Informatik 1, Sprache: C

Stefan Küttel

Version: April 10, 2026

1. Zahlensysteme und Ganzzahlliterale

Umrechnungshilfe Binär / Dezimal / Hexadezimal

1) Grundlagen (Stellenwertsystem) Jede Ziffer hat einen **Stellenwert** als Potenz der Basis.

- Dezimal (Basis 10): Stellenwerte $\ldots, 10^3, 10^2, 10^1, 10^0$
- Binär (Basis 2): Stellenwerte $\ldots, 2^3, 2^2, 2^1, 2^0$
- Hex (Basis 16): Stellenwerte $\ldots, 16^3, 16^2, 16^1, 16^0$

Merksatz: *Dezimal: **10er-Potenzen**, Binär: **2er-Potenzen**, Hex: **16er-Potenzen**.*

2) Binär → Dezimal (Summe der gesetzten Bits)

$$(101101)_2 = 1 \cdot 2^5 + 0 \cdot 2^4 + 1 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0$$

$$= 32 + 0 + 8 + 4 + 0 + 1 = 45_{10}$$

3) Dezimal → Binär (Division durch 2, Reste von unten nach oben)

Beispiel: 46_{10}

$$\begin{array}{r|l} 46 : 2 = 23 & \text{Rest } 0 \\ 23 : 2 = 11 & \text{Rest } 1 \\ 11 : 2 = 5 & \text{Rest } 1 \\ 5 : 2 = 2 & \text{Rest } 1 \\ 2 : 2 = 1 & \text{Rest } 0 \\ 1 : 2 = 0 & \text{Rest } 1 \end{array} \quad \Rightarrow \quad (46)_{10} = (101110)_2$$

Merksatz: *Bei Dez → Bin: Reste **von unten nach oben** lesen.*

4) Binär ↔ Hex (immer in 4-Bit-Gruppen)

Regel: Hex-Ziffer ↔ genau 4 Bits (Nibble).

$$(11010110)_2 = (1101\ 0110)_2 = (D6)_{16}$$

$$(D6)_{16} = D \cdot 16^1 + 6 \cdot 16^0 = 13 \cdot 16 + 6 = 214_{10}$$

Merksatz: *Binär ↔ Hex geht **immer** über 4er-Blöcke.*

Hex-Ziffern: $A = 10,\ B = 11,\ C = 12,\ D = 13,\ E = 14,\ F = 15$

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

Typische Prüfungsabkürzung (Hex → Bin → Dez): *Hex nach Bin (4er-Blöcke), dann Bin nach Dez (Bitsummen).*

Tabelle: Variablendefinitionen

Kategorie	Beispiel	OK?

Gut (klar)	`int a = 1;`
	`int b = 2, c = 3;` (geht, aber oft unübersichtlich)
	`double radius = 2.5;`
	`unsigned int flags = 0xFF;`
		Ja

Gut (konstant)	`const int N = 10;` `const double PI = 3.14159;`	Ja

Funktioniert, aber schlechter Stil (unklar)	`int a = 0;`
	`int x1x2 = 7;`
	`int l = 1;` (leicht mit 1 verwechselbar)
		Ja

Funktioniert, aber gefährlich / verwirrend	`int O = 0;` (O vs 0) `int I = 1;` (I vs l vs 1) `int temp = 0; temp = temp++;` (undef./unspez. Verhalten je nach Kontext)	Ja*

Compilerfehler (ungültiger Name)	`int 2x = 5;`
	`int my-var = 3;`
	`int float = 1;` (Keyword)
		Nein

Compilerfehler (fehlendes Semikolon / Syntax)	`int x = 5` `int y = ;`	Nein
Compilerfehler (doppelt im selben Scope)	`int x = 1; int x = 2;`	Nein

Merksätze:

