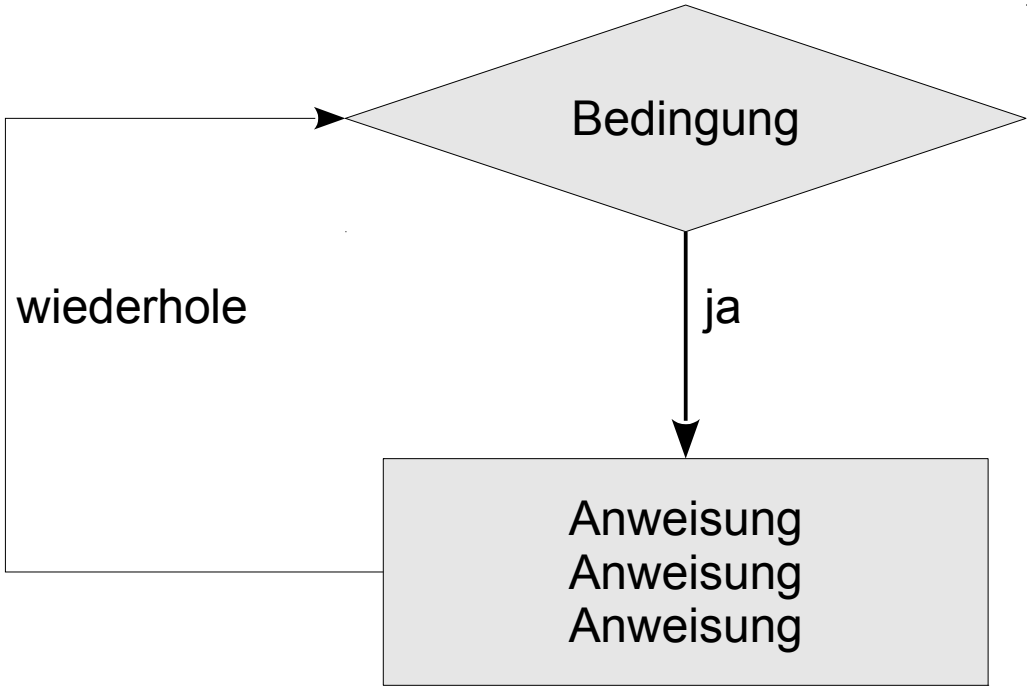
C++ - Einführung in die Programmiersprache

„Schleifen“



Anweisungen in C++

```
#include <iostream>
```

Präprozessor-Anweisungen

```
int ergebnis;
ergebnis- = 5 + 5;
std::cout << ergebnis << std::endl;
return 0;
```

Interpretation durch den Compiler

Präprozessor-Anweisungen

```
#include <iostream> // Ein- und Ausgabe
#include <cmath>   // Mathematische Funktionen
```

- Beginn mit dem Hash-Zeichen.
- Pro Zeile eine Anweisung. Das Zeilenende markiert auch das Ende der Anweisung.
- Am Anfang der Quelldatei werden alle Präprozessor-Anweisungen aufgelistet.
- Der Präprozessor ersetzt die Anweisung durch den entsprechenden Textabschnitt.

Anweisungen

- Beschreibung eines Arbeitsschrittes entsprechend der Syntax einer Sprache. Falls die Syntax nicht korrekt ist, wird ein Fehler angezeigt.
- Befehle für den Computer, die von dem Compiler interpretiert werden.
- Anweisungen werden von oben nach unten in einer Quelldatei ausgeführt.

Deklarationsanweisungen

```
const double PI = 3.14159265359;

double wert = 0.0;
char zeichen;

int koordinaten[10][31];
```

- Deklaration von Konstanten, Variablen und Arrays in einem Codeblock.
- Die Anweisung wird mit einem Semikolon beendet.

Ausdrucksanweisungen

```
std::cin >> b;
flaeche = b * h;
sinus = sin(wert);
```

- Rechts vom Gleichheitszeichen wird ein Ausdruck definiert. Der Wert des Ausdruckes wird einer Variablen mit Hilfe des Gleichheitszeichen zugewiesen.
- Rechts vom Gleichheitszeichen wird eine Funktion aufgerufen. Die Funktion gibt einen Wert zurück, der in einer Variablen gespeichert wird.
- Ein- und Ausgabe.

