#include <Bibliothek>
<stdio.h> // Input/output functions (e.g., printf, scanf)
<math.h> // Mathematical functions (e.g., sin, sqrt)
<stdbool.h> // Boolean data type (true, false)
<time.h> // Time and date functions
<stdlib.h> // Memory allocation, random numbers, system commands
<stdint.h> // Fixed-width integer types (e.g., int8_t, uint32_t)
<string.h> // String manipulation functions (e.g., strcpy, strlen)

```
printf("Was ich schreiben möchte %Datentyp", variable);
```
//schreibt Text in "..." plus variable im angegebenen Typ

```
scanf("%Datentyp", &variable);
```
// speichert Wert vom vorgegebenen Datentyp in «Variable» (an diese Adresse)

Typ	Beschreibung	Wertebereich üblich	printf	scanf	
int	Ganzzahl	] -2^{31} 2^{31} [	%d	%d	17
unsigned int	Ganzzahl nur positiv	[0 2^{32} [			17u
long	Lange Ganzzahl	] -2^{32} 2^{32} [	%ld	%ld	17L
float	Fliesskommazahlen		%f	%f	17.0f
double	Fließkommazahlen (doppelte Präzision)			%lf	17.0
char	Einzelnes Zeichen		%c	%c	
char* (string)	Zeichenkette		%s	%s	
(void*)	Pointer		%p		
hexadezimal	Ganze Zahl im hexadezimalen Format		%x		0x11
oktal	Ganzzahl Basis 8				022

Erweitern des Datentyps in printf():
%5d Integer, rechtsbündig, 5 Zeichen
%05d Integer, rechtsbündig, 5 Zeichen, mit 0 aufgefüllt
Erweitern des Datentyps in scanf():

```
scanf(" %50[^\n]s", &value);
```

Leerzeichen am Anfang werden ignoriert, eingabe wird mit «Enter» beendet

Typumwandlung:

```
double a = 6.7, b = 2.3, c;
c = a+b// 9
c = (int) a+b // 8.0
```

dezimal	15	20
hexadezimal	0xF	0x14
oktal	017	024

Konstanten initialisieren
innerhalb von Funktionen z:B.: const int MAXIMUM = 1000;
ausserhalb von Funktionen (üblich im Programmkopf): #define MAX = 1000

Operatoren

Operator		Bsp	Assotiativität	Prezedenz
*	Mal		rechts	13
/ (int)	Division Ganzzahlen	1 / 2 → 0		13
/ (double/float)	Div. Gleitkommazahlen	1 / 2.0 → 0.5		13
%	Modulo (Rest bei int-Div.)	5 % 2 → 1		13
+ -	Plus, Minus			12
=	Zuweisung		links	2

Inkrement / Dekrement

			Erster Schritt	2.Schritt
`e = f = g = h = 3`				
`a = e++`	a=3	e=4	Zuweisung $a=e_{alt}$	e++
`b = f--`	b=3	f=2	Zuweisung $b=f_{alt}$	f--
`c = ++g`	c=4	g=4	++g	Zuweisung $c=g_{neu}$
`d = --h`	d=2	h=2	--h	Zuweisung $d=h_{neu}$

Operatoren	Präzedenz	Assoziativität
++ (Inkrement) -- (Dekrement) ~ (bitweise NOT) ! (logisches NOT) + (Vorzeichen) - (Vorzeichen)	15	rechts
[] (Array-Zugriff) () (Funktionsaufruf) . (Direkter Zugriff auf ein Objekt) -> (Pointer-Zugriff auf ein Objekt)	14	links
* (Multiplikation) / (Division) % (Modulo, Rest)	13	links
+ (Addition) - (Subtraktion)	12	links
<< (Bitweise Linksverschiebung) >> (Bitweise Rechtsverschiebung)	11	links
Vergleichsoperatoren < (kleiner) <= (kleiner gleich) > (grösser) >= (grösser gleich)	10	links
== (Gleichheit) != (Ungleichheit)	9	links
& (Bitweise AND)	8	links
^ (Bitweise XOR)	7	links
| (Bitweise OR)	6	links
&& (logisches AND)	5	links
|| (logisches OR)	4	links
?: (Bedingungsoperator)	3	links
= (Zuweisung) += -= *= /= %= <<= >>= &= |= ^= (Kombinierte Zuweisungsoperatoren)	2	rechts