- *Variablennamen: Buchstaben/Ziffern/_ , aber nicht mit Ziffer beginnen.*
- *Keine Schlüsselwörter (z. B. `int`, `float`) als Namen verwenden.*
- *Lesbarkeit zählt in Prüfungen: `count` besser als `a`.*

2. Datentypen und Variablen

Datentypen

```c
int alter = 21;              // ganze Zahl (signed)
unsigned int punkte = 100;   // ganze Zahl ohne Vorzeichen
char note = 'A';             // einzelnes Zeichen (ASCII)
```

Anwendungsbeispiel

- `int` → Zähler, Indizes
- `char` → Zeichen, Bytes
- `unsigned` → Bits, Speicher, Masken

Merksatz: *unsigned speichert keine negativen Zahlen.*

3. Ausdrücke und Zuweisungen

Arithmetik

```c
int preis = 10;              // Einzelpreis
int gesamt = preis * 3 + 5;  // 3 Stueck + Zuschlag
```

Inkrement

```c
int i = 0;          // Startwert
int a = i++;        // a = 0, i = 1  (Post-Inkrement)
int b = ++i;        // b = 2, i = 2  (Pre-Inkrement)
```

Merksatz: *i++ → danach erhöhen, ++i → vorher erhöhen*

4. Ein- und Ausgabe – stdio.h

Bibliothek `stdio.h`: Standard-Ein-/Ausgabe über Konsole oder Dateien.

printf

```c
printf("Punkte: %d\n", punkte); // Ausgabe: int-Variable mit Zeilenumbruch
```

scanf

```c
scanf("%d", &alter); // int einlesen; & = Adresse der Variable
```

5. Bedingungen und Verzweigungen

if / else

```c
if (alter >= 18) {          // Bedingung pruefen
  printf("volljaehrig");    // Text ausgeben
}                           // Blockende
```

switch

```c
switch (note) {                  // Auswahl nach Zeichen
  case 'A': punkte += 10; break; // A: Bonuspunkte geben, dann abbrechen
  case 'B': punkte += 5;  break; // B: weniger Bonus, dann abbrechen
  default: break;                // sonst: nichts tun
}                                // switch Ende
```

Merksatz: *break verhindert Durchfallen.*

6. Schleifen

for

```c
for (int i = 0; i < 5; i++) { // i von 0 bis 4
  printf("%d\n", i);          // i ausgeben
}                             // Schleifenblock Ende
```

Anwendungsbeispiel

- Arrays durchlaufen
- Zählen
- Wiederholungen

while

```c
while (eingabe != 0) {        // laeuft, solange eingabe nicht 0 ist
  scanf("%d", &eingabe);      // neue Zahl einlesen
}                              // while Ende
```

Merksatz: *while prüft vor dem ersten Durchlauf.*

7. Funktionen

Definition

```c
int quadrat(int x) {          // Funktion: nimmt int x
  return x * x;               // Rueckgabe: x^2
}                             // Funktionsende
```

Anwendungsbeispiel

- Code-Wiederverwendung
- Strukturierung
- bessere Lesbarkeit

Seiteneffekte (Zeiger)

```c
void reset(int *x) {          // nimmt Zeiger auf int
  *x = 0;                     // Wert an der Adresse auf 0 setzen
}                             // Funktionsende
```

8. Fließkommazahlen – math.h

Bibliothek `math.h`**:** Mathematische Funktionen mit `double`.

```c
#include <math.h>             // Mathe-Bibliothek einbinden
double r = sqrt(16.0);        // Quadratwurzel, Ergebnis: 4.0
```

Beispiel: $(-1)^n$ ohne pow()

```c
int vorzeichen = (n % 2 == 0) ? 1 : -1; // gerade n -> +1, ungerade -> -1
```

Anwendung:

- Wechselnde Vorzeichen
- Reihenentwicklung

9. Arrays

```c
int werte[3] = {10, 20, 30};  // Array mit 3 int-Werten
```

Anwendungsbeispiel

- Messwerte
- Punktelisten
- Tabellen

10. Zeiger

```c
int x = 5;     // normale int-Variable
int *p = &x;   // p speichert Adresse von x
```

Anwendungsbeispiel

- Funktionsparameter
- Dynamische Daten
- Arrays / Strings

Merksatz: *Zeiger speichern Adressen.*

11. Strings – string.h

Bibliothek `string.h`**:** Arbeiten mit Zeichenketten.

```c
char name[] = "Max";    // String (endet mit '\0')
printf("%s\n", name);   // String ausgeben
```

Anwendungsbeispiel

- Texte
- Benutzernamen
- Dateipfade

12. Bitoperationen

Bitoperationen arbeiten **bitweise** auf den einzelnen Bits einer Ganzzahl. Sie sind besonders wichtig für **Hardware-nahe Programmierung**, **Flags**, **Masken** und **effiziente Berechnungen**.

Binäre Darstellung (Grundlage)

Beispiel:

$$13_{10} = 00001101_2$$

Jede Stelle ist ein Bit (0 oder 1).
Merksatz: *Bitoperatoren arbeiten nicht mit Zahlen, sondern mit Bits.*

Übersicht der Bitoperatoren

Operator	Name	Wirkung
&	AND	Bit = 1, wenn beide Bits 1 sind
|	OR	Bit = 1, wenn mindestens ein Bit 1 ist
^	XOR	Bit = 1, wenn Bits verschieden sind
~	NOT	Invertiert alle Bits
«	Left Shift	Verschiebt Bits nach links
»	Right Shift	Verschiebt Bits nach rechts

Bitweises AND (&) – Maskieren

```c
int a = 12;       // 0x0C = 00001100
int b = 10;       // 0x0A = 00001010
int c = a & b;    // 0x08 = 00001000 (= 8)
```

Anwendung: Bestimmte Bits „herausfiltern“

```c
// Pruefen, ob Bit 3 gesetzt ist
// 0x08 = 00001000
if (x & 0x08) {                  // AND-Maske: bleibt nur Bit 3 uebrig
  printf("Bit 3 ist gesetzt");   // Meldung, falls Bit gesetzt
}                                // if Ende
```

Merksatz: *AND maskiert Bits weg.*

Bitweises OR (|) – Bits setzen

```c
int flags = 0x00;        // 00000000 (alle Bits aus)
flags = flags | 0x04;    // 00000100 (setzt Bit 2)
```

Anwendung:

- Statusbits setzen
- Flags aktivieren

Merksatz: *OR setzt Bits auf 1.*

Bitweises XOR (^) – Umschalten

```c
int x = 0x0F;       // 00001111
x = x ^ 0x01;       // 00000001 -> Ergebnis: 00001110
```

Eigenschaft:

$$x \oplus x = 0$$

Typische Anwendung:

- Bits toggeln
- Einfaches Tauschen von Variablen (theoretisch)

Merksatz: *XOR schaltet Bits um.*

Bitweises NOT (~) – Invertieren

```c
unsigned char x = 0x0F;  // 00001111
x = ~x;                  // 11110000 (alle Bits invertiert)
```

Achtung: NOT invertiert *alle* Bits inklusive Vorzeichenbit!
Prüfungsfalle: *Bei signed-Typen kann ~ zu negativen Zahlen führen.*

Bitverschiebung nach links («)

```c
int x = 0x03;    // 00000011
x = x << 2;      // 00001100 (= 12), entspricht * 4
```

Regel:

$$x << n \quad \Rightarrow \quad x \cdot 2^n$$

Anwendung:

- Schnelles Multiplizieren
- Bitpositionen erzeugen

Merksatz: *Links-Shift multipliziert mit einer Zweierpotenz.*

Bitverschiebung nach rechts (»)

```c
int x = 0x0C;    // 00001100
x = x >> 2;      // 00000011 (= 3), entspricht / 4 (abgerundet)
```

