Strukturierte Programmierung

Die strukturierte Programmierung ist ein Programmierparadigma (~ Programmierstil), das ca. ab 1970 populär wurde. Dabei spielte auch der Entwurf der Programmiersprache Pascal 1971, die Studenten die strukturierte Programmierung beibringen soll, eine Rolle.

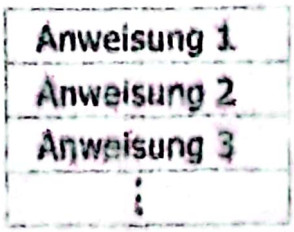
Die wichtigsten 4 Merkmale der strukturierten Programmierung sind:

- Einsatz von **Sequenzen** (hintereinander ausführende Programmanweisungen)
- Einsatz von **Verzweigungen**
- Einsatz von **Iterationen**
- Zerlegung des Programms in Teilprogramme (**Funktionen**). Wenn eine Funktion mehrmals im Programm aufgerufen wird, entsteht dadurch weniger Quellcode, die Wahrscheinlichkeit einen Fehler zu machen sinkt und Änderungen lassen sich leichter vornehmen.

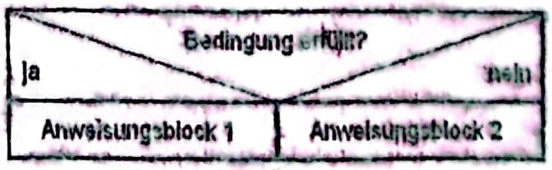
Zur Darstellung von strukturierten Programmen sind zwei Darstellungsformen üblich:

1. **Struktogramme**

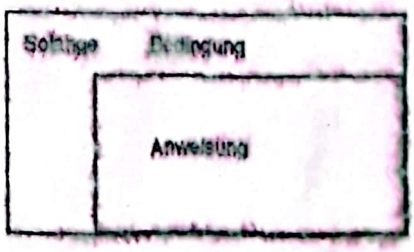
   - Sequenz/Block



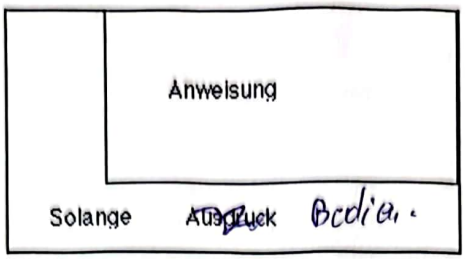
   - Verzweigung / if...else



   - Kopfgesteuerte Schleife / while

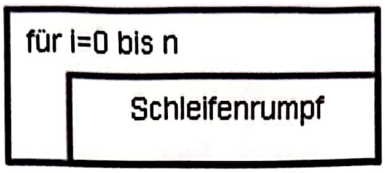


- Fußgesteuerte Schleife / do...while



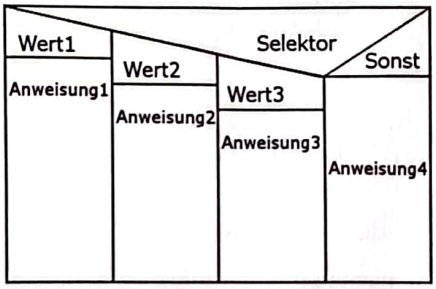
- Zählschleife / for

für i=0 bis n, Schritt 1
SW: 1



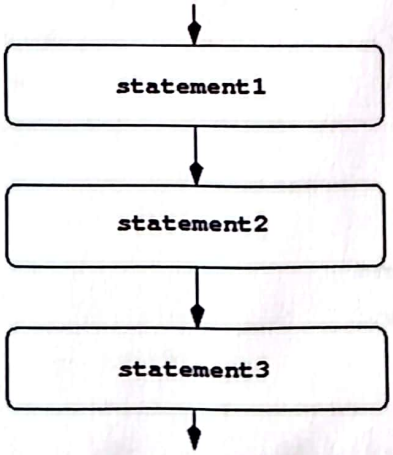
(Anmerkung: Korrekterweise ist das „=“ mit einem Zuweisungspfeil ← zu ersetzen)