Zusammengesetzte Operatoren

```
a = a+b
a += b
```

Logische Verknüpfungen

A	B	Z = A && B (AND)	Z = A || B (OR)	Z = !A (NOT)
1	1	1	1	0
1	0	0	1	0
0	1	0	1	1
0	0	0	0	1

Entscheidungen

if/else

```
if (bedingung) {
    block_a
} else {
    block_b
}
```

if(a) {...} prüft, ob a ≠ 0 ist

FALSCH!!! if(a=5) {printf(%d,a)} // gibt 5 aus. a wird auf 5 gesetzt, da 5 ≠ 0, wird Block ausgeführt.
Richtig für Test ist if(a==5) {...}

Gleichheit bei Fliesskommazahlen

```
const double EPS = 1E-10; // ein "genügend" kleiner Wert
if (fabs(testWert - zielWert) < EPS)  // testet ob testWert und zielWert genügend nah beieinander liegen.
```

Ternärer Operator

min = «Bedingung» **?** «*wert wenn true*» **:** «*wert wenn false*» ;

Bsp.

```
min = a < b ? a : b ;
```

Das ist in diesem Fall eine Kurzform für:

```
if (a < b ) {
      min = a;
} else {
      min = b;}
```

Mehfachauswahl

```
int variable;
switch (variable) {
case 1: printf("A");
      break;
case 2: printf("B");



case 3: printf("C");
      break;
default: printf("Z");
}
return 0;}
```

Variable =	Output
1	A
2	BC (fall through, kein break bei case 2)
3	C
andres	Z

Aufzähltypen

Ohne Startwert erhält das erste Element den Wert 0

```
enum frucht { Apfel, Birne, Zitrone }; //Apfel=0, Birne=1, Zitrone=2
```

Startwert kann festgelegt werden

```
enum frucht { Apfel=1, Birne, Zitrone }; //Apfel=1, Birne=2, Zitrone=3
```

Werte können einzeln definiert werden: `enum frucht { Apfel=2, Birne=14, Zitrone=25 };`

Funktionen

als Teil von Bibliotheken

selbst definiert:

Beschreibung, Kommentar

Funktionskopf

- Typ der Funktion, d.h. der Typ desRückgabewertes
- Name der Funktion
- Liste der Parameter, d.h. der Übergabewerte

Funktionsrumpf

- Lokale Variablen ❶
- Anweisungsteil = Implementierung

```
// Potenzen berechnen
int potenz (int basis, int exponent) {
int result=1; ❶
for(;exponent; exponent--) {
result = result*basis;}
      return result;}
```

Funktion vom Typ «void» hat keinen Rückgabewert

Funktionsaufruf nach Deklaration

Datentypen müssen übereinstimmen oder konvertiert werden

```
double x; int y;
z = potenz ( (int) x, y);
```

Schleifen

while	do...while	for

`int i=0, x;`
`while(i < x){`
`printf("*");`
`i++;`
`}`
	`int i=0, x;`
	`do{`
	`printf("*");`
	`i++;`
	`}while (i < x);`
		`int i, x;`
		`for (i = 0; i < x; i++) {`
		`printf("*");`
		`}`

wenn die Anzahl nicht bekannt ist (durch eine Bedingung geprüft wird)	wenn der Schleifenrumpf mindestens einmal ausgeführt werden soll	wenn die Anzahl der **Wiederholungen bekannt**

