

Anweisungen/Statements

- Programme bestehen aus
 - Vereinbarungen
 - Anweisungen
- Anweisungen müssen die algorithmischen Grundbausteine realisieren
 - Berechnung von Werten (Ausdrücke)
 - Alternative
 - Iteration
- Mit Hilfe von Funktionen/ggf. Prozeduren wird ein Programm in sinnvolle Einheiten aufgeteilt.

Ausdrucksanweisung

- Dient der Berechnung von Werten.
- Die Ausdrucksanweisung besteht aus einem Ausdruck, gefolgt von einem Semikolon.

```
I=U/R;  
P=U*I;  
P=U*(I=U/R);  
  
i+=2; j++;  
  
i+=sizeof x;  
  
i<<=2;
```

Block

- In c werden Vereinbarungen und Anweisungen von geschweiften Klammern umschlossen und bilden so einen Block.
- Ein Block darf überall dort stehen, wo eine Anweisung stehen darf.
- In klassischem c stehen in einem Block erst die Vereinbarungen, danach kommen die Anweisungen.

Block

Das ist eine Initialisierungsliste – kein Block!

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int Noten[]={5,2,3,4,5,5,2,3,4,5,0}; //38/10
5
6  int main()
7  {
8      int Akku=0, Count=0;
9      while(Noten[Count]!=0)
10     {
11         Akku=Akku+Noten[Count];
12         Count=Count+1;
13     }
14     if(Count>0)
15     {
16         Akku=Akku/Count;
17         printf("Durchschnitt: %d\n",Akku);
18     }else printf("Fehler - Division durch 0\n");
19     return 0;
20 }
```

Auch der Körper der main-Funktion kann als Block aufgefasst werden

Block

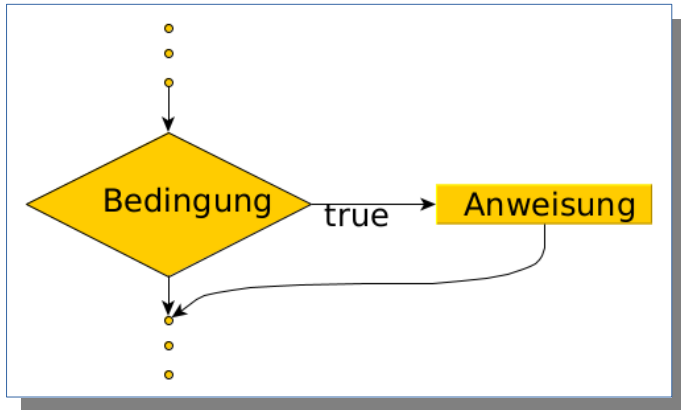
Block

Als Beispiel betrachten wir die Berechnung des arith. Mittels aus den einführenden Betrachtungen.

Bedingte Anweisung

- Bedingte Anweisungen gestatten die Ausführung von eingebetteten Anweisungen in Abhängigkeit einer Bedingung.
- Man unterscheidet zwischen der
 - Verkürzten if-Anweisung.
 - Vollständigen if-Anweisung.
- Bedingte Anweisungen werden mit dem Schlüsselwort **if** eingeleitet, es folgt eine Bedingung in runden Klammern und eine Anweisung (verkürzte if-Anweisung)
- An Stelle der Anweisung darf auch ein Block stehen.

Verkürzte bedingte Anweisung

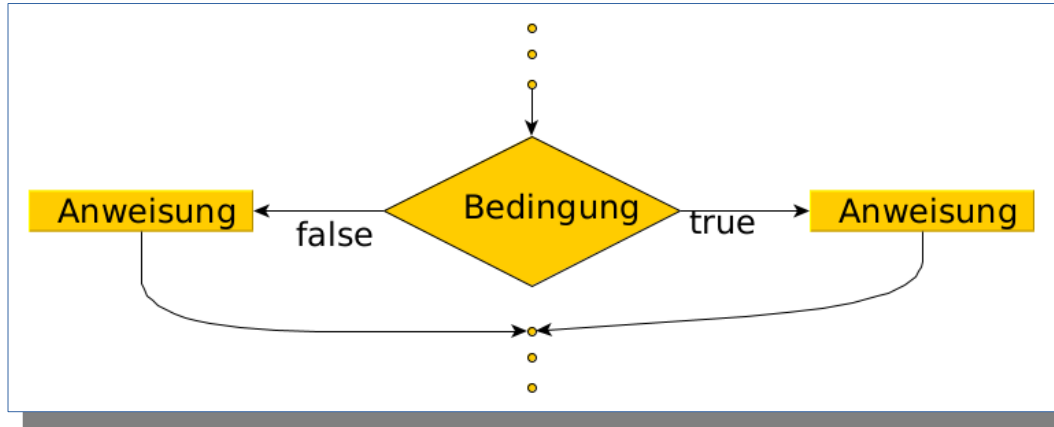


