2 Datentypen und Grundbefehle

2.1 Datentypen

Variablen sind benannte Speicherplätze, die man über den Variablennamen lesen bzw. verändern kann. Kennzeichnend für eine Variable sind ihr Name, ihr Datentyp, ihre Adresse und ihr Inhalt, sowie ihre Attribute.

Standard-Datentypen

leer		
`void`		kein Datentyp
ganze Zahlen		
`short`	`short int`	kleiner Bereich
`int`		mittlerer Bereich
`long`	`long int`	größer Bereich
`long long`	`long long int`	sehr großer Bereich (nicht ANSI)!
natürliche Zahlen		
`unsigned short`	`unsigned short int`	kleiner Bereich
`unsigned int`		mittlerer Bereich
`unsigned long`	`unsigned long int`	größer Bereich
`unsigned long long`	`unsigned long long int`	sehr großer Bereich (nicht ANSI)!
Gleitpunktzahlen		
`float`		kleiner Bereich
`double`		mittlerer Bereich
`long double`		größer Bereich (nicht ANSI)!
Zeichen		
`char`		intern ganze Zahl
`unsigned char`		intern natürliche Zahl

Für die Objekte eines Programms (Variablen- und Funktionsnamen) stehen einige voreingestellte Datentypen zur Verfügung mit denen weitere Datentypen erzeugt werden können. Abgeleiteten Datentypen sind Vektoren von

Objekten desselben Typs, Funktionen, die einen bestimmten Typ zurückgeben, Zeiger auf Objekte eines bestimmten Typs, sowie Strukturen, die aus unterschiedlichen Datentypen konstruiert werden.
Daneben werden in C selbst noch weitere Datentypen definiert, z. B. mit `size_t` ein Typ für die Anzahl der Bytes eines Datentyps.
(typ) var wandelt *var* in den Typ *typ* um (Casting).

2.2 Geltungsbereiche

Man unterscheidet externe und interne Objekte. Externe erscheinen in der Quelldatei außerhalb von Funktionsdefinitionen, interne erscheinen innerhalb einer Funktionsdefinition oder eines mit geschweiften Klammern begrenzten Blockes.
Der Geltungsbreich eines externen Objekts erstreckt sich von seiner Definition bis zum Ende der Quelldatei. Der Geltungsbreich eines internen Objekts ist der Bereich, in dem es erklärt wurde, und ersetzt dort etwa existierende externe Objekte gleichen Namens. Variablen und Funktionen, die nur in der Quelldatei sichtbar und zugänglich sein sollen, werden mit `static` vereinbart.

2.3 Befehle (1)

Zuweisungen:

***var* = *expression*;** Der Ausdruck *expression* wird ausgewertet, das Ergebnis unter *var* abgelegt. Die Typen von *var* und *expression* sollten übereinstimmen. x `=` f`(`y `=` z`);`. Die Variable y erhält den Wert z, die Variable x hat den Wert der Funktionsauswertung. Die Auswertung erfolgt von rechts nach links.

***var* $\odot$= *expression*;** Die Variable *var* wird mittels $\odot$ mit *expression* verknüpft und unter *var* abgelegt. Dabei kann $\odot$ die Standardoperatoren annehmen, soweit sie einen Sinn machen.

While-Schleife:

while (***condition*** **) {**
***statements*;**
} Es werden *statements* ausgeführt, solange die Bedingung in *condition* erfüllt ist.

Do-while-Schleife:

do {
***statements*;**
}
while (*condition*); Es werden *statements* ausgeführt, solange die Bedingung in *condition* erfüllt ist. Die Überprüfung erfolgt jetzt am Ende der Schleife.

2.4 Operatoren (1)

Vorzeichen Operatoren:

+, - unäre Operatoren, werden von rechts nach links ausgewertet.

Multiplikation, Division:

⋆, / binäre Operatoren, werden von links nach rechts ausgewertet.

Modulus Operator:

% binärer Operator für ganzzahlige Operanden liefert den Rest nach Division, wird von links nach rechts ausgewertet.

Addition, Subtraktion:

+, - binäre Operatoren, werden von links nach rechts ausgewertet.

Relationen:

$<, >, <=, >=, ==, !=$: binäre Tests, liefern den Wert 1, wenn sie wahr sind. Die Priorität von `==` und `!=` ist geringer als die der restlichen Operatoren. Auswertung erfolgt von links nach rechts.

Quer- und Wechselsumme

$$n = \sum_{\nu=0}^{m} a_\nu g^\nu \in \mathbb{N},\ g > 2,$$
$$a_m \in \{1, 2, \ldots, g-1\},\ a_\nu \in \{0, 1, \ldots, g-1\},\ \nu < m,$$
$$\mathcal{Q}(n) = \sum_{\nu=0}^{m} a_\nu,\ \text{Quersumme},\ \mathcal{W}(n) = |\sum_{\nu=0}^{m} (-1)^\nu a_\nu|,\ \text{Wechselsumme},$$
$$\mathcal{Q}(1279) = 19,\ \mathcal{Q}(4711) = 13,\ \mathcal{W}(1016) = 4,\ \mathcal{W}(4711) = 3.$$

Listing 6: Quersumme (quersumme1.c)

```
#include <stdio.h>
static long unsigned int quersum(long unsigned int);

int main(void) {
        long unsigned int n, s;
        printf("Eingabe (natuerliche Zahl): ");
        (void) scanf("%lu", &n);
        s = quersum(n);
        printf("Q(%lu) = %lu\n", n, s);
        return 0; }

long unsigned int quersum(long unsigned int zahl) {
        long unsigned int a, b, summe;
        a = b = summe = 0;
        do { a = zahl / 10;
             b = zahl % 10;
             zahl = a;
             summe = summe + b;
        } while (a != 0);
        return summe; }
```

Listing 7: Quersumme (quersumme2.c)

```
#include <stdio.h>
static long unsigned quers(long unsigned);

int main(void) {
        long unsigned n;
        printf("Eingabe (natuerliche Zahl): ");
        (void) scanf("%lu", &n);
        printf("Q(%lu) = %lu\n", n, quers(n));
        return 0; }

long unsigned quers(long unsigned zahl) {
        long unsigned summe = 0lu, decem = 10lu;
        while (zahl != 0) {
             summe += zahl % decem;
             zahl /= decem; }
        return summe; }
```

2.5 Operatoren (2)

logische Operatoren:

&&, || der Operator && ist stärker, die Auswertung erfolgt von links nach rechts und endet, wenn der Wahrheitswert feststeht.

