

Programmieren

Sammlung gegliedert nach Modul

Fabian Suter, 13. Juni 2024

<https://github.com/FabianSuter/Programmieren.git>

1 ProgC

1.1 Wichtige Kurzbefehle

cd "Path"	Pfad anwählen
cd ..	um eine Ebene nach oben (zurück)
mkdir "Ordnername"	Ordner erstellen
rmkdir "Ordnername"	Ordner löschen
rm -rf *	Alles innerhalb vom aktuellen Ordner löschen
rm "Datei"	Datei löschen
mv "Name alt" "Name neu"	Datei umbenennen
cp "Datei alt" "Datei neu"	Datei kopieren und benennen
clang -Wall -o "Outputname" "Inputdatei"	clang-Compiler mit Warnungen
clang -Wall -o "Outputname" "Inputdatei" -lm	-lm für Mathebibliothek (<math.h> für sqrt)
ls	Listet alle Files im akt. Verzeichnis auf
ls -l	Inkl. Informationen wie Grösse u.a.
ls -a	Inkl. versteckten Dateien
ls -al	Beide Varianten

1.2 Zahlensysteme

$2^0 = 1$	$2^1 = 2$	$2^2 = 4$	$2^3 = 8$	$2^4 = 16$	$2^5 = 32$	$2^6 = 64$	$2^7 = 128$
-----------	-----------	-----------	-----------	------------	------------	------------	-------------

Grösse	Abk.	Genauer Wert	Näherung
Kilobyte	kB	$2^{10} = 1024$ Bytes	10^3 Bytes
Megabyte	MB	$2^{20} = 1\,048\,576$ Bytes	10^6 Bytes
Gigabyte	GB	$2^{30} = 1\,073\,741\,824$ Bytes	10^9 Bytes
Terabyte	TB	$2^{40} = 1\,099\,511\,627\,776$ Bytes	10^{12} Bytes

Oktal	3 Bits	X_8	X_O	X_q	X_{oct}	$0X$
Hex	4 Bits	X_{16}	X_h	X_H	X_{hex}	$0xX$

Hexadezimal

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F

ASCII (7-Bit) American Standard Code for Information Interchange

Ordnet gängigen Schriftzeichen einen Zahlenwert zu, um diese in einem Digitalrechner präsentieren zu können. Die Tabelle ist wichtig, um für geg. Schriftzeichen den in der Maschine repräsentierten Zahlenwert zu ermitteln (und umgekehrt).

Nachfolger: Unicode (8-, 16-, 32-Bit)

1.3 Datentypen

1.3.1 Datentypen

Typ	Anz. Bytes	Bereich	printf	Spezielles
Ganze Zahlen				
byte	min. 1	0 ... +255		
short	min. 2	$-2^{15} \dots +2^{15} - 1$	%d; %i	Hex: %x; %X
int	min. 4	$-2^{31} \dots +2^{31} - 1$	%d	Hex: %x; %X
long	min. 8	$-2^{63} \dots +2^{63} - 1$	%ld; %li	Hex: %x; %X
Dezimalzahlen			(Expon.: %e)	
float	4	$1.2E - 38 \dots 3.4E + 38$	%f	6 Dez.stellen
double	8	$2.3E - 308 \dots 1.7E + 308$	%lf	15 Dez.stellen
Spezial				
char	1	Einzelne Buchstaben	%c	
boolean	1	True / False		
string		Zeichenkette; Text	%s	
Vorzeichen, Versch.				
unsigned char	1	0 ... +255	%c	
signed char	1	-128 ... +127	%c	
unsigned int	4	0 ... + $2^{32} - 1$	%u	
short int	2	$-2^{15} \dots +2^{15} - 1$	%hd	
unsigned short int	2	0 ... + $2^{16} - 1$	%hu	
long int	4	$-2^{31} \dots +2^{31} - 1$	%ld	
unsigned long int	4	0 ... + $2^{32} - 1$	%lu	
long long int	8	$-2^{63} \dots +2^{63} - 1$	%lld	
unsigned long long int	8	0 ... + $2^{64} - 1$	%llu	
long double	16	$3.3E - 4932 \dots 1.1E + 4932$	%Lf	18 Dez.stellen

Ganzzahlen können überlaufen!

