Programmstruktur

Ein C-Programm besteht aus einer Menge an Funktionen, wobei die Funktion `main()` eine spezielle Rolle einnimmt, sie ist der Programmeintrittspunkt.

Funktionen bestehen aus einem *Deklarator* und einem *Block*, diese werden von geschweiften Klammern umschlossen. Der Funktionsblock ist eine Liste von *Deklarationen*, *Anweisungen* und weiteren Blöcken, die sequentiell abgearbeitet werden. Sowohl Deklarationen als auch Anweisungen werden mit Semikoli beendet.

Anweisungen

Es gibt mehrere Arten von Anweisungen:

Leere Anweisung Ein Semikolon, tut nichts

Ausdrucksanweisung Besteht aus einem Ausdruck, der ausgewertet wird

Zusammengesetzte Anweisung Ein Block, kann überall dort benutzt werden, wo Anweisungen erlaubt sind

Ausdrücke

Ausdrücke sind Folgen von *Operatoren* und *Operanden* (Variablen, etc.), die …

- einen Wert berechnen (z. B. `a + b * c`), oder
- ein Objekt oder eine Funktion bezeichnen (dazu später mehr), oder
- eine Nebenwirkung auslösen (z. B. `a = b`, `printf("Hallo")`), oder
- eine Kombination davon.

Die Operanden können Konstanten (z. B. `42`, `'?'`), Literale / Stringkonstanten (z.B. `"ABC"`), Variablen, oder andere Ausdrücke sein.

Operatoren

Operatoren verbinden Operanden zu Ausdrücken, ranghöhere Operatoren binden dabei stärker als rangniedrigere Operatoren (Wie Punkt-vor-Strich, nur mit mehr Ebenen). Die Assoziativität „links nach rechts“ bedeutet: `a + b + c + d` ist wie `((a + b) + c) + d`; „rechts nach links“: `a = b = c = d` ist wie `a = (b = (c = d))`.

Die Arten der Operatoren können wir gliedern in Pre- und Postfix-Operatoren (einstellig) und Infix (zweistellig). Manche Operatoren arbeiten logisch/arithmetisch, manche kondizional, manche weisen zu, manche bestimmen die Reihenfolge von Ausführung. Außerdem arbeiten manche Operatoren auf den eigentlichen Zahlenwerten, und manche auf den bitweise.

Postfix-Operatoren sind einstellige Operatoren die *hinter* dem Operanden stehen.

- `()` ruft Funktionen auf (z. B. `puts()`); funktioniert wie `call` in Assembler
- `[]` indiziert Arrays (z. B. `argv[1]`)
- `++` (z. B. `i++`) inkrementiert seinen Operanden und liefert den vorherigen Wert zurück
- `--` wie `++`, dekrementiert aber

Prefix-Operatoren sind einstellige Operatoren die *vor* dem Operanden stehen.

- `!` ist logische Verneinung. `!0` ist `1`, alles andere wird `0`.
- `~` ist bitweises Komplement. Jedes Bit im Operand wird invertiert. Wie `not` in Assembler.
- `+` `-` einstelliges + und −, wie in der Mathematik. `-` ist wie `neg` in Assembler.
- `++` `--` (z. B. `++i`) inkrementieren oder dekrementieren ihre Operanden und lieferen den neuen Wert zurück. Wie `inc` bzw. `dec` in Assembler.
- `()` wandelt den Typen (z. B. `(int)argv`)
- `&` gibt die Adresse seines Operanden zurück. Wie `lea` in Assembler.
- `*` dereferenziert seinen Operand. `*x` ist wie `[x]` in Assembler.

arithmetische Operatoren Haben die gleiche Rangfolge wie in der Mathematik üblich. Wir dabei auf Ganzzahlen gerechnet, so wird gegen 0 gerundet (also abgeschnitten). Werden zwei Operanden verschiedenen Types verrechnet, so wird die Berechnung mit dem genaueren Typ ausgeführt.

- `+` `-` sind Addition und Subtraktion, wie `add` und `sub` in Assembler.
- `*` ist Multiplikation, wie `imul` in Assembler.
- `/` `%` sind Division und Modulo, wie `idiv` in Assembler. Division durch 0 ist *undefiniertes Verhalten.*

Bitshifts Operieren auf Bit-Ebene und schieben Bitmuster nach links oder rechts.

