Anweisungen

- Ausdrucksanweisung
- Deklarationsanweisung
- Block
- Bedingte Anweisung
- Fallunterscheidung
- Schleifen
- Sprunganweisungen

Anweisungen

Die Ausdrucksanweisung

Ausdrücke werden durch ein angehängtes Semikolon zur Anweisung. Das Semikolon beendet die Anweisung.

Mit einer Ausdrucksanweisung ist in der Regel eine Aktivität verbunden, z.B. eine Zuweisung oder eine Ausgabe.

Beispiele:

```
a = b = c;
cout << a << b;
```

Ausdrucksanweisungen wie z. B.

```
x < y;
a == b + 1;
```

sind zwar erlaubt, ergeben aber keinen Sinn. Der Compiler erzeugt für solche Anweisungen keinen Code

Das Semikolon allein bildet auch schon eine Anweisung, und zwar die **leere Anweisung**. Diese bewirkt *nichts*.

Sie wird aus syntaktischen Gründen eingeführt:
Wenn an einer Stelle des Programms eine Anweisung stehen *muss*, dort aber nichts zu tun ist, wird die leere Anweisung verwendet.

Anweisungen

Block und Deklarationsanweisung

Keine oder mehrere Anweisungen können durch geschweifte Klammern ({ und }) zu einem **Block** zusammengefaßt werden.

Ein Block kann immer dort verwendet werden, wo eine einzelne Anweisung erlaubt ist, denn der Block *ist* eine Anweisung, wenn auch eine zusammengesetzte.

In einem Block können lokale Variablen deklariert werden. Die Variablendeklaration wird auch als **Deklarationsanweisung** bezeichnet.

Lokale Variablen müssen vor ihrer Verwendung entweder bei der Deklaration oder durch eine Wertzuweisung initialisiert werden.

Vorsicht:

Der C++-Compiler gibt *keine* Fehlermeldung aus, wenn eine lokale Variable vor ihrer Initialisierung verwendet wird.

Lokale Variablen können nur in dem Block verwendet werden, der ihre Deklaration enthält. Die Variablen, die in einem Block deklariert werden, müssen *unterschiedliche* Bezeichner haben. Sie können aber denselben Bezeichner haben wie Variablen, die in einem anderen Block deklariert sind.

Anweisungen

Der Block

Unterscheidung von

- **Gültigkeitsbereich** (engl. *scope*) und
- **Sichtbarkeitsbereich** (engl. *visibility*)

Gültigkeits- und Sichtbarkeitsregeln für Bezeichner

- Bezeichner sind nur *nach der Deklaration* und nur *innerhalb des Blocks* gültig, in dem sie deklariert wurden. Sie sind *lokal* bezüglich des Blocks.

- Bezeichner sind auch gültig für innerhalb des Blocks neu angelegte innere Blöcke.

- Die Sichtbarkeit (engl. *visibility*) z. B. von Variablen wird eingeschränkt durch die Deklaration von Variablen gleichen Namens. Im Sichtbarkeitsbereich der inneren Variablen ist die äußere Variable unsichtbar.

Anweisungen

Der Block

Gültigkeits- und Sichtbarkeitsregeln für Bezeichner

Beispiel:

```
#include <iostream>
using namespace std;

int main(int argc, char *argv[])
{
    // short argc = 4711;

    {
        bool argc = true;

        {
            char argc = 'a';

            {
                double argc = 14.15;

            }
        }
    }
}
```

Anweisungen

Der Block

Gültigkeits- und Sichtbarkeitsregeln für Bezeichner

Beispiel:

```
#include <iostream>
using namespace std;

int a = 1, b = 2;      // globale Variablen

int main()
{
  cout << "globales a = " << a << endl;
  int a = 10;
  cout << "lokales a = " << a << endl;
  cout << "globales ::a = " << ::a << endl;
  {
    int b = 20;
    int c = 30;
    cout << "lokales b = " << b << endl;
    cout << "lokales c = " << c << endl;
    cout << "globales ::b = " << ::b << endl;
  }
  cout << "globales b wieder sichtbar: b = "
       << b << endl;

  //cout << "c = " << c << endl;
}
```

Anweisungen

Die bedingte Anweisung

`if (` ***Bedingung*** **`)`**
  Anweisung$_1$
`else`
  Anweisung$_2$

Die Bedingung muss vom Typ `bool` sein oder in `bool` umgewandelt werden können.

Die Bedingung wird ausgewertet. Wenn sie *wahr* ist, d.h. den Wert `true` oder einen Wert ungleich `0` liefert, wird Anweisung$_1$ ausgeführt und der `else`-Teil übersprungen.