Leere Anweisung

```
;
{}
```

- Nur das Semikolon ohne eine Anweisung.
- Besser: Leere geschweifte Klammern als Kennzeichnung für einen leeren Block.
- Leere Anweisung führen an den falschen Stellen zu Fehlermeldungen!

Bedingte Anweisungen

```
double divident;
double divisor = 0.0;

std::cout << "\nBitte geben Sie ein Divident ein: ";
std::cin >> divident;

if(!std::cin.fail()){
    std::cout << "\nBitte geben Sie ein Divisor ein: ";
    std::cin >> divisor;

    if (divisor > 0){
        std::cout << divident << " / " << divisor << " = "
                << (divident / divisor);
    }
}
```

Erläuterungen

- Im Kopf der Anweisung wird eine Bedingung formuliert.
- Bedingungen sind Aussagen, die mit Ja oder Nein beantwortet werden können.
- Wenn die Bedingung wahr ist, wird der Rumpf der Anweisung ausgeführt.
- Wenn die Bedingung falsch ist, werden die nachfolgenden Anweisungen ausgeführt.

Bedingungen

- Wahre oder falsche Aussagen. Zum Beispiel: „Das Fahrrad ist blau. Ja / Nein“.
- Vergleiche von Werten. Zum Beispiel: a größer b.
- Ausdruck, der einen boolschen Wert zurückgibt.
- Vergleich von Werten. Beantwortung von Fragen wie zum Beispiel „Ist der Messwert größer als 0“.

Vergleichsausdrücke

(	ausdruck	)	>	(	ausdruck	)
(	20	)	>	(	5	)
(	6 - 7	)	>	(	0	)

Operanden in Vergleichsausdrücken

- Variablen, die als Platzhalter für einen definierten Wert genutzt werden.
- Konstanten, die einen bestimmten Wert verkörpern. Der Wert kann nicht verändert werden.
- Literale wie zum Beispiel 2, 3.5, 'A' . Die Werte werden direkt in in die Anweisung geschrieben. Literale werden einmalig an einer bestimmten Stelle im Code benötigt.

Operatoren in Anweisungen

- Zuweisungsoperator. Mit Hilfe des Gleichheitszeichens wird einer Variablen ein Wert zugewiesen.
- Arithmetische Operatoren berechnen einen Wert aus ein oder zwei Operanden.
- Vergleichsoperatoren vergleichen zwei Werte.
- Logische Operatoren verknüpfen verschiedene Ausdrücke.
- Bit-Operatoren nutzen die binäre Darstellung von Zahlen.

Relationale Operatoren in Vergleichen

==	Ist gleich
!=	Ist ungleich
<	Ist kleiner
<=	Ist kleiner gleich
>	Ist größer
>=	Ist größer gleich

Hinweise

- Zusammengesetzte Operatoren wie ==, <= und so weiter dürfen nicht durch ein Leerzeichen getrennt werden.
- Der Zuweisungsoperator = darf nicht mit dem Operator „ist gleich“ == verwechselt werden.
- Mit Hilfe von Klammern kann die Lesbarkeit des Ausdrucks erhöht werden.
- Gleitkommazahlen nähren sich einem Wert an. Aus diesen Grund sollte eine Überprüfung auf Gleichheit vermieden werden.

Verknüpfung von Bedingungen

```
bool ergebnis;
double zahl;
char zeichen;

ergebnis = ((zahl > 1) && (zahl < 10));
ergebnis = ((zeichen == 'f') || (zeichen == 'F'));
ergebnis = (!(zeichen == 'C'));
```

Verknüpfen von Ausdrücken

(	ausdruck	)	&&	(	ausdruck	)

- Fragen, die mit Ja (true) oder Nein (false) beantwortet werden können, werden miteinander verknüpft.
- Zuerst werden die Vergleichsausdrücke links und rechts ausgewertet. Das Ergebnis der Auswertung wird verknüpft.
- Die Verknüpfung gibt immer einen boolschen Wert zurück.

Verknüpfungsoperatoren

&&	und / and
||	oder /or
!	nicht

Und-Verknüpfung

True	=	(True)	&&	(True)	;
False	=	(True)	&&	(False)	;
False	=	(False)	&&	(True)	;
False	=	(False)	&&	(False)	;

Beispiel

```
ergebnis = ((zahl > 1) && (zahl < 10));
```

- Der linke sowohl als auch der rechte Ausdruck müssen wahr sein.
- Falls der linke Ausdruck falsch ist, wird der rechte Ausdruck nicht mehr ausgewertet werden.