- `<<` schiebt nach links; `a << n` schiebt `a` um `n` Stellen nach links. Wie `shl` in Assembler.

  Mathematisch äquivalent zu einer Multiplikation mit 2^n.
- `>>` schiebt nach rechts; `a >> n` schiebt `a` um `n` Stellen nach rechts. Wie `shr` bzw. `sar` in Assembler.

  Mathematisch äquivalent zu einer Division durch 2^n mit Rundung gegen $-\infty$.

Relationale Operatoren Vergleichen Zahlen und geben 0 (für falsch) und 1 (für richtig) zurück. Zahlen können beliebig verglichen werden, Zeiger nur auf (Un)gleichheit. Wie `cmp` in Assembler

- `==` `!=` vergleichen auf Gleichheit oder Ungleichheit
- `<=` `>=` vergleichen auf größer-gleich oder kleiner gleich
- `<` `>` vergleichen auf kleiner oder größer

Bitweise Operatoren Arbeiten auf den Bits ihrer Operanden. Sie sind nur auf Ganzzahlen zulässig.

- `&` bitweises und, wie `and` in Assembler
- `|` bitweises inklusives oder, wie `or` in Assembler.
- `^` bitweises exklusives oder, wie `xor` in Assembler.

Logische Operatoren Interpretieren 0 oder Nullzeiger also „falsch“ und alles andere als „wahr“. Ihr zweiter Operand wird nur falls nötig ausgewertet (Kurzschlussverhalten / Lazy Evaluation).

- `&&` logisches und. `a && b` wertet `a` aus. Wenn `a` falsch ist, gebe 0 zurück. Sonst werte `b` aus und gebe 0 oder 1 zurück, je nachdem, ob `b` falsch oder wahr ist.
- `||` logisches inklusives oder. `a || b` wertet `a` aus. Wenn `a` wahr ist, gebe 1 zurück. Sonst werte `b` aus und gebe 0 oder 1 zurück, je nachdem, ob `b` falsch oder wahr ist.

Der konditionale Operator (Ternärer Operator) Hat drei Argumente und wirkt wie eine `if`-Anweisung als Ausdruck.

- `?:` Der Ausdruck `a ? b : c` liefert `b` zurück, falls `a` wahr ist, sonst `c`. Wie `cmovCC` in Assembler.

Zuweisungsoperatoren

- `=` weißt einen Wert zu. Der Ausdruck `a = b` weißt `a` den Wert `b` zu. Wie `mov` in Assembler.
- Ein Operator `a X= b` ist äquivalent zu `a = a X b`.
- `+= -=`
- `*= /= %=`
- `&= |= ^=`
- `<<= >>=`

Kommaoperator Vor allem dort nützlich, wo genau eine Anweisung erwartet wird, aber mehrere Ausdrücke ausgewertet werden sollen.

Achtung! Nicht jedes Komma ist ein Kommaoperator.

- `,` wertet zwei Ausdrücke aus. `a, b` wertet erst `a` und dann `b` aus und gibt den Wert von `b` zurück.

Operator	Assoziativität
`() [] -> .`	links nach rechts
`! ~ ++ -- - (type) * & sizeof _Alignas`	rechts nach links
`* / %`	links nach rechts
`+ -`	links nach rechts
`<< >>`	links nach rechts
`< <= > >=`	links nach rechts
`== !=`	links nach rechts
`&`	links nach rechts
`^`	links nach rechts
`|`	links nach rechts
`&&`	links nach rechts
`||`	links nach rechts
`?:`	rechts nach links
`= += -= *= /= %= <<= >>= &= ^= |=`	rechts nach links
`,`	links nach rechts

Dekarationen

Syntax für primitive Datentypen:

