```
================================================================================
Variablendeklarationen
================================================================================

Es gibt prinzipiell 3 Arten, eine Variable zu deklarieren:

    int i;                  // deklariert eine uninitialisierte Variable i
    int i = 5;              // deklariert eine Variable i mit dem Wert 5
    int i(5);               // deklariert eine Variable i mit dem Wert 5

Uninitialisierte Variablen haben nach der Deklaration einen beliebigen Wert.
Insbesondere ist nicht sichergestellt, dass dieser bei jedem Programmdurchlauf
gleich ist. Es kann also sein, dass i einmal den Wert 0 und einmal den Wert
-428984243 annimmt. Daher sollte man uninitialisierte Variablen nie direkt
verwenden, sondern immer durch Zuweisung oder Rechnung initialisieren

Gut:
----
{
    int i;
    int j = 4;

    i = j * j;                      // i ist jetzt immer 16, j bleibt 4
    printf ("4 * 4 = %d\n", i);     // hier wird "4 * 4 = 16" ausgegeben
}

Schlecht:
---------
{
    int i = 4;
    int j;

    i = j * j;                      // j ist irgendwas also ist auch i irgendwas
    printf ("4 * 4 = %d\n", i);     // hier wird irgendwas ausgegeben
}

Der Compiler erkennt solche Fehler und gibt Warnungen aus.

Der Unterschied zwischen den anderen beiden ist:

    int i = 5;  // gebräuchlich für primitive Typen wie int, C++ Objekte
    int i(5);   // gebräuchlich praktisch nur für C++ Objekte

Für C++ Objekte entspricht das erste einer Deklaration eines Objektes, was dann
zugewiesen wird, das zweite ist ein Objekt was direkt mit einem Konstruktor vom
Wert 5 konstruiert wird. Gcc optimiert (fast immer) so, dass für C++ Objekte
die effizientere Form (die untere) tatsächlich verwendet wird, egal welche man
schreibt.

Mehrere Variablen deklariert man so

    int i = 5, j = 10;

oder uninitialisiert so

    int i, j, k;
```



```
================================================================================
Variablengültigkeit
================================================================================

Variablen sind immer innerhalb eines bestimmten Bereiches gültig, für den sie
deklariert wurden, ab der sie deklariert wurden.

Beispiele:

void a()
{
    /* ai ist innerhalb der for-Schleife gültig */
    for (int ai = 0; i < 10; i++)
    {
    }
}

void b()
{
    /* bi ist innerhalb der Funktion b gültig */
    int bi = 2;

    if (irgendwas())
    {
        /* ci ist nur innerhalb dieses if-Zweigs gültig */
        int ci = 7;

        while (irgendwas())
        {
            /* di ist nur innerhalb dieser while-Schleife gültig */
            int di = ci * ci;
        }
    }
}

/* di ist eine globale Variable, die für alle Funktionen (ab hier) zur
   Verfügung steht */
int di;
```

```
================================================================================
if, else und else if
================================================================================

Es gibt 3 grundlegende Arten, if zu verwenden.

(1) reines if

    if (x > 10)
    {
        printf ("wird ausgeführt wenn x > 10 ist");
    }

(2) if mit else-Zweig

    if (x > 10)
    {
        printf ("wird ausgeführt wenn x > 10 ist");
    }
    else
    {
        printf ("wird ausgeführt wenn es nicht der Fall ist, dass x > 10 ist");
    }

(3) if mit else-if-Zweig

    if (x > 10)
    {
        printf ("wird ausgeführt wenn x > 10 ist");
    }
    else if (x > 5)
    {
        printf ("wird ausgeführt wenn x > 5 (aber x nicht > 10 ist) ist");
    }

Man kann beliebig viele else-if Zweige (3) verwenden, und dann maximal ein
else, welches dann ausgeführt wird, wenn keiner der anderen Zweige ausgeführt
wurde.

================================================================================
for-Schleifen
================================================================================

for (int i = 0; i < 10; i++)   // Schleife, die die Zahlen von 0 - 9 ausgibt
{
    printf ("%d\n", i);
}

for (int i = 9; i >= 0; i--)   // Schleife, die die Zahlen von 9 - 0 ausgibt
{
    printf ("%d\n", i);
}
```



```
Im allgemeinen besteht eine for-Schleife aus den 3 Teilen:

for (<Initialisierung>; <Schleifenbedingung>; <Schritt>)
    ...

<Initialisierung>

    Wird einmal am Anfang der Schleife ausgeführt. Variablen die innerhalb der
    Initialisierung deklariert werden (also beispielsweise "int i = 0;" sind
    nur innerhalb der for-Schleife gültig).

<Schleifenbedingung>

    Wird vor dem Eintritt in die Schleife geprüft; wenn die Schleifenbedingung
    false ist, bricht die Schleife ab, andernfalls werden die Anweisungen
    innerhalb der Schleife ausgeführt.

