1.2.F Programmstrukturen/Berechnungen und Ausgabe – Ergänzungen und Bilder

1.2.F.1 Darstellung von Umlauten und anderen Nicht-ASCII-Zeichen

Will man in C-Stringkonstanten Umlaute verwenden, muss man sie im Quelltext wie in Tabelle 1 codieren. Beispiel: `src/umlaute.c`

Zeichen	ISO-8859-1 (Windows)	UTF-8 (Linux)	IBM-PC (BIOS)
Ä	\304	\303\204	\216
Ö	\326	\303\226	\231
Ü	\334	\303\234	\232
ä	\344	\303\244	\204
ö	\366	\303\266	\224
ü	\374	\303\274	\201
ß	\337	\303\237	\341
Ω	—	\316\251	\352

Tabelle 1: Darstellung von Umlauten in C-Stringkonstanten

```c
#include <stdio.h>
int main(void)
{
    printf("Der Wert von Rv betr\303\244gt %i \316\251.\n", 470);
    return 0;
}
```

1.2.F.2 Bildschirm-Spielereien mit Terminal-Codes

Mit Hilfe der folgenden Escape-Sequenzen kann man die eigenen C-Programme etwas bunter gestalten als es der C-Standard eigentlich zulässt. Unter Linux läuft das folgende Beispiel (`src/terminalcodes.c`) ohne Zusatz; unter Windows braucht man ein Zusatzprogramm wie ANSICON.

```c
#include <stdio.h>
int main(void)
{
    printf("\033c");      /* loescht den Bildschirm */
    printf("\033[1m");    /* Fettschrift ein */
    printf("\033[31m");   /* rote Schrift */
    printf("\033[42m");   /* gruener Hintergrund */
    printf("Hallo!\n");
    printf("\033[0m");    /* Schrift wieder normal */
    return 0;
}
```

In Tabelle 2 findet man weitere Escape-Sequenzen (mit dem Wort *hier* ist jeweils die aktuelle Cursor-Position gemeint). Weitere Informationen kann man nachlesen unter:

- `https://www-user.tu-chemnitz.de/~heha/hs/terminal/terminal.htm`
- `http://wiki.bash-hackers.org/scripting/terminalcodes`

Aktion	Escape-Sequenz
Bildschirm löschen, Reset	`\033c`
Löschen der Zeile	`\033[2K`
Löschen der Zeile bis hier	`\033[1K`
Löschen der Zeile ab hier	`\033[K`
Löschen des Bildschirms	`\033[2J`
Löschen des Bildschirms bis hier	`\033[1J`
Löschen des Bildschirms ab hier	`\033[J`
Schrift invertiert ein	`\033[?5h`
Schrift invertiert aus	`\033[?5l`
Schrift normal	`\033[m`
Schrift fett ein	`\033[1m`
Schrift fett aus	`\033[21m`
Schrift unterstrichen ein	`\033[4m`
Schrift unterstrichen aus	`\033[24m`
Schrift invertiert ein	`\033[7m`
Schrift invertiert aus	`\033[27m`
Schrift unsichtbar ein	`\033[8m`
Schrift unsichtbar aus	`\033[28m`
Schrift schwarz	`\033[30m`
Schrift rot	`\033[31m`
Schrift grün	`\033[32m`
Schrift gelb	`\033[33m`
Schrift blau	`\033[34m`
Schrift violett	`\033[35m`
Schrift türkis	`\033[36m`
Schrift weiß	`\033[37m`
Hintergrund schwarz	`\033[40m`
Hintergrund rot	`\033[41m`
Hintergrund grün	`\033[42m`
Hintergrund gelb	`\033[43m`
Hintergrund blau	`\033[44m`
Hintergrund violett	`\033[45m`
Hintergrund türkis	`\033[46m`
Hintergrund weiß	`\033[47m`
Cursor ein	`\033[?25h`
Cursor aus	`\033[?25l`
Cursor-Position setzen (x=2, y=5)	`\033[5,2H`
Cursor auf linke obere Ecke	`\033[H`
Cursor 7 Zeilen nach oben	`\033[7A`
Cursor 7 Zeilen nach unten	`\033[7B`
Cursor 7 Zeichen nach rechts	`\033[7C`
Cursor 7 Zeichen nach links	`\033[7D`
Position und Schriftart merken	`\033[s`
Position und Schriftart zurückholen	`\033[u`
Bildschirm mit `E` füllen	`\033#8`
Piepser ausgeben	`\007`

Tabelle 2: Tabelle mit Escape-Sequenzen

1.2.F.3 Auswertung von Ausdrücken

Eine Berechnung wie `10+20*30` ist für den C-Compiler ein *Ausdruck*. Der obige Ausdruck enthält drei Zahlen und zwei Operatoren. Jeder Ausdruck hat ein Rechenergebnis, man nennt ihn den *Wert* des Ausdrucks. Falls der Ausdruck wie hier nur konstante Zahlen enthält, kann er schon vom Compiler berechnet werden. Enthält der Ausdruck auch so genannte Variablen (Zahlen, die im Register oder im RAM gespeichert sind, siehe später), wird sein Wert erst berechnet, wenn das Programm läuft.