Oder-Verknüpfung

True	=	(True)	||	(True)	;
True	=	(True)	||	(False)	;
True	=	(False)	||	(True)	;
False	=	(False)	||	(False)	;

Beispiel

```
ergebnis = ((zeichen == 'f') || (zeichen == 'F'));
```

- Einer der beiden Ausdrücke muss wahr sein.
- Sobald der linke Ausdruck wahr ist, wird der rechte Ausdruck nicht mehr ausgewertet.

Negation

False	=	(	!	(True)	)	;
True	=	(	!	(False)	)	;

Beispiel

```
ergebnis = (!(zeichen == 'C'));
```

- Falsch wird zu wahr und umgekehrt.
- Die Negation kann häufig durch eine „Umkehrung“ der Operatoren ersetzt werden.
- In diesem Beispiel wird der „ist gleich“-Operator durch den „ist nicht gleich“-Operator ersetzt.

Runde Klammern

```
boolean = (Bedingung)

variable = (ausdruck) operator (ausdruck)

int main(int argc, char** argv)
```

- Runde Klammern fassen Ausdrücke zusammen.
- Runde Klammern erhöhen die Lesbarkeit von komplexen Ausdrücken.
- Anfang und Ende einer Parameterliste einer Funktion oder Methode wie zum Beispiel `main()`.

Rangfolge

Priorität	Operator
1	Postfix-Inkrement ++ Postfix-Dekrement -- Klammerung ()
2	Prefix-Inkrement ++ Prefix-Dekrement -- Vorzeichen + Vorzeichen - Negation !
3	Multiplikation * Division / Modula %
4	Addition + Subtraktion -

Rangfolge

Priorität	Operator
5	Größer > Größer gleich >= Kleiner < Kleiner gleich <=
6	Ist gleich == Ist ungleich !=
7	Und && Oder ||
8	Zuweisung =

Assoziativität

	Operator
→	Inkrement ++, Dekrement – Multiplikation * Division Modulo % Addition + Subtraktion - Kleiner <, Kleiner Gleich <= Größer >, Größer gleich >= ist gleich ==, ist nicht gleich != Und &&, Oder ||
←	Nicht ! Inkrement ++, Dekrement -- Zuweisung =

Wiederholung von Anweisungen

- Schleifen. Iterationsanweisungen.
- Kopf- oder fußgesteuerte Schleifen. Die Anzahl der Schleifendurchläufe ist abhängig von einer Bedingung.
- Zählschleifen. Die Anzahl der Durchläufe ist exakt festgelegt.
- Seit C++ 11: Für jedes Element in einem Array. Foreach-Schleifen.

Kopfgesteuerte Schleife

```
int summe = 0;
int durchlauf = 1;

while(durchlauf <= 10){
    summe = summe + durchlauf;
    durchlauf++;
}
```

