

1.Pointer (Zeiger)

Variablen sind Speicherstellen bestimmter Größe, die durch ihren Datentyp festgelegt sind. In Programmen werden diese Speicherstellen meistens durch den vergebenen Namen angesprochen. Für den Computer (Betriebssystem) gibt es keine Namen für die Speicherstellen, sondern sie werden durchnummeriert, sodass jede Speicherstelle durch eine Zahl charakterisiert ist, die man Adresse nennt. Adressen sind also Zahlen, die bestimmte Stellen im Hauptspeicher (RAM) angegeben (meistens in hexadezimaler Schreibweise). Da Variablen Speicherstellen sind, gehört zu jeder Variable eine Adresse. Wird in einem C-Programm eine Variable definiert, wird eine Speicherzelle bestimmter Größe vereinbart und ihr beim linken die Adresse zugeordnet. Wird die Adresse in einer Variable abgespeichert, dann spricht man von einem Pointer auf diese Variable.

1.1. Operatoren

- &** gibt die Adresse an z.B.: **&zahl**
***** gibt den Inhalt einer Adresse an z.B.: ***pointer**

```
int zahl = 25;
int *pointer;
pointer = &zahl;
*pointer = 12;
printf("%p", pointer); //gibt ABCD aus
printf("%d", *pointer); //gibt 12 aus
```

Name	Adresse	Inhalt
zahl	ABCD	25
pointer	ABCE	ABCD

Name	Adresse	Inhalt
zahl1	3D22	12
zahl2	3D23	15
pointer	3D24	3D23

```
zahl1 //12
zahl2 //15
&zahl //3D22
&zahl2 //3D23
pointer //3D23
*pointer //15
&pointer //3D24
*zahl1 //ERROR
&*pointer //3D23
*&pointer //3D23
*&zahl1 //12
&*zahl1 //ERROR
```

Arrays

Der Arrayname steht für die Adresse des 1. Elements.

z.B.: `array` //entspricht `&array[0]`

Mit `*Arrayname` wird der Inhalt des 1. Elements angegeben.

z.B.: `*array` //entspricht `array[0]`

Arrayname+i steht für Adresse i+1. Element.

z.B.: `array+i` //entspricht `&array[i]`

Mit `*(array+i)` wird der Inhalt des i+1. Elements angegeben.

z.B.: `*(array+i)` //entspricht `array[i]`

zahl1	7F00	13
zahl2	7F01	-8
zahlen	7F02	7F03
[0]	7F03	-2
[1]	7F04	17
[2]	7F05	23
pointer	7F06	7F01

```

zahl1           //13
&zahl1         //7F00
*zahl1         //ERROR
zahlen          //7F03
*zahlen        //-2
&zahlen        //7F02
*(zahlen+1)    //17
*zahlen+1      //-1
zahlen[2]      //23
*&zahl2       //-8
&*zahl2       //ERROR
&*zahlen      //7F03
*pointer       //-8
*(pointer+2)   //-2
**(pointer+1)  //-2
*&(pointer+1) //-2
2*zahl2       //-16
3***(pointer+5) //3*(-8) = 24
*(&zahlen-1)   //-8

```

2. Unterprogramme

Unterprogramme sind Programmteile mit einem eigenen Namen. Der Name der Unterprogramme muss eindeutig sein.

2.1.1 Verwendung

2.1.1.1 Für Programmteile die wiederholt ausgeführt werden

2.1.1.2 Um die Programme übersichtlicher zu gestalten

2.1.2 Reihenfolge

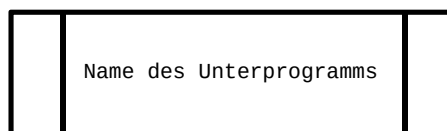
1. Includes + Defines
2. Prototypen
3. Unterprogramme
4. Hauptprogramm

2.1.3 Ausführung

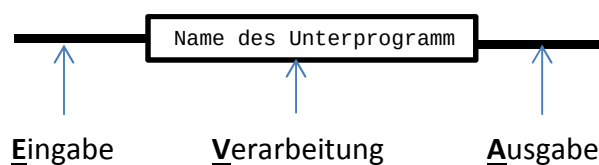
1. Die Programmausführung beginnt mit dem Hauptprogramm.
2. Die Unterprogramme werden erst ausgeführt, wenn sie aufgerufen werden.
Ein Unterprogramm kann auch von einem anderen Unterprogramm ausgeführt werden.

