1 Funktionen

1.1 Leere Fkt Deklaration

Beliebig viele Parameter - nur ein Rückgabewert Eine Funktion kann auch keine Parameter haben In C gekennzeichnet durch das Schlüsselwort void In C++ gekennzeichnet durch eine leere Parameterliste Eine Funktion kann auch keinen Rückgabewert haben Gekennzeichnet durch das Schlüsselwort void Wichtig: Bei Prototypen unterscheidet C zwischen einer leeren Parameterliste und einer Parameterliste mit void . Ist die Parameterliste leer, so bedeutet dies, dass die Funktion eine nicht definierte Anzahl an Parametern besitzt. Das Schlüsselwort void gibt an, dass der Funktion keine Werte übergeben werden dürfen. Beispiel:

```
int foo1();
int foo2(void);

int main(void)
{
    foo1(1, 2, 3); // kein Fehler
    foo2(1, 2, 3); // Fehler
    return 0;
}
```

Während der Compiler beim Aufruf der Funktion foo1 in Zeile 6 keine Fehlermeldung ausgegeben wird, gibt der Compiler beim Aufruf der Funktion foo2 in Zeile 7 eine Fehler- Meldung aus. (Der Compiler wird höchstwahrscheinlich noch zwei weitere Warnungen oder Fehler ausgeben, da wir zwar Prototypen für die Funktionen foo1 und foo2 haben, die Funktion aber nicht definiert haben.)

Diese Aussage gilt übrigens nur für Prototypen: Laut C Standard bedeutet eine leere Liste bei Funktionsdeklarationen die Teil einer Definition sind, dass die Funktion keine Parameter hat. Im Gegensatz dazu bedeutet eine leere Liste in einer Funktionsdeklaration, die nicht Teil einer Definition sind (also Prototypen), dass keine Informationen über die Anzahl oder Typen der Parameter vorliegt - so wie wir das eben am Beispiel der Funktion foo1 gesehen haben.

Noch ein Hinweis für Leser, die ihre C Programme mit einem C++ Compiler compilieren: Bei C++ würde auch im Fall von foo1 eine Fehlermeldung ausgegeben, da dort auch eine leere Parameterliste bedeutet, dass der Funktion keine Parameter übergeben werden können.

1.2 Parameter

Default pass by value

1.3 Prototype

```
int max(int, int);
```

1.4 Definition

```
int max(int a, int b)
{
    ...
}
```

1.5 Static

Static variable bleibt im Memory bestehen und wird zwischen Fkt aufrufen nicht zurück gesetzt wird nur beim ersten mal initialisiert.

2 Floating point

2.1 float

- Schreibweise: `2.0355e2f`
- Size: 4Bytes

The size of the float is 32-bit, out of which:

1. The most significant bit (MSB) is used to store the sign of the number.
2. The next 8 bits are used to store the exponent.
3. The remaining 23 bits are used to store the mantissa.

Converting to Binary form, we get:

$$\begin{aligned} 65 &= 1000001 \\ 0.125 &= 001 \\ \text{So,} & \\ 65.125 &= 1000001.001 \\ &= 1.000001001 \times 10^6 \\ \text{Normalized Mantissa} &= 000001001 \end{aligned}$$

Now, according to the standard,
we will get the baised exponent by adding the exponent to 127,

$$\begin{aligned} &= 127 + 6 = 133 \\ \text{Baised exponent} &= 10000101 \end{aligned}$$

And the signed bit is 0 (positive)

So, the IEEE 754 representation of 65.125 is,

```
0 10000101 00000100100000000000000
```

2.2 double

- Literale mit Punkt und Nachkommastellen: 203.55
- Alternativ mit Zehnerpotenz multipliziert: 2.0355e2 (auch E möglich)

The size of the double is 64-bit, out of which:

1. The most significant bit (MSB) is used to store the sign of the number.
2. The next 11 bits are used to store the exponent.
3. The remaining 52 bits are used to store the mantissa.

From above,

$$65.125 = 1.000001001 \times 10^6$$

$$\text{Normalized Mantissa} = 000001001$$

Now, according to the standard, bais is 1023. So,

$$= 1023 + 6 = 1029$$

$$\text{Baised exponent} = 10000000101$$

And the signed bit is 0 (positive)
So, the IEEE 754 representation of 65.125 is,

```
0 10000000101 00000100100000000000000000000000000000000000000000000000
```

2.3 long double

- Schreibweise: `2.0355e2f`
- Size: 10Bytes

Maximale und minimale Werte in der Datei float.h Ähnlich wie in limits.h für die Ganzzahltypen