Aufbau

Schleifenkopf

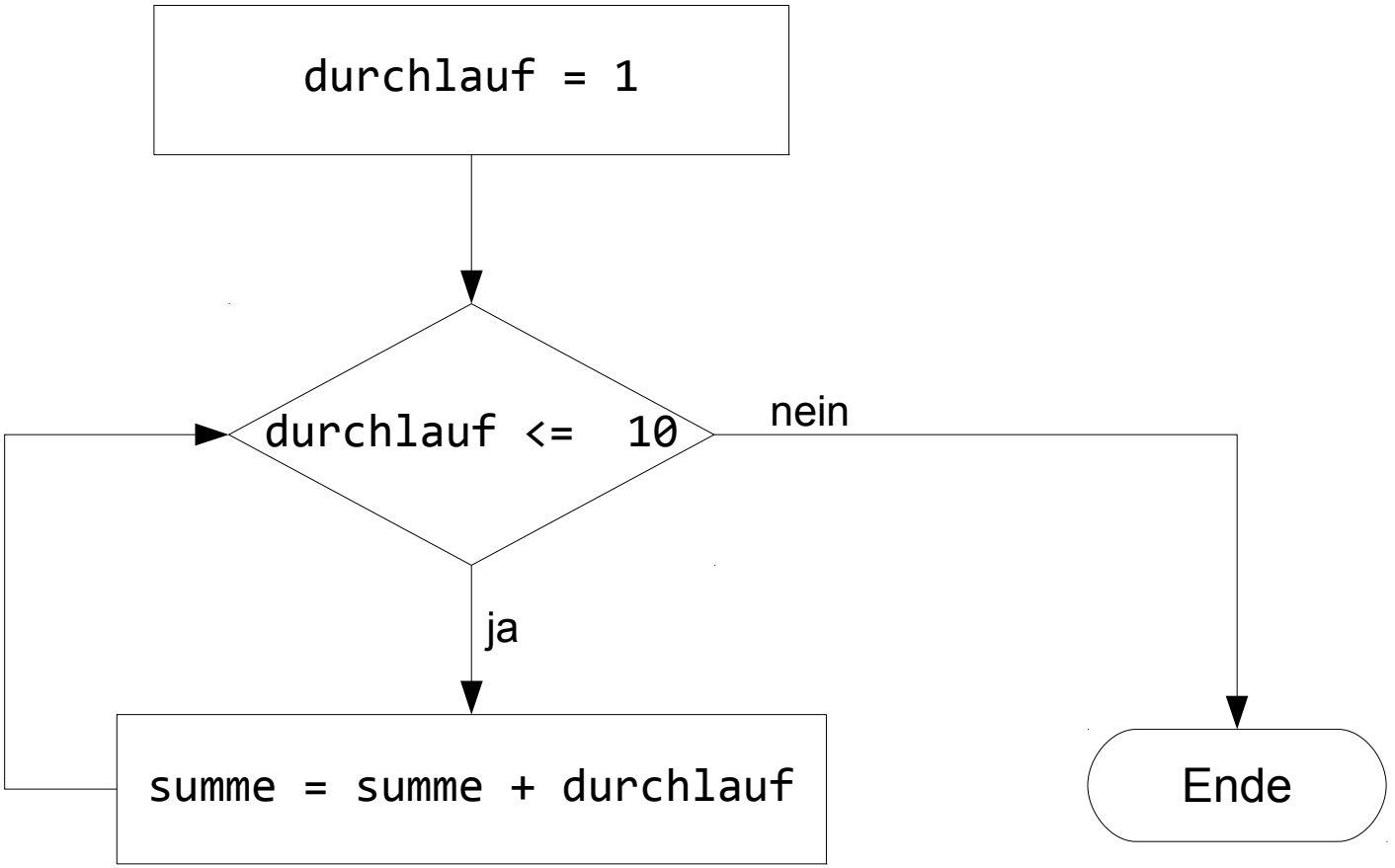
```
int durchlauf = 1;

while(durchlauf <= 10){
```

- Der Schleifenkopf beginnt mit dem Schlüsselwort `while`.
- Dem Schlüsselwort folgt in runden Klammern das Abbruchkriterium der Schleife.

Schleifenrumpf

- Beginn und Ende mit den geschweiften Klammern.
- Anweisungsblock, der wiederholt werden soll.
- In dem Block wird die zu überprüfende Variable so verändert, so dass die Schleife nach x Durchläufen abbricht. Andernfalls wird die Schleife nie abgebrochen.

Ablaufdiagramm



Endloschleife

```
while(1){

```

- Die Bedingung `(1)` symbolisiert den boolschen Wert „wahr“.
- Die Bedingung trifft immer zu.
- Die Schleife läuft endlos. Ausnahme: Die Schleife wird mit Hilfe von `break` im Schleifenrumpf abgebrochen.

Leere Schleife

```
int summe = 0;
int durchlauf = 1;

while(durchlauf <= 10);
{
    summe = summe + durchlauf;
    durchlauf++;
}
```

- Direkt im Anschluss an den Schleifenkopf folgt ein Semikolon.
- Der Schleifenrumpf zu dieser Schleife ist leer.
- Die while-Schleife läuft endlos.

x Sekunden warten

```
#include <iostream>
#include <ctime>


int main() {
    double sekunden = 4.0;

    if (sekunden > 0){
        clock_t  wartezeit = sekunden * CLOCKS_PER_SEC;
        std::cout << "\nWartezeit beginnt.";

        clock_t startzeit = clock();

        while((clock() - startzeit) < wartezeit){
            std::cout << "\ngewartet: ";
        }

        std::cout << "\nWartezeit beendet";
    }
```

Systemzeit

- Der Typ `clock_t` ist in der Bibliothek `<ctime>` definiert. Der Typ repräsentiert konstante, aber systemabhängige Zeiteinheiten.
- Die Funktion `clock()` liefert die Prozessorzeit zurück, die das Programm verbraucht hat.
- Die symbolische Konstante `CLOCKS_PER_SEC` definiert die Zeiteinheit pro Sekunde.

