Einführung in die Programmierung

Bachelor of Science

Prof. Dr. Rethmann

Fachbereich Elektrotechnik und Informatik
Hochschule Niederrhein

WS 2009/10

Übersicht

Motivation

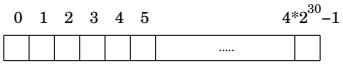
```
#include <stdio.h>
void main(void) {
    int zahl, inp = 130;
    int s0, s1, s2, s3, ..., s31;

    zahl = inp;

    s0 = zahl % 2;
    zahl /= 2;
    s1 = zahl % 2;
    zahl /= 2;
    ...
    s31 = zahl % 2;
    zahl /= 2;

    printf("dezimal %d = binaer %d%d%d...%d\n",
        inp, s31, s30, s29, ..., s0);
}
```


Motivation

```
#include <stdio.h>
void main(void) {
    int inp, zahl, i, s[32];

    printf("Zahl? ");
    scanf("%d", &inp);

    zahl = inp;
    for (i = 0; i < 32; i++) {
        s[i] = zahl % 2;
        zahl /= 2;
    }

    printf("dezimal %d = binaer ", inp);
    for (i = 31; i >= 0; i--)
        printf("%d", s[i]);
    printf("\n");
}
```


Arrays

Vereinfachtes Bild der Speicherorganisation:



Speicherzellen

- sind fortlaufend nummeriert und adressierbar
- können einzeln oder in zusammenhängenden Gruppen bearbeitet werden

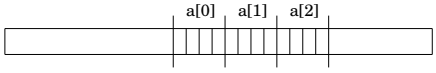
Arrays

Ein Array belegt einen zusammenhängenden Speicherbereich.

Syntax: `Typ Name[Anzahl_Elemente];`

Beispiel: `int a[3];`

Semantik: es wird Platz für drei Werte vom Typ `int` im Speicher reserviert



Bei einem 32-Bit-System werden typischerweise je `int` vier Byte reserviert, im obigen Beispiel als $3 \cdot 4 = 12$ Byte.

Arrays

- ein Array, das aus Elementen unterschiedlicher Datentypen besteht, gibt es in C nicht
- die Größe eines Arrays muss zum Zeitpunkt der Übersetzung bekannt sein (Ausnahme: C99-Standard)
- in C gibt es keine wachsenden Arrays (müsste über Zeiger realisiert werden)
- man greift mittels des Index-Operators `[ ]` auf die einzelnen Elemente des Arrays zu
- das erste Element des Arrays steht auf Position 0, das letzte auf Position `Anzahl_Elemente - 1`
- die Initialisierung eines Arrays kann direkt bei dessen Deklaration erfolgen

Arrays

```
#include <stdio.h>

void main(void) {
    int i, summe = 0;
    int arr[5] = {1, 2, 3, 4, 5};

    for (i = 0; i < 5; i++)
        summe += arr[i];
    printf("Summe = %d\n", summe);
}
```

Anmerkungen:

- die Größenangabe kann entfallen, falls das Array bei der Deklaration direkt initialisiert wird
- in der deutschsprachigen Literatur werden Arrays oft auch als Felder oder Vektoren bezeichnet

Mehrdimensionale Arrays

Anhängen weiterer Indizes.

Beispiel: zweidimensionales Feld `matrix` vom Typ `int`

```
int matrix[10][20];
matrix[0][0] = 42;
matrix[1][0] = matrix[0][0] + matrix[0][1];
```

Mehrdimensionale Arrays und ihre Initialisierung:

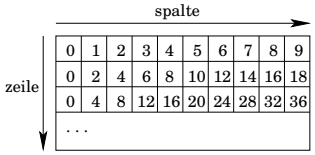
```
float x[2][4] = { {1.0, 2.0, 3.0, 4.0},
                  {5.0, 6.0, 7.0, 8.0} };
int y[][4] = {
    {1, 2, 3, 4},
    {5, 6, 7, 8}
};
char z[][7] = { "Hello,", "world!" };
```


Mehrdimensionale Arrays

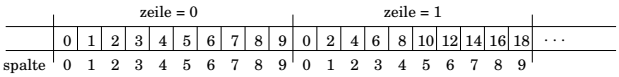
```
#include <stdio.h>

void main(void) {
    int matrix[5][10];
    int z, s;    /* Zeile, Spalte */

    for (s = 0; s < 10; s++)
        matrix[0][s] = s;
    for (z = 1; z < 5; z++)
        for (s = 0; s < 10; s++)
            matrix[z][s] = matrix[z-1][s] * 2;

    for (z = 0; z < 5; z++) {
        for (s = 0; s < 10; s++)
            printf("%4d", matrix[z][s]);
        printf("\n");
    }
}
```


