atoi`, `atol`, `atoll`, `atoq`
- `gets`
- `scanf`, `fscanf`, ... mit `%s`
- `sprintf`, `vsprintf`
- `strcat`
- `strcpy`, `strncpy`

Ein paar Problemkandidaten möchte ich gerne mit Beispiel vorstellen. Was Sie hier sehen, sind Funktionen, die Sie eigentlich *gar nicht* verwenden solltet. Mehr dazu gleich.

- `gets`
- `scanf`, `fscanf`, `sscanf`, ... mit `%s`
- `sprintf`, `vsprintf`
- `strcat`
- `strcpy`

Buffer-Overflows

Besser: `snprintf`, `vsnprintf`, `fgets`

Beginnen wir mit dieser Gruppe von Funktionen. Buffer-Overflows sind mit ihnen schwer bis gar nicht vermeidbar.

Sehen wir uns dazu ein Beispiel an.

Beispiel scanf mit %s

```
#include <stdio.h>

void main() {
   char text[3];
   scanf(“%s”, text);   /* Richtig: %3s */
}
```

Buffer-Overflow für

$\texttt{strlen}(\textit{input}) \geq 3$

```
# echo 12345678901234567890123 | ./crash
Segmentation fault
```


Sehen wir uns `scanf` mit `%s` an. `scanf` liest hier Benutzereingaben und schreibt einen String in das `char`-Array `text`.

Sieht jemand ein Problem? <PAUSE>

Richtig: Dieser String kann länger sein, als das Array Platz hat; ab einer Länge von drei schreibt `scanf` über das Ende des Arrays hinaus. Auf meiner Maschine bringen 24 Zeichen das Programm zum Absturz.

Sicher ist hier, statt `%s` das explizite `%3s` zu verwenden.

Beispiel sprintf

```
void log_error(const char * error) {
   char text[256];
   sprintf(text, “ERROR: %s”, error);
   log_message(text);
}
```

Buffer-Overflow für

$\texttt{strlen(error)} > 256 - 7 - 1 = 248$

(Halbherzige) Abhilfe:

```
   snprintf(text, sizeof(text), .....);
```


Ein ähnliches Problem hat die `sprintf`-Familie.

Wie lang muss `error` sein, damit dieser Code knallt? <PAUSE>

Richtig: 249 zeichen oder mehr.

Mit snprintf wäre das nicht passert. Eigentlich muss hier aber dynamisch ausreichend Speicher angefordert werden.

- `strncpy`

Nullterminator nicht garantiert

Besser: `snprintf`

Diese Funktion ist auf ihre ganz eigene Art problematisch. Besser gleich `snprintf` verwenden.

Sehen wir uns ein Beispiel an.

Beispiel `strncpy`

```
char text[5];

strncpy(text, "Test", sizeof(text));
```
⟶ schreibt { 84, 101, 115, 116, 0 }

```
strncpy(text, "Hallo", sizeof(text));
```
⟶ schreibt { 72, 97, 108, 108, 111 }

⟶ nicht nullterminiert

```
snprintf(text, sizeof(text), "%s",
                                "Hallo");
```
⟶ schreibt { 72, 97, 108, 108, 0 }

Hier bitten wir `strncpy`, bis zu 5 Zeichen zu kopieren. Weil „Test“ nur vier Zeichen lang ist, läuft das glatt: `strncpy` kopiert 4 Zeichen und den Nullterminator.

Wieder bitten wir um das Kopieren von bis zu 5 Zeichen, bieten aber mehr Text an: „Hallo“ hat 5 Zeichen. `strncpy` schreibt 5 Zeichen, aber keinen Nullterminator.

Anders mit `snprintf`: Nur 4 Zeichen werden kopiert — das fünfte Zeichen ist der Nullterminator.

- `atoi`
- `atol`
- `atoll`
- `atoq`

Fehlerbehandlung fehlt

Besser: `strtol`, `strtoll`

Kommen wir zu etwas Lustigerem.

Bei atoi und seinen Verwandten fehlt jegliche Fehlerbehandlung. Warum ist mir schleierhaft.

Sehen wir uns das näher an.