Gleitpunktzahlen haben meist Rundungsfehler. Nie auf Gleichheit prüfen!

Wertebereich:

- unsigned $0 \dots (2^n - 1)$ $n=8 : 0 \dots 255$
- signed $-2^{n-1} \dots + (2^{n-1} - 1)$ $n=8 : -128 \dots +127$

Stellenbreite: %7.2f → `uuuu.uu`, %4d → `uuuu` (res. immer 4 Zahlenbreiten)

1.3.2 Typumwandlung

float f = 41.7;

Implizit: Eine Kommazahl ohne f am Ende hat den Typ double

int x = (int) f;

Explizit: x hat den Wert 41, Nachkommastellen werden abgeschnitten

1.3.3 Namen

- Buchstaben a-z, A-Z
- Ziffern 0-9
- Underscore
- alpha ≠ Alpha

Nicht als Namen erlaubt: die reservierten Schlüsselwörter

Im C90-Standard sind 32 reservierte Schlüsselwörter definiert. Sie sind stets klein geschrieben und dürfen nicht als Namen (z.B. für Variablen) verwendet werden.

auto	double	int	struct
break	else	long	switch
case	enum	register	typedef
char	extern	return	union
const	float	short	unsigned
continue	for	signed	void
default	goto	sizeof	volatile
do	if	static	while

Im C11-Standard sind die folgenden Schlüsselwörter dazugekommen:

inline	_Alignof	_Complex	_Noreturn
restrict	_Atomic	_Generic	_Static_assert
_Alignas	_Bool	_Imaginary	_Thread_local

1.3.4 Konstanten

Literale Konstanten:

- Ganzzahlige Konstanten (default: int)

254 035 0x3f -34 14L 14U 14UL
dezimal okt hex long unsigned unsigned long

- Zeichenkonstanten

'c' '\n' '\x4a' '\14' '\\ ' 'L'a'
ASCII hex ASCII okt Double Byte

- Gleitpunktkonstanten (default: double)

254.89 -13.0 3.45e23 4.65f 3.14159L
Exp-Schreibweise Float-Konstante long double

Symbolische Konstanten:

```
//Mit #define
#define PI (3.14159)

//Mit enum
enum{
    listLength = 40;
    commLength = 30;
    dateLength = 20;
}
```

1.4 Variablen

	Lokale Variable	Globale Variable
Sichtbarkeit	Zwischen Definition und Ende des aktuellen Blocks	Zwischen Definition und Ende der aktuellen Compile-Unit; über Deklaration extern auch in anderen Compile-Units importierbar
Lebensdauer	Laufzeit des zugehörigen Funktionsaufrufs	Laufzeit des Programms
Automatische Initialisierung	keine	automatische Initialisierung mit Wert 0

1.5 Operatoren und Operanden

- unär (monadisch): hat einen einzigen Operator
 - Inkremental ++
 - De-, Referenzieren (&, *)
 - !wahrheitswert (Negation)
- binär (dyadisch): hat zwei Operanden
 - z.B. 3+4 od. a+b
- ternär (triadisch): hat drei Operanden
 - Mini-If: wahrheitswert?wert1:wert2
(wert1 für wahr, wert2 für falsch)
(z.B. x?"wahr":"unwahr")