3 Casting

```
3/2 // ist 1
3.0/2.0 // ist 1.5

short a, b;
a + b // int
4 / a // int
4 / 2.0 // double
4L / 2.0 // double
4 / 2.0L // long double

int i;
double x;
i = 3 / 2.0; // i ist dann 1 (Zuweisung von 1.5 zu i)
x = 3 / 2; // x ist dann 1.0 (Zuweisung von 1 zu x)

int m=20, n=7, k;
double x;
double pi=3.14;
x = m/n; // 2.0
x = (double)20/7; // 2.8571
x = (double)(20/7); // 2.0
k = m + pi; // 20.0 + 3.14 = 23.14 => k = 23
k = m + (int)pi; // 20 + 3 = 23 => k = 23
```

4 Pointer Array

4.1 Array Initialisierung

```
int a[5] = {4,7,12,77,2}; /* Alle 5 Elemente werden initialisiert */
int b[] = {4,7,12,77,2}; /* Laenge wird impliziert auf 5 gesetzt */
int c[5] = {4,3,88,5}; /* OK, letztes Element erhaelt Wert 0 */
int d[9] = {0}; /* Alle Elemente 0 */
int d[5] = {4,3,88,5,3,6}; /* Kompilierfehler */
```

4.2 Pointer

- `&` ist der Adressoperator
- `&anzahl` bedeutet: die Adresse von anzahl
- `*` ist der Dereferenzierungsoperator
- `*aktuelleAnzahl` bedeutet: das worauf aktuelleAnzahl zeigt

4.3 Void Pointer

Normalerweise bestehen Zeiger aus Adresse und Datentyp beim Dereferenzieren eines int-Zeigers erhält man ein int verschiedene Zeigertypen sind nicht zuweisungskompaktigel Man kann z.B. keinen int-Zeiger einem double-Zeiger zuweisen man kann aber auch Zeiger ohne Datentyp anlegen: void-Zeiger Typkonvertierungen von und zu void-Zeigern sind möglich.

4.4 Null Pointer

- Verweist auf die Adresse 0
- Normalerweise vom Typ `void *`
- Signalisiert, dass ein Zeiger gerade nicht auf einen brauchbaren Wert zeigt

4.5 Write to Adresse

```
int *px;
px = (int *) 0x1000; // Adresse 0x1000
*px = *px | 0x40; // Bit 6 setzen
px = px + 1; // Adresse 0x1004
*px = *px & 0xF7; // Bit 3 zuruecksetzen
```

4.6 ZF

- Arrays sind im Speicher hintereinander liegende Werte eines bestimmten Typs
- Ihr Name kann als konstanter Zeiger auf das erste Element betrachtet werden
- Zeiger sind Variablen, welche Adressen enthalten und einen Typ haben
- Auf das Ziel eines Zeigers greift man zu, indem man den Zeiger dereferenziert (*)
- Die Adresse einer Variablen kann man mit dem Adressoperator ermitteln (&)
- Die Grösse eines Zeigers ist architekturbahngig (häufig 4 oder 8 Bytes)
- Auch Zeiger müssen korrekt initialisiert werden

5 Escape sequences

Sequence	ASCII	Description
\n	10	new line
\r	13	carriage return
\t	09	horizontal tab
\v	11	vertical tab
\f	12	form feed (Curser to start of next page)
\b	08	backspace
\a	07	bell
\'	39	apostrophe
\"	34	quote
\\	92	backslash
\nnn		char value in octal (n = 0..7)
\xhh		char value in hex

6 Format

Sequence	Output
%d or %i	int
%c	character
%e or %E	double in format [-] d.ddd e±dd
%f	double in format [-] ddd.ddd
%ot	int as octal
%s	string
%p	As pointer address
%u	unsigned int
%x pr %X	int as hex
%%	%

`%ni` output as int / flush right / n characters wide
`%0ni` output as int / flush right / n characters wide filled with 0
`%.nf` output as float / n digits after comma
`%+i` output as int / forces plus or minus sign
`%#x` int as hex / forces `0x` before number

7 Priorities

Priority	Symbol	Associativity	Meaning
15	(Postfix) ++, - -	L - R	Increment, Decrement
	(Postfix) - -		Decrement
	( )		Function call
	[ ]		Indexing
14	(Prefix) ++, - -	R - L	Increment, Decrement
	+, - (positiv/negativ number)		positiv/negativ number
	!		logic not
	(Type)		type conversion
	sizeof		
13	*, /, %	L - R	Rechenoperation
12	+, -	L - R	Rechenoperation
10	<	L - R	kleiner
	<=		kleiner gleich
	>		grösser
	>=		grösser gleich
9	==	L - R	gleich
	!=		ungleich
5	&&	L - R	logisches UND
4	||	L - R	logisches ODER
3	?:	R - L	Bedingung
2	=	R - L	Zuweisung
	*=, /=, %=, +=, -=,		Kombinierte Zuweisung
1	,	L - R	Komma-Operator

7.1 Weird operators

`i++` → Value is returned before and then incremented
`++i` → Value is first incremented and then returned