2.1.4 Struktogramm

1. Für jedes Unterprogramm wird ein eigenes Struktogramm gezeichnet.
2. Aufruf des Unterprogramms:



2.1.5 EVA-Diagramm



2.2 Unterprogramm ohne Parameter

Es wird nichts an das Unterprogramm übergeben und es kommt nichts zurück.

```
#include <stdio.h>
void sterne(); //Prototyp

void sterne()
{
    int i;
    for(i=0; i<15; i++)
    {
        printf("*");
    }
    printf("\n");
}

int main()
{
    sterne();        //Aufruf
    sterne();        //Aufruf
    sterne();        //Aufruf
    return 0;
}
```

Datentyp Name Prototyp

Ausgabe

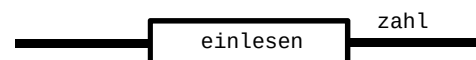
```
*****
*****
*****
```

2.3 Unterprogramm mit Rückgabewert (return)

```
#include <stdio.h>
int einlesen(); //Prototyp

int einlesen()
{
    int zahl;
    printf("Bitte eine Zahl eingeben: ");
    scanf("%d", &zahl);
    return zahl;
}

int main()
{
    int zahl1, zahl2;
    zahl1 = einlesen();
    zahl2 = einlesen();
    printf("Ergebnis = %d\n", zahl1*zahl2);
    return 0;
}
```



Ausgabe

```
Bitte eine Zahl eingeben: 5
Bitte eine Zahl eingeben: 6
Ergebnis = 30
```

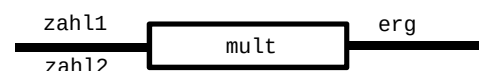
Mit return kann nur 1 Wert zurückgegeben werden.

2.4 Unterprogramm mit einem return und mehreren Eingangsparameter (call-by-value)

```
#include <stdio.h>
float mult(float zahl1, float zahl2); //Prototyp

float mult(float zahl1, float zahl2)
{
    float erg;
    erg = zahl1*zahl2;
    return erg;
}

int main()
{
    float zahl1, zahl2, erg;
    printf("Bitte 2 Zahlen eingeben: ");
    scanf("%f %f", &zahl1, &zahl2);
    erg = mult(zahl1, zahl2);
    printf("Ergebnis = %f\n", erg);
    return 0;
}
```



Ausgabe

```
Bitte 2 Zahlen eingeben: 5.5 6.7
Ergebnis = 36.85
```

Beim Aufruf wird der 1. Wert in die 1. Variable, der 2. Wert in die 2. Variable usw. gespeichert.

Aufgaben:

1. Zierzeilenprogramm, in dem im Hauptprogramm die Anzahl & das Zeichen eingegeben wird.

```
#include <stdio.h>
void zier(char zeichen, int anzahl); //Prototyp

void zier(char zeichen, int anzahl)
{
    int i;
    for(i=0; i<anz; i++)
    {
        printf("%c", zeichen);
    }
    printf("\n");
}

int main()
{
    int anz;
    char zeichen;
    printf("Bitte Zeichen eingeben: ");
    scanf("%c", &zeichen);
    printf("Bitte Anzahl eingeben: ");
    scanf("%d", &anz);
    zier(zeichen, anz);    //auch möglich: zier(" ", 80);
    return 0;
}
```

Ausgabe

```
Bitte Zeichen eingeben: *
Bitte Anzahl eingeben: 7
*****
```

2. Schreiben Sie ein Programm zu Berechnung der Benzinverbrauchs eines Autos.
Im Hauptprogramm werden zuerst folgende Werte eingelesen & überprüft:
 - Alter Kilometerstand
 - Neuer Kilometerstand
 - Liter
 - Preis der Tankfüllung