Regel:

$$x >> n \quad \Rightarrow \quad \lfloor x/2^n \rfloor$$

Achtung:

- Bei `signed`: Vorzeichenauffüllung möglich
- Bei `unsigned`: immer Nullen

Typisches Anwendungsbeispiel: Flags

```c
#define READ    0x01    // 00000001 (Leserecht Bit 0)
#define WRITE   0x02    // 00000010 (Schreibrecht Bit 1)
#define EXEC    0x04    // 00000100 (Ausfuehrrecht Bit 2)

int rechte = READ | WRITE; // 00000011 (READ+WRITE gesetzt)

// Pruefen
if (rechte & READ) {        // maskiert Bit 0 (READ)
  printf("Lesen erlaubt"); // Ausgabe, falls READ gesetzt
}                           // if Ende
```

Warum sinnvoll?

- Viele Zustände in einer Variable
- Sehr speichereffizient

Typische Prüfungsfallen

- Verwechslung von & (bitweise) und && (logisch)
- Vergessen von Klammern bei Kombination mit anderen Operatoren
- NOT auf signed-Typen

Prüfungs-Merksatz: *Bitoperatoren brauchen fast immer Klammern!*

Zusammenfassung (Ultra-Kurz)

- AND → maskieren
- OR → setzen
- XOR → umschalten
- NOT → invertieren
- « → $\times 2^n$
- » → $\div 2^n$

13. struct, enum, typedef

struct – Zusammengesetzte Datentypen

Ein `struct` fasst mehrere Variablen (auch unterschiedlicher Typen) zu einem neuen, zusammengehörigen Datentyp zusammen.
Motivation: Statt viele einzelne Variablen zu verwalten, beschreibt ein `struct` *ein reales Objekt* mit mehreren Eigenschaften.

Definition eines struct

```c
struct Punkt {              // neuer Datentyp Punkt
  int x;                    // x-Koordinate
  int y;                    // y-Koordinate
};                          // struct Ende (Semikolon!)
```

Bedeutung:

- `struct Punkt` ist ein neuer Datentyp
- `x` und `y` sind die Bestandteile (Member)

Merksatz: *Ein struct ist eine selbst definierte Datenstruktur aus mehreren Variablen.*

Deklaration und Zugriff

```c
struct Punkt p1;            // Variable vom Typ struct Punkt
p1.x = 3;                   // Member x setzen
p1.y = 4;                   // Member y setzen
```

Zugriff: *Auf Member wird mit dem Punkt-Operator . zugegriffen.*

Anwendungsbeispiel

```c
// Speicherung eines Koordinatenpunkts
struct Punkt pos;           // Punkt-Variable anlegen
pos.x = 10;                 // x setzen
pos.y = 20;                 // y setzen
```

Typische Anwendungen:

- Koordinaten (x, y)
- Messwerte (Wert + Einheit)
- Datensätze (Name, Alter, ID)

struct und Funktionen

Übergabe als Wert (Kopie):

```
void move(struct Punkt p) { // Parameter als Kopie (Wert)
  p.x++;                    // erhoeht nur die Kopie
}                           // Funktionsende
```

Übergabe als Zeiger (Original wird verändert):

```
void move(struct Punkt *p) { // Parameter als Zeiger (Original)
  p->x++;                    // erhoeht Original (ueber Zeiger)
}                            // Funktionsende
```

Merksatz: *Mit Zeiger auf struct werden Änderungen nach außen sichtbar.*

Pfeil-Operator ->

Der ->-Operator kombiniert Dereferenzierung und Memberzugriff.

$$p->x \equiv (*p).x$$

struct mit typedef

```
typedef struct Punkt Punkt; // Alias: Punkt statt struct Punkt
Punkt a;                    // Variable vom Typ Punkt
a.x = 5;                    // Member x setzen
```

Vorteil:

- Kürzerer, besser lesbarer Code
- Kein ständiges Schreiben von `struct`

Häufige Fehler in Prüfungen

- Vergessen des `struct`-Schlüsselworts (ohne typedef)
- Verwechslung von `.` und `->`
- Zugriff auf nicht initialisierte Member

Prüfungs-Merksatz: *Objekt → Punkt, Zeiger → Pfeil.*

14. Matrizen und String-Arrays

```
int matrix[2][2] = {{1,2},{3,4}}; // 2x2 Matrix initialisiert
char *namen[] = {"Anna","Bob"};   // Array von String-Zeigern
```

Ergänzungen für die Prüfung (Wichtig!)

1. Speicher und `sizeof`

```
int x;                          // int-Variable
printf("%zu\n", sizeof(x));     // Groesse von x in Bytes
printf("%zu\n", sizeof(int));   // Groesse von int in Bytes
```

Merksätze:

- `sizeof` liefert die Größe in **Bytes**
- `sizeof` ist kein Funktionsaufruf
- `sizeof(variable)` ≠ `sizeof(pointer)`

Prüfungsfalle:

```
int a[10];                      // Array mit 10 ints
int *p = a;                     // Zeiger auf erstes Element

sizeof(a);                      // 10 * sizeof(int)
sizeof(p);                      // sizeof(int*) (Zeigergroesse)
```

2. Arrays vs. Zeiger

```
void f1(int a[]) { }            // Array-Parameter (wird zu int*)
void f2(int *a) { }             // Zeiger-Parameter (gleich wie oben)
```

Merksatz: *Arrays werden bei Funktionsübergabe immer zu Zeigern.*
Aber:

```
int a[5];                       // Array mit 5 ints
sizeof(a);                      // 5 * sizeof(int)
```

```
void f(int *a) {                // Parameter ist Zeiger
  sizeof(a);                    // sizeof(int*) (nicht Arraygroesse!)
}                               // Funktionsende
```

3. Strings (`char`-Arrays mit `\0`)

```
char s[] = "Hi";                // 'H' 'i' '\0'
```

Speicher:

Wichtig:

- Jeder String endet mit `\0`
- `"Hi"` benötigt **3 Bytes**

Prüfungsfalle:

```
char s[2] = "Hi";               // falsch: kein Platz fuer '\0'
```

4. `scanf` – typische Fehler

Richtig:

```
int x;              // Variable anlegen
scanf("%d", &x);    // red int at adress
```

Strings sicher einlesen:

```
char name[20];        // Puffer
scanf("%19s", name); // max. 19 Zeichen +
      '\0'
```

Falsch:

```
scanf("%d", x); // x not adress
```

Merksätze:

- Bei `scanf` fast immer `&`
- Strings niemals ohne Laengenbegrenzung einlesen

5. Initialisierung vs. Zuweisung

```
int x = 5;                      // Initialisierung
x = 7;                          // Zuweisung
```

Prüfungsfalle:

```
int x;                          // nicht initialisiert
printf("%d", x);                // undefiniertes Verhalten
```

Merksatz: *Nicht initialisierte Variablen haben keinen definierten Wert.*

6. `break` vs. `continue`

```
for (int i = 0; i < 10; i++) {  // i von 0 bis 9
  if (i == 5) continue;         // i==5 ueberspringen
  printf("%d", i);              // sonst i ausgeben
}                               // Schleifenende
```

Unterschied:

- `break` beendet die Schleife
- `continue` startet den naechsten Durchlauf

7. Rueckgabewerte von Funktionen beachten

```
if (scanf("%d", &x) != 1) {     // prueft, ob genau ein int gelesen wurde
  // Fehlerbehandlung             // falls nicht: Eingabefehler
}                                 // if Ende
```

Merksatz: *Viele C-Funktionen liefern Statuswerte und sollten geprueft werden.*

8. ASCII

ASCII: `'A'` = 65, `'a'` = 97
Abstand: $97 - 65 = 32$ (0x20)