Mehrdimensionale Arrays

wir können uns obige Matrix als zweidimensionale Tabelle vorstellen:



im Speicher werden die Werte allerdings zeilenweise hintereinander abgelegt, also eindimensional:

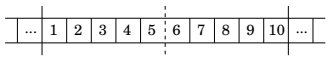


Mehrdimensionale Arrays

Felder werden zeilenweise abgespeichert. Der letzte Index ändert sich schneller als der erste: `int a[#zeilen][#spalten]`

Beispiel:

```
int a[][5] = {
   {1, 2, 3, 4, 5},
   {6, 7, 8, 9, 10}
};
```



Initialisierung: Nur die erste, also die äußerste Größenangabe darf weggelassen werden. Der Rest nicht, da sonst der Index-Operator `[]` die Position nicht berechnen kann.

Beispiel: Position von `a[i][j]`: `i * #spalten + j`

Mehrdimensionale Arrays

```
#include <stdio.h>

void main(void) {
    int z, s;      /* Zeile, Spalte */
    int matr[4][5] = { {1, 2, 3, 4, 5},
                       {2, 3, 4, 5, 6},
                       {3, 4, 5, 6, 7} };

    for (s = 0; s < 5; s++) {
        int summe = 0;

        for (z = 0; z < 3; z++)
            summe += matr[z][s];
        matr[z][s] = summe;
    }

    /* Ausgabe ..... */
}
```


Übersicht

- Arrays
- *Datentypen, Operatoren und Kontrollstrukturen*
- Funktionen
- Zeiger
- Zeichenketten
- Dateioperationen
- Standardbibliotheken
- strukturierte Programmierung
- modulare Programmierung

Bezeichner und Namen

- Bezeichner dienen der eindeutigen Identifizierung von Objekten innerhalb eines Programms.
- Funktionsnamen, Variablennamen, Sprungmarken usw. sind Folgen von Zeichen, die aus Buchstaben, Ziffern und dem Unterstrich bestehen.
- Ein Bezeichner beginnt immer mit einem Buchstaben oder einem Unterstrich.
- Vermeiden Sie Namen, die mit zwei Unterstrichen beginnen! Solche Namen sind evtl. systemintern benutzt.
- Bezeichner dürfen nicht mit Schlüsselwörtern wie `int`, `while` oder `if` übereinstimmen.
- Groß-/Kleinschreibung wird unterschieden.

Bezeichner und Namen

- Der Name soll den Zweck der Variablen bzw. der Funktion andeuten!
- Er sollte so gewählt sein, dass eine Verwechselung durch Tippfehler unwahrscheinlich ist.
- Wenn eine Variable kommentiert werden muss, dann ist der Name der Variablen schlecht gewählt!

gute Namen:

- `summe(int x, int y)`
- `strlength(char *str)`
- `getHostByName(char *name)`

Datentypen

Damit der Compiler Unverträglichkeiten bei Zuweisungen und Operationen erkennen kann, ist C eine *typisierte* Programmiersprache:

- Alle in einem C-Programm verwendeten Größen wie z.B. Konstanten, Variablen und Funktionen haben einen Typ.
- Die Typen von Variablen und Funktionen werden in deren Deklaration bzw. Definition festgelegt.
- Die Größen der primitiven Datentypen sind nicht festgelegt, sie sind abhängig von der C-Implementierung und von der Wortlänge des Rechners.

Character-Typ

Der Datentyp `char` hat die Größe ein Byte und kann ein Zeichen aus dem Zeichensatz der Maschine speichern.

Der Wert einer Zeichenkonstanten ist der numerische Wert des Zeichens im Zeichensatz der Maschine (bspw. ASCII).

Schreibweise:

- ein Zeichen innerhalb von einfachen Anführungszeichen
- Ersatzdarstellungen für Steuerzeichen (`\n`, `\t`, ...)
- oktale Darstellung `'\ooo'`: ein bis drei oktale Ziffern
- hexadezimal `'\xnn'`: ein oder zwei Hex-Ziffern

Character-Typ

```
/* 'X' in ASCII  */
char c1 = 'X';
char c2 = '\130';
char c3 = '\x58';

/* Sonderzeichen */
char c4 = '\"';
char c5 = '\\';
char c6 = '\'';

/* Steuerzeichen */
char c7 = '\n';
char c8 = '\t';
char c9 = '\r';
```


Character-Typ

Zeichen sind ganzzahlig, arithmetische Operationen sind definiert.