1.5.1 Modulo

% gibt den Restwert einer Rechnung aus

10 % 3 = 1, 20 % 7 = 6

1.5.2 Priorität & Assoziativität

Priorität	Operatoren	Assoziativität
Priorität 1	() [] -> ++ -- (typename) {} ! ~ -- sizeof + - (typename) * &	Funktionsaufruf Array-Index Komponentenzugriff Inkrement, Dekrement als Postfix compound literal ⁹⁸
Priorität 2	* / % + - << >> < <= > >=	Negation (logisch, bitweise) Inkrement, Dekrement als Präfix Vorzeichen (unär) cast Dereferenzierung, Adresse Multiplikation, Division modulo
Priorität 3	== != & ^ && ?: += -= *= /= %= &= & = >>=	Summe, Differenz (binär) bitweises Schieben Vergleich kleiner, kleiner gleich Vergleich größer, größer gleich Gleichheit, Ungleichheit bitweises UND bitweises Exklusives-ODER bitweises ODER logisches UND logisches ODER bedingte Auswertung einfache Wertzuweisung kombinierte Zuweisungsoperatoren
Priorität 15	,	Komma-Operator

Assoziativität: Reihenfolge der Operationen bei gleicher Priorität

1.5.3 Inkrementieren & Dekrementieren

Postinkrement:

```
printf("%d", i++)
//i wird zuerst geschrieben, anschliessend inkrementiert
```

Präinkrement:

```
printf("%d", ++i)
//i wird zuerst inkrementiert, anschliessend geschrieben
```

1.5.4 Logisch vs. bitweise Operatoren

|| = OR
&& = AND

Logisch:

```
signed char a = 0; //bedeutet unwahr
signed char b = -27; //bedeutet wahr
if(a&&b){
    printf("A-und-B-sind-wahr");
}
```

Bitweise:

```
unsigned char a = 128; // 1000'0000
unsigned char b = 16;  // 0001'0000
printf("%d\n", a | b); // 1001'0000
```

1.6 Schleifen

1.6.1 For-Schleife

```
for (Ausdruck_init; solange Ausdruck; Ausdruck_update)
    Anweisung
```

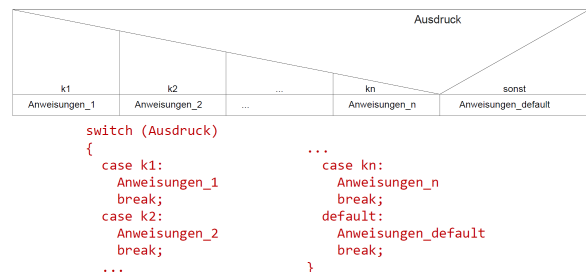
Ausdruck_init
solange Ausdruck
Anweisung
Ausdruck_update

entspricht

```
Ausdruck_init;
while (solange Ausdruck)
{
    Anweisung
    Ausdruck_update
}
```

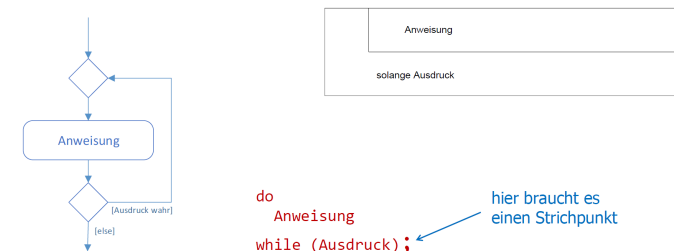
Für Zählschleifen, bzw. wenn die Anzahl Durchläufe bekannt ist

1.6.2 Switch-Schleife



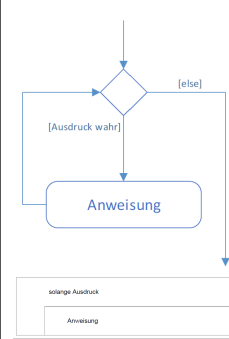
Für Anwendungen, wo eine Überprüfung mehrere Cases haben kann, z.B. Abfrage eines Zustandes (Standby, Off, Idle, Run, ...)

1.6.3 Do-While-Schleife



Keine Zählschleife, das Programm führt min. 1 Durchlauf aus und arbeitet solange Ausdruck = true

1.6.4 While-Schleife



Für alle anderen Fälle. Spezialfall: `while(1)` ist ein konstanter Loop im Programm

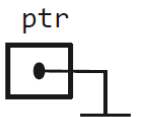
1.6.5 Sprunganweisungen

- **break:** Schleifen abbrechen, zurückhaltend einsetzen!
- **continue:** nächsten Schleifendurchgang starten, sehr zurückhaltend einsetzen!
- **return:** aus Funktion zum Aufruf springen
- **goto:** zu einer Marke springen, VERMEIDEN!