Fußgesteuerte Schleife

```
summe = 0;
durchlauf = 1;

do{
    summe = summe + durchlauf;
    durchlauf++;
}while(durchlauf <= 10);
```

Aufbau

Schleifenkopf

```
do {

}
```

- Jede fußgesteuerte Schleife beginnt mit dem Schlüsselwort do.
- Eine Bedingung für den Durchlauf ist nicht vorhanden.
- Der Schleifenrumpf wird mindestens einmal durchlaufen.

Schleifenfuß

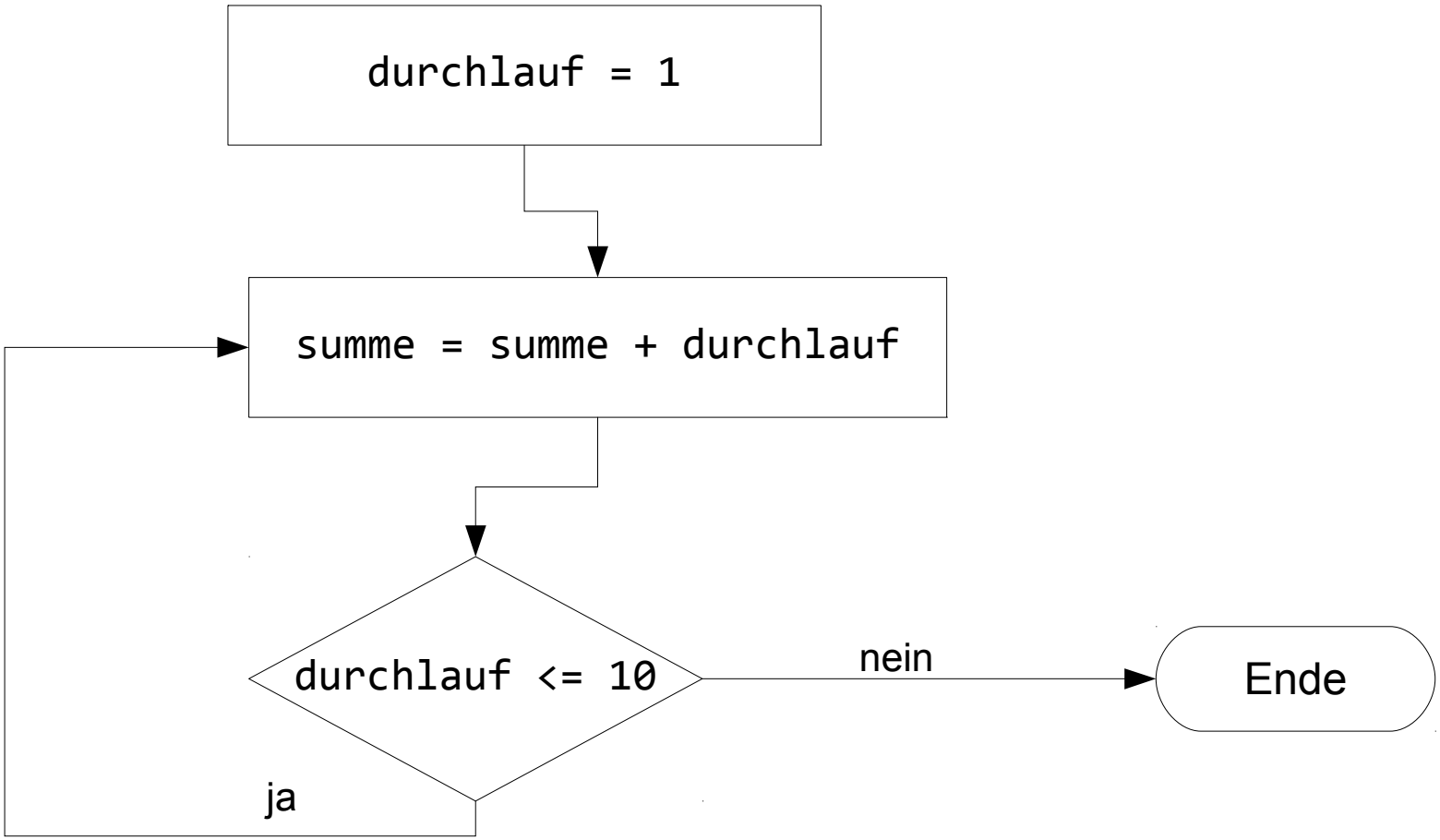
```
int durchlauf = 1;

do {

}while(summe <= 10);
```