Beispiel:

$$\texttt{'a' + 'b' = 'Ã'} \iff 97 + 98 = 195$$

Eine Zeichenkette ist ein Array von Zeichen:

- wird als Folge von beliebig vielen Zeichen zwischen doppelten Anführungszeichen geschrieben:

```
"Eine konstante Zeichenkette"
"Eine " "konstante" " Zeichenkette"
```

- intern hat jede Zeichenkette am Ende ein Null-Zeichen `'\0'`, wodurch die Länge prinzipiell nicht begrenzt ist

Integer-Typ

Schreibweise:

- Buchstabe `l` oder `L` am Ende bedeutet `long`
- Buchstabe `u` oder `U` am Ende bedeutet `unsigned`
- Ziffer 0 am Anfang bedeutet oktal
- Zeichen 0x oder 0X am Anfang bedeuten hexadezimal

Beispiele:

- `123456789UL`
- `022` bedeutet $(22)_8 = (18)_{10}$
- `0x1F` bedeutet $(1F)_{16} = (31)_{10}$

Fließkomma-Typ

Schreibweise:

- Suffix `f` oder `F` vereinbart `float`
- Suffix `l` oder `L` vereinbart `long double`
- Dezimalpunkt und/oder Exponent vorhanden

Beispiele:

- 1e-3 $= 1 \cdot 10^{-3} = 0{,}001$
- 234E4 $= 234 \cdot 10^4 = 2\ 340\ 000$
- -12.34e-67F
- 234.456
- .123e-42

Vereinbarungen

- Alle Variablen müssen vor ihrem Gebrauch deklariert werden. Dazu wird der Datentyp angegeben, gefolgt von einer Liste von Variablen dieses Typs:

```
int lower, upper, step;
char c, line[256];
```

- Variablen können beliebig auf mehrere Vereinbarungen verteilt werden, dann kann jede einzelne Vereinbarung kommentiert werden.

```
int lower;          /* untere Grenze    */
int upper;          /* obere Grenze     */
int step;           /* Schrittweite     */
char c;             /* Laufvariable     */
char line[256];     /* Zeile aus Datei  */
```


Vereinbarungen

- Wenn eine Variable kommentiert werden muss, dann ist der Name der Variablen schlecht gewählt!
- Eine Variable kann bei ihrer Vereinbarung auch initialisiert werden.

  Beispiel:

```
double epsilon = 1.0e-5;
```

- Mit dem Attribut `const` kann bei der Vereinbarung einer Variablen angegeben werden, dass sich ihr Wert nicht ändert.

  Beispiel:

```
const double e = 2.71828182845904523536;
```

Arithmetische Operatoren

Operator	Beispiel	Bedeutung
+	+i	positives Vorzeichen
-	-i	negatives Vorzeichen
+	i + 5	Addition
-	i - j	Subtraktion
*	i * 8	Multiplikation
/	i / j	Division
%	i % 6	Modulo
=	i = 5 + j	Zuweisung
+=	i += 5	i = i + 5
-=	i -= j	i = i - j
*=	i *= 6	i = i * 6
/=	i /= j	i = i / j

Arithmetische Operatoren

- Vorrang der Operatoren in abnehmender Reihenfolge:
  - unäre Operatoren + und - (Vorzeichen)
  - *, / und %
  - binäre Operatoren + und -

Beispiel:

```
5 * -7 - 3   ⟺   (5 * (-7)) - 3
```

- Arithmetische Operationen werden von links her zusammengefasst.