1.7 Pointer

1.7.1 Nullpointer

```
int* ptr = NULL;
```



1.7.2 Ref- und Dereferenzieren

Referenzieren

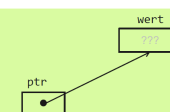
• `int wert;`



• `int* ptr;`



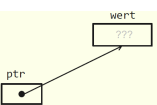
• `ptr = &wert;`



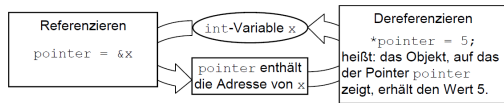
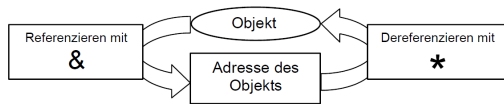
`&` verknüpft den Pointer mit einer Variable

Dereferenzieren

```
int wert;
int* ptr;
ptr = &wert;
```



`*` liefert den Inhalt der Speicherzelle der Adr.



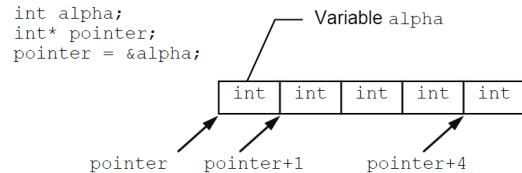
1.7.3 Zuweisungen

```
int    a;
double d;
int*   pi = &a;
int*   pj;
double* pd = &d;
void*  pv;

pv = pd;    //erlaubt, da pv void-Pointer
pj = pi;    //erlaubt, gleicher Typ
pd = pi;    //nicht erlaubt, untersch. Typen
pi = pv;    //erlaubt, da pv void-Pointer
pd = (double*)pj; //erlaubt da Type-Cast
```

1.7.4 Addition, Subtraktion

Von einem Pointer können ganze Zahlen addiert oder subtrahiert werden. Der Pointer **ptr** bewegt sich bei **ptr+n** immer um **n * sizeof(Typ)** Bytes.



Weitere Optionen:

Pointer funktionieren auch mit anderen Operatoren.

Vergleiche mit ==, !=, <, >, >=, etc. funktionieren bei Pointern desselben Typs.

1.8 Arrays

Arrays arbeiten mit Array-Index. In C beginnt dieser bei 0 und endet bei $n - 1$:

```
int alpha[5]; // Array "alpha" mit 5 El. vom Typ int
alpha[0] = 14; // 1. Element (Index 0) = 14
alpha[4] = 3;  // letztes Element (Index 4)
alpha[5] = 4;  // Bereichsueberschr.! -> undefined behaviour
```

Memorymap:

14	???	???	???	3
0	1	2	3	4
alpha				

1.8.1 Initialisierungsvarianten

```
int a1[5] = {0, 8, 5, 1, 2};
int a2[5] = {1, 8};           //Index 1 bis 3 sind auf "0"
int a3[5] = {};               //Alle Elemente sind auf "0"
int a4[] = {12, 3, 2};        //Groesse anhand der Anz. Elemente => "[3]"
```

1.8.2 Grösse eines Arrays

sizeof() liefert bei Arrays die Grösse in **Bytes**

Zur Bestimmung der Anzahl Elemente kann die Grösse des Arrays durch die Grösse eines Wertes geteilt werden:

```
int main() {
    int arr[] = {4, 3, 2, 1};
    printf("arr hat %lu Elemente\n", sizeof(arr)/sizeof(arr[0]));
    return 0;
}
```

1.8.3 Mehrdimensionale Arrays

Arrays können auch als Matrizen verwendet werden, wobei der erste Wert der Zeilenindex und der zweite der Spaltenindex ist.

```
int alpha[3][4] = {
    {1, 3, 5, 7},
    {2, 4, 6, 8},
    {3, 5, 7, 9}
};
```

1	3	5	7
2	4	6	8
3	5	7	9

// äquivalent dazu ist die folgende Definition:

```
int alpha[3][4] = {1, 3, 5, 7, 2, 4, 6, 8, 3, 5, 7, 9};
```

1.8.4 char - Arrays

Ein String in C ist immer ein Array von Zeichen. (**char** - Array).

