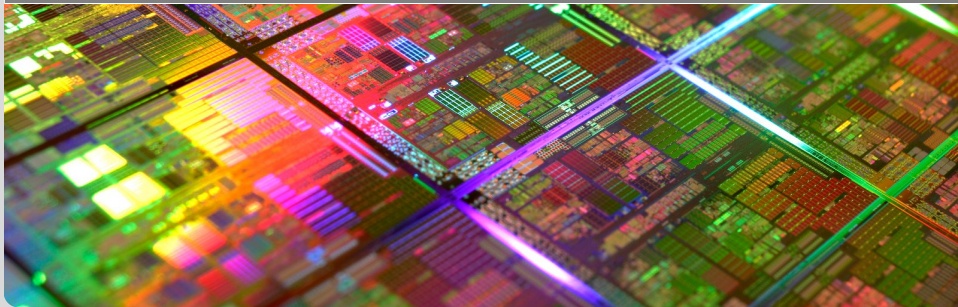


# RO-Tutorien 3 / 6 / 12

Tutorien zur Vorlesung "Rechnerorganisation"

Christian A. Mandery

WOCHE 2 AM 06./07.05.2013



- Datentypen in C
- Kontrollstrukturen in C
- Operatoren in C
- C-Zeiger
- Übungsaufgaben

- signed/unsigned char (immer 1 Byte)
- signed/unsigned short [int] (oft 2 Byte)
- signed/unsigned int (oft 4 Byte)
- signed/unsigned long [int] (oft 4 oder 8 Byte)
- signed/unsigned long long [int] (oft 8 Byte)
- Die Größe der einzelnen Datentypen ist bis auf Mindestgrößen und die Größe von char nicht vorgeschrieben!
- Es gilt aber immer:  
`sizeof(short int) ≤ sizeof(int) ≤ sizeof(long int)`

- signed/unsigned char (immer 1 Byte)
- signed/unsigned short [int] (oft 2 Byte)
- signed/unsigned int (oft 4 Byte)
- signed/unsigned long [int] (oft 4 oder 8 Byte)
- signed/unsigned long long [int] (oft 8 Byte)
- Die Größe der einzelnen Datentypen ist bis auf Mindestgrößen und die Größe von char nicht vorgeschrieben!
- Es gilt aber immer:

`sizeof(short int) ≤ sizeof(int) ≤ sizeof(long int)`

- float (“single precision”, 4 Byte)
- double (“double precision”, 8 Byte)
- Der Aufbau von Gleitkommazahlen ist standardisiert (IEEE 754) und wird in “Digitaltechnik und Entwurfsverfahren” tiefergehend behandelt

- Zeiger
- Arrays (Strings als Spezialfall)
- Selbstdefinierte Strukturen (`struct`)
- Enumerationen (`enum`)
- Unions (`union`)

- Zeiger
- Arrays (Strings als Spezialfall)
- Selbstdefinierte Strukturen (`struct`)
- Enumerationen (`enum`)
- Unions (`union`)

- Zeiger
- Arrays (Strings als Spezialfall)
- Selbstdefinierte Strukturen (`struct`)
- Enumerationen (`enum`)
- Unions (`union`)



- Zeiger
- Arrays (Strings als Spezialfall)
- Selbstdefinierte Strukturen (`struct`)
- Enumerationen (`enum`)
- Unions (`union`)

- Zeiger
- Arrays (Strings als Spezialfall)
- Selbstdefinierte Strukturen (`struct`)
- Enumerationen (`enum`)
- Unions (`union`)

- Sollten bereits aus Java (1. Semester) bekannt sein
- Bedingte Ausführung und Fallunterscheidung:
  - `if (<Bedingung>) ... [else ...]`
  - `switch (<Ausdruck>) {case <Wert>: ...; case <Wert>: ...; default: ...}`
- Schleifen:
  - `for (<Initialisierung>; <Bedingung>; <Iteration>) ...`
  - `while (<Bedingung>) ...`
  - `do ... while (<Bedingung>);`

- Sollten bereits aus Java (1. Semester) bekannt sein
- Bedingte Ausführung und Fallunterscheidung:
  - `if (<Bedingung>) ... [else ...]`
  - `switch (<Ausdruck>) {case <Wert>: ...; case <Wert>: ...; default: ...}`
- Schleifen:
  - `for (<Initialisierung>; <Bedingung>; <Iteration>) ...`
  - `while (<Bedingung>) ...`
  - `do ... while (<Bedingung>);`

# Weitere Schlüsselwörter zur Kontrollflusssteuerung

- `break`: Innerste Schleife sofort verlassen
- `continue`: Nächsten Schleifendurchlauf sofort starten
- `return`: Funktion (ggf. mit Rückgabewert) sofort verlassen
- `goto`: Spärlich verwenden!