```
// wenn in der char-Variablen c  
// eine Ziffer steht, wird die  
// Zeichenkette "Ziffer" ausgegeben.  
if (c >= '0' && c <= '9') puts ("Ziffer");
```

Bedingte Anweisung

- Die vollständige if-Anweisung ergänzt die verkürzte Form durch ein else-Konstrukt.
- Auf die Anweisung hinter der Bedingung folgt das Schlüsselwort else und wieder eine Anweisung.
- An Stelle der Anweisung kann wieder ein Block stehen, die eingebetteten Anweisungen in der if-Anweisung können auch wieder bedingte Anweisungen sein.

Vollständige bedingte Anweisung



```
// a, b und max sind int-Variable  
// Es wird das Maximum von a und b ermittelt.  
if (a>b) max=a;  
else     max=b;
```


Bedingte Anweisung

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  char buf[128];
5
6  int main()
7  {
8      int a,b,max;
9
10     printf("Input a:");
11     fgets(buf, 128,stdin); a=atoi(buf);
12     printf("Input b:");
13     fgets(buf, 128,stdin); b=atoi(buf);
14     if (a>b) max=a;
15     else     max=b;
16     printf("Max (%d, %d): %d\n",a,b,max);
17
18     return 0;
19 }
```

```
$ gcc maximum.c
$ ./a.out
Input a:12
Input b:21
Max (12, 21): 21
$
```

oder

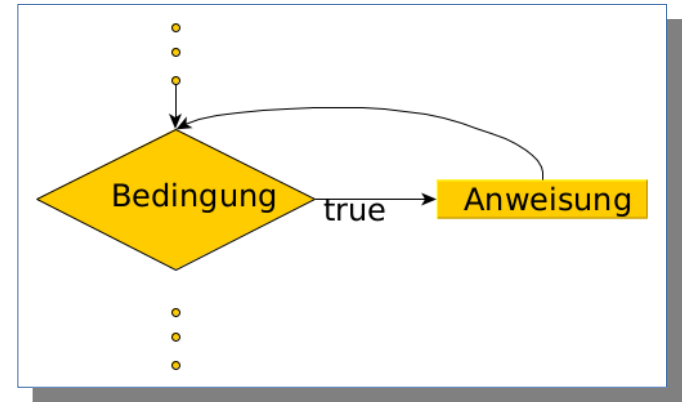
```
$ gcc maximum.c -o maximum
$ ./maximum
Input a:12
Input b:21
Max (12, 21): 21
$
```

Schleifen

- Schleifen dienen der wiederholten Ausführung von Anweisungen
- Sie enthalten eine Bedingung. Von deren Wert hängt ab, ob die eingebettete Anweisung (wiederholt) ausgeführt, oder die Schleife verlassen wird.
- Man unterscheidet zwei Typen von Schleifen
 - Abweisende Schleife
 - Nicht abweisende Schleife

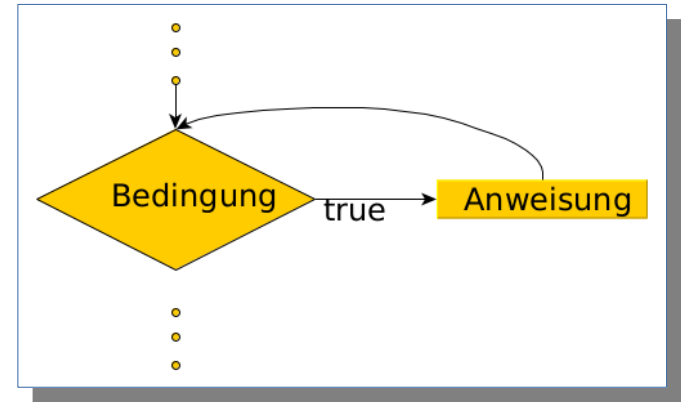
Abweisende Schleifen

- Bei der abweisenden Schleife wird immer zuerst die Bedingung bewertet.
- Es kann sein, dass die eingebettete Anweisung kein mal ausgeführt wird.
- Die abweisende Schleife heißt mitunter auch kopfgesteuerte Schleife.