Beispiel `atoi`

`atoi("Hallo")` $\longrightarrow 0$

`atoi("0")` $\longrightarrow 0$

Problem:

`atoi(`s`)` $= 0 \longrightarrow s = ?$

Idee:

Vorher s mit "0" vergleichen?

Sowohl aus „Hallo“ als auch aus dem String „0“ extrahiert `atoi` den Wert 0. Wir wissen also nur, dass die Konvertierung erfolgreich war, wenn das Ergebnis *nicht* 0 ist.

Wenn die Rückgabe 0 ist, wissen wir nichts.

Frage an Sie: lässt sich dieses Problem umgehen?

<PAUSE>

Ja, wir könnten vorher mit „0“ vergleichen. Wir müssten allerdings noch deutlich mehr Fälle abdecken:

Beispiel atoi

```
atoi("2")       ⟶ 2
atoi("0002")    ⟶ 2
atoi("+2")      ⟶ 2
atoi("  2")     ⟶ 2
atoi("2Hallo")  ⟶ 2
```


Aus all diesen Strings extrahiert `atoi` den Wert 2.

Was für 2 gilt, gilt auch für 0.

Überzeugt Sie das? <PAUSE>

Ausweg strtol

```
#include <stdlib.h>
#include <string.h>

int str_to_long(const char * text,
     long * output) {
   char * end;
   long number = strtol(text, &end, 10);
   if ((end != text) && (*end == '\0')) {
      *output = number;
      return 1;
   }
   return 0;
}
```


Die Lösung ist, `strtol` zu verwenden.

Stop. Für diesen Code braucht man etwas Ruhe und die Manpage von `strtol`. Hier in der Vorlesung würde ich gern mit anderem Stoff weitermachen. Meine Folien sind auch online zu finden.

Const Correctness

Kommen wir nun zu Const Correctness, zu korrekter Verwendung des Modifikators `const`.

Const Correctness (1/6)

Modifikator `const`
verbietet Schreibzugriff

Zu allererst sollte ich erwähnen, was `const` eigentlich tut: `const` verbietet Schreibzugriff auf ausgewählte Variablen.

Const Correctness (2/6)

```
const int const foo;
```


Bei dieser Deklaration des Integers `foo` wäre const an zwei Stellen erlaubt: links vom `int` und rechts vom `int`.

Auf welcher Seite man es schreibt, ist hier Geschmacksache.

Const Correctness (3/6)

```
_____ int _____ * _____ foo;
  |         |       |
 <2a>      <2b>    <1>
```


Bei Pointern — einfachen Pointern — kommt nun eine weitere Stelle hinzu, an der `const` stehen darf.

Ich benenne diese Stellen mal eben mit „1“, „2a“ und „2b“.

Const Correctness (4/6)

<1> _____ int _____ * const foo;

Verboten:

```
foo = ...;
```

<2a> const int _____ * _____ foo;

<2b> _____ int const * _____ foo;

Verboten:

```
foo[0] = ...;
```


Steht const an Position 1 — also ganz rechts bei `foo` — darf der Wert von `foo` selbst nicht mehr verändert werden.

Steht const hingegen beim `int` so darf der Inhalt *hinter* `foo` nicht verändert werden.

Diese beiden Fälle lassen sich auch kombinieren ...

Const Correctness (5/6)

<2a,1> const int _____ * const foo;

<2b,1> _____ int const * const foo;

Verboten:

```
   foo = ...;
foo[0] = ...;
```


Steht beim int und bei `foo` `const`, so darf weder `foo` selbst noch der Inhalt hinter `foo` verändert werden.

Schauen wir uns noch ein Beispiel mit Doppelpointern an ...

Const Correctness (6/6)

```
const int * const * foo;
```

Erlaubt:

```
foo = ...;
```

Verboten:

```
foo[0] = ...;
foo[0][0] = ...;
```


Was darf ich mit dieser Variable machen?

<PAUSE>

Richtig, ich darf `foo` selbst verändern.

Was ich nicht verändern darf, ist der Inhalt hinter `foo` und der Inhalt hinter dem Inhalt von `foo`.

Gibt es Fragen dazu?

<PAUSE>

Ich schulde Ihnen noch eine Erklärung zu Funktion `my_strlen` von vorhin ...