Ein String in C muss **immer** mit **\0** abgeschlossen werden und braucht eine Stelle des Arrays!

// Folgende Varianten sind gleichwertig:

```
char name[15] = {1, 2, 3, 4, 5, 0};
char name[15] = {'M', 'e', 'i', 'e', 'r', '\0'};
char name[15] = "Meier";
```

1.8.5 Array mit Schleife durchlaufen (Bsp.)

```
enum{groesse = 5};
int alpha[groesse];
```

```
for(int i = 0; i < groesse; ++i)
    printf("%d\n", alpha[i]) // keine "{}", da nur eine Zeile
```

1.8.6 Weitere Array-Regeln

- Ein Array als Ganzes kann keine Werte annehmen, nur einzelne Elemente
- Die üblichen Operatoren können nicht auf Arrays angewendet werden
- Funktionen in C können **keine** Arrays als Aufrufparameter haben!
- Wird bei einem Funktionsaufruf ein Array als Parameter übergeben, wird das Array *implizit zu einem Pointer auf das Element an Index 0 konvertiert*
- Der Name des Arrays kann als *konst. Adresse* von Index 0 des Arrays verwendet werden:
`alpha[i] == *(alpha + i)`
Achtung!

- Der Pointer **ptr** bewegt sich bei **ptr+n** immer um **n * sizeof(Typ)** Bytes!
- Wenn der Pointer über den Bereich hinauszeigt, ist das zwar legal, das Resultat ist aber undefiniert.

- Zuweisung eines Arrays auf einen Pointer:

```
int* ip;
int ia[5] = {-3, 6, 9, 2, 5};
ip = ia; // ein Array wird implizit zu einem Pointer des Basis-Typs gewandelt
ip = &ia[0]; // tut dasselbe
ip = &ia; // Achtung: tut nicht dasselbe! Führt zu einem Compiler-Fehler!
```

- Benutzung eines Pointers im Array-Stil:

```
int* ip;
int ia[5] = {-3, 6, 9, 2, 5};
ip = ia; // ein Array wird implizit zu einem Pointer des Basis-Typs gewandelt
ia[3] = 100; // Zuweisung des Wertes 100 auf Element an Indexposition 3
ip[4] = 200; // Zuweisung des Wertes 200 auf Element an Indexposition 4 über Pointer
*(ip+4) = 200; // tut dasselbe, Schreibweise mit Dereferenzierungsoperator
```

1.9 Structs

Arrays enthalten mehrere Elemente desselben Datentyps. Structs können im Gegensatz auch **unterschiedliche** Datentypen enthalten.

```
#include <stdio.h>
```

```
struct Angestellter
{
    int personalnummer;
    char name[20];
    char vorname[20];
    char strasse[20];
    int hausnummer;
    int postleitzahl;
    char wohnort[20];
    float gehalt;
};
```

Typdeklaration

```
int main()
{
    struct Angestellter a1 = {20202175, "Geiger", "Stefan", "Seestrasse", 12, 8640, "Rapperswil", 100000.0};
    printf("Vorname: %s, Nachname: %s, Gehalt: %d\n", a1.vorname, a1.nachname, a1.gehalt);

    struct Angestellter* a1Ptr = &a1;
    printf("Vorname: %s, Nachname: %s, Gehalt: %d\n", (*a1Ptr).a1.vorname, (*a1Ptr).a1.nachname, (*a1Ptr).a1.gehalt);
    return 0;
}
```

Variablendefinition mit Initialisierung

Zugriff auf einzelne Elemente

Zugriff auf einzelne Elemente bei Pointer auf struct
(Die Variante mit Pfeiloperator ist zu bevorzugen!)

