C-Vorkurs, Tag 2, 12. April 2016

Jonas Cleve, Sebastian Faase, Thomas Tegethoff

Heutiger Inhalt

- Variablen und einfache Datentypen
- Arrays, Zeiger, Strings
- Bedingte Ausführung
- Schleifen
- Funktionen

Variablen und Datentypen

Variablen in C

- C ist eine streng typisierte Sprache
  - Jede Variable hat einen festgelegten Datentyp
- Variablen sollten "sprechende" Namen haben
  - Gut: `begruessung`, `automarke`, `nenner`, ...
  - Schlecht: `var1`, `str2`, `laufen`, ...
- Es gibt verschiedene Standards für Variablenbenennung
  - Camel-Case: `isRunning`, `feindAnzahl`, ...
  - Underscore: `is_running`, `feind_anzahl`, ...
  - Sucht euch *einen* aus, benutzt sie *konsequent*!

Deklaration, Definition, Initialisierung

- Jedes Variable muss mit ihrem Typ *deklariert* bzw. *definiert* werden:

```
int a, b;
float c;
```

Deklaration, Definition, Initialisierung

- Jedes Variable muss mit ihrem Typ *deklariert* bzw. *definiert* werden:

```
int a, b;
float c;
```

- Vor der ersten Verwendung *sollte* jede Variable *initialisiert* werden

```
a = 15;
```

Deklaration, Definition, Initialisierung

- Jedes Variable muss mit ihrem Typ *deklariert* bzw. *definiert* werden:

```
int a, b;
float c;
```

- Vor der ersten Verwendung *sollte* jede Variable *initialisiert* werden

```
a = 15;
```

- *Deklaration* und *Initialisierung* passiert oft gleichzeitig

```
int anzahl_der_studierenden = 56;
```

Datentypen – Ganzzahlen

- Einfachster Datentyp: Ganzzahlen
- Gibt Datentypen für verschieden große Zahlen
- Vermutlich schon mal gehört: `short`, `int`, `long`
- Problem: Genaue Größe einer `int`-Variable variiert systemabhängig
  - Unklar, ob die Zahl 100 000 in einem `int` gespeichert werden kann, da er auch nur 16 Bit haben könnte

Datentypen – `inttypes.h`

- Wollen wissen / festlegen, wie viele Bit / Byte die verwendeten Variablen haben
- Daher verwenden wir Datentypen aus `inttypes.h`:
  - `int8_t`, `uint8_t`, `int16_t`, `uint16_t`, `int32_t`, `uint32_t`, `int64_t`, `uint64_t`
  - Die Zahl steht für die Anzahl der Bits in der Zahl
  - `intX_t` sind Zahlen *mit*, `uintX_t` Zahlen *ohne* Vorzeichen

Datentypen – `inttypes.h`

- Wollen wissen / festlegen, wie viele Bit / Byte die verwendeten Variablen haben
- Daher verwenden wir Datentypen aus `inttypes.h`:
  - `int8_t`, `uint8_t`, `int16_t`, `uint16_t`, `int32_t`, `uint32_t`, `int64_t`, `uint64_t`
  - Die Zahl steht für die Anzahl der Bits in der Zahl
  - `intX_t` sind Zahlen *mit*, `uintX_t` Zahlen *ohne* Vorzeichen

```
#include <inttypes.h> // Datentypen einbinden
uint8_t  guests;      // 0 <= Anzahl Gäste <= 255
uint64_t file_size;   // 64bit für Dateien >= 4,3 GB
int16_t  geburtsjahr; // im Bereich etwa -32768 bis 32767
```

Datentypen – Kommazahlen

- Kommazahlen: `float`

```
// Wichtig: Punkt statt Komma
float percentage = 0.345;     // 34,5%
```

Datentypen – Kommazahlen

- Kommazahlen: `float`

```
// Wichtig: Punkt statt Komma
float percentage = 0.345;     // 34,5%
```

- Kommazahlen mit doppelter Genauigkeit: `double`

```
double pi = 3.141592653589793;
```