Const Correctness in APIs

```
int my_strlen(      char * str);
```

Funktion darf *Inhalt* von `str` verändern

`my_strlen("ABC")` gibt Compile-Fehler

```
int my_strlen(const char * str);
```

Funktion darf Inhalt von `str` *nicht* verändern

`my_strlen("ABC")` erlaubt und sicher

Hier sehen Sie zwei verschiedene Prototypen von `my_strlen`: einen mit `const` und einen ohne. Wo liegt der Unterschied?

Aus Sicht des Compilers darf die Variante ohne `const` den Inhalt von `str` verändern. Wenn wir diese Funktion mit der String-Konstante „ABC“ aufrufen, wirft der Compiler einen Fehler.

Der Typ von String-Konstanten ist `const char *`. Diese Variante von `my_strlen` erwartet zuviel vom übergebenen String: sie erwartet, den String verändern zu dürfen.

Bei der Variante mit `const` *darf* die Funktion den Inhalt von `str` nicht ändern. Ein Aufruf mit der String-Konstante „ABC“ ist daher erlaubt und sicher.

Initialisierung von Strings

Kommen wir nun zur Initialisierung von Strings.

In C ist das auf zwei sehr verschiedene Arten möglich. Sehen wir uns das näher an ...

Initialisierung von Strings

Variante „Mit Array“

```
char a[] = ‘‘Hallo’’;
sizeof(a) = sizeof(char)*(5 + 1)
```

Inhalt les- und schreibbar

Variante „Nur Pointer“

```
const char * p = ‘‘Hallo’’;
sizeof(p) = sizeof(char *)
```

Inhalt *nicht* schreibbar

Hier initialisieren wir einen String *a* mit „Hallo“. Im Speicher belegt *a* dann 5 + 1 Byte, 5 Byte für jeden Buchstaben, 1 Byte extra für den Nullterminator.

Diese Variante möchte ich „Mit Array“ nennen. *a* ist ein vollwertiges Array: der String in *a* ist les- und schreibbar.

Die zweite Variante nenne ich „Nur Pointer“. Ihre Syntax ist anders: wir deklarieren hier einen ganz normalen Pointer *p*. Das ist *p* auch im Hinblick auf seine Größe — ein ganz normaler Pointer: auf einer amd64-Machine 8 Byte groß. Der String hinter *p* liegt in einem besonderem, schreibgeschützten Speichersegment und ist daher *nicht* schreibbar.

Weil sein Inhalt nicht schreibbar ist, fehlt aus Sicht der Const Correctness bei dieser Variante auch ein `const`.

Schauen wir uns dazu noch einmal etwas Code an ...

Variante „Nur Pointer“

```
#include <stdio.h>

int main() {
  char * a = ‘‘hallo’’;
  char * b = ‘‘hallo’’;

  printf(‘‘a=%p\n’’
         ‘‘b=%p\n’’, a, b);

  a[0] = ‘X’;

  return 0;
}
```


Hier definieren wir zwei Strings in der Variante „Nur Pointer“, beide mit dem Text „hallo“.

Dann geben wir ihre Adressen aus. Zur Erinnerung: %p im Format-String steht für Pointer. Für ihren Inhalt hätten wir stattdessen %s geschrieben.

Danach überschreiben wir den ersten Buchstaben von *a* mit einem großen „X“. Schauen wir uns an, was passiert . . .

Ausgabe

```
# gcc rombad.c -o rombad
# ./rombad
a=0x80484dc
b=0x80484dc
Segmentation fault
```


Kompilieren, ausführen, . . . da sind *a* und *b*.

Wie Sie sehen — sie haben dieselbe Adresse!

a und *b* zeigen also auf den gleichen Speicherbereich. Wenn wir nun den Text hinter *a* verändern, ändert sich auch *b*.

Aber dazu kommt es nicht: Es knallt — wir bekommen einen Segmentation Fault.

Dieser Speicher ist nicht schreibbar, ich sagte es ja bereits.

Warum warnt uns der Compiler nicht? Wir hatten ihn bisher nicht darum gebeten. Holen wir das nach . . .