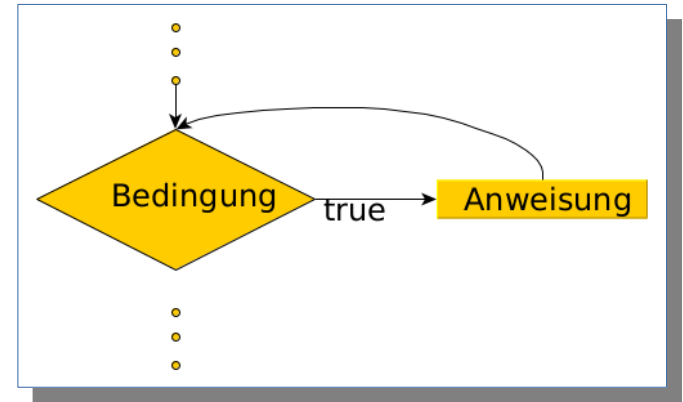
Abweisende Schleifen

- Bei der abweisenden Schleife wird immer zuerst die Bedingung bewertet.
- Es kann sein, dass die eingebettete Anweisung kein mal ausgeführt wird.
- Die abweisende Schleife heißt mitunter auch kopfgesteuerte Schleife.



While-Schleifen

- Typischer Vertreter der Abweisenden Schleife ist die While-Schleife.
- Sie wird durch das Schlüsselwort `while` eingeleitet, es folgen die Bedingung in runden Klammern und die eingebettete Anweisung.



Beispiel e^x

- Werte der Funktion e^x kann man numerisch nach einer Reihenentwicklung berechnen.

$$\bullet \exp(x) = \sum_{n=0}^{\infty} \frac{x^n}{n!} = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \frac{x^4}{4!} + \dots$$

- Benötigt werden die Variablen x , y oder Summe, Summand der Reihe und ein Zähler für die Fakultät.
- Da Potenz und Fakultät sehr stark steigen, sollen Zähler und Nenner nicht getrennt berechnet, sondern der n -te Summand soll immer aus seinem Vorgänger berechnet werden.

```

1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <math.h>
4  char vBuf[128];
5
6  int main()
7  {
8      double y=1.0, summand=1.0, x;
9      int i=1;
10     printf("Eingabe von x:");
11     fgets(vBuf,128,stdin);
12     x=atof(vBuf);
13     printf("x: %lf\n",x);
14     while (summand>0.00005)
15     {
16         summand=summand*x/i;
17         y+=summand;
18     }
19     i++;
20 }
21 printf("Ergebnis: y=%lf\n",y);
22 return 0;
23 }

```

Das komplette Beispiel e^x

- Zeile 8 und 9 enthalten die nötigen Variablendefinitionen.
- Zeile 10 bis 13 dienen der Eingabe des x-Wertes zu dem der Funktionswert berechnet werden soll.
- Zeile 14 bis 20 enthalten die Schleife zur näherungsweise Berechnung des Funktionswertes (ausführlich nächste Folie).
- Die eingebettete Anweisung ist hier ein Block.
- Zeile 21 dient der Ausgabe des Ergebnisses.

Die While-Schleife in e^x

```
14 while (summand>0.00005)
15 {
16     summand=summand*x/i;
17     y+=summand;
18     |
19     i++;
20 }
```

- Bedingung

- Die Summanden werden immer kleiner (die Reihe konvergiert) und beeinflussen so die Genauigkeit des Ergebnisses immer weniger.
- Ist der Summand kleiner als 0.00005, so kann die Berechnung abgebrochen werden oder anders formuliert, solange der Summand größer als 0.00005 ist, wird die Schleife erneut ausgeführt.
- Summand ist mit 1.0 initialisiert, das ist größer als 0.00005, somit ist die Bedingung true und die eingebettete Anweisung wird ausgeführt.