<Schritt>

    Wird jedesmal nach den innerhalb der Schleife stehenden Anweisungen
    ausgeführt.

Die Ausführungsreihenfolge ist also

<Initialisierung>
<Schleifenbedingung>                wenn wahr
<Anweisungen innerhalb der Schleife>
<Schritt>
<Schleifenbedingung>                wenn wahr
<Anweisungen innerhalb der Schleife>
<Schritt>
<Schleifenbedingung>                wenn wahr
<Anweisungen innerhalb der Schleife>
<Schritt>
...
<Schleifenbedingung>                wenn false bricht die Schleife hier ab

Es ist möglich, die Teile <Initialisierung>, <Schleifenbedingung> und <Schritt>
wegzulassen. Beispielsweise:

for (int i = 0; i*i < j;)
{
    if (k < 30)
        i += berechneIrgendwas (j, k, l);
    else
        i -= berechneIrgendwas (j, k, l);
}

Wenn die Schleife nur eine Anweisung enthält, können die geschweiften Klammern
entfallen.

for (int i = 0; i < 10; i++)
    printf ("%d\n", i);

Wenn die Schleife keine Anweisung enthält, sieht sie so aus (kommt aber
seltener vor):

for (int i = 0; i < 10; i++)
    ;
```

```
================================================================================
while und do-while-Schleifen
================================================================================

Neben for-Schleifen gibt es while und do-while Schleifen.

while (i < 10)
{
    ...
}

do
{
    ...
} while (i < 10);

Beide besitzen eine Schleifenbedingung ähnlich wie eine for-Schleife. Was ist
der unterschied? Die while-Schleife prüft die Schleifenbedingung vor dem ersten
Schleifendurchlauf, die do-while-schleife erst danach.

Also: while-Schleife:

<Schleifenbedingung>                wenn wahr
<Anweisungen innerhalb der Schleife>
<Schleifenbedingung>                wenn wahr
<Anweisungen innerhalb der Schleife>
<Schleifenbedingung>                wenn wahr
<Anweisungen innerhalb der Schleife>
...
<Schleifenbedingung>                wenn false bricht die Schleife hier ab

Also: do-while-Schleife:

<Anweisungen innerhalb der Schleife>
<Schleifenbedingung>                wenn wahr
<Anweisungen innerhalb der Schleife>
<Schleifenbedingung>                wenn wahr
<Anweisungen innerhalb der Schleife>
...
<Schleifenbedingung>                wenn false bricht die Schleife hier ab

Man sieht, dass die do-while-Schleife eine Prüfung der Schleifenbedingung
(ganz am Anfang) nicht vornimmt.

Im allgemeinen ist zu sagen, dass alle Schleifen, also for, while und do-while
Schleifen sich ineinander umformen lassen.
```



```
================================================================================
Pseudozufall
================================================================================

// Zufall:

#include <stdlib.h>

int rand();            // gibt eine Zahl zwischen 0 und RAND_MAX zurück.
int srand (int seed);  // initialisiert den Zufallsgenerator

// Zeitmessung:

#include <time.h>

time_t time (time_t *tloc);  // gibt die Anzahl der Sekunden seit
                             // (00:00:00 UTC, January 1, 1970) zurück

Aus diesen drei Funktionen lässt sich folgendes Programm konstruieren:

// Programm welches zwei Würfel wirft.

#include <stdio.h>
#include <stdlib.h>
#include <time.h>

int main()
{
    srand (time (NULL)); // initialisiert den Zufallsgenerator

    int w1 = rand() % 6 + 1;
    int w2 = rand() % 6 + 1;

    printf ("Würfel: %d %d\n", w1, w2);
}

Anmerkungen:

Ohne den srand()-Aufruf wären die Ergebnisse immer gleich.

Es reicht eigentlich immer aus bei time() aus, NULL als Argument zu übergeben.
```

```
================================================================================
Arrays
================================================================================

Wenn man viele Variablen vom gleichen Typ benötigt, lässt sich dies mit einem
Array bewerkstelligen. Ein Array enthält einfach nur lauter Variablen vom
gleichen Typ.

Syntax:

    int x[10];

Deklariert 10 Integer Variablen, die dann als x[0] bis x[9] angesprochen werden
können. Mehrdimensionale Arrays deklariert man so:

    int schachbrett[8][8];

deklariert ein Array mit dem Namen Schachbrett, welches 8 * 8 Integer aufnehmen
kann. Man kann Arrays initialisieren, indem man die Werte listet:

    int x[5] = { 33, 20, 19, 9, 2 };

================================================================================
reverse.cc
================================================================================

/* Programm welches ein Array benutzt und einmal vorwärts und einmal rückwärts
 * ausgibt.