Ist die Bedingung *falsch*, wird Anweisung$_2$ ausgeführt; Anweisung$_1$ wird übersprungen.

Soll mehr als eine Anweisung ausgeführt werden, muss ein Block verwendet werden.

Der `else`-Teil kann auch fehlen. Dann bewirkt die bedingte Anweisung nichts, falls die Bedingung *falsch* ist.

Anweisungen

Die bedingte Anweisung

Beispiele:

```
if(ch >= 'A' && ch <= 'Z')
  ch += 32;

if(x > y)
  max = x;
else
  max = y;

if(sum > 100)
{
  cout << sum << endl;
  sum = 0;
}
```

Implizite Umwandlung arithmetischer Bedingungs-ausdrücke nach `bool`.

Die Variable `i` sei z.B. vom Typ `int`.

`if (i)` bedeutet `if (i != 0)`

`if (!i)` bedeutet `if (i == 0)`

Anweisungen

Die bedingte Anweisung

Beispiele:

Seiteneffekte in Bedingungsausdrücken

```
int i, j;

i = 0; j = 2;
if(i++ || j++) i++;
cout << "i == " << i
     << ", j == " << j << endl;

i = 1; j = 2;
if(i++ || j++) i++;
cout << "i == " << i
     << ", j == " << j << endl;

i = 0; j = 2;
if(i++ && j++) i++;
cout << "i == " << i
     << ", j == " << j << endl;

i = 1; j = 2;
if(i++ && j++) i++;
cout << "i == " << i
     << ", j == " << j << endl;
```

Anweisungen

Die bedingte Anweisung

Durch **Einrücken** der zu einem Block gehörenden Anweisungen wird die Lesbarkeit eines Programms erhöht.

Einrücken entweder so:

```
if(Bedingung)
{                       // "{" unter dem "i" des "if"
  Anweisung;
  ...
}                       // "}" unter dem "{"
```

oder so:

```
if(Bedingung) {         // "{" hinter der Bedingung
  Anweisung;
  ...
}                       // "}" unter dem "i" des "if"
```

Ein abschreckendes Beispiel (C-Code):

```
int i;main(){for(;i["]<i;++i){--i;}"];read('-'-'-',i+++"hell\
o, world!\n",'/'/'/'));}read(j,i,p){write(j/p+p,i---j,i/i);}
```

Anweisungen

Die bedingte Anweisung

Oft muss unter mehr als zwei Alternativen ausgewählt werden.

Eine Folge von Tests kann dadurch aufgebaut werden, dass dem `else`-Teil einer vorhergehenden `if`-Anweisung ein weiteres `if` hinzugefügt wird.

Beispiel:

```
if(punkte >= 90)
  note = 1.0;
else if(punkte >= 80)
  note = 2.0;
else if(punkte >= 70)
  note = 3.0;
else if(punkte >= 60)
  note = 4.0;
else
  note = 5.0;
```

Anweisungen

Die bedingte Anweisung

geschachtelte bedingte Anweisungen

Beispiel:

```
if (x == 1)
  if (y == 1)
    cout << "x == y == 1";
else
  cout << "x != 1";
```

```
if (x == 1)
{
  if (y == 1)
    cout << "x == y == 1";
}
else
  cout << "x != 1";
```

Ohne Klammerung gehört ein `else` immer zum letzten `if`, dem kein `else` zugeordnet ist.

Anweisungen

Die bedingte Anweisung

häufige Fehler in Verbindung mit `if`

Beispiele:

```
if (a == b)
  cout << "a ist gleich b";

if (a = b)
  cout << "a ist gleich b";

if (a == b);
  cout << "a ist gleich b";



int i = -1;
unsigned u = 0;

if (u < i)
  cout << u << '<' << i << endl;
```

Anweisungen

Die Fallunterscheidung

Soll unter mehreren Möglichkeiten ein Fall ausgewählt werden, kann man geschachtelte bedingte Anweisungen verwenden.

Beispiel:
Umrechnung der römischen Ziffern in Dezimalwerte

```
char roemischeZiffer;
int dezimalWert;

if(roemischeZiffer == 'I')
  dezimalWert = 1;
else if(roemischeZiffer == 'V')
  dezimalWert = 5;
else if(roemischeZiffer == 'X')
  dezimalWert = 10;
else if(roemischeZiffer == 'L')
  dezimalWert = 50;
else if(roemischeZiffer == 'C')
  dezimalWert = 100;
else if(roemischeZiffer == 'D')
  dezimalWert = 500;
else if(roemischeZiffer == 'M')
  dezimalWert = 1000;
else
  cout << roemischeZiffer
       << " ist keine roemische Ziffer.\n";
```