1.10 Strings und Speicher

Für alle Funktionen in diesem Kapitel: `#include <string.h>`

Vorsicht: Jeder Charakter im String benötigt je ein Byte, die `'\0'`-Terminierung ein zusätzliches Byte

1.10.1 Strings kopieren

```
char* strcpy(char* dest, const char* src);
```

```
char* strncpy(char* dest, const char* src, size_t n);
```

String copy

- kopiert von **src** nach **dest**, inklusive `'\0'`
(bei `strncpy()` maximal **n** chars)
- return: **dest**
- **dest** muss bei `strcpy()` auf einen genügend grossen Bereich zeigen
(Ansonsten werden Speicherbereiche nach **dest** überschrieben)

1.10.2 Strings zusammenfügen

```
char* strcat(char* dest, const char* src);
```

```
char* strncat(char* dest, const char* src, size_t n);
```

String concatenate

- hängt von **src** nach **dest** an, inklusive `'\0'`
(bei `strncat()` maximal **n** chars)
Das ursprüngliche `'\0'` von **dest** wird überschrieben
- return: **dest**
- **dest** muss bei `strcat()` auf einen genügend grossen Bereich zeigen
(Ansonsten werden Speicherbereiche nach **dest** überschrieben)

1.10.3 Strings vergleichen

```
char* strcmp(const char* s1, const char* s2);
```

```
char* strncmp(const char* s1, const char* s2, size_t n);
```

String compare

- vergleicht die beiden Strings, auf die **s1** und **s2** zeigen,
bei `strncmp()` nur die ersten **n** char's
- return:
 - `< 0` : **s1** ist lexikographisch kleiner als **s2**
 - `== 0` : **s1** und **s2** sind gleich
 - `> 0` : **s1** ist lexikographisch grösser als **s2**

1.10.4 Stringlänge bestimmen

```
size_t strlen(const char* s);
```

String length

- bestimmt die Länge des Strings s, d.h. die Anzahl char's. '\0' wird nicht mitgezählt
- return: Länge des Strings

1.10.5 Speicher bearbeiten

- Aufrufparams sind vom Typ void* statt char*
- Die mem-Funktionen arbeiten byteweise
- Das '\0'-Zeichen wird nicht speziell behandelt wie bei den str-Funktionen
- Die Bufferlänge muss als Parameter übergeben werden

```
//Speicherbereich kopieren (ohne Ueberlappung!)
void* memcpy(void* dest, const void* src, size_t n);

//Speicherbereich verschieben
void* memmove(void* dest, const void* src, size_t n);

//Speicherbereiche vergleichen
int memcmp(const void* s1, const void* s2, size_t n);

//Erstes Auftreten von Zeichen c in Bereich s suchen
void* memchr(const void* s, int c, size_t n);

//Speicherbereich mit Wert belegen
void* memset(void* s, int c, size_t n);
```

1.11 Static

static-Funktionen

- static-Funktionen sind nur in der Compile-Unit, in welcher sie definiert sind, sichtbar
- Alle Funktionen, welche von aussen nicht sichtbar sein müssen, sollten deshalb als static definiert werden
- Ein versuchter Zugriff auf statische Elemente von einer weiteren Datei ergibt einen Linkerfehler
- Alle Funktionen static definieren, welche keine Schnittstelle nach aussen bilden!

static-Variablen

- Globale Variablen
 - Analog zu den Funktionen: nur am Definitionsort gültig
- Lokale Variablen
 - **Achtung: Komplette andere Bedeutung desselben Schlüsselwortes!**
 - Lokale static-Variablen haben eine Lebensdauer wie eine globale Variable. Dadurch bleibt der Wert der lokalen Variable erhalten, auch wenn die Funktion verlassen wird. (→ Gleiches Verhalten wie globale Variable, abgesehen von der Sichtbarkeit)
 - Werden automatisch mit 0 initialisiert
 - → Kompromiss zwischen globaler und lokaler Variable