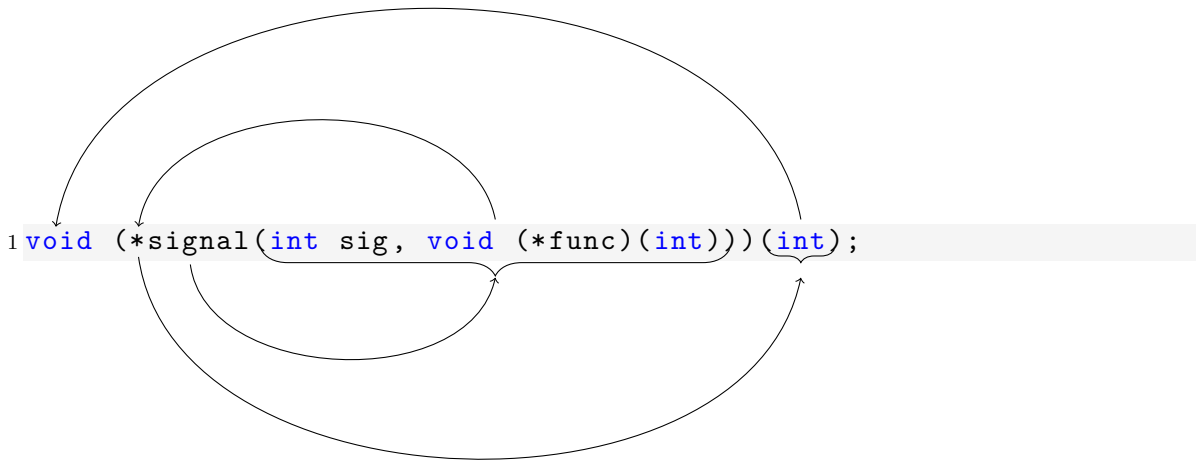
Typ Bezeichner [`=` *Ausdruck*] [`,` *Bezeichner* [`=` *Ausdruck*] `,` …] `;`

Spezielle Deklaratoren existieren für Pointer, Arrays, und Funktionen:

Typ Deklarator [`=` *Ausdruck*] [`,` *Deklarator* [`=` *Ausdruck*] `,` …] `;`

Intuititiv sieht ein Deklarator aus wie ein Ausdruck, den man auf die Variable anwenden kann, um den zugrundeliegenden Typen zu erhalten. Bspw. deklariert `int *foo;` die Variable `foo` als Pointer auf `int`, denn `*foo` ist ein `int`.

Deklaratoren müssen von Innen nach außen gelesen werden: `int *bar()` deklariert `bar` als Funktion, die einen Zeiger auf einen `int` zurückgibt, denn `*bar()` ist ein `int`. Hierbei folgen die Rangfolge der Deklaratoren der Operatorenrangfolge.



`signal()` ist also eine Funktion, die als Argumente …

1. einen `int sig` und
2. einen Pointer `func` auf eine Funktion, die
   a) ein `int` nimmt und
   b) `void` zurückibt

… bekommt. Außerdem gibt sie einen Pointer auf eine Funktion zurück, die

1. einen `int` nimmt und
2. `void` zurückgibt.

Datentypen

Primitive Datentypen Alle ganzzahligen Datentypen sind einfach so (außer `char`) oder mit `signed` vorzeichenbehaftet. Sie können mit `unsigned` vorzeichenlos gemacht werden

- `char` Mindestens ein Byte, damit kleinster Typ. Häufig für Buchstaben verwendet, speziell in ASCII-Codierung.
- `\_Bool` Boolean. So groß wie ein `char`, aber nur 0 oder 1.
- `int` Ganzzahl. Mindestens 16 Bit, üblicherweise 32 Bit.
- `float` Gleitkommazahl einfacher Präzision. Eigentlich immer 32 Bit.
- `double` Gleitkommazahl doppelter Präzision Eigentlich immer 64 Bit.

Es gibt noch Qualifikatoren, mit denen man `int` und z. T. auch `double`-Typen in der größe verändern kann:

- `short int` kurze Ganzzahl. Mindestens und eigentlich immer 16 Bit.
- `long int` lange Ganzzahl. Mindestens 32 Bit, üblicherweise 32 oder 64 Bit.
- `long long int` sehr lange Ganzzahl. Mindestens und eigentlich immer 64 Bit.
- `long double` Gleitkommazahl hoher Präzision. Mindestens 64 Bit, auf x86 80 Bit.

Wichtige abgeleitete Datentypen

Aus `<stddef.h>`:

- `size_t` Vorzeichenloser Ganzzahltyp, der groß genug für die Größe jedes Objektes ist. Idealer Typ für Schleifenzähler o. ä.