Die While-Schleife in e^x

```
14 while (summand>0.00005)
15 {
16     summand=summand*x/i;
17     y+=summand;
18     |
19     i++;
20 }
```

- Berechnung des Summanden (Zeile 16)

- Der Summand soll immer aus seinem Vorgänger berechnet werden.
- Per Initialisierung (Zeile 8) ist Summand zunächst 1.
- Wird er mit x multipliziert, erhält er den Wert x ($1*x \rightarrow x$). Im nächsten Durchlauf wird er wieder mit x multipliziert, da wird aus x dann x^2 , dann x^3 usw.
- Weiter wird der Summand bei jedem Durchlauf durch i , das mit dem Wert 1 initialisiert ist und bei jedem Durchlauf incrementiert wird, dividiert.
- Damit entsteht im Nenner die jeweilige Fakultät.

Die While-Schleife in e^x

```
14 while (summand>0.00005)
15 {
16     summand=summand*x/i;
17     y+=summand;
18     |
19     i++;
20 }
```

- Berechnung der Summe (Zeile 17)

- In Zeile 17 entsteht schrittweise das Ergebnis.
- Ergänzt man in Zeile 18 eine Ausgabe, so kann man das Verfahren gewissermaßen beobachten:

```
printf("Summand:%lf Zw.Erg:%lf\n", summand, y);
```

- In Zeile 19 erfolgt die Incrementierung von i. Man könnte auch **i=i+1;** verwenden, aber **i++;** ist in c eher üblich.

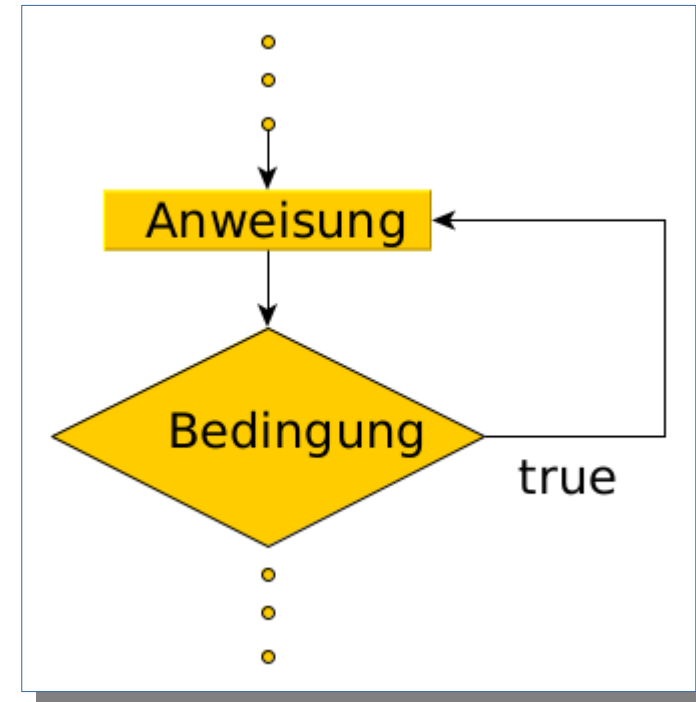
e^x ausprobieren

- Erfassen und speichern des Quelltextes unter z.Bsp.: while1.c

```
$ ./while1
Eingabe von x:1.0
x: 1.000000
Summand:1.000000 Zw.Erg:2.000000
Summand:0.500000 Zw.Erg:2.500000
Summand:0.166667 Zw.Erg:2.666667
Summand:0.041667 Zw.Erg:2.708333
Summand:0.008333 Zw.Erg:2.716667
Summand:0.001389 Zw.Erg:2.718056
Summand:0.000198 Zw.Erg:2.718254
Summand:0.000025 Zw.Erg:2.718279
Ergebnis: y=2.718279
$
```