Exkurs – Zahlen in der CPU

- CPU kann nur mit einzelnen Zahlen rechnen
- Es gibt dafür CPU-Register für Ganzzahlen (z.B. bis 64 Bit) und Kommazahlen
- Teilweise liegen Variablenwerte im Speicher
- Müssen zuerst in die Register geladen werden

Datentypen – Arrays

- Wollen mehrere gleichartige Dinge speichern
- Beipiel: Eine Liste von 5 Geburtsjahren

```
// 5 Plätze, je -32768 <= jahr <= 32767
int16_t geburtsjahr[5];
```

Datentypen – Arrays

- Wollen mehrere gleichartige Dinge speichern
- Beipiel: Eine Liste von 5 Geburtsjahren

```
// 5 Plätze, je -32768 <= jahr <= 32767
int16_t geburtsjahr[5];
```

- Zugriff über die Indizes 0 bis 4

```
geburtsjahr[0] = 1985;
geburtsjahr[1] = 1973;
// ...
geburtsjahr[4] = 1996;
```

Datentypen – Arrays

- Wollen mehrere gleichartige Dinge speichern
- Beipiel: Eine Liste von 5 Geburtsjahren

```
// 5 Plätze, je -32768 <= jahr <= 32767
int16_t geburtsjahr[5];
```

- Zugriff über die Indizes 0 bis 4

```
geburtsjahr[0] = 1985;
geburtsjahr[1] = 1973;
// ...
geburtsjahr[4] = 1996;
```

- Wie speichert eine Variable mehrere Werte?

Datentypen – Arrays II

```
uint8_t x = 5;  int32_t y = 9;  int16_t arr[5];
```

Datentypen – Arrays II

```
uint8_t x = 5;  int32_t y = 9;  int16_t arr[5];
```

Leerer Speicher

Datentypen – Arrays II

```
uint8_t x = 5;  int32_t y = 9;  int16_t arr[5];
```

Speicher mit belegter Variable x

Datentypen – Arrays II

```
uint8_t x = 5;  int32_t y = 9;  int16_t arr[5];
```

Speicher mit belegten Variablen x und y

Datentypen – Arrays II

```
uint8_t x = 5;  int32_t y = 9;  int16_t arr[5];
```

Speicher mit belegten Variablen x, y und arr

```
printf("%"PRId8"; %"PRId32"; %d", x, y, arr);
=> "5; 9; 18"
```

Datentypen – Arrays II

```
uint8_t x = 5;  int32_t y = 9;  int16_t arr[5];
```

Speicher mit belegten Variablen x, y und arr

```
printf("%"PRId8"; %"PRId32"; %d", x, y, arr);
=> "5; 9; 18"
```

Datentypen – Zeiger

- `arr` enthält keine Daten, *nur* die Adresse, wo diese liegen
- `arr` ist ein *Zeiger* auf Daten vom Typ `int16_t`, der Typ der Variablen ist `int16_t*`
- Wie kann ich auf Daten zugreifen, von denen nur die Adresse bekannt ist?
  - ```
    printf("arr[0] = %"PRId16" \n", *arr);
    printf("arr[1] = %"PRId16" \n", *(arr+1));
    printf("arr[2] = %"PRId16" \n", *(arr+2));
    // ...
    ```
  - `*(arr+2) = 17; // arr[2] = 17`
    ```

Datentypen – Zeiger II

```
int8_t x = 5;
int8_t *pX;
```

Speicher mit den Variablen x und pX

Datentypen – Zeiger II

```
int8_t x = 5;
int8_t *pX = &x;
```

pX zeigt jetzt auf x

Datentypen – Zeiger II

```
int8_t x = 5;
int8_t *pX = &x;

pX = 6;
```

pX zeigt auf Speicherstelle 6

Datentypen – Zeiger II

```
int8_t x = 5;
int8_t *pX = &x;

*pX = 6;
```

Die Variable, auf die `pX` zeigt, ist jetzt 6

Datentypen – Strings

- Text wird als Array von Zeichen (`char`) dargestellt

