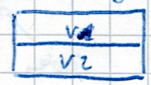
Programmieren

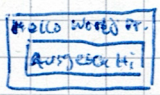
Programmablaufplan

- : Anweisung / Operation
- : Verzweigung
- : Eingabe / Ausgabe
- : Unterprogramm
- : Start, Stop, Return etc.
- : Übergangsstelle zwischen Programmteilen

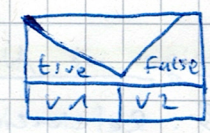
Struktogramm



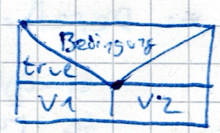
: Verarbeitungsschritte V1 und V2



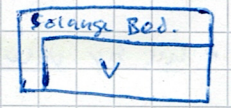
: Block: Haupt- oder Unterprogramm



: Verzweigung



: bedingte Verarbeitung



: while Schleife

---

for vs while

```
for (Initialisierung, Bedingung, Schleifenanweisung)
{
   ...
}
```

(i=0; | i<=10; | i++)

```
Initialisierung;
while (Bedingung)
{
   ...
   Schleifenanweisung;
}
```

(i=0; | i<=10; | i++;)

break = gesamte Schleife verlassen
continue = Abbruch eines Durchlaufs

Funktionen

Aufbau → Definition

```
Rückgabetyp Funktionsname (typ1 Bezeichner1, ...)
{
   ...
   return Ausdruck; // vom Typ Rückgabetyp
}
```

Funktionsprototyp:
z.B. `int name(int, int);`
↳ Deklaration

```
scanf("%d", &speicher)
```

Typ, der eingelesen werden soll | wo es gespeichert werden soll (welche Variable)

```
printf("%d", speicher)
```

Deklaration: `int a;`

Deklaration + Definition + Initialisierung: `int a = 1;` ohne & !

Typumwandlung (Cast)

Compiler nimmt Typumwandlung vor → impliziter Cast

```
Bsp: int a;
     char f = 'A';
     a = f;
```

Typumwandlung von Programmierer erzwungen → expliziter Cast

```
Bsp: int a;
     double d = 1,23;
     a = (int) d;
```

- Compiler rechnet mit "größtem" Typ

Arrays

- variable Anzahl von Elementen eines Datentyps

Index:
0 – (Anzahl-1) z.B. `int feld[10];`
↳ Index 0-9

Initialisierung:
z.B. `int feld[4] = {0}` → alle Elemente auf 0
`int feld[4] = {0,1,3,5}`
`int feld[] = {0,1,3,5}`
`char name[] =`

- Strings sind arrays vom Typ "char"

<string.h>

- **Länge eines Strings**

```
a = strlen(s1);
```
← ergibt Länge ohne '\0' Byte

- **String kopieren**

```
strcpy(ziel_string, quell_string);
```

- **String an Anderen anhängen**

```
strcat(ziel-string, quell_string);
```

- **Strings vergleichen**

```
int a;
a = strcmp(s1, s2);
```

-1 wenn s1 < s2
0 wenn s1 = s2
1 wenn s1 > s2

Kommandozeilenparameter

```
int main(int argc, char *argv[]);
```

argc: Anzahl der Kommandozeilenparameter + 1

argv[]: die Parameter selbst

Strukturen

Beispiel:

```
// c
struct Adresse
{
    char strasse[20];
    int hausnummer;
    int postleitzahl;
    char stadt[20];
};
```

Variablen können direkt wie definiert werden

```
struct Adresse adresse1, adresse2;
// zwei Adressen des Typs Adresse
struct Adresse adressbuch[30];
// Array mit 30 Adressen
```

Definition mit Initialisierung

```
struct Adresse adresse1 = {
    "Villinger Str.",
    9,
    78054,
    "Schwenningen"
};
```

Mehrdimensionale Arrays

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

[3] = Zeilen, [4] = Spalten

Zeiger und Adressen

&: Referenzieren → liefert Adresse

*: Dereferenzieren → liefert Inhalt

- ermöglichen mehrere Rückgabewerte bei Funktionen

- **Call by value:** standardmäßig werden Argumente als Kopie übergeben

- **Call by Reference:** Argumente können innerhalb einer Funktion verändert werden, so dass die Änderungen auch außerhalb sichtbar sind (Beispiel den Wert von 2 Variablen swappen)

Arrays und Zeiger

- Array-Name ist Zeiger zum ersten Element des Arrays

```
// c
int numbers[4] = {25, 50, 75, 100}
```

Wert des ersten Objekts im Array auf 13 → `*numbers = 13;`

`*(numbers + 1) = 17;`

Wert des zweiten Objekts auf 17

```
printf("%d", *numbers);
//
```

Den Wert des ersten Objekts anzeigen

Zugriff auf Komponentenvariablen

- `adresse1.hausnummer = 7;`
- `strcpy(adresse1.stadt, "Stuttgart");`
- `printf("%5d", adresse1.postleitzahl);`

Zeiger und structs

```
struct Koordinaten punkt = {1.5, 3.0};
struct Koordinaten *pA;
(*pA).x = 7.0;
```

oder `pA->x = 7.0;`

```
struct Koordinaten
{
    float x;
    float y;
};
```