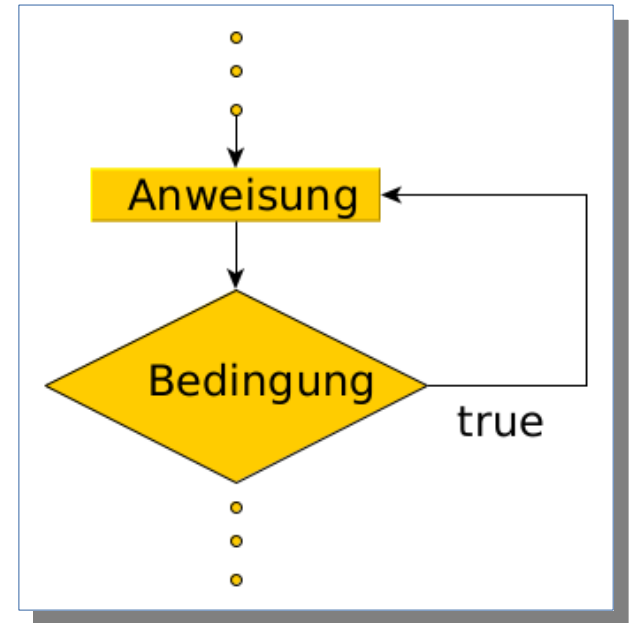
Nicht abweisende Schleifen

- Bei der nichtabweisenden Schleife wird immer zuerst die Anweisung ausgeführt.
- Die eingebettete Anweisung also wenigstens ein mal ausgeführt.
- Die Berechnung und Auswertung der Bedingung erfolgt danach.
- Ergibt die Bedingung den Wert true, wird die Anweisung erneut ausgeführt.
- Die abweisende Schleife heißt mitunter auch fußgesteuerte Schleife.



Do-While-Schleifen

- Typischer Vertreter der Nicht-abweisenden Schleife ist die Do-While-Schleife.
- Sie wird durch das Schlüsselwort `do` eingeleitet, es folgen die Anweisung und in runden Klammern die Bedingung. Ein Semikolon schließt die Anweisung ab.



```

1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <math.h>
4  char vBuf[128];
5
6  int main()
7  {
8      double y=1.0, summand=1.0, x;
9      int i=1;
10     printf("Eingabe von x:");
11     fgets(vBuf,128,stdin);
12     x=atof(vBuf);
13     printf("x: %lf\n",x);
14     do
15     {
16         summand=summand*x/i;
17         y+=summand;
18         i++;
19     }while (summand>0.00005);
20     printf("Ergebnis: y=%lf\n",y);
21     return 0;
22 }

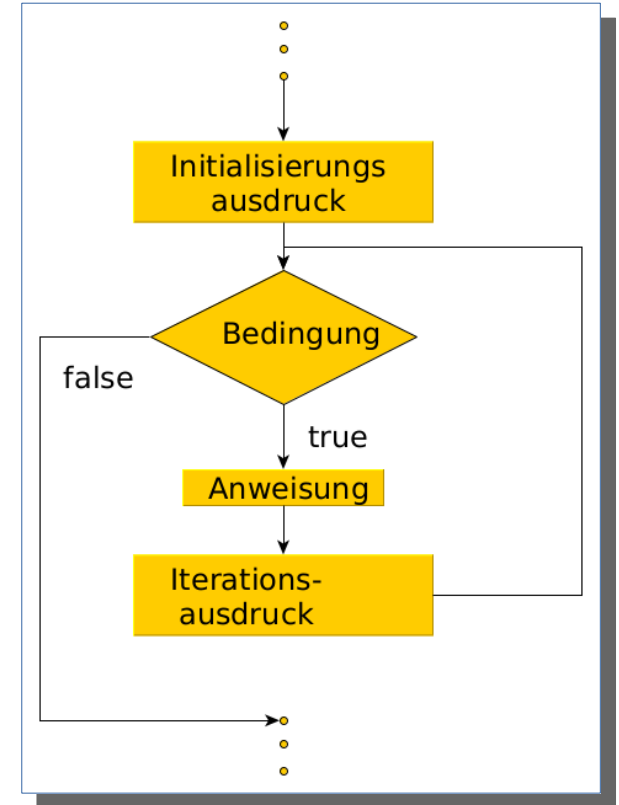
```

Das komplette Beispiel e^x

- Nebenstehend ist das Beispiel in der Variante mit der do-While-Schleife abgebildet.
- Zeile 14 bis 19 enthalten die Schleifenkonstruktion.
- Am Algorithmus ist nichts geändert worden.
- Die eingebettete Anweisung der do-while-schleife ist auch hier der Block in Zeile 15 bis 19, in dem die Berechnungen erfolgen.

for-Schleifen

- In vielen Programmiersprachen gibt es zusätzlich sogenannte Zählschleifen, die mit dem Schlüsselwort for eingeleitet werden.
- In c ist die for-Schleife weiter gefasst und ist eine universelle, abweisende Schleife.



