C-Grundlagen

Ausblick

Wir beschäftigen uns nun damit wie wir ...

- ... eigene *Variablen* definieren können
- ... diese formatiert ausgaben können
- ... mit Hilfe von *Ausdrücken* rechnen können
- ... damit dann weitere *Anweisungen* an den Computer geben können
- ... diese wiederholt ausführen oder nur unter Bedingungen, indem wir *Kontrollanweisungen* einführen

Was haben wir da geschrieben?

```c
#include <stdio.h>

int main(void)
{
  puts("Hello World");
  return (0);
}
```

- Zeile 1: Möchten Funktionalität für Ein-/Ausgabe benutzen
- Zeile 3–7: Definition `main()`
  - Zeile 3: Signatur: Rückgabewert und Parameter
  - Zeile 4, 7: Beginn und Ende des Funktionsblockes
  - Zeile 5: Funktionsaufruf, gibt Argument auf der Konsole aus
  - Zeile 6: Beenden Funktion, geben 0 zurück

Programmumgebung: Allgemein

- Ein Programm besteht grob aus einer Menge an Funktionen
- Die Funktion mit der die Programmausführung beginnt, ist der *Programmeintrittspunkt*
- Nur sie wird vom Betriebssystem aufgerufen, aber sie kann andere Funktionen innerhalb des Programmes ausführen
- ... oder vorgefertigte Funktionen, als Teil von „Bibliotheken“

Programmumgebung: C, Hosted

- Der Programmeintrittspunkt heißt `main()`
- Für die Signatur dieser Funktion gibt es mehrere Möglichkeiten:
  - `int main(void)`
  - `int main(int argc, char *argv[])`
  - `int main(int argc, char **argv)`
  - `int main(const int argc, const char *const argv[argc+1])`
- Was diese bedeuten gucken wir uns gleich genauer an
- Jede Bibliothek bietet besteht außerdem sog. „Header“, welche beschreiben, welche Funktionen sie bereitstellt

Eigene Programme schreiben

- Wir schreiben erstmal nur die Funktion `main()` und schreiben keine neuen anderen Funktionen
- Stattdessen beschäftigen wir uns mit einfachen Anweisungen und Variablen
- Um Variablen zu benutzen, muss man sie dem Compiler bekannt machen (*deklarieren*)
- Sowohl Anweisungen als auch Deklarationen werden mit Semikoli beendet.

```
1 float pi, daumen;          // (Fliess-)Kommazahlen
2 daumen = 13.372;
3 pi = 3.141;
4 int antwort = pi*daumen;   // ~42
```

Variablen definieren

```
<typ> <variablenname> [ = <wert> ];
```

- der *Typ* ist zunächst immer `int`
- wir lernen morgen weitere Typen kennen
- Variablen können initialisiert werden
- Beispiel:

```
int foo;
int bar = 23, baz = 42, quux;
```

Sichtbarkeit

- Scope: Variablen sind nur sichtbar innerhalb ihres Blockes!
- Nur darin kann man sich auf die Variable über ihren Namen beziehen.
- Weitere Vernestung von Unterblöcken ist aber okay!

```
1 int a = 42;
2 int d;
3 {
4     int b = 10;
5     d = b/2 + 42;
6 }
7 int e = a + d; // b und c nicht sichtbar!
```

Formatierte Ausgabe

- Um berechnete Ergebnisse auszugeben, müssen sie in eine menschenlesbare Form umgewandelt werden
- `printf()` aus `stdio.h` ist eine mächtige Funktion, die das kann
- Dazu wird als erstes Argument ein sog. Formatstring übergeben, dieser bestimmt *wie* ausgegeben wird
- Bspw.: `printf("i ist: %d\n", i);`
  - Das `"%d"` ist ein Platzhalter, der das nächste Argument als Dezimalzahl interpretiert ausgibt
  - Das `"\n"` ist eine spezielle Escapesequenz welche die Zeile beendet (newline)
  - Der Rest in dem String wird direkt so ausgegeben
  - Für $i = 42$ also: „`i ist: 42`“

Formatierte Ausgabe

Wichtige Direktiven für `printf()`:

- `%d` nächstes Argument ist Zahl und wird in *dezimal* ausgegeben
- `%x` nächstes Argument ist Zahl und wird in *hexadezimal* ausgegeben
- `%s` nächstes Argument ist *Zeichenkette (string)* und wird ausgegeben
- `%c` nächstes Argument ist ASCII-Code und das *zugehörige Zeichen (character)* wird ausgegeben
- `%%` ein Prozentzeichen wird ausgegeben (kein Argument)

$\longrightarrow$ siehe **printf**(3) für mehr Details

Schleifen

- Manchmal möchte man wiederholt die gleichen Anweisungen ausführen
- Hierzu bietet C unterschiedliche Schleifen, heute werden wir kurz `for` vorstellen
- Wir nutzen diese Signatur von `main()`: `int main(int argc, char *argv[])`
- `argc` („Argument Count“) gibt an wie viele Argumente dem *Programm* auf der Konsole übergeben wurden
- `argv` („Argument Vector“) hält eben diese als Strings vor

Schleifen II

```c
int main(int argc, char *argv[])
{
    for (int i = 0; i < argc; i++) {
        printf("Arg %d: \"%s\"\n", i, argv[i]);
    }
    return (0);
}
```

- Die `for`-Anweisung besteht aus 3 Ausdrücken:

  ```
  for ( <Initialisierung> ; <Bedingung> ; <Inkrement> )
  ```