Beispiel:

```
1 + 3 + 5 + 7   ⟺   ((1 + 3) + 5) + 7
```

zur Erinnerung: die Addition auf Gleitkommazahlen ist nicht assoziativ aufgrund von Rundungsfehlern bei der Denormalisierung

Inkrement- und Dekrementoperatoren

Ausdruck	Bedeutung
++i	erhöhe i um 1, *bevor* i im Ausdruck weiterverwendet wird (Präfix-Notation)
i++	erhöhe i um 1, *nachdem* i im Ausdruck weiterverwendet wird (Postfix-Notation)
--i	vermindere i um 1, *bevor* i im Ausdruck weiterverwendet wird (Präfix-Notation)
i--	vermindere i um 1, *nachdem* i im Ausdruck weiterverwendet wird (Postfix-Notation)

Inkrement- und Dekrementoperatoren

Ausdrücke werden unter Umständen schwer einsichtig:

```
int x, y;    /* Variablendeklaration */

x = 1;
y = ++x + 1;
             /* hier: x = 2, y = 3 */
x = 1;
y = x++ + 1;
             /* hier: x = 2, y = 2 */
x = 1;
x = ++x + 1;
             /* hier: x = 3 */
```

Diese Operatoren können nur auf Variablen angewendet werden, nicht auf Ausdrücke. **Verboten:** `(i+j)++`

Vergleichsoperatoren

Op	Beispiel	Bedeutung
<	i < 7	kleiner als
<=	i <= 7	kleiner gleich
==	i == 7	gleich

Op	Beispiel	Bedeutung
>	i > j	größer als
>=	i >= j	größer gleich
!=	i != j	ungleich

Prioritäten:

- Vergleichsoperatoren <, >, <=, >= haben gleiche Priorität
- Äquivalenzoperatoren ==, != haben geringere Priorität
- arithmetische Op. haben höhere Priorität als Vergleiche

Beispiele:

- `i < t - 1` wird bewertet wie `i < (t - 1)`
- `2+2 < 3 != 5 > 7` entspricht `(4 < 3) != (5 > 7)`

Boolesche Werte in C

in C wird

- *false* durch den Wert 0 und
- *true* durch einen Wert ungleich 0 dargestellt
- bizarre Ausdrücke möglich: 3 < 2 < 1 ist true, denn (3 < 2) = false = 0 und 0 < 1

Oft findet man in C-Programmen verkürzte Anweisungen.

Beispiel:

- `while (x)` ⟺ `while (x != 0)`
- `while (strlen(s))` ⟺ `while (strlen(s) > 0)`

Logische Verknüpfungen

Operator	Beispiel	Ergebnis (Bedeutung)
&&	a && b	liefert 1, falls a und b wahr sind, sonst 0 (logisches UND)
||	a || b	liefert 1, falls a oder b wahr sind, sonst 0 (logisches ODER)
!	!a	liefert 1, falls a falsch ist, sonst 0

Prioritäten:

- && hat Vorrang vor ||
- Vergleichs- und Äquivalenzoperatoren: höherer Vorrang

Beispiele:

- `i < t-1 && c != EOF` → keine Klammern notwendig
- `!valid` ⟺ `valid == 0`

Verkürzte Auswertung

Ausdrücke werden nur solange bewertet, bis das Ergebnis feststeht!

Es gilt:

- `X && Y` ist gleich 0, falls `X == 0`
- `X || Y` ist gleich 1, falls `X == 1`

Verkürzte Auswertung ist notwendig, um Laufzeitfehler zu vermeiden.

Beispiele:

- `(x != 1) && (1/(x-1) > 0)`
- `scanf("%d", &n) > 0 || exit(errno)`

Operatoren zur Bitmanipulation

Ganze Zahlen können als Bitvektoren aufgefasst werden:

$$
\begin{array}{rcl}
 & \vdots & \\
-3 & = & 1111\ 1111\ 1111\ 1101 \\
-2 & = & 1111\ 1111\ 1111\ 1110 \\
-1 & = & 1111\ 1111\ 1111\ 1111 \\
0 & = & 0000\ 0000\ 0000\ 0000 \\
1 & = & 0000\ 0000\ 0000\ 0001 \\
2 & = & 0000\ 0000\ 0000\ 0010 \\
3 & = & 0000\ 0000\ 0000\ 0011 \\
 & \vdots &
\end{array}
$$

in C: Manipulation einzelner Bits möglich

Operatoren zur Bitmanipulation

nur auf Integer-Typen anwendbar:

Op	Beispiel	Bedeutung
<<	`i << j`	Links-Shift von `i` um `j` Stellen
>>	`i >> j`	Rechts-Shift von `i` um `j` Stellen
&	`i & j`	bitweises UND von `i` und `j`
|	`i | j`	bitweises ODER von `i` und `j`
^	`i ^ j`	bitweises Exklusiv-ODER von `i` und `j`
~	`~i`	Einerkomplement von `i`

Prioritäten:

- Shift-Operatoren vor Äquivalenzoperatoren == und !=
- Äquivalenzoperatoren Vorrang vor &, | und ^
- *Beispiel:* `(x & MASK) == 0` statt `x & MASK == 0`