GCC Flag -Wwrite-strings

```
# gcc -Wall -Wextra -Wwrite-strings \
      rombad.c -o rombad
rombad.c: In function ‘main’:
rombad.c:4: warning: initialization discards
    qualifiers from pointer target type
rombad.c:5: warning: initialization discards
    qualifiers from pointer target type
```


Compilieren wir also noch einmal. `-Wall` und `-Wextra` möchte ich generell empfehlen, für dieses Beispiel wird aber noch `-Wwrite-strings` benötigt. Ja, wirklich.

Und da sind die Warnungen, die wir erwarten können. „discards qualifiers“ bedeutet: Da fehlt `const`.

Variante „Nur Pointer“

```
#include <stdio.h>

int main() {
  const char * a = “hallo”;
  const char * b = “hallo”;

  printf(“a=%p\n”
         “b=%p\n”, a, b);

  a[0] = ‘X’;

  return 0;
}
```


So sah unser Code eben aus.

Hier gehört eigentlich `const` hin.

Wenn wir dieses `const` einfügen, merkt der Compiler hier unten, dass wir böse Dinge tun, und verbietet uns das.

Damit wissen Sie jetzt ausgezeichnet Bescheid über die Initialisierung von Strings.

Mehrdimensionale Arrays

Für die Repräsentation von mehrdimensionalen Arrays gibt es in C zwei Varianten, die sich stark unterscheiden.

Wenn man über diesen Unterschied stolpert, steht man relativ im Dunkeln: Zu diesem Thema schweigen tielweise auch dickere C-Bücher. Im Internet dazu etwas zu finden, ist auch nicht einfach.

Nicht zuletzt auch, weil ich dieses Thema spannend finde, möchte ich daher kurz auf diese zwei Repräsentationen eingehen.

Mehrdimensionale Arrays

Zwei Varianten:

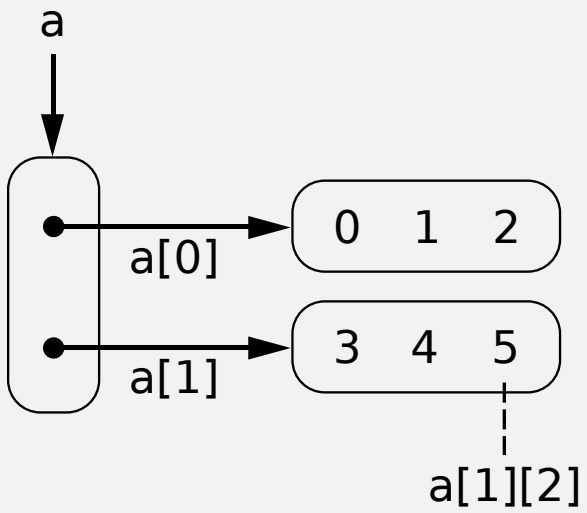
- Array von Arrays („deep array“)
- linear („flat array“)

Mehrdimensionale Arrays sind im Speicher entweder ein Array von Arrays (*deep arrays*) oder linear — quasi ein-dimensional (*flat arrays*).

Variante „Array von Arrays“

Schauen wir uns zuerst die Variante „Array von Arrays“ an.

2D-Array als Array von Arrays



Speichern wollen wir eine 2×3 Matrix, also ein Array mit 2 Zeilen und 3 Spalten.

Bei dieser Variante liegen beide Zeilen als je ein 1D-Array im Speicher.

Oben drauf setzen wir dann ein 1D-Array aus Pointern, die auf die Zeilen zeigen. Dieses Zeilen-Pointer-Array ist das eigentliche Array, hier *a* genannt.

`a[0]` zeigt auf die erste Zeile, `a[1]` auf die zweite.

Hier unten sitzt `a[1][2]`.

Im Code sieht das dann so aus: ...

2D-Array als Array von Arrays

```
void demo_deep(int * const * a) {
  a[1][2] *= 2;
}

int main() {
  int row0[] = {0, 1, 2};
  int row1[] = {3, 4, 5};
  int * const d[] = {row0, row1};
  demo_deep(d);
  return 0;
}
```


Wieder das Gerüst, dann definieren wir die zwei Zeilen und dann das Zeilen-Pointer-Array.