Aus `<stdint.h>`:

- `int`N`_t` Vorzeichenbehafteter Ganzzahltyp mit genau N Bits, $N \in \{8, 16, 32, 64\}$.
- `uint`N`_t` Vorzeichenloser Ganzzahltyp mit genau N Bits, $N \in \{8, 16, 32, 64\}$.
- `intmax_t` Vorzeichenbehafteter Ganzzahltyp maximaler Größe.
- `uintmax_t` Vorzeichenloser Ganzzahltyp maximaler Größe.

Formatierte Ausgabe

printf() Diese Funktion versteht u. A. folgende Formatierungsbefehle:

- `%c` Platzhalter für einen `char` als Zeichen
- `%d` Platzhalter für einen `int` als Dezimalzahl
- `%u` Platzhalter für einen `unsigned int` als Dezimalzahl
- `%x` Platzhalter für einen `unsigned int` als Hexadezimalzahl
- `%f` Platzhalter für einen `double`
- `%s` Platzhalter für eine Zeichenkette
- `%%` Gibt ein Prozentzeichen aus
- `\n` Beendet eine neue Zeile / fängt eine neue an.

Kontrollanweisungen

Die `if`-Anweisung Wenn *Ausdruck* wahr ist, wird *Anweisung 1* ausgeführt. Gibt es einen `else`-Teil, wird sonst *Anweisung 2* ausgeführt. Syntax:

```
if (Ausdruck)
    Anweisung 1
[ else
    Anweisung 2 ]
```

Die `while`-Anweisung Solange *Ausdruck* wahr ist, führe *Anweisung* aus *Ausdruck* wird am Anfang der Schleife geprüft, d. h. ist *Ausdruck* vor Anfang der Schleife falsch, so wird die Schleife nie ausgeführt. Syntax:

```
while (Ausdruck)
    Anweisung
```

Die `do`-Anweisung Führe *Anweisung* aus, solange *Ausdruck* wahr ist. *Ausdruck* wird am Ende der Schleife geprüft, d. h. ist *Ausdruck* vor Anfang der Schleife falsch, wird die Schleife doch mindestens einmal ausgeführt. Syntax:

```
do
    Anweisung
(Ausdruck);
```

Die `for`-Anweisung Wird üblicherweise als Zählschleife gebraucht:

- `for (i = 0; i < n; i++) { ... }`
- `for (lap = 0; lap < n; lap++, next_lap()) { ... }`

Syntax:

```
for (Ausdruck 1; Ausdruck 2; Ausdruck 3)
    Anweisung
```

Ist äquivalent zu:

```
Ausdruck 1;
while (Ausdruck 2) {
    Anweisung
    Ausdruck 3;
}
```

Sprunganweisungen Ermöglichen Sprünge im Code, hierzu werden Sprungmarken gesetzt. Sie können nicht vor Deklarationen oder am Ende eines Blockes stehen. Syntax:

```
Marke: Anweisung
```

- `break` ; beendet die innerste Schleife oder `switch`-Anweisung.
- `continue` ; springt an das Ende des Schleifenrumpfes und beginnt die nächste Iteration.
- `return` [*Ausdruck*] ; beendet die Funktion und gibt ggf. *Wert* zurück, wie `ret` in Assembler, mit Setzen des Rückgaberegisters.
- `goto` *Marke* ; setzt die Ausführung bei *Marke* fort, wie `jmp` in Assembler.

Die `switch`-Anweisung Wertet *Ausdruck* aus und springt zur passenden `case`-Marke in *Anweisung* (die ein Block sein sollte). Gibt es keine, wird ggf. zur `default`-Marke gesprungen.

```
switch ( Ausdruck ) {
case Ausdruck 1 :
    Anweisung 1
    break;
case Ausdruck 2 :
    Anweisung 2 // fallthrough
case Ausdruck 2 :
    Anweisung 3
// ...
default:
    Anweisung n
}
```

Funktionsdefinitionen Funktionen können nur auf höchster Ebene definiert werden. Beim Funktionsaufruf werden alle Argumente in die Parameter der Funktion kopiert, hierbei werden Arrays werden als Pointer übergeben. Soll in ein Argument geschrieben werden, so muss ein Zeiger auf dieses übergeben werden werden. Alle Variablen einer Funktion werden beim Funktionsende zerstört! Syntax:

```
Rückgabetyp Funktionsname ( [ Argument , … ] )
{
  …
}
```

Soll die Funktion keine *Argumente nehmen, schreibt man den speziellen Parameter* `void`