Operatoren zur Bitmanipulation

Beispiele:

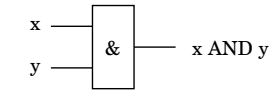
```
short i = 1;      /* i= 0000 0000 0000 0001 */
i = i << 3;       /* i= 0000 0000 0000 1000 */
i = i >> 2;       /* i= 0000 0000 0000 0010 */
i = i | 5;        /* i= 0000 0000 0000 0111 */
i = i & 3;        /* i= 0000 0000 0000 0011 */
i = i ^ 5;        /* i= 0000 0000 0000 0110 */
i = ~i;           /* i= 1111 1111 1111 1001 */
```

- der Ausdruck `1 << n` entspricht also dem Wert 2^n
- mittels des Ausdrucks `wert & (1 << n)` kann genau das `n`-te Bit von `wert` analysiert werden

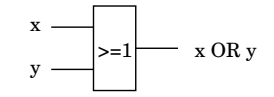
Operatoren zur Bitmanipulation

Beispiel: $x = 39 = (100111)_2$ und $y = 45 = (101101)_2$

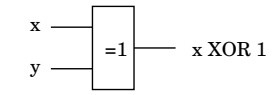
```
    x   100111
    y = 101101
x & y = 100101
```




```
    x   100111
    y = 101101
x | y = 101111
```




```
    x   100111
    y = 101101
x ^ y = 001010
```



Links-Shift

Mathematische Deutung für positive, ganze Zahlen:

$$z = (x_n x_{n-1} \ldots x_0)_2 \Rightarrow z = x_n \cdot 2^n + x_{n-1} \cdot 2^{n-1} + \cdots + x_0 \cdot 2^0$$

Links-Shift oder Multiplikation mit 2:

$$
\begin{array}{rcl}
z \cdot 2 & = & x_n \cdot 2^{n+1} + x_{n-1} \cdot 2^n + \cdots + x_0 \cdot 2^1 + 0 \cdot 2^0 \\
 & = & (x_n\ x_{n-1}\ x_{n-2}\ \ldots\ x_0\ 0)_2
\end{array}
$$

allgemein: Links-Shift um m Stellen bedeutet eine Multiplikation mit 2^m

Rechts-Shift

Mathematische Deutung für positive, ganze Zahlen:

$$z = (x_n x_{n-1} \ldots x_0)_2 \Rightarrow z = x_n \cdot 2^n + x_{n-1} \cdot 2^{n-1} + \cdots + x_0 \cdot 2^0$$

Rechts-Shift oder ganzzahlige Division durch 2:

$$
\begin{array}{rcl}
z/2 & = & x_n \cdot 2^{n-1} + x_{n-1} \cdot 2^{n-2} + \cdots + x_1 \cdot 2^0 \\
 & = & (x_n\ x_{n-1}\ x_{n-2}\ \ldots\ x_1)_2
\end{array}
$$

allgemein: Rechts-Shift um m Stellen bedeutet eine ganzzahlige Division durch 2^m

- Wert vom Typ `unsigned` → es wird Null nachgeschoben
- vorzeichenbehafteter Wert
  - *arithmetic shift:* Vorzeichenbit wird nachgeschoben
  - *logical shift:* Null-Bits werden nachgeschoben

Operatoren zur Bitmanipulation

```
#include <stdio.h>

void main(void) {
    int i, zahl;

    printf("Wert? ");
    scanf("%d", &zahl);

    for (i = 31; i >= 0; i--) {
        if (zahl & (1 << i))
            printf("1");
        else printf("0");
    }
    printf("\n");
}
```


Sonstige Operatoren in C

Es gibt keine

- Operationen, mit denen zusammengesetzte Objekte wie Zeichenketten, Listen oder Vektoren bearbeitet werden können
- Operationen für Ein- und Ausgabe
- eingebauten Techniken für Dateizugriff

⇒ abstrakte Mechanismen müssen als explizit aufgerufene Funktionen zur Verfügung gestellt werden

C-Implementierungen enthalten eine relativ standardisierte Sammlung solcher Funktionen (ANSI-Standard).