Arrays

```
int   a[10] ; //Platz für 10 int-Werte in Array a
int d[9] = {0}; // «9 mal 0»
int d[3] = {1, 2, 3, 4}; // Kompilierfehler
int d[3] = {1, 2, 3}; // d[0]=1, d[1]=2...
```

Zeiger

Adressoperator & liefert Adresse eines Objekts
Dereferenzierungsoperator * greift auf das Objekt zu, auf das der Zeiger zeigt
Bsp. scanf("%lf",**&**value); //speichert eingegebenes an Adresse von value

```
int a[10];
int *pa;
pa = &a[0];

a[3] = 5;  //setzt das 4. Element auf 5
*(pa+3) = 5;  //gleich wie a[3]=5;
```

Pointer am besten initialisieren:

```
int *pi = NULL;
```

Der void-Zeiger *void** ist zu jedem Datentyp kompatibel

Struct

```
struct Complex {
double real;
double imag;
};
struct Complex x;
x.real = 7.0;
x.imag = 6.4;
```

Auch möglich:

```
struct Complex x = {7.0, 6.4};
```

Typedef

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

Zeiger auf Datenstrukturen

```
Datum geburtstag;
Datum *pgb = &geburtstag; /* Zeiger auf geburtstag */
(*pgb).tag = 20;//setzt tag auf 20
pgb -> tag = 20; //auch möglich
```

Pass by Value, structs

```
int multiply (int x, int y) {
int result=++x*++y;
      return result;}

int a=2, b=3, c;
c = multiply(a,b); //a=2, b=3, c=(3*4)=12
```

a und b werden als Kopie übergeben

Pass by Reference
Zeiger als Parameter

```
void inc(int *val) {
      (*val)++;}
int main(){
      int i = 4;
      inc(&i);
      printf("%d",i);} // 5
```

Zeiger als Rückgabewert

```
Zeit* besser(Zeit* t1, Zeit* t2){
if(t1->sec+60*t1->min+3600*t1->h  <  t2->sec+60*t2->min+3600*t2->h)){
return t1;}
else { return t2;}
}
```

!! Keine Pointer auf lokale Variablen zurückgeben!! Der Speicher wird nach Aufruf wieder freigegeben.

Array Parameter

```
void function(int feld[]) {…}
```

Übergabe von einem Pointer auf `feld[0]`

Strings und Stringfunktionen

String muss um mindestens 1 grösser sein als Array, wegen '\0'

```
char name[4];
name = "Sam"; /* Fehler !!
char neu[4] = "Sam" /* zulässig */
```

Dynamischer Speicher in C

```
int main() {
int *arr; int n = 5;  // Number of elements
// Allocate memory for 5 integers
arr = (int *)calloc(n, sizeof(int));
if (arr == NULL) {
printf("Memory allocation failed!\n");
return 1;}
...
free(arr);  // Free allocated memory
*arr = NULL; //Controlled crash in the event of errors
return 0;
}
```

Bit-Operationen

a = 6 = 0110
b = 3 = 0011

	a b	0110 0011	6 3	
AND	a&b	0010	2	
OR	a|b	0111	7	
XOR	a^b	0101	5	
NOT	~a	1001	9	
	b<<2	1100	12	b<<c=b*2^c
	a	0110		
	a>>2	0001		a>>c=a/2^c

Bits setzen, löschen, abfragen

```
#define BIT2 0x04 // Binär : 00000100
c = c & ~BIT2; // Bit auf 1 setzten
c = c & ~BIT2;// Bit auf 0 setzen
c = c ^ BIT2; //Bit invertieren
if ((c & BIT2) != 0) { …. } //Block wird ausgeführt, wenn Bit 2 = 1
```

2^0	2^1	2^2	2^3	2^4	2^5	2^6	2^7
1	2	4	8	16	32	64	128
0000’0001	0000’0010	0000’0100	0000’1000	0001’0000	0010’0000	0100’0000	1000’0000

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0000	0001	0010	0011	0100	0101	0110	0111	1000	1001	1010	1011	1100	1101	1110	1111
0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F