1.12 Iterativ vs. Rekursiv

Iterativ:

$0! = 1$

$n! = 1 \cdot 2 \cdot \dots \cdot (n-1) \cdot n$

Rekursiv:

$0! = 1$

$n! = (n-1)! \cdot n$

```
unsigned long
faku(unsigned int n){
    unsigned long fak = 1UL;
    for(unsigned int i = 2; i <= n; ++i)
        fak = fak * i;
    return fak;
}
```

```
unsigned long
faku(unsigned int n){
    if(n > 1)
        return n * faku(n-1);
    else
        return 1UL;
}
```

1.13 Präprozessor

#define ALT NEU	Ersetzt ALT durch NEU
#include "Datei"	Fügt Inhalt aus Datei an die aktuelle Position
#include <Datei>	Gleich wie oben
#ifdef Marke ... #endif	Prüft, ob Marke definiert ist
#ifndef Marke ... #endif	Prüft, ob Marke nicht definiert ist
#error(Nachricht)	Bricht den Compile-Vorgang ab

1.13.1 Beispiel

```
#include <stdio.h>

#ifndef __cplusplus
#error Ich brauche keinen C++-Compiler
#endif

#ifdef __APPLE_
#error Ich mag keine Äpfel
#endif

#ifdef __WIND32__
#error Ich mag kein Windows
#endif

int main(){
    printf("Ich bin sehr wählerisch\n");
    return 0;
}
```

1.14 Trivia

1.14.1 Funktionsprototypen

Funktionsprototypen legen die Schnittstelle einer Funktion fest. Sie sind zwingend erforderlich, wenn Funktionen vor ihrer Definition aufgerufen werden.

1.14.2 Call-by-Reference vs. Call-by-Value

Call-by-Reference:	Call-by-Value:
An eine zu rufende Funktion werden Pointer auf Variablen übergeben, an denen Werte stehen können <pre>int s1 = 3; int s2 = 7; int summe; add(&s1, &s2, &summe);</pre>	An eine zu rufende Funktion werden Werte übergeben <pre>int summe = add(3,7);</pre>

1.14.3 C-Compiler

Der C-Compiler überprüft den Quelltext auf syntaktische Korrektheit.
Zusätzlich übersetzt er den Quelltext in Maschinencode für eine bestimmte Zielplattform.

1.14.4 Gleitpunktzahlen

Eine Gleitpunktzahl ist ein Datentyp in Exponentialdarstellung, bestehend aus Vorzeichen, Mantisse und Exponent. Diese drei Bestandteile werden im Computer separat im Speicher binär dargestellt, gem. Standard IEEE 754. In C gibt es hierfür die Datentypen `float` und `double`.

Vorteil: sehr grosser Wertebereich von $-\infty$ bis ∞ , ± 0 , NaN (Not a number)

Nachteil: Rundungsfehler treten zwangsläufig auf und sind sehr schwer abschätzbar. Sie sollten daher nie auf Gleichheit geprüft werden!

1.15 Code-Snippets

1.15.1 Array und Pointer 1

```
#include <stdio.h>  
  
int main(){  
    enum{array_size = 6};  
    int test[array_size] = {1,2,3,4,5,6};  
    for(int i =0; i<array_size; ++i)  
        printf("Element %u: %i\n", i, test[i]);  
  
    printf("Groesster: %d", *findAbsMax(test, array_size));  
    return 0;  
}
```

Main-Funktion zum Finden eines **betragsmässig** grössten Wertes innerhalb eines Arrays.

```
int* findAbsMax(int* arr, size_t size){  
    int* max_ptr = &arr[0];  
    for(size_t i = 0; i < size; ++i){  
        if((arr[i] >=0 && *max_ptr >=0 && arr[i] > *max_ptr)  
        || (arr[i] <=0 && *max_ptr <=0 && arr[i] < *max_ptr)  
        || (arr[i] >=0 && *max_ptr <=0 && arr[i] > *max_ptr * -1)  
        || (arr[i] <=0 && *max_ptr >=0 && arr[i] * -1 > *max_ptr))  
            max_ptr = &arr[i];  
    }  
    return max_ptr;  
}
```