for-Schleife

- Die for-Schleife wird durch das Schlüsselwort for eingeleitet. Ihm folgen in runden Klammern 3 durch Semikolon getrennte Ausdrücke
 - Initialisierungsausdruck, er wird nur ein mal beim Schleifenstart vor der Berechnung der Bedingung bewertet.
 - Ist der Bedingungsausdruck true, wird die auszuführende Anweisung ausgeführt, ist er false wird die Schleife verlassen.
 - Iterationsausdruck, er wird immer nach Ausführung der Anweisung bewertet und besteht sehr oft nur aus einer Incrementoperation.
- Nach der schließenden Klammer steht die eingebettete, auszuführende Anweisung.


```

1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <math.h>
4  char vBuf[128];
5
6  int main()
7  {
8      double y, summand=1.0, x;
9      int i;
10     printf("Eingabe von x:");
11     fgets(vBuf,128,stdin);
12     x=atof(vBuf);
13     printf("x: %lf\n",x);
14     y=1.0,summand=1.0;
15     for (i=1; summand>0.00005; i++)
16     {
17         summand=summand*x/i;
18         y+=summand;
19     }
20     printf("Ergebnis: y=%lf\n",y);
21     return 0;
22 }
23

```

Das komplette Beispiel e^x

- Nebenstehend ist das Beispiel in der Variante mit der for-Schleife abgebildet.
- Zeile 15 bis 19 enthalten die Schleifenkonstruktion.
- Am Algorithmus ist nichts geändert worden.
- Die eingebettete Anweisung der do-while-schleife ist auch hier der Block in Zeile 16 bis 19, in dem die Berechnungen erfolgen.
- Die Variable i wird hier im Initialisierungsausdruck mit 1 belegt und im Interationsaudruck jeweils incremented.

- Das Beispiel kann noch vielfältig modifiziert werden.
- Initialisierungs- und Incrementoperator können auch den Kommaoperator enthalten, somit wird auch nachfolgende Form möglich:

```
15     for (i=1,y=1.0, summand=1.0;  
16         summand>0.00005;  
17         summand=summand*x/i, y+=summand, i++)  
18     {  
19     }
```

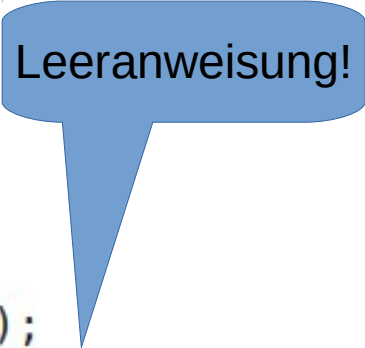
- Der Block als auszuführende Anweisung bliebe hier leer, die Berechnung erfolgt im Incrementausdruck der for-Schleife. Guter Programmierstil ist das aber nicht.

Leeranweisung

- An Stellen, wo eine Anweisung stehen muss, aber nichts passieren soll, haben wir auf der vergangene Seite einen leeren Block eingefügt.
- Eleganter ist in solchen Fällen eine Leeranweisung.
- Sie wird durch ein Semikolon dargestellt.

Leeranweisung

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <math.h>
4  char vBuf[128];
5
6  int main()
7  {
8      int i;
9      printf("Eingabe:");
10     fgets(vBuf, 128, stdin);
11
12     for (i=0; vBuf[i]; i++);
13     vBuf[i-1]=0;
14     i--;
15     printf("Length of %s: %d\n", vBuf, i);
16     return 0;
17 }
```



Leeranweisung!

- Zeile 10 dient der Eingabe einer Zeichenkette.
- Zeile 12 enthält die Schleife, sie ermittelt die Länge der Zeichenkette. Das Ende ist mit einem Nullbyte (terminierende 0) markiert.
- Als Bedingung reicht in c **vBuf[i]**; . Ist der Wert 0, gilt das als false, alles andere wird als true interpretiert, das entspricht letztlich **vBuf[i]!=0**;
- Die auszuführende Anweisung ist leer (Leeranweisung).

| | | | | |
|-----|-----|-----|-----|-----|
| M | a | x | \n | 00 |
| [0] | [1] | [2] | [3] | [4] |

- In der nachfolgenden Zeile 13 wird der mit eingegebene Zeilenwechsel ('\\n') entfernt.
- Die ermittelte Länge beträgt 4. An der Stelle Länge-1 (hier [3]) wird das Zeichen '\\n' mit einer 0 überschrieben.
- In Zeile 14 wird die Länge nun korrigiert, die Länge von Max ist somit 3.
- Randbemerkung: zur Ermittlung der Stringlänge wird gern `strlen` verwendet. Um das Zeilenwechselzeichen zu entfernen, ergibt sich damit: **`vBuf[strlen(vBuf) - 1] = 0;`**

break- Anweisung