Dieses Array *d* kann ich dann an eine Funktion `demo_deep` übergeben.

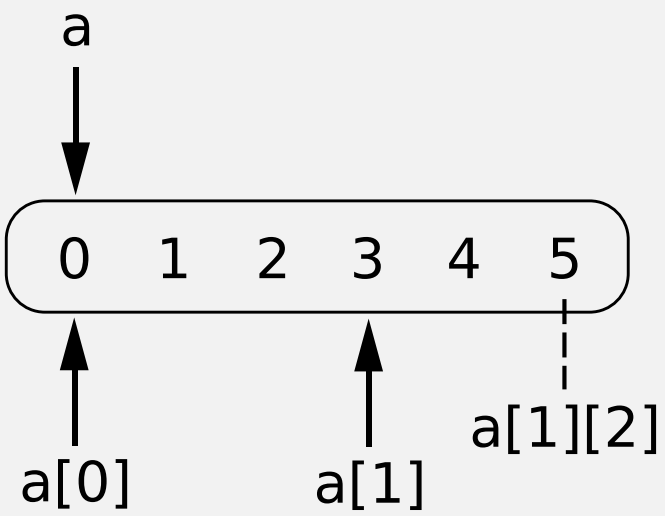
Diese Funktion tut nicht viel Sinnvolles: sie verändert einfach einen der Einträge unserer Matrix.

Das const habe ich eingefügt, weil `demo_deep` sonst erlaubt wäre, die Struktur der Matrix zu verändern, zum Beispiel Zeilen der Matrix zu vertauschen.

Variante „Linear“

Die Variante „linear“ sieht im Speicher völlig anders aus:

2D-Array linear



... Beide Zeilen liegen an einem Stück hintereinander im Speicher.

a zeigt auf das erste Element. Die Ausdrücke `a[0]`, `a[1]`, `a[1][2]` funktionieren wie erwartet, dereferenzieren aber anders als erwartet.

a ist kein Doppelpointer. Wäre er das, müsste bei *a* im Speicher ein Pointer liegen. Dort liegt aber ein Datum unseren Matrix.

Wie sieht das im Code aus?

2D-Array linear

```
void demo_flat(int a[][3]) {
  a[1][2] *= 2;
}

int main() {
  int f[][3] = { {0, 1, 2},
                 {3, 4, 5} };
  demo_flat(f);
  return 0;
}
```


Wir initialisieren die Matrix in einem Stück ... und übergeben sie dann an `demo_flat`.

Auffällig ist der Paramater *a* dieser Funktion: das ist kein Doppelpointer. Und die 3 ist nötig, damit der Code kompiliert. Der Compiler weiß sonst nicht, wieviele Elemente er je Zeile springen muss.

Das bedeutet auch, dass wir bei dieser Variante zur Zeit der Kompilierung wissen müssen, wie groß das Array sein soll: Das kann ein Ausschlusskriterium für diese Variante sein.

Variante „Hybrid“

Eine dritte „hybride“ Variante ergibt sich, wenn man beide Varianten kombiniert.

2D-Array-Hybrid

```
void demo_deep(int * const * a);
void demo_flat(int a[][3]);

int main() {
  int f[][3] = { {0, 1, 2},
                 {3, 4, 5} };
  int * const d[] = {f[0], f[1]};
  demo_deep(d);
  demo_flat(f);
  return 0;
}
```


Wir initialisieren *f* als Flat Array wie zuvor.

Oben drauf setzen wir dann ein Zeilen-Pointer-Array, das in *f* zeigt. Die Ausdrücke `f[0]` und `f[1]` liefern, was wir dazu brauchen.

Das resultierende Array *d* ist mit unserer früheren Funktion `demo_deep` kompatibel, während *f* selbst weiterhin `demo_flat` bedienen kann.

Zur Erinnerung: So sahen die Prototypen aus.

argc und argv

Zu guter Letzt möchte ich vorstellen, wie man auf Programmargumente zugreifen kann, die beim Aufruf übergeben wurden.

argc und argv

```
#include <stdio.h>

int main(int argc, char ** argv) {
  int i = 0;
  printf(“%d parameters\n”, argc - 1);
  for (; i < argc; i++) {
    printf(“[%d] %s\n”, i, argv[i]);
  }
  return 0;
}
```


Das ist die `main()`-Funktion, wie Sie sie inzwischen dutzende Male gesehen habt.