```
char text[10] = "Hallo!";
```

Variable mit 10 Zeichen Platz für den Text

Datentypen – Strings

- Text wird als Array von Zeichen (`char`) dargestellt

```
char text[10] = "Hallo!";
```

Variable mit 10 Zeichen Platz für den Text

Datentypen – Strings

- Können daher einzelne Zeichen verändern

```
char text[10] = "Hallo!";    text[1] = 'e';
```

Zweites Zeichen des Strings wurde geändert

Datentypen – Strings II

- Ein `char` enthält Zahlen von `0` bis `255`, entspricht also 256 verschiedenen Zeichen (siehe auch ASCII-Tabelle)
- Wissen, wo String anfängt, aber nicht, wo er aufhört
- Strings hören auf, wenn der erste Array-Eintrag die Zahl `0` ist

Kontrollstrukturen

Bedingungen: `if`, `else if`, `else`

- Programmausführung ist im Allgemeinen abhängig von Variablenwerten, Benutzereingaben, etc.

```
if (x == y) {
    // Mache etwas, wenn x und y gleich sind
    // ACHTUNG: Vergleich ist ==, nicht =
} else if (5 < a) {
    // Mache etwas, wenn x != y und a > 5
} else {
    // Mache etwas, wenn x != y und a <= 5
}
```

Bedingungen: `if`, `else if`, `else` II

- Vergleichsoperatoren: `<`, `>`, `<=`, `>=`, `==`, `!=`
- Logische Operatoren: `&&`, `||`, `!`

Bedingungen: `if`, `else if`, `else`

- Vergleichsoperatoren: `<`, `>`, `<=`, `>=`, `==`, `!=`
- Logische Operatoren: `&&`, `||`, `!`
- *Achtung:* Folgendes **funktioniert nicht**:

```
char text1[10] = "Hallo!";
char text2[10] = "Hallo!";

if (text1 == text2) {
    // Texte sind gleich
} else {
    // Texte sind nicht gleich
}
```

siehe `man 3 strncmp` / `man 3 strncasecmp`

Bedingungen: switch

- Oftmals möchte man eine Variable auf viele Werte testen

```
uint8_t z = // Irgendwo kommt die Zahl her

switch (z) {
    case 0:
        printf("Das ist sehr wenig!");
        break; // <- Wichtig, sonst bei "case 1" weiter
    case 1:
        printf("Besser, als nichts...");
        break;
    case 2: case 3: case 4: case 5:
        printf("Geht noch mit einer Hand.");
        break;
    default:
        printf("Zu viel!");
}
```

Schleifen: `while`

- Programmcode mehrfach ausführen, solange eine Bedingung zutrifft

```
int32_t  a = 34;
uint32_t b = 5;
int64_t  r = 1;

while (b > 0) {
    r = r * a;
    b--; // b = b - 1
}
```

Schleifen: `for`

- Auch Zählschleife genannt

```
for (int16_t i = 0; i < 100; i++) {
    // Hier ist i = 0, 1, ..., 99
}
```

(semantisch) äquivalent zu folgendem Code

```
int16_t i = 0;
while (i < 100) {
    // Hier ist i = 0, 1, ..., 99
    i++;
}
```

Schleifen: `do while`

- Wie die `while`-Schleife, Bedingung wird jedoch erst nach der ersten Ausführung getestet

```
int32_t  a = 34;
uint32_t b = 5;
int64_t  r = 1;

do {
    r = r * a;
    b--; // b = b - 1
} while (b > 0);
// Nur korrekt, wenn b initial nicht 0 ist
```

Funktionen

```
Rückgabetyp Name(Parametertyp Parametername, ...) {
    // Funktionskörper
}
```

Beispielsweise

```
int64_t hoch(int32_t  a, uint32_t b) {
    int64_t  r = 1;

    while (b > 0) {
        r = r * a;
        b--;
    }

    return r;
}
```

Was haben wir (bisher) heute gelernt?

Was haben wir (bisher) heute gelernt?

- Variablen und einfache Datentypen
- Arrays, Zeiger, Strings
- Bedingte Ausführung
- Schleifen
- Funktionen