- Mehrfachverzweigung / switch … case

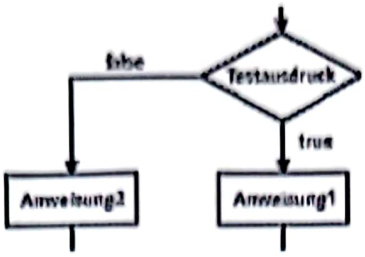


2. Flussdiagramme

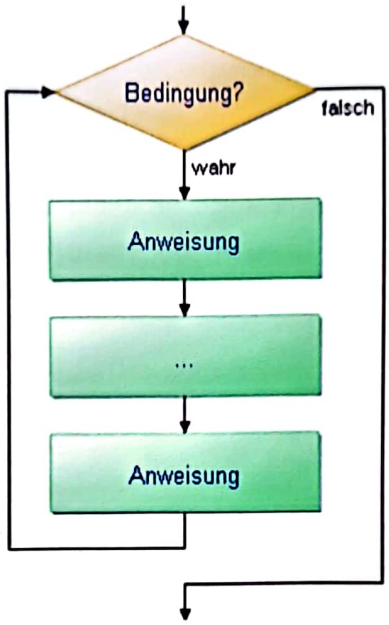
- Sequenz/Block



- Verzweigung / if...else



- Schleife



Syntax der strukturierten Programmierung

- Verzweigung

```
if (<Bedingung>)
      <Anweisung>
else <Anweisung>
```

- Mehrfachverzweigung

```
switch (<Selektor>)
{
      case <Wert 1> : <Anweisung 1> break;
      case <Wert 2> : <Anweisung 2> break;
      default : <Anweisung 3>
}
```

  - Der Ausdruck nach dem `case` (Wert 1) muss eine Konstante sein (keine Variable)

- In C darf nur ein ganzzahliger Datentyp als Selektor verwendet werden (int, char, enum)
- In Java darf außerdem ein String als Selektor verwendet werden

- Alternative verschachtelte Mehrfachverzweigung:

```
if (<Bedingung>)
      <Anweisung 1>
else if (<Bedingung>)
      <Anweisung 2>
else
      <Anweisung 3>
```

- Sequenzen (=Anweisungsblock)

```
{
      <Optionale Liste von Deklarationen>
      <Optionale Liste von Anweisungen>
}
```

- Kopfgesteuerte Schleife

```
while ( < Laufbedienung > )
      <Anweisung>
```

  - Hier wird schon vor der eventuellen Ausführung der Anweisungen im Schleifenrumpf die Laufbedienung (im Schleifenkopf) geprüft.

- Fußgesteuerte Schleifen

```
do
      <Anweisung>
while ( < Laufbedienung > );
```

  - Hier wird erst nach dem Durchlauf des Schleifenrumpfes eine Bedienung geprüft. Deswegen wird diese Form der Schleife immer dann verwendet, wenn die Überprüfung der Bedienung erst nach dem ersten Durchlauf möglich ist.

- Zählschleifen

Kollektion

```
for ( <Initialisierung> ; <Laufbedienung> ; <Schrittweite> )
      <Anweisung>
```

  - i,j,k = Laufvariable
  - Auch hier wird vor dem Durchlaufen des Schleifenrumpfes die Bedienung geprüft. (=vorprüfende Schleife)

- Mengenschleifen (FOREACH)

- Sonderform der Zählschleife
- „für jedes Element der Menge“
- In C nicht möglich, in Java schon:

```
for(<Datentyp Element> <Bezeichnung Element> : <Kollektion>)
    <Anweisung>
```

foget

EVA-Prinzip

Bei Programmen, die in der Kommandozeile ausgeführt werden (also wie z.B. in vielen Fällen C-Programme) wird oft nach dem EVA-Prinzip verfahren:

1. Eingabe
2. Verarbeitung
3. Ausgabe

Zuerst wird dem Benutzer die Möglichkeit gegeben, Eingaben zu tätigen oder aus Möglichkeiten zu wählen. Dann werden die Eingaben verarbeitet und die Ergebnisse ausgegeben. Das steht z.B. im Gegensatz zu einer GUI, die auf Ereignisse reagiert.

Fehler in Programmen

- **Syntaxfehler** sind Verstöße gegen die syntaktischen Regeln einer Programmiersprache. Programme mit Syntaxfehlern werden vom Compiler mit einer entsprechenden Fehlermeldung zurückgewiesen.
- **Semantische Fehler** sind Fehler, in denen eine programmierte Anweisung zwar syntaktisch fehlerfrei, aber inhaltlich trotzdem fehlerhaft ist. Zum Beispiel syntaktisch nicht erkennbare falsche Parameterreihenfolge. (Achtung! Logische Fehler und Laufzeitfehler sind keine semantischen Fehler!)

Algorithmus

Ein Algorithmus ist ganz allgemein eine Verfahrensanweisung zur Lösung eines Problems. In der Informatik wird als Algorithmus meist eine abstrakte Lösungsstrategie und nicht ein konkret ausimplementiertes Programm bezeichnet.

Sortierverfahren

Sortierverfahren oder Sortieralgorithmen haben die Aufgabe, eine Menge an Daten nach einem bestimmten Merkmal zu sortieren. Zum Beispiel könnte man eine Liste von Kunden nach deren Nachnamen sortieren. Dazu gibt es unterschiedliche Algorithmen, die sich in zwei Kriterien unterscheiden:

- Zeitkomplexität - Wie viele Operationen sind nötig?