Kontrollstrukturen

dienen der Steuerung des Programmablaufs

In C: Kontrollstrukturen für *wohlstrukturierte* Programme

- Zusammenfassen von Anweisungen → `{ }`
- Entscheidungen → `if/else`
- Auswahl aus einer Menge möglicher Fälle → `switch`
- Schleifen mit Test des Abbruchkriteriums
  - am Anfang → `while`, `for`
  - am Ende → `do`
- das vorzeitige Verlassen einer Schleife mittels `break` oder `continue` ist nicht wohlstrukturiert!

Zusammenfassen von Anweisungen

werden Anweisungsfolgen mit geschweiften Klammern `{ }` zusammengefasst, dann gilt der geklammerte Block als eine Anweisung

```
{
    x = 0;
    i--;
    printf(...);
}
```

Struktogramm:

Anweisung 1
Anweisung 2
Anweisung 3

Dies wird im wesentlichen bei Schleifen `for, while, do` und Auswahlanweisungen `if, else` genutzt.

abweisende/kopfgesteuerte Schleife

Syntax:

```
while (ausdruck)
    anweisung
```

Semantik:

- `anweisung` wird ausgeführt, solange `ausdruck` wahr ist.
- `ausdruck` wird vor jedem Schleifendurchlauf bewertet.
- ist der Kontrollausdruck vorm ersten Durchlaufen der Schleife nicht erfüllt, dann wird die `anweisung` gar nicht ausgeführt

abweisende/kopfgesteuerte Schleife

Beispiel:

```
int i = 0;
int sum = 0;

while (i < 10) {
    sum += i;
    i++;
}
```

Struktogramm:

while Bedingung	
	Anweisung 1
	Anweisung 2
	Anweisung 3

fußgesteuerte Schleife

Syntax:

```
do
    anweisung
while (ausdruck);
```

Semantik:

- Die Anweisung in der Schleife wird ausgeführt, bis der Kontrollausdruck `ausdruck` nicht mehr erfüllt ist.
- Die Schleifenanweisung `anweisung` wird mindestens einmal ausgeführt!

fußgesteuerte Schleife

Beispiel:

```
int i = 0;
int sum = 0;

do {
    sum += i;
    i++;
} while (i < 10);
```

Struktogramm:

do	Anweisung 1
	Anweisung 2
	Anweisung 3
while Bedingung	

do/while vs. repeat/until

In C gibt es keine `repeat/until`-Schleifen. Diese können aber durch `do/while`-Schleifen nachgebildet werden:

```
do                          entspricht      repeat
    .....                                       .....
while B                                     until !B
```

Beispiel:

```
do {                                        repeat {
    sum += i;                   entspricht      sum += i;
    i++;                                        i++;
} while (i < 10);                           } until (i >= 10);
```


Zähl-Schleife

`for`-Schleife: bietet Möglichkeit, einfache Initialisierungen und Zählvorgänge übersichtlich zu formulieren

Syntax:

```
for (ausdruck1; ausdruck2; ausdruck3)
    anweisung
```

Semantik:

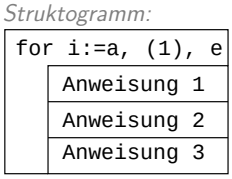
- `ausdruck1` ist ein Initialisierungsausdruck, der vor Beginn der Schleife einmal ausgeführt wird
- solange `ausdruck2` erfüllt ist, wird der Schleifenrumpf durchlaufen
- `ausdruck3` wird nach jedem Schleifendurchlauf bewertet und wird dazu verwendet, die Schleifenvariablen zu ändern

Zähl-Schleife

Beispiel:

```
int i, sum = 0;

for (i = 0; i < 10; i++) {
    sum += i;
}
```

Struktogramm:



Anmerkung: Bei keinem Schleifentyp sind die geschweiften Klammern { und } Bestandteil der Syntax! Gleichbedeutend zu oben:

```
int i, sum = 0;
for (i = 0; i < 10; i++)
    sum += i;
```

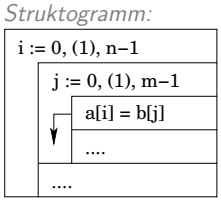

break

Mittels der `break`-Anweisung kann die innerste Schleife vorzeitig und unmittelbar verlassen werden.

Beispiel:

```
for (i = 0; i < n; i++) {
    for (j = 0; j < m; j++) {
        if (a[i] == b[j])
            break;
        .....
    }
    .....
}
```

Struktogramm:



keine strukturierte Programmierung

continue

`continue`-Anweisung: die nächste Wiederholung der umgebenden Schleife wird unmittelbar begonnen.