Das hier ist die Variante, die Zugriff auf Programmargumente erlaubt.

`argc` ist die Anzahl der Parameter, die wir übergeben bekommen: es ist die Anzahl der Einträge in `argv`.

Weil das erste Element unser Programmename ist, gibt es also `argc - 1` echte Parameter.

In einer Schleife laufen wir über das Array, das ein echtes „Deep Array“ aus Zeilen-Pointern ist.

`argv[i]` ist vom Typ `char *`, was hier nullterminierte Strings sind. Diese übergeben wir direkt an `printf()`.

Ausgabe

```
# gcc print_args.c -o print_args
# ./print_args free 'open source' software
3 parameters
[0] ./print_args
[1] free
[2] open source
[3] software
```


Kompilieren, ausführen, ... 3 Parameter.

Die Shell hat wegen der Anführungszeichen „open“ und „source“ als ein Element verstanden.

Element 0 enthält den Namen des Programms, so wie es aufgerufen wurde.

Weiterführende Themen

Weiterführende Themen: Ein paar Dinge habe ich absichtlich ausgelassen:

Weiterführende Themen

- Character Encodings, Unicode, `wchar_t`
- Sicherer Umgang mit Strings
- Der Typ `size_t`
- Void-Pointer
- Funktions-Pointer

. . . Wir haben uns nicht mit Character Encodings und Unicode beschäftigt. Für uns war Text ASCII-enkodiert. Das ist nicht immer so.

Die dynamische Längenberechnung von Strings kann zu einem Sicherheits-Problem werden, wenn man nicht sorgfältig damit umgeht. Auch das habe ich ausgeblendet.

Der Rückgabetyp unserer Funktion `my_strlen` hätte `size_t` sein sollen, nicht int. Was es damit auf sich hat, hat hier keinen Platz mehr gefunden.

Auch Void- und Funktions-Pointer habe ich außen vor gelassen. Beides sind einfache Themen, die Sie sich schnell selbst aneignen lassen.

Zusammenfassung

Wir haben aber auch viel geschafft. Wir haben neue Grundlagen aufgebaut und einige Stolpersteine umgedreht.

Was Sie unbedingt mitnehmen müssen:

Zusammenfassung

- `sizeof(`x`)` liefert die Größe von x in Byte
- Operator `&` liefert eine Adresse
- Operator `*` folgt einer Adresse
  (er *dereferenziert*)
- Pointer sind typisiert
- Arrays sind Pointer auf ihr erstes Element
- Strings sind nullterminininiert
- Const Correctness bei Pointern ist *keine* Geschmacksfrage

. . . sizeof(x) liefert die Größe von x in Byte.

Der Operator & liefert eine Adresse, der Operator * dagegen folgt einer Adresse. Man sagt er dereferenziert.

Pointer sind typisiert. Wir haben gesehen, wie sich Pointer-Arithmetik das zu nutze macht.

Arrays sind auch Pointer auf ihr erstes Element. Sie erinnern sich an die Formel von vorhin: `a[k] = *(a + k)`.

Strings sind nullterminiert, ihre Größe wird dynamisch berechnet. Auch das haben wir uns genau angesehen.

Const Correctness bei Pointern ist mehr als nur Geschmack, sondern macht einen echten Unterschied.

Wenn Sie das, was auf dieser Folie steht, wirklich verstanden haben — vielleicht sogar anderen erklären können — dann bin ich stolz darauf, Ihnen das beigebracht zu haben.

Eine Bitte

Veröffentlichen Sie Ihren Code
als Freie Software.

Eine letzte Bitte an Sie: Veröffentlichen Sie den Code, den Sie schreiben, als Freie Software. Für Fragen zu diesem Thema bin ich immer zu haben.

Danke!

Zeit für Ihre Fragen!

http://blog.hartwork.org/

Ich danke Ihnen für die Aufmerksamkeit.

Zeit für Ihre Fragen!