- **Arithmetische Operatoren:** +, -, \*, /, %, a++, ++a, a--, --a
- **Logische Operatoren:** !, ==, !=, <, >, >=, <=, &&, ||
- **Bitweise Operatoren:** &, |, ^, <<, >>, ~
- **Zuweisungsoperatoren:** =, += (etc.), <<= (etc.), &= (etc.)
- **Sonstiges:** &, \*, a?b:c

- Arithmetische Operatoren: +, -, \*, /, %, a++, ++a, a--, --a
- Logische Operatoren: !, ==, !=, <, >, >=, <=, &&, ||
- Bitweise Operatoren: &, |, ^, <<, >>, ~
- Zuweisungsoperatoren: =, += (etc.), <<= (etc.), &= (etc.)
- Sonstiges: &, \*, a?b:c

- Arithmetische Operatoren: +, -, \*, /, %, a++, ++a, a--, --a
- Logische Operatoren: !, ==, !=, <, >, >=, <=, &&, ||
- Bitweise Operatoren: &, |, ^, <<, >>, ~
- Zuweisungsoperatoren: =, += (etc.), <<= (etc.), &= (etc.)
- Sonstiges: &, \*, a?b:c



- Arithmetische Operatoren: +, -, \*, /, %, a++, ++a, a--, --a
- Logische Operatoren: !, ==, !=, <, >, >=, <=, &&, ||
- Bitweise Operatoren: &, |, ^, <<, >>, ~
- Zuweisungsoperatoren: =, += (etc.), <<= (etc.), &= (etc.)
- Sonstiges: &, \*, a?:b:c

- Arithmetische Operatoren: +, -, \*, /, %, a++, ++a, a--, --a
- Logische Operatoren: !, ==, !=, <, >, >=, <=, &&, ||
- Bitweise Operatoren: &, |, ^, <<, >>, ~
- Zuweisungsoperatoren: =, += (etc.), <<= (etc.), &= (etc.)
- Sonstiges: &, \*, a?b:c

- Zeiger sind wichtiges Konzept in C!
- Ein Zeiger ist eine Variable, deren Wert eine Speicheradresse ist
- Werden bei der Deklaration durch einen Stern gekennzeichnet
- Bei Deklaration angeben, auf welchen Datentyp er zeigen wird (Achtung: Keine Prüfung zur Laufzeit!) oder einen void-Zeiger deklarieren
- Beispiele:
  - `int*, char*, char**, void**`
- Häufig: `int *a` statt `int* a`
  - `int* a, b` erzeugt einen Zeiger `a` und eine `int`-Variable `b`!
  - `int *a, *b` bzw. `int *a, b` sind besser lesbar

- Zeiger sind wichtiges Konzept in C!
- Ein Zeiger ist eine Variable, deren Wert eine Speicheradresse ist
- Werden bei der Deklaration durch einen Stern gekennzeichnet
- Bei Deklaration angeben, auf welchen Datentyp er zeigen wird (Achtung: Keine Prüfung zur Laufzeit!) oder einen void-Zeiger deklarieren
- Beispiele:
  - `int*`, `char*`, `char**`, `void**`
- Häufig: `int *a` statt `int* a`
  - `int* a`, `b` erzeugt einen Zeiger `a` und eine `int`-Variable `b`!
  - `int *a`, `*b` bzw. `int *a`, `b` sind besser lesbar

- Zeiger sind wichtiges Konzept in C!
- Ein Zeiger ist eine Variable, deren Wert eine Speicheradresse ist
- Werden bei der Deklaration durch einen Stern gekennzeichnet
- Bei Deklaration angeben, auf welchen Datentyp er zeigen wird (Achtung: Keine Prüfung zur Laufzeit!) oder einen void-Zeiger deklarieren
- Beispiele:
  - `int*`, `char*`, `char**`, `void**`
- Häufig: `int *a` statt `int* a`
  - `int* a`, `b` erzeugt einen Zeiger `a` und eine `int`-Variable `b`!
  - `int *a`, `*b` bzw. `int *a`, `b` sind besser lesbar

- Mit dem Adressoperator (&) ermittelt man die Adresse einer Variable
- Mit dem Dereferenzierungsoperator (\*) greift man auf die Speicherzelle zu, deren Adresse ein Zeiger enthält
- Nicht verwechseln: Stern bei der Deklaration vs. Stern als Dereferenzierungsoperator!
- Pfeil-Operator (a->b): Kurzschreibweise für (\*a) . b
- Aber: **Wozu braucht man Zeiger überhaupt?**

- Mit dem Adressoperator (&) ermittelt man die Adresse einer Variable
- Mit dem Dereferenzierungsoperator (\*) greift man auf die Speicherzelle zu, deren Adresse ein Zeiger enthält
- Nicht verwechseln: Stern bei der Deklaration vs. Stern als Dereferenzierungsoperator!
- Pfeil-Operator (a->b): Kurzschreibweise für (\*a) . b
- Aber: [Wozu braucht man Zeiger überhaupt?](#)