Beispiel:

```
for (i = 0; i < n; i++) {
    if (a[i] < 0)
        continue;
    .....
}
```

- keine strukturierte Programmierung
- in Struktogrammen nicht darstellbar

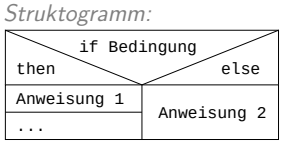
Auswahlanweisungen

Mittels Auswahlanweisungen kann der Ablauf eines Programms abhängig von Bedingungen geändert werden.

Syntax:

```
if (ausdruck)
    anweisung1
else
    anweisung2
```

Struktogramm:



Semantik:

- `ausdruck` wird bewertet: Falls er wahr ist, wird `anweisung1` ausgeführt, sonst `anweisung2`.
- Der `else`-Zweig kann entfallen → `anweisung1` wird übersprungen, falls der Ausdruck `ausdruck` nicht erfüllt ist.

Auswahlanweisungen

In C ist der Wert einer Zuweisung der Wert, der an die linke Seite zugewiesen wird.

- `if (i = 1)` ist immer erfüllt → Vorsicht!
- `if (i == 1)` die wahrscheinlich gewünschte Anweisung
- `if (1 == i)` verhindert solche Fehler → besser!

Eine Zuweisung kann als Teil eines Ausdrucks verwendet werden!

Beispiele:

- `while ((c = getchar()) != EOF)`
- `if ((len = strlen(s)) > 0)`

⇒ Probleme beim Debuggen!

Auswahlanweisungen

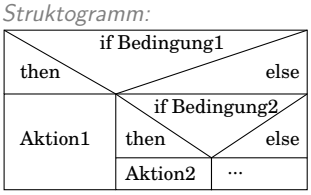
```
#include <stdio.h>   /* Wechselgeld berechnen */
int main(void) {
    int z;

    printf("Eingabe: z = ");
    scanf("%d", &z);

    if (z >= 500) {
        printf("%d mal 500\n", z / 500);
        z %= 500;
    }
    if (z >= 200) {
        printf("%d mal 200\n", z / 200);
        z %= 200;
    }
    ...
    return 0;
}
```


Auswahlanweisungen

Syntax:

```
if (ausdr_1)
    aktion_1
...
else if (ausdr_n)
    aktion_n
else aktion
```

Struktogramm:



Semantik:

- `ausdr_1`, `ausdr_2`, ... werden der Reihe nach bewertet
- `ausdr_i` erfüllt → `aktion_i` ausführen und die Abarbeitung der Kette abbrechen
- ist kein Ausdruck wahr → die Anweisung im `else`-Teil ausführen. Der `else`-Teil ist optional.

Auswahlanweisungen

Beispiel: Schaltjahr-Bestimmung mittels Auswahlanweisungen

```
if (jahr % 4 != 0)
    tage = 365;
else if (jahr % 100 != 0)
    tage = 366;
else if (jahr % 400 != 0)
    tage = 365;
else tage = 366;
```

Konditional-Ausdrücke

werden mittels ternärem Operator `?:` gebildet

Syntax:

```
expr_0 ? expr_1 : expr_2
```

Semantik:

- Ausdruck `expr_0` auswerten
- falls `expr_0` gilt, dann `expr_1` auswerten, sonst `expr_2`
- es wird entweder `expr_1` oder `expr_2` ausgewertet

Beispiele:

- `max = (i > j) ? i : j;`
- `x = (x < 20) ? 20 : x;`

Konditional-Ausdrücke

Beispiel:
Schaltjahr-Bestimmung mittels Konditional-Ausdruck

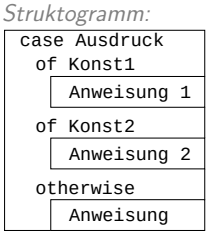
```
tage = (jahr % 4 != 0)
       ? 365
       : ((jahr % 100 != 0)
              ? 366
              : ((jahr % 400 != 0)
                     ? 365
                     : 366));
```


Auswahl aus mehreren Alternativen

Syntax:

```
switch (ausdruck) {
   case konst1: anweisung1
   case konst2: anweisung2
   ...
   case konstN: anweisungN
   default: anweisung
}
```

Struktogramm:



Semantik:

1. `ausdruck` wird ausgewertet → muss konstanten ganzzahligen Wert liefern (`switch`-Ausdruck)
2. `case`-Marken werden abgefragt (besteht aus `case`, ganzzahligem konstanten Ausdruck und Doppelpunkt).