- Dem Schlüsselwort `while` (solange) folgt die Bedingung in runden Klammern.
- Nach jedem Durchlauf wird die Bedingung überprüft.
- Der Schleifenfuß muss mit einem Komma abgeschlossen werden.

Schleifenrumpf

- Beginn und Ende mit den geschweiften Klammern.
- Anweisungsblock, der wiederholt werden soll.
- In dem Block wird die zu überprüfende Variable so verändert, so dass die Schleife nach x Durchläufen abbricht. Andernfalls wird die Schleife nie abgebrochen.

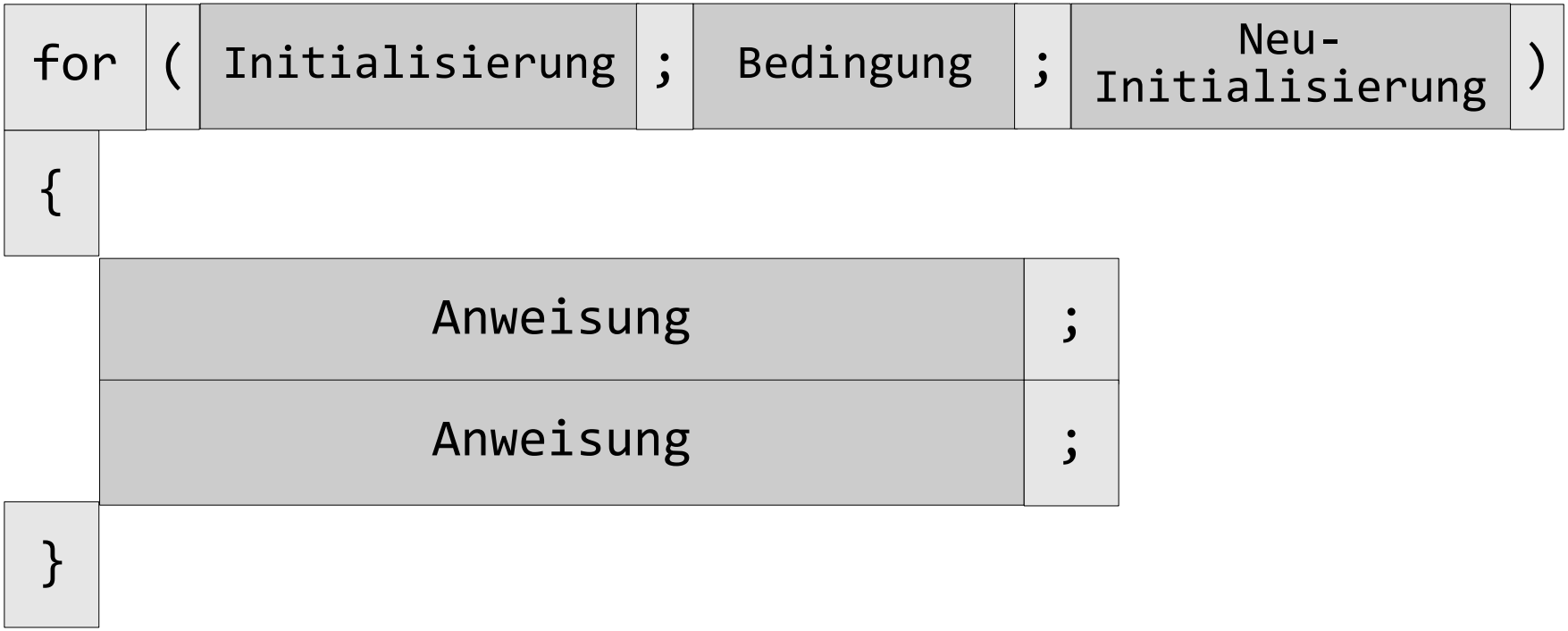
Ablaufdiagramm



Zählschleife

```
int summe = 0;

for (int durchlauf = 1;
     durchlauf <= 10;
     durchlauf++){


    summe = summe + durchlauf;
}
```