- Mit dem Adressoperator (&) ermittelt man die Adresse einer Variable
- Mit dem Dereferenzierungsoperator (\*) greift man auf die Speicherzelle zu, deren Adresse ein Zeiger enthält
- Nicht verwechseln: Stern bei der Deklaration vs. Stern als Dereferenzierungsoperator!
- Pfeil-Operator (a->b): Kurzschreibweise für (\*a) . b
- Aber: **Wozu braucht man Zeiger überhaupt?**



- Zeiger können wie andere C-Datentypen gecastet werden (schlechter Stil!)
- Im Unterschied zu Java-Referenzen keinerlei Schutzmechanismen:
  - Keine Garantie, dass Zeiger auf deklarierten Typ zeigt
  - Keine Garantie, dass Zeiger-Ziel aligned (ausgerichtet) ist
  - Keine Garantie, dass Zeiger auf gültigen Speicher zeigt (Nullzeiger sind explizit erlaubt und werden gezielt eingesetzt)
- Was passiert, wenn man (im x86 Protected Mode) auf ungültigen Speicher zugreift?

- Zeiger können wie andere C-Datentypen gecastet werden (schlechter Stil!)
- Im Unterschied zu Java-Referenzen keinerlei Schutzmechanismen:
  - Keine Garantie, dass Zeiger auf deklarierten Typ zeigt
  - Keine Garantie, dass Zeiger-Ziel aligned (ausgerichtet) ist
  - Keine Garantie, dass Zeiger auf gültigen Speicher zeigt (Nullzeiger sind explizit erlaubt und werden gezielt eingesetzt)
- Was passiert, wenn man (im x86 Protected Mode) auf ungültigen Speicher zugreift?

- Zeiger können wie andere C-Datentypen gecastet werden (schlechter Stil!)
- Im Unterschied zu Java-Referenzen keinerlei Schutzmechanismen:
  - Keine Garantie, dass Zeiger auf deklarierten Typ zeigt
  - Keine Garantie, dass Zeiger-Ziel aligned (ausgerichtet) ist
  - Keine Garantie, dass Zeiger auf gültigen Speicher zeigt (Nullzeiger sind explizit erlaubt und werden gezielt eingesetzt)
- Was passiert, wenn man (im x86 Protected Mode) auf ungültigen Speicher zugreift?

- 1 Welche Datentypen wurden in der Vorlesung genannt und welche Werte können darin gespeichert werden?
- 2 Was ist bei diesen Datentypen zu beachten?
- 3 Wie legt man in C eine Variable von einem entsprechenden Datentyp an und weist dieser einen Wert zu?

- 1 Welche Datentypen wurden in der Vorlesung genannt und welche Werte können darin gespeichert werden?
- 2 Was ist bei diesen Datentypen zu beachten?
- 3 Wie legt man in C eine Variable von einem entsprechenden Datentyp an und weist dieser einen Wert zu?

- 1 Welche Datentypen wurden in der Vorlesung genannt und welche Werte können darin gespeichert werden?
- 2 Was ist bei diesen Datentypen zu beachten?
- 3 Wie legt man in C eine Variable von einem entsprechenden Datentyp an und weist dieser einen Wert zu?

Welche Kontrollstrukturen sind in der Programmiersprache C verfügbar und wie werden diese verwendet?

- 1 Wie wird ein Zeiger auf einen Datentyp in C deklariert?
- 2 Was bedeutet der Ausdruck `&variable` in C?
- 3 Erklären Sie die folgenden Zeilen C-Code und was am Ende ausgegeben wird (erklären Sie dabei, welche Bedeutung die Zeichen `&` und `*` haben):

```
int a = 12;  
int* p;  
p = &a;  
printf("p = %d, a = %d", p, *p);
```



- 1 Wie wird ein Zeiger auf einen Datentyp in C deklariert?
- 2 Was bedeutet der Ausdruck `&variable` in C?
- 3 Erklären Sie die folgenden Zeilen C-Code und was am Ende ausgegeben wird (erklären Sie dabei, welche Bedeutung die Zeichen `&` und `*` haben):

```
int a = 12;  
int* p;  
p = &a;  
printf("p = %d, a = %d", p, *p);
```

- 1 Wie wird ein Zeiger auf einen Datentyp in C deklariert?
- 2 Was bedeutet der Ausdruck `&variable` in C?
- 3 Erklären Sie die folgenden Zeilen C-Code und was am Ende ausgegeben wird (erklären Sie dabei, welche Bedeutung die Zeichen `&` und `*` haben):

```
int a = 12;  
int* p;  
p = &a;  
printf("p = %d, a = %d", p, *p);
```

# Übungsaufgabe 4

Erstellen Sie ein Programm, das die Elemente in einem zweidimensionalen Array um den Wert 5 erhöht.

```
int b[X][Y] = { {0, 1, 2, 3},  
               {4, 5, 6, 7} };
```

```
/* ... */
```



Quelle: <http://xkcd.com/138/>