1.15.2 Array und Pointer 2

Programm liest Wert um Wert ein und gibt sie wieder zurück. `init` in Pointer-Schreibweise, `ausgabe` in Array-Schreibweise.

```
#include <stdio.h>  
enum{groesse = 3};  
  
void init(int* alpha, int dim){ //alpha in Pointer-Schreibweise  
    for(int i = 0; i < dim, ++i){  
        printf("Eingabe-Wert mit Index-%d von arr:", i);  
        scanf("%d", alpha++);  
    }  
}  
  
void ausgabe(const int alpha[], int dim){ //alpha in Array-Schreibweise  
    for(int i = 0; i < dim; ++i)  
        printf("arr[%d]-hat-Wert: %d\n", i, alpha[i])  
}  
  
int main(void){  
    int arr[groesse];  
    init(arr, sizeof(arr)/sizeof(arr[0]));  
    ausgabe(arr, sizeof(arr)/sizeof(arr[0]));  
    return 0;  
}
```

1.15.3 Bitweise Zahl ausgeben

Funktion gibt die Zahl bitweise aus, beginnend mit MSB
In diesem Fall 1000'0000

```
unsigned char x = 128;  
for(int i = 0; i < 8; i++){  
    int bitValue = 1 & x;  
    printf("%d", bitValue);  
    x = x >> 1;  
}  
return 0;
```

1.15.4 Drehmoment berechnen

Programm:

```
#include<stdio.h>  
int main(){  
    float kraft;  
    float abstand;  
    printf("Kraft F in N: ");  
    scanf("%f", &kraft);  
    printf("Abstand s in m: ");  
    scanf("%f", &abstand);  
    if(abstand < 0.0f){  
        printf("Fehler: Abstand ist negativ!");  
        return -1;  
    }  
    else{
```

```

    printf("Das Drehmoment ist: %f-Nm\n", kraft*abstand);
    return 0;
}

```

Ausgabe:

Kraft F in N: -17
 Abstand s in m: 0.5
 Das Drehmoment ist: -8.500 Nm

1.15.5 Pointer & Memorymaps

```

#include <stdio.h>
void geheim(double* p, double z){
    z = 2.0 * z;
    *p = z;
}

int main(void){
    double u = 5.0;
    double v = 7.0;
    geheim(&u, v);
    printf("u=%f, v=%f\n", u, v);
    return 0;
}

```

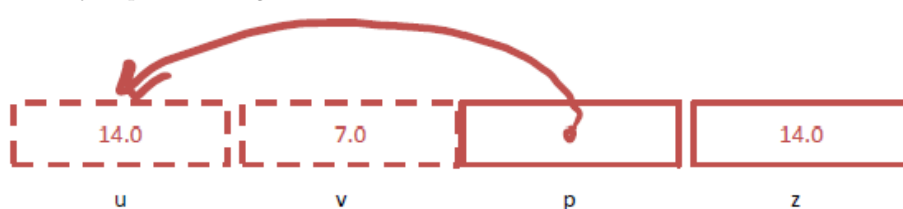
Memorymap kurz vor *geheim*-Aufruf



Memorymap kurz nach *geheim*-Aufruf



Memorymap kurz vor *geheim*-Verlassen



Memorymap kurz nach *geheim*-Verlassen



Ausgabe des Programms:

u=14.0,v=7.0<newline>

1.15.6 Grösster gemeinsamer Teiler

```

typedef unsigned int uint;
uint func(uint m, uint n){
    uint r;
    uint h;
    do{
        if(m<n){
            h=m;
            m=n;
            n=h;
        }
        r=m%n;
        if(r!=0){
            m=n;
            n=r;
        }
    }while(r!=0);
    return n;
}

```