 */
#include <stdio.h>

int main()
{
    int x[5] = { 33, 20, 19, 9, 2 };

    printf ("Das Array x enthält vorwärts gelesen:");
    for (int i = 0; i < 5; i++)
        printf (" %d", x[i]);
    printf ("\n");

    printf ("und rückwärts gelesen:");
    for (int i = 4; i >= 0; i--)
        printf (" %d", x[i]);
    printf ("\n");
}
```



```
================================================================================
Aufgabe 1: einfache Funktionen
================================================================================
(a) Implementiere (und teste) eine Funktion, welche mittels einer
    while-Schleife die ersten n Zahlen aufaddiert

(b) Implementiere (und teste) die Funktion

    int hoch (int basis, int exponent)
    {
    }

    welche basis^exponent berechnet. Also beispielsweise 3^7 sollte 2187 sein.
    Das C++ Symbol ^ steht allerdings nicht für diese Operation (sondern für
    bitweises XOR), sodass Du die Funktion durch einzelne Multiplikationen
    berechnen musst.

(c) Wieviele Multiplikationen braucht Deine Implementation um 3^7 zu
    berechnen? Versuche die Funktion so zu implementieren, dass die Anzahl der
    Multiplikationen minimal wird.

    Beispielsweise kann man 3 ^ 7 wie folgt berechnen:

    3 * 3 = 9       // 3 ^ 2
    9 * 9 = 81      // 3 ^ 4
    81 * 9 = 729    // 3 ^ 6
    729 * 3 = 2187  // 3 ^ 7

    Damit werden nur 4 Multiplikationen verwendet.

================================================================================
Aufgabe 2: Arrays
================================================================================
(a) Schreibe ein Programm, welches 10 Würfel wirft, und die Ergebnisse in
    ein Array schreibt, und dann ausgibt.

(b) Erweitere das Programm so, dass es ausgibt, wieviele Würfel 1, wieviele 2,
    wieviele 3 usw. waren. Verwende hierbei (mindestens) eine do-while Schleife.

(c) Schreibe ein Programm, was eine Zeile einliest und rückwärts ausgibt. Für
    ein Beispiel zum Lesen einer Zeile, siehe Aufgabe 3.
```

```
================================================================================
Aufgabe 3: Ein Rechentrainer
================================================================================
Das folgende Programm soll als Grundlage für diese Übung dienen:

/* Programm welches dem Nutzer zufällige Rechenaufgaben stellt, und prüft,
 * ob sie richtig beantwortet werden
 */
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

int main()
{
    // initialisiert den Zufallsgenerator (sodass immer andere Aufgaben gestellt
 werden)
    srand (time (NULL));

    // zufällig auswähen ob '+' oder '-' Aufgabe gestellt wird
    int rechenart = rand() % 2;
    int ergebnis;

    if (rechenart == 0)
    {
        /* wähle z1 und z2 zufällig zwischen 1 und 10 */
        int z1 = rand() % 10 + 1;
        int z2 = rand() % 10 + 1;
        ergebnis = z1 + z2;
        printf ("%d + %d = ", z1, z2);
    }
    else
    {
        /* wähle z1 und z2 zufällig zwischen 1 und 10 */
        int z1 = rand() % 10 + 1;
        int z2 = rand() % 10 + 1;
        z1 += z2;                   // stellt sicher, dass das Ergebnis am Ende
positiv ist
        ergebnis = z1 - z2;
        printf ("%d - %d = ", z1, z2);
    }

    // lese Eingabe vom Benutzer
    char eingabe[10];
    fgets (eingabe, 10, stdin);

    // prüfe, ob Eingabe und richtiges Ergebnis übereinstimmen
    if (atoi (eingabe) == ergebnis)
        printf ("Korrekt!\n");
    else
        printf ("Falsch -> richtiges Ergebnis ist: %d\n", ergebnis);
}
```



```
Neu eingeführte Funktionen sind hierbei:

    fgets           liest eine Zeile vom Benutzer
    atoi            wandelt eine Zeichenkette in einen Integer um

Das Programm soll nun erweitert werden, um mehr Features anzubieten.

Einige Anregungen hierzu:
 - mehrere Aufgaben nacheinander abfragen
 - auch Multiplikationen und Divisionen abfragen
 - mehrere Schwierigkeitsstufen anbieten

    man bedenke dabei aber, dass es im Kopf schwieriger ist 382 * 482 zu
    rechnen als 382 + 482.  Die Schwierigkeitsstufen sollten also natürlich
    gewählt sein.

 - ein Endergebnis präsentieren, nachdem alle Aufgaben gerechnet wurden
 - berücksichtigen, wie schnell oder langsam der Benutzer rechnet
 - das Programm besser strukturieren (Unterfunktionen einführen)
 - das Feedback zufällig variieren (beispielsweise manchmal "Sehr gut!" sagen
   und manchmal "Richtig, weiter so!")
```