Wie wird nun der Wert von `10+20*30` berechnet? Als Mensch kennt man die Regel „Punktrechnung vor Strichrechnung“ und berechnet zuerst das Zwischenergebnis $20 \cdot 30 = 600$. Anschließend setzt man das Zwischenergebnis ein und berechnet $10 + 600 = 610$. Hier benimmt sich die Programmiersprache absichtlich genauso: `20*30` ist ein Teilausdruck, der zuerst berechnet wird. Man sagt, der Operator `*` hat *Vorrang* vor dem Operator `+`. In C gibt es eine Vorrangtabelle der verschiedenen Operatoren. In dieser Tabelle liegen die Operatoren `*` (mal), `/` (geteilt durch) und `%` (modulo) eine Ebene oberhalb der Operatoren `+` (plus) und `-` (minus). Der Wert wird also berechnet, er ist `600`. In einem zweiten Schritt wird nun der Ausdruck `10+600` berechnet; sein Wert ist `610`. Eine graphische Darstellung dieses Ablaufs liefert der sogenannte Strukturbaum[1] (Abbildung 1).

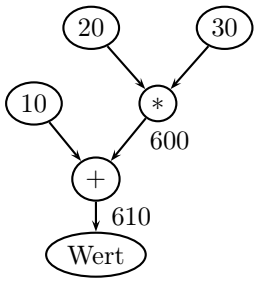

Abbildung 1: Strukturbaum zu `10+20*30`

Genaus kann man den Strukturbaum zu `1*2+3*4` ermitteln (Abbildung 2). Hier müssen zuerst

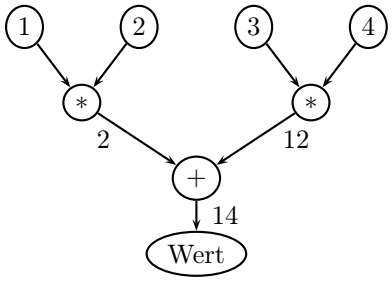

Abbildung 2: Strukturbaum zu `1*2+3*4`

`1*2` und `3*4` berechnet werden, bevor die Addition erfolgen kann[2].

Bei den Operatoren `-` (minus), `/` (geteilt durch) und `%` (modulo) ist noch eine Frage zu beachten: Falls mehrere Operatoren der gleichen Ebene nebeneinander stehen, welcher Operator hat Vorrang? Ist `3-2-1` gleich $3 - 2 - 1 = 0$ oder $3 - (2 - 1) = 1$? Hier verhält sich C wiederum absichtlich so, wie man es aus der mathematischen Schreibweise gewohnt ist: Der linke Operator hat Vorrang vor dem rechten Operator. Man sagt, diese Operatoren in C sind *links-assoziativ*. Abbildung 3 zeigt das am Beispiel von `3-2-1`.

[1] Oft wird der Strukturbaum auch mit der Wurzel nach oben dargestellt. In der Aussage ergibt sich dadurch kein Unterschied.

[2] Übrigens ist in C nicht festgelegt, welcher der Teilausdrücke zuerst berechnet wird (links oder rechts).

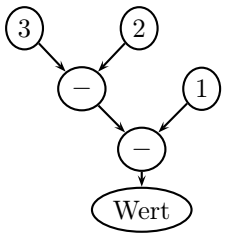

Abbildung 3: Strukturbaum zu `3-2-1`

Mit Hilfe runder Klammern kann man die Regeln zu Vorrang und Assoziativität überwinden. Wie in der gewohnten mathematischen Schreibweise gilt auch in C, dass das, was in Klammern steht, zuerst berechnet wird. Allerdings müssen in C alle Vorrangklammern rund sein; eckige, geschweifte oder spitze Klammern sind in C anderen Zwecken vorbehalten. Abbildung 4 zeigt den Strukturbaum des Ausdrucks `(5+6)*(7+8)`.

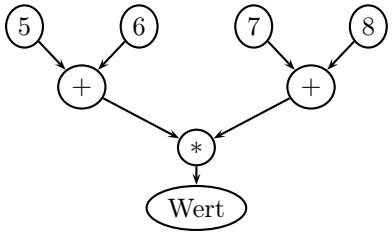

Abbildung 4: Strukturbaum zu `(5+6)*(7+8)`

Eine Besonderheit ist die Klammer, die regelmäßig hinter dem Wort `printf` steht. `printf` ist der Name eines Funktionsbaustein. Die Klammer hinter dem Namen bedeutet, dass der Funktionsbaustein (in C sagt man verkürzend: diese Funktion) an dieser Stelle benutzt wird (in C sagt man: er wird aufgerufen). Das heißt, dass die Anweisungen, die in dem Baustein aufgelistet sind, ausgeführt werden und danach an der gleichen Stelle (hinter dem Aufruf) weitergemacht wird.

Die Ausdrücke in der Klammer heißen *Parameter*. Beim Aufruf von `printf` geben sie an, was ausgegeben werden soll. Es gibt Funktionsbausteine ohne Parameter (dann bleibt die Klammer leer), mit einem und mit mehreren Parametern. Hat man mehrere Parameter, dann werden sie durch Kommata voneinander getrennt, damit der Compiler weiß, wo ein Parameter aufhört und der nächst anfängt. Alle Parameter (des Aufrufs) zusammen bilden die *Parameterliste*[3].

Auch zu einem Funktionsaufruf kann man einen Strukturbaum angeben. Abbildung 5 zeigt den Strukturbaum des Ausrufs `printf("%d", 7)`.

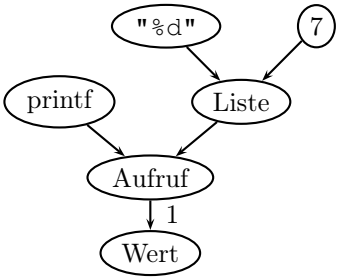

Abbildung 5: Strukturbaum zum Funktionsaufruf

[3]Der Funktionsbaustein `printf` ist dabei ein Sonderfall: Er kann sowohl mit einem als auch mit mehreren Parametern umgehen; seine Parameterliste hat also eine variable Länge.