```
12 while (1)
13 {
14     summand=summand*x/i;
15     y+=summand;
16     printf("Summand:%lf Zw.Erg:%lf\n",summand, y);
17     if (summand<0.00005) break;
18 }
```

- Die Anweisung break dient dem „außerplanmäßigen“ Verlassen der unmittelbar umgebenden Schleife.
- Im obigen Beispiel wurde in Zeile 12 die Schleifenbedingung durch 1 (true) ersetzt. Man spricht von einer Endlosschleife.
- In Zeile 17 wurde eine if-Anweisung eingefügt. Ist der Summand < 0.00005 wird die Schleife via break verlassen.

continue-Anweisung

- Die continue-Anweisung verlässt die auszuführende Anweisung der unmittelbar umgebenden Schleife und setzt die Programmausführung mit einer erneuten Berechnung der Schleifenbedingung fort.
- Continue wird eher selten benötigt.
- Ein Beispiel findet sich unter:

http://openbook.rheinwerk-verlag.de/c_von_a_bis_z/008_c_kontrollstrukturen_011.htm#mjd84e64cf5f6936ffe215d31e7708dc0f

Fallunterscheidung switch-case

- Die Fallunterscheidung ist eine etwas komplexere Anweisung.
- Sie beginnt mit dem Schlüsselwort switch, worauf ein ganzzahliger Wert (Ausdruck) in runden Klammern folgt. Danach schließt sich eine Liste von Anweisungen in geschweiften Klammern an.

```
Switch (a)
{
    // Anweisungen
    . . .
}
```


Fallunterscheidung switch-case

- Innerhalb der Anweisungsliste können einzelne Anweisungen mit sogenannten caseLabels markiert sein.
- Jedes Caselabel besteht aus dem Schlüsselwort case und einer ganzzahligen Konstante.
- Die Caselabels bilden Einsprungmarken in die Anweisungsliste.
- Als Beispiel diene die Berechnung der Summe der Tage vollständig abgelaufener Monate eines Jahres.

Beispiel zur Berechnung der Tage seit dem 01.01.1900 unter der Annahme, dass im Zeitraum 1901-2099 jedes durch 4 teilbare Jahr ein Schaltjahr ist. Die Variablen d, m und y enthalten die Datumswerte für Tag, Monat und Jahr.

```
17 days=(y-1900)*365+(y-1900)/4;
18 switch(m)
19 {
20     case 12: days+=30;
21     case 11: days+=31;
22     case 10: days+=30;
23     case 9: days+=31;
24     case 8: days+=31;
25     case 7: days+=30;
26     case 6: days+=31;
27     case 5: days+=30;
28     case 4: days+=31;
29     case 3: days+=28;
30     case 2: days+=31;
31     case 1: days+=d;
32 }
33 if (y>1900 && (y%4)==0 && m<3)days--;
```

Der vollständige Code des Beispiels enthält noch unbekanntes Material. Er liegt im Downloadbereich Beispiele zur Vorlesung 2020

- Nebenstehende switch-Konstruktion enthält 12 case-Labels.
- Je nach Monat wird am CaseLabel in die Anweisungsliste hineingesprungen. (im Juli bei case 7:).
- Von dort wird die Anweisungsliste sequenziell nach unten ausgeführt.
- Dabei werden die Anzahl der Tage der vollständig vergangenen Monate aufsummiert.
- Im Januar wird der aktuelle Tag addiert oder 0, weil im Januar noch kein Monat vollständig vergangen ist, falls man den Tag nicht bereits in Zeile 17 addiert hat.

break in switch/case

- Im vergangenen Beispiel werden die Anweisungen ab der zutreffenden Einsprungmarke bis zum Ende der Switch-Konstruktion durchlaufen.
- `break` beendet die Ausführung von Anweisungen in der `switch`-Konstruktion.
- Es kann, muss aber nicht zu jedem `case`-Label ein `break` geben.