Ausgabe (Menü):

- 1- Verbrauch pro 100 km
- 2- Benzinpreis
- 3- gefahrene km
- 1. Ende

2.5 Unterprogramm mit mehreren Ausgabeparameter (call by reference)

```
int uhrzeit(int *std, int *min);    //Prototyp
int uhrzeit(int *std, int *min)
{
    int falsch;
    *std = *min/60;
    *min = *min%60;
    falsch = (*std>23)||(*std<0);
    return falsch;
}

int main()
{
    int stunden, minuten, ungueltig;
    printf("Bitte Zeit eingeben: ");
    do {
        scanf("%d", &minuten);
        ungueltig = uhrzeit(&stunden, &minuten);
    } while(ungueltig);
    printf("Uhrzeit: %d:%d\n", stunden, minuten);
    return 0;
}
```

Ausgabe

```
Bitte Zeit eingeben: 76
Uhrzeit: 1:16
```



Aufgaben:

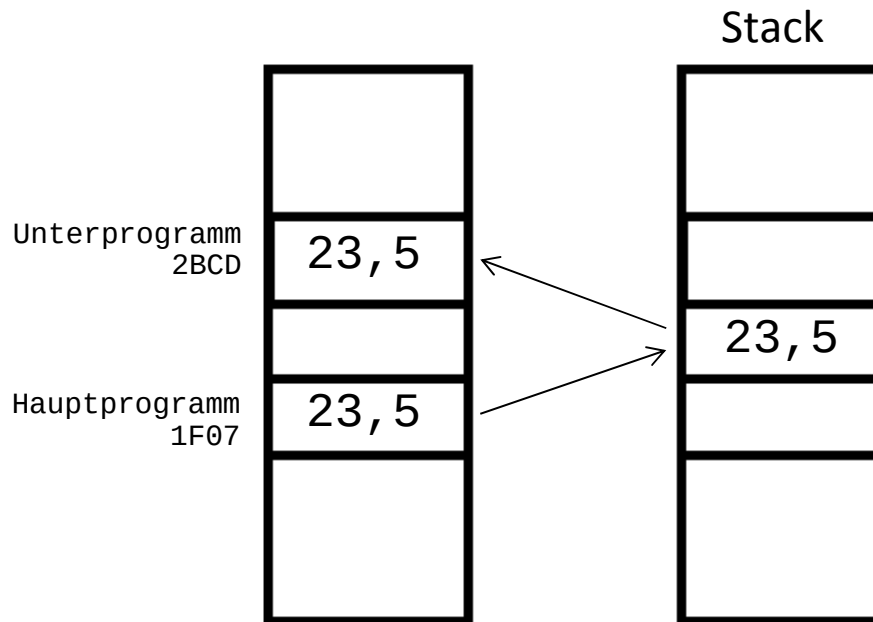
1. Ein Unterprogramm soll aus dem Radius, der im Hauptprogramm eingegeben wurde, den Umfang und die Fläche eines Kreises berechnen und im Hauptprogramm ausgeben.
2. Schreiben Sie ein Unterprogramm, in welches ein Winkel in Radiant übergeben wird und in Grad, Minuten und Sekunden umrechnet und als return den Quadranten zurückgibt.

Normalerweise werden Daten per Call-by-Value übergeben. Call-by-Reference wird nur verwendet, wenn man mehrere Daten zurückliefern muss. Gründe sind, dass das Programm einfacher ist und es weniger Fehler gemacht werden können.