Aufbau


for
(
Initialisierung
;
Bedingung
;
Neu-
Initialisierung
)
{
Anweisung
;
Anweisung
;
}

Schleifenkopf

- Der Schleifenkopf beginnt mit dem Schlüsselwort `for`.
- Dem Schlüsselwort folgen drei Anweisungen. Die drei Anweisungen werden mit Hilfe der runden Klammern zusammengefasst.
- Jede der Anweisungen endet mit einem Semikolon. Die letzte Anweisung wird automatisch „beendet.“

1. Anweisung: Deklaration und Initialisierung

for	(	Initialisierung	;	)
for	(	int lauf = 0	;	)

- Die erste Anweisung ist eine Deklarationsanweisung.
- In der ersten Anweisung wird die Zählvariable für die Durchläufe definiert.
- Die Zählvariable wird gleichzeitig initialisiert. Der Anfangswert der Zählvariablen ist definiert.
- Hinweis: Die Zählvariable kann nur im Schleifenrumpf dieser Schleife genutzt werden.

2. Anweisung: Abbruchbedingung

for	(	Initialisierung	;	Bedingung	;	)
for	(	int lauf = 0	;	lauf <= 10	;	)

- Die zweite Anweisung legt das Abbruchkriterium fest.
- Mit Hilfe der Bedingung wird die Anzahl der Durchläufe festgelegt.

3. Anweisung: Neu-Initialisierung

for	(	Initialisierung	;	Bedingung	;	Neu-Initialisierung	)
for	(	int lauf = 0	;	lauf <= 10	;	lauf++	)

- Durch den angegebenen Ausdruck wird die Zählvariable neu initialisiert.
- Für die Neu-Initialisierung der Zählvariablen kann jede Form von Ausdrucksanweisung genutzt werden. In diesem Beispiel wird der Inkrement-Operator genutzt.

Hinweise

- Wenn die Initialisierungsanweisung durch eine leere Anweisung ersetzt wird, sollte die Zählvariable vor Aufruf der Schleife deklariert und initialisiert werden.
- Wenn die Bedingung durch eine leere Anweisung ersetzt wird, sollte das Abbruchkriterium durch eine bedingte Anweisung im Schleifenrumpf überprüft werden. Andernfalls läuft die Schleife endlos.
- Wenn die Neu-Initialisierung der Zählvariablen durch eine leere Anweisung ersetzt wird, muss die Zählvariablen im Schleifenrumpf mit Hilfe einer Ausdrucksanweisung verändert werden.

Schachtelung von Schleifen

```
int x = 0;
int y = 0;

do{

   for(int anzahl = 1; anzahl <= anzahlMax; anzahl++ ){

          std::cout << "\n Bitte geben Sie eine y_Koordinate ein:";
          std::cin >> y;

          if (std::cin.fail()){
              std::cin.clear();
              std::cin.ignore(std::numeric_limits<std::streamsize>::max(),
                              '\n');
              std::cout << "\nFalsche Eingabe";
              break;
          }
          std::cout << "\nKoordinaten: " << x << ", "  << y;
    }

    x++;
}while(x < maxAnzahl);
```

Hinweis

- Schleifen und bedingte Anweisungen können beliebig tief verschachteltet werden.
- Aber ab einer bestimmten Schachtelungstiefe wird der Code unlesbar.

Vorzeitiger Abbruch einer Schleife

- Jede Schleife kann mit Hilfe des Schlüsselwortes `break` vorzeitig abgebrochen werden.
- Bei verschachtelten Schleifen, wird nur die Schleife abgebrochen, zu der das Schlüsselwort `break` gehört. Alle darüber liegende Schleifen werden nicht abgebrochen.

Beispiel

```
int teilbarOhneRest = 0;

for(int divident = 101; divident < 200; divident++ )
{
    for(int divisor = 2; divisor < 10; divisor++ )
    {
        teilbarOhneRest = divident % divisor;

        if (teilbarOhneRest == 0)
        {
            cout << divident << " % " << divisor << '\n';
            break;
        }
    }
}
```

Bricht diese Schleife ab