```

1  #include <stdio.h>
2  #include <stdlib.h>
3
4  char buf[128];
5  char ErrDivZero[]="Divide by zero - no operation performed\n";
6  char ErrWrongOp[]="wrong operator - no operation performed\n";
7  int main()
8  {
9      int result;
10     char operator;
11     int value;
12     result=0;
13     while(1)
14     {
15         printf("result:%d\n",result);
16         printf("enter operator and number:");
17         fgets(buf,sizeof(buf),stdin);
18         sscanf(buf,"%c %d",&operator,&value);
19         {
20             switch (operator)
21             {
22                 case 'Q':
23                 case 'q':exit(0);          break;
24                 case '+':result+=value; break;
25                 case '-':result-=value; break;
26                 case '*':result*=value; break;
27                 case '/':
28                 case '%':if (value==0)
29                     puts(ErrDivZero);
30                     else
31                     if (operator=='/')result/=value;
32                     else
33                     result%=value; break;
34                 default: puts(ErrWrongOp);
35             }
36         }
37     }
38     return 0;
39 }

```

- Das Beispiel zeigt ein kleines Rechenprogramm ohne Berücksichtigung der Operatorpriorität.
- Es wird immer eine Operation und ein Operand eingegeben.
- Der Anfangswert ist 0, es ist also sinnvoll, mit einer additiven Operation zu beginnen.
- In der Switchkonstruktion wird ausgewertet, welche Operation eingegeben worden ist und entsprechend gerechnet.
- Für die Operationen Division und Modulo wird zunächst geprüft, ob der Divisor ungleich 0 ist, und danach mit einer if-Anweisung die eingegebene Operation ausgeführt.

```

$ ./a.out
result:0
enter operator and number:+10
result:10
enter operator and number:*3
result:30
enter operator and number:/6
result:5
enter operator and number:q
$

```

Kommentare

- Ein Programm kann/sollte Kommentare enthalten, die das Programm erklären.
- In c gibt es zweierlei Arten von Kommentaren:
 - */* das ist ein Kommentar, der über mehrere Zeilen gehen kann, er wird in Stern-Schrägstrich geklammert */*
 - *// Das ist auch ein Kommentar, der mit der Zeile endet.*

Kommentare

- Jedes Programm sollte im Kopf einen Kommentar mit dem Namen des Programmierers (Autors) haben.
- Bei komplexeren Programmen, die aus mehreren Quelldateien bestehen, ist es mitunter sinnvoll, das Build-Kommando und ev. Einen Programmaufruf mit der Kommandozeile als Kommentar einzufügen.
- Jede Funktion sollte mit einem Kommentar, der die Parameter und die Aufgabe der Funktion beschreibt, versehen sein.
- Darüber hinaus sollten Passagen, die nicht selbsterklärend sind, kommentiert werden.

Kommentare

- Kommentieren hilft, Programmfehler zu vermeiden, weil man sich noch mal genau überlegt, was an der betreffenden Stelle genau passiert.
- Was man im Moment des Programmierens noch meint, zu verstehen, kann nach zwei Wochen schon völlig unverständlich sein, wenn es nicht kommentiert ist.
- In Softwareunternehmen gibt es sogenannte Coding Standards, die Dinge, wie Namensgebung und Kommentierung sowie weitere Aspekte der Quelltextgestaltung regeln.

Einrückung/Quelltextgestaltung

- In Blöcken sollte prinzipiell eingerückt werden.
- Es gibt verschiedene Gestaltungsvarianten:

```
if (values[i]>values[j])
{
    values[i]^=values[j];
    values[j]^=values[i];
    values[i]^=values[j];
}
```

```
if (values[i]>values[j]){
    values[i]^=values[j];
    values[j]^=values[i];
    values[i]^=values[j];
}
```


Zusammenfassung Statements

- Nachfolgende Anweisungen wurden betrachtet:
 - Ausdrucksanweisung zur Berechnung von Werten
 - If-Anweisung (verkürzt/vollständig) für Alternativen
 - Schleifen (while, do-while, for)
 - break/continue
 - switch/case für Fallunterscheidung
- Es gibt in c noch eine goto-Anweisung, deren Verwendung gilt als kein guter Programmierstil, sie wurde deshalb nicht behandelt.
- Weiterhin gibt es noch die Anweisung return, sie wird später behandelt.