Unterschied zwischen Call-by-Value & Call-by-Reference

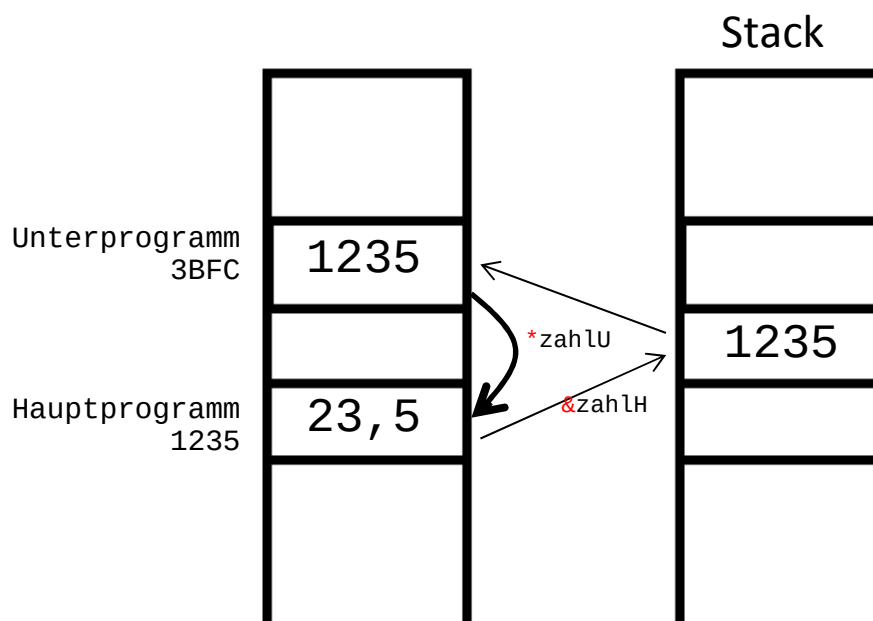
Call-by-Value(Werteparameter):

Variablen werden an Unterprogramme übergeben, aber nicht mehr zurück. Praktisch geschieht das so, dass das aufrufende Programm den Inhalt einer Variable in einen Speicher schreibt, den sogenannten **Stack**, von dem aus das Unterprogramm den Wert holt.



Call-by-Reference (Variablenparameter)

Variablen werden an ein Unterprogramm übergeben und wieder zurück. Hier wird im aufzurufenden Programm über den **Stack** mitgeteilt, auf welcher Adresse die Variable liegt, die übergeben wird. Das Unterprogramm greift also auf dieselbe Variable zu wie das aufrufende Programm. Änderungen der Variablenwertes sind daher auch sofort im aufrufenden Programm erfolgt und nicht erst nach Ende des Unterprogrammes (wie es bei return passiert).



Übergabe von Arrays an Unterprogramme

Arrays werden immer Call-by-Reference übergeben.

```
void einlesen(int *zahlen); //1. Möglichkeit
void einlesen(int zahlen[]); //2. Möglichkeit

int einlesen(int *zahlen)
{
    *zahlen = 15;
    *(zahlen+1) = 17;
    printf("Zahl eingeben: ");
    scanf("%d", zahlen + 2);
}

int main()
{
    int zahlen[3], i;
    einlesen(zahlen);
    for(i=0; i<3; i++)
    {
        printf("%d\n", zahlen[i]);
    }
    return 0;
}
```

Ausgabe

```
Zahl eingeben: 42
15
17
42
```

Aufgabe (Hex-Format-Überprüfung):

Jede Zeile besteht aus mehreren Bytes in Hexadezimalschreibweise, wobei das letzte Byte eine Checksumme ist. Mithilfe der Checksumme kann die Richtigkeit der Zeile überprüft werden (Summe aller Bytes + Checksumme ist vielfaches von 256).

Schreiben Sie ein Programm, das das Intel-Hex-File einliest und jede Zeile auf Richtigkeit überprüft.

Die einfache Checksummenberechnung erkennt nur 1bit-Fehler zu 100%. 2 und mehr-bit-Fehler können sich gegenseitig aufheben.