switch-Anweisung

3. stimmt die `case`-Konstante mit dem `switch`-Ausdruck überein, dann wird der Programmfluss hinter der `case`-Marke fortgesetzt

   auch die Anweisungen der folgenden case-Teile werden ausgeführt!
4. soll die Kette nach Abarbeitung des `case`-Teils beendet werden, so müssen wir das explizit durch eine `break`-Anweisung erzwingen
5. alle `case`-Konstanten einer Kette müssen verschiedene Werte haben

switch-Anweisung

6. wird keine übereinstimmende `case`-Marke gefunden, so wird die Anweisung hinter der `default`-Marke ausgeführt (`default`-Marke und `default`-Anweisung sind optional!)
7. die `case`-Konstante darf in einigen Implementierungen keine `const`-Variable sein
   erlaubt: mittels `#define` festgelegte Konstanten

switch-Anweisung

Beispiel:

```
switch (c) {
  case 'm': klein = 1;
  case 'M': n = 1000;
            break;
  case 'd': klein = 1;
  case 'D': n = 500;
            break;
  case 'c': klein = 1;
  case 'C': n = 100;
            break;
  case 'l': klein = 1;
  case 'L': n = 50;
            break;
  case 'x': klein = 1;
  case 'X': n = 10;
            break;
  case 'v': klein = 1;
  case 'V': n = 5;
            break;
  case 'i': klein = 1;
  case 'I': n = 1;
            break;
  default: n = 0;
}
```


Format von C-Programmen

C-Programme können formatfrei geschrieben werden, d.h. sie müssen keine bestimmte Zeilenstruktur haben.

Schlüsselwörter und Namen sind getrennt zu schreiben!

Trennzeichen sind:

- Leerzeichen
- horizontaler Tabulator
- vertikaler Tabulator
- Seitenvorschub
- Zeilenende
- Kommentare

In vielen Firmen gibt es Konventionen

- wie Quelltexte zu formatieren sind
- wie Variablen, Tabellen, Dateien usw. zu benennen sind

Format von C-Programmen

guter Einstieg für übersichtliche Formatierung:

`http://java.sun.com/docs/codeconv/index.html`

Ziel unserer Konventionen: größtmögliche

- Deutlichkeit
- Übersichtlichkeit
- Testfreundlichkeit

Bitte zu Hause ansehen und in Praktikum und Übung anwenden!

Kommentare

Jeder Dummkopf kann Code schreiben, den ein Computer versteht.
Gute Programmierer schreiben Code, den Menschen verstehen!

Bezeichner müssen aussagekräftig sein:

- Wenn eine Variable kommentiert werden muss, ist der Name der Variablen schlecht gewählt!
- das Gleiche gilt auch für Funktionsparameter

Code-Blöcke kommentieren, keine einzelnen Zeilen!

Kommentare

So nicht:

```
i += 1;                 /* i um eins erhoehen   */
for (j=0; j<=n; j++)    /* zaehle j von 0 bis n */
```

sondern so:

```
/* tausche x und y */
if (x > y) {
    x = x - y;
    y = x + y;
    x = y - x;
}
```

oder so:

```
/* berechne s = x*x */
s = 0;
i = 1;
for (j=1; j<=x; j++) {
    s += i;
    i += 2;
}
```

heute: Wird ein Kommentar benötigt, um einen Code-Block zu erklären, sollte der Code umgeschrieben werden. → Refactoring: Methode extrahieren

Ohne Worte

gefunden unter `http://www.arkko.com/ioccc.html`:
International Obfuscated C Code Contest

```
#include<stdio.h>
#include<string.h>
main(){char*O,l[999]="''acgo\177~|xp .-"
"\0R^8)NJ6%K4O+A2M(*OID57$3G1FBL";while (O=
fgets(l+45,954,stdin)){*l=O[strlen(O)[O-1]=0,
strspn(O,l+11)];while(*O)switch((*l
&&isalnum(*O))-!*l){case-1:{char*I=
(O+=strspn(O,l+12)+1)-2,O=34;while
(*I&3&&(O=(O-16<<1)+*I---'-')<80);
putchar(O&93?*I&8||!(I=memchr(l,O,44))?'?'
:I-l+47:32);break;case 1:;}*l=(*O&31)[l-15+
(*O>61)*32];while(putchar(45+*l%2),
(*l=*l+32>>1)>35);case 0:
putchar((++O,32));}putchar(10);}}
```