- In einem weiteren Block: Die zu wiederholende Anweisung

Ausführung

Ausführung auf der Konsole:

```
$ ./prog Das sind meine Argumente "in Anfuehrungszeichen"
Argument 0: ./prog
Argument 1: Das
Argument 2: sind
Argument 3: meine
Argument 4: Argumente
Argument 5: in Anfuehrungszeichen
```

If-Aneweisung

- Nimmt nur einen Ausdruck und führt den Block aus, wenn es wahr ist, sonst nicht
- Falls darauf ein `else` folgt, wird anstelle dieser Block ausgeführt

```
1  int main(int argc, char *argv[])
2  {
3      if (argc < 3) {
4          printf("Maximal ein Argument uebergeben\n");
5      } else {
6          printf("Mehr als ein Argument uebergeben\n");
7      }
8      /* alles folgende wird so oder so ausgefuehrt */
9      printf("Hallo Welt!\n");
10 }
```

Parsen von Zahlen: `atoi()`

Die übergebenen Argumente liegen in Form von Zeichenketten vor. Um eine Zeichenkette einer Dezimalzahl in einen Integer umzuwandeln gibt es die Funktion `atoi()`:

```c
#include <stdlib.h> // int atoi(char *str);

int main(void)
{
  int a = atoi("3"); // a = 3
}
```

Operanden und Operatoren: Ausdrücke

- Nun da wir (arithmetische) *Operanden* haben, benötigen wir noch *Operatoren*
- Zusammen haben wir (arithmetische) Ausdrücke
- Operanden können
  - Konstanten (42, `'?'`) und Stringliterale (`"FOO"`),
  - Variablen, oder auch
  - weitere Ausdrücke sein.
- Operatoren sind nicht nur arithmetische Operatoren, sondern auch Zuweisungen und Abfragen oder Kondizionale.

Operanden und Operatoren: Operatoren

Nützliche Operatoren

`+`, `-`, `*`, `/`, `%` Arithmetik (addieren, subtrahieren, multiplizieren, dividieren, Modulo)

`&`, `|`, `^`, `~` bitweise Logik (und, inklusives oder, exklusives oder, nicht)

`&&`, `||`, `!` Kurzschlusslogik (und, oder, nicht)

`==`, `!=` Vergleiche (gleich, ungleich)

`<`, `>`, `<=`, `>=` Ordnung (kleiner, größer, kleinergleich, größergleich)

`=`, *op*`=` Zuweisungen (`a op= b` ist wie `a = a op b`)

`?:` ternärer Operator (`a ? b : c` ist `b` falls `a` wahr, sonst `c`)

`<<`, `>>` Schübe (nach links, nach rechts)

`++`, `--` Inkrement, Dekrement (je als Prä- und Post- Inkrement und Dekrement)

$\longrightarrow$ siehe Skript für Details

Die Auswertung von Ausrücken

- Die Auswertung eines Audrucks berechnet ihren Wert
- Hierbei werden die einzelnen Teile des Ausdrucks wiederum einzeln ausgewertet
- ...also auch Funktionsausführungen, Zuweisungen, etc.

```
1 int    a = 42;
2 int    b =  8;
3 int    c = (a+b)/5 + (7*3) + 12*5/3 + atoi("24");
4 _Bool  d = a == c;
5 int    e = (a = b);
6 b = c;
7 a == c;
8 42;
```

Anweisung

- Ein Ausdruck an sich kann nie ausgeführt werden
- Hierzu benötigen wir eine Anweisung
- Einfachste Anweisungen sind Ausdrücke mit einem Semikolon beendet
- Jede Anweisung wird sequentiell abgearbeitet

```
1 int   a = 42;
2 int   b =  8;
3 int   c = (a+b)/5 + (7*3) + 12*5/3 + atoi("24");
4 _Bool d = a == c;
5 int   e = (a = b);
6 b = c;
7 a == c;
8 42;
```

Anweisungen im Verbund: Blöcke

- Anweisungen werden in Blöcken gruppiert
- Ein Block ist äquivalent zu einer Anweisung und kann anstelle einer solchen auftreten
- Blöcke werden mit geschweiften Klammern umschlossen
- Haben Eigenschaften bezüglich der „Sichtbarkeit“ von Variablen

```
1 int a = 42;
2 int d;
3 {
4     int b = 10;
5     d = b/2 + 42;
6 }
```

Kontrollfluss

- Wenn wir von „Kontrollfluss“ reden, meinen wir die Abfolge von „Anweisungen“ bzw. Blöcken
- Um Abfolge der Anweisungen zu verändern gibt es Kontrollanweisungen
- Das sind z. Bsp. If-Else und Schleifen (For, While)
- Eine `if`-Anweisung kontrolliert, ob das folgende Statement ausgeführt werden soll
- Hierfür evaluiert es einen Ausdruck auf Ungleichheit mit 0 (true)
- Analog wiederholt eine `while`-Schleife das Statement, solange die Bedingung wahr ist
- Äquivalent können wir anstatt der Statements auch Blöcke kontrollieren
- … welche wiederum aus anderen (Control-) Statements bestehen

```c
int a = 0;
if (a == 0)
    puts("a = 0");
while (a < 10)
    a = a + 1;                    // alt.: a += 1, ++a / a++
for (int i = 0; i < 10; i++)
    puts("i < 10");
    puts("foo");                  // Nicht mehr i.d. Schleife!

while (a > 0) {                   // Block anstatt Statement
    puts("a > 0");
    a--;
}
```