  Das wird meistens in der Landau-Notation angegeben:

  - O(n) = Ein Durchlauf durch das Feld
  - $O(n^2)$ = Feldlänge ² Durchläufe durch das Feld
  - O(n*log(n))
  - ...

- Platzkomplexität - Wie viel zusätzlicher Speicherplatz wird benötigt?

Wie schnell die unterschiedlichen Algorithmen sind, hängt auch davon ab, wie sehr das Feld schon „vorsortiert" ist. Beispiele für Sortieralgorithmen sind:

- Mergesort O(n*log(n))
- Heapsort O(n*log(n))
- Quicksort O(n*log(n))
- Timsort O(n*log(n))

BubbleSort

Ein sehr einfacher und schlechter Sortieralgorithmus ist Bubblesort, der zu keinem zusätzlichen Verbrauch von Speicherplatz führt. Im besten Fall werden nur O(n) Durchläufe benötigt. Im Normalfall und im schlechtesten Fall aber $O(n^2)$. BubbleSort wird sinnvoll eingesetzt, wenn entweder n klein ist oder wenn die Daten mit hoher Wahrscheinlichkeit schon annähernd sortiert sind.

In der Bubble-Phase wird die Eingabe-Liste von links nach rechts durchlaufen. Dabei wird in jedem Schritt das aktuelle Element mit dem rechten Nachbarn verglichen. Falls die beiden Elemente das Sortierkriterium verletzen, werden sie getauscht. Am Ende der Phase steht bei auf- bzw. absteigender Sortierung das größte bzw. kleinste Element der Eingabe am Ende der Liste.

Die Bubble-Phase wird solange wiederholt, bis die Eingabeliste vollständig sortiert ist. Dabei muss das letzte Element des vorherigen Durchlaufs nicht mehr betrachtet werden, da die restliche zu sortierende Eingabe keine größeren bzw. kleineren Elemente mehr enthält.

Je nachdem, ob auf- oder absteigend sortiert wird, steigen die größeren oder kleineren Elemente wie Blasen im Wasser (daher der Name) immer weiter nach oben, das heißt, an das Ende der Liste. Es werden stets zwei Zahlen miteinander in „Bubbles" vertauscht.

Dabei gibt es folgendes Problem: Große Elemente zu Beginn wirken sich nicht negativ aus, da sie schnell nach hinten getauscht werden. Jedoch kleine Elemente am Ende bewegen sich nur langsam nach vorn. Um den Worst-Case-Fall mit einem umgekehrt sortierten Eingabefeld besser verarbeiten zu können, wurde BubbleSort weiterentwickelt (Combsort, Cocktailsort).

Implementierung von BubbleSort in C:

```
void bubble_sort_1(int array[], int length)
{
  int i,j;
  for (i = length ; i>1; i--)
  {
    for (j = 0 ; j<i-1; j ++)
    {
      if (array[j] > array[j+1])
        swap(array[j], array[j+1]);
    }
  }
}
```

Die äußere Schleife verringert schrittweise die rechte Grenze für die Bubble-Phase, da nach jedem Bubblen an der rechtesten Position das größte Element der jeweils unsortierten Rest-Eingabe steht. In der inneren Schleife wird der noch nicht sortierte Teil des Feldes durchlaufen. Dabei werden zwei benachbarte Daten vertauscht, wenn sie in falscher Reihenfolge sind (also das Sortierkriterium verletzen).

In dieser einfachen Variante wird jedoch nicht ausgenutzt, dass wenn das Feld bereits sortiert ist (also keine Vertauschungen mehr stattgefunden haben), keine weiteren Schleifendurchläufe mehr stattfinden müssen. Die folgende Variante berücksichtigt das:

```
void bubble_sort_2(int array[], int length)
{
  bool swapped; int i,j;
  for (i = length ; i>1; i--)
  {
    swapped = false;
    for (j = 0 ; j<i-1; j ++)
    {
      if (array[j] > array[j+1])
      {
        swap(array[j], array[j+1]);
        swapped = true;
      }
    }
    if (!swapped)
      break;
  }
}
```

Special für die Datentypfanatiker in C

Name	Minimum	Maximum	Speicher-bedarf
(signed) char	-128	127	8 Bit
unsigned char	0	255	8 Bit
(signed) int	-2.147.483.648	2.147.483.647	32 Bit
unsigned int	0	4.294.967.295	32 Bit
(signed) short int	-32.678	32.767	16 Bit
unsigned short int	0	65.536	16 Bit
(signed) long int	-2.147.483.648	2.147.483.647	32 Bit
unsigned long int	0	4.294.967.295	32 Bit
float	$3{,}4 \cdot 10^{-38}$	$3{,}4 \cdot 10^{38}$	32 Bit
double	$1{,}7 \cdot 10^{-308}$	$1{,}7 \cdot 10^{308}$	64 Bit
long double	$3{,}4 \cdot 10^{-4932}$	$3{,}4 \cdot 10^{4932}$	80 Bit