Statische Variablen

Statische Variablen werden nicht bei jedem Aufruf neu angelegt. Eine Initialisierung einer statischen Variable erfolgt nur beim 1. Mal.

```
void zaehler(); //Prototyp

void zaehler()
{
    static int z = 0;
    z++;
    printf("%d. Aufruf\n", z);
}
```

Parameterübergabe ans Hauptprogramm

```
int main(const int argc, const char * const argv[])
{
    int i = 0;
    for(i=0; i<argc; i++)
    {
        printf("%d. Parameter = %s\n", i+1, argv[i]);
    }
    return 0;
}
```

Test.exe	1.Wert	2.Wert	3.Wert
1. Parameter	2. Parameter	3. Parameter	4. Parameter

In argv[0] steht immer der Programmname

Dynamischer Speicher

```
#include "i:/fsst/malloc.h"
```

a) Speicher anlegen:

```
pointer = (TYP *) malloc(anzahl); //Anzahl der Bytes
```

oder:

```
pointer = (TYP *) calloc(anzahl, b); //Anzahl der Variablen, Größe der Variable
```

b) Überprüfung:

```
if(pointer==NULL)
{
    puts("kein Speicher");
}
else
{
    //Programm
}
```

c) Freigeben

```
free(pointer);
pointer = NULL;
```

d) Größe verändern

```
pointer2 = realloc(pointer, anzahl); //neue Größe in Byte
```

```

int *werte;
werte = (int *) malloc(15*sizeof(int)); //1. Möglichkeit
werte = (int *) calloc(15, sizeof(int)); //2. Möglichkeit
if(werte==NULL)
{
    puts("kein Speicher");
}
else
{
    werte[0] = 25;
    *(werte+1) = 35;
    free(werte);
    werte = NULL;
}

```

Aufgabe:

Es soll ein String mit maximal 80 Zeichen in ein Unterprogramm eingegeben werden, wobei der Speicher in einem anderen Unterprogramm dynamisch reserviert werden soll.

Hauptprogramm:

- Speicher anlegen
- String lesen
- String ausgeben
- Speicher löschen
- Ende

Mehrdimensionale Arrays

```
float zahlen[ZEILE][SPALTE];
```

[Bild mehrdimensionale Arrays]

```
float zahlen2[2][3] = {{1,2,3}, {4,5,6}};
```

```

void einlesen(float z[][SPALTEN]); //1. Möglichkeit
void einlesen(float (*z)[SPALTEN]); //2. Möglichkeit

void einlesen(float z[][SPALTEN])
{
    int i, j;
    for(i=0; i<ZEILEN; i++)
    {
        for(j=0; j<SPALTEN; j++)
            printf("Zahl[%d][%d]: ", i, j);
        scanf("%f", z[i][j]); //auch möglich: &z[i][j]
    }
}

//Aufrufen mit:
einlesen(zahlen1);

```

Dynamisch Mehrdimensionales Array:

```
zahlen = (float (*)[]) malloc(ZEILEN*SPALTEN*sizeof(float));
```

Strukturen

Mehrere logisch zusammenhängende Daten können unter einem gemeinsamen Namen (Typ) behandelt werden.

```
struct Datum
{
    int tag;
    int monat;
    int jahr;
};

struct Datum geb;

geb.tag = 7;
geb.monat = 3;
geb.jahr = 1999;

printf("%d.%d.%d\n", geb.tag, geb.monat, geb.jahr);
```

```
typedef struct
{
    int tag, monat, jahr;
} Datum;

Datum geb;
```

```
struct Datum
{
    int tag, monat, jahr;
}
geb = {7,3,1999}, geb2;
```

```
struct
{
    int tag, monat, jahr;
} geb;
```

Letzte Möglichkeit nur möglich, wenn die Variablen sofort angelegt werden.

Bei einer Definition mit „typedef“ steht der Typ nach der Variable, ohne „typedef“ ein Variablenname.

Die Datentypdefinition erfolgt nach den INCLUDE-Files, aber vor den Prototypen.

```
typedef struct
{
    int tag, monat, jahr;
}
Datum;

typedef struct
{
    char vorname[20];
    char nachname[20];
    Datum geb;
}
Person;

int main()
{
    Person schueler;
    gets(schueler.vorname);
    strcpy(schueler.nachname, "Moetz");
    schueler.geb.tag = 9;
    schueler.geb.monat = 2;
    schueler.geb.jahr = 1997;
    return 0;
}
```