In/Dekrement Operatoren:

var++; ***var***−−; Es wird der Wert von *var* um 1 erhöht, bzw. erniedrigt, falls diese Anweisung allein erscheint. In Ausdrücken wird *var* benutzt, anschließend wird *var* um ±1 verändert.

In/Dekrement Operatoren:

++***var***; −−***var***; Es wird der Wert von *var* um ±1 verändert, in Ausdrücken wird mit den veränderten Werten gearbeitet.

Vorsicht ist geboten, wenn die Variable, auf die ein Inkrement oder Dekrement angewendet wird, mehrfach im Ausdruck auftaucht, z. B.

```
x = x++ + --x;
```

Der Wert ist compilerabhängig. `splint` weist auf solche Fehler hin.

2.6 Befehle (2)

Die folgenden Anweisungen illustrieren wir an – wie wir sehen werden – immer besser durchdachten Programmen zu einem ungelösten Problem. Für $n \in \mathbb{N}$ läßt sich die Folge

$$f_0 = n,\ f_{i+1} = \begin{cases} f_i/2, & f_i \text{ gerade} \\ 3f_i + 1, & \text{sonst} \end{cases},\ i = 0, 1, \ldots$$

definieren. Es wird vermutet, daß diese Folge nach endlich vielen Schritten nur noch den Zyklus $1, 4, 2$ wiederholt.

For-Schleife:

for (***start***; ***condition***; ***do*** **) {**
statements;
} Es werden *statements* für eine Laufvariable ausgeführt. Die Laufvariable wird in *start* initialisiert, sie wird gemäß des *do* solange verändert, wie *condition* erfüllt ist.

Verlassen einer Schleife:

break; Die innerste umgebende Schleife wird verlassen.

Erhöhen der Laufvariablen:

continue; Es wird zum Ende der Schleife gesprungen, die Laufvariable wird ggf. erhöht.

If-Abfrage:

if (*condition*) {
**   *statements*;**
}
Wenn *condition* wahr ist, werden die *statements* ausgeführt.

If-else-Abfrage:

if (*condition*) {
**   *statements1*;**
} else {
**   *statements2*;**
} Wenn *condition* wahr ist, werden die *statements1* ausgeführt, andernfalls *statements2*.

If-else if-Abfrage:

if (*condition1*) {
**   *statements1*;**
} else if (*condition2*) {
**   *statements2*;**
} else if (*condition3*) {
**   *statements3*;**
} else { ***statements*;**
} Wenn *conditioni* wahr ist, werden die *statementsi* ausgeführt, wenn keine Bedingung erfüllt wird, *statements*. `else` ist optional, die Zahl der Bedingungen kann variiert werden.

Listing 8: 3 n + 1 Problem (3n+1a.c)

```c
#include <stdio.h>
/* 3 n + 1 Problem, erster Versuch */
/* Input: 134217727 ?? */

int main(void)
{
        int i, n;
        printf("Eingabe einer natuerlichen Zahl: ");
        (void) scanf("%d", &n);
        for (i = 1; i <= 200; i++) {
                if (n == 1)
                        break;
                else if (n % 2 == 0) {
                        n = n / 2;
                } else {
                        n = 3 * n + 1;
                }
                printf("%d\n", n);
        }
        if ((i == 201) && (n != 1))
                printf("%d Schritte zu wenig!\n", i-1);
        return 0;
}
```

Switch:

switch (*var*) {

case *const1* :

***statements1*;**

case ***const2*** **:**

***statements2*;**

...

default:

***statements*;**

} Wenn *var* = *consti* werden *statementsi* ausgeführt, die wird man i.a. mit `break` beenden. Wird keine der Bedingungen angenommen können default-statements angegeben werden.

Listing 9: 3 n + 1 Problem (3n+1b.c)

```
#include <stdio.h>
/* 3 n + 1 Problem, zweiter Versuch */
/* input: 134217727 ?? */

int main(void)
{
        int n;
        printf("Eingabe einer natuerlichen Zahl: ");
        (void) scanf("%d", &n);
        while (n != 1) {
                switch (n % 2) {
                case 0:
                        n = n / 2;
                        break;
                case 1:
                        n = 3 * n + 1;
                        break;
                }
                printf("%d\n", n);
        }
        return 0;
}
```

Beachten Sie bitte die Meldungen des splint!

Goto Sprung:

goto *marke* es wird zu der angegebenen Marke *marke* gesprungen. Ein `goto` läßt sich immer in anderer Form schreiben. Beim Sprung aus verschachtelten Schleifen kann es aber nützlich sein.

Marken:

***marke*:** es wird eine Marke *marke* gesetzt, zu der gesprungen werden kann.

Unsere beiden Programme haben noch schwerwiegende Fehler. Testen Sie die Zahl 134217727. Das zweite Programm gerät in eine Endlos-Schleife, das erste liefert wegen Überlaufs nicht die gewünschten Ergebnisse.