Anweisungen

Die Fallunterscheidung

Bequemer und besser lesbar ist hier die **Fallunterscheidung**.

```
switch(roemischeZiffer)
{
  case 'I': dezimalWert = 1;
            break;
  case 'V': dezimalWert = 5;
            break;
  case 'X': dezimalWert = 10;
            break;
  case 'L': dezimalWert = 50;
            break;
  case 'C': dezimalWert = 100;
            break;
  case 'D': dezimalWert = 500;
            break;
  case 'M': dezimalWert = 1000;
            break;
  default:  cout << roemischeZiffer
                 << " ist keine "
                 << "roemische Ziffer.\n";
            break;
}
```

Anweisungen

Die Fallunterscheidung

```
switch ( Ausdruck )
{
  case KonstantenAusdruck1 : Anweisungen1
  case KonstantenAusdruck2 : Anweisungen2
    ...
  case KonstantenAusdruckn : Anweisungenn
  default : Anweisungenn+1
}
```

Der Ausdruck nach `switch` wird ausgewertet; er muss von einem ganzzahligen Typ sein (`char`, `short`, `int`, `long` oder `enum`).

Ist das Ergebnis ein Wert, der hinter einem `case` steht, wird die Abarbeitung an dieser Stelle fortgesetzt.

Kommt der Wert nicht hinter einem `case` vor, so werden die Anweisungen nach `default` ausgeführt. Fehlt `default`, hat die Fallunterscheidung dann keine Wirkung.

Die Konstantenausdrücke in den `case`-Konstrukten müssen alle verschiedene Werte liefern, sonst meldet der Compiler einen Fehler.

Anweisungen

Die Fallunterscheidung

Beispiel: Olympische Spiele zwischen 1980 und 2002

```
switch(jahr)
{
  case 1980:
  case 1984:
  case 1988:
  case 1992: cout << jahr <<
               ": Sommer- und Winterspiele";
             break;
  case 1994:
  case 1998:
  case 2002: cout << jahr <<
               ": Winterspiele";
             break;
  case 1996:
  case 2000: cout << jahr <<
               ": Sommerspiele";
             break;
  default:   cout << jahr <<
               ": keine olympischen Spiele";
             break;
}
```

Die Fallunterscheidung

Normalerweise wird die Anweisungsfolge hinter einem `case` mit dem Schlüsselwort **`break`** beendet.

Die Abarbeitung der Fallunterscheidung wird dann beendet und an die Anweisung hinter der schließenden geschweiften Klammer `}` der Fallunterscheidung verzweigt.

Fehlt `break`, werden die nächsten Anweisungen bis zu einem späteren `break` oder dem Ende der Fallunterscheidung der Reihe nach ausgeführt.

Beispiel: (fehlerhaft)

```
roemischeZiffer = 'I';

switch(roemischeZiffer)
{
  case 'I': dezimalWert = 1;
  case 'V': dezimalWert = 5;
  case 'X': dezimalWert = 10;
  case 'L': dezimalWert = 50;
  case 'C': dezimalWert = 100;
  case 'D': dezimalWert = 500;
  case 'M': dezimalWert = 1000;
  default:  cout << roemischeZiffer
                 << " ist keine "
                 << "roemische Ziffer.\n");
}
```

Ausgabe:

```
I ist keine roemische Ziffer
```

Anweisungen

Schleifen

Schleifen sind Kontrollstrukturen, in denen Anweisungen wiederholt ausgeführt werden.

C++ kennt die folgenden Schleifentypen:

- **`while`**-Schleife
- **`do-while`**-Schleife
- **`for`**-Schleife

Die `while`-Schleife

Bei einer `while`-Schleife wird jeweils *vor* der Schleife überprüft, ob sie zu wiederholen ist.

`while (` ***Bedingung*** **`)`**
Anweisung

Die Bedingung muss vom Typ `bool` sein oder in `bool` umgewandelt werden können.

Wenn sie *wahr* ist, d.h. den Wert `true` oder einen Wert ungleich `0` liefert, wird die Anweisung ausgeführt und der `while`-Ausdruck erneut ausgewertet.

Ist die Bedingung *falsch*, wird die Schleife abgebrochen und die Abarbeitung des Programms hinter der `while`-Schleife fortgesetzt.

Anweisungen

Die `while`-Schleife

Beispiel: Quadratwurzel einer Zahl berechnen

```
#include <iostream>
#include <cmath>
using namespace std;

int main()
{
    double zahl;
    cout << " Zahl eingeben: ";
    cin >> zahl;
    while (zahl < 0)
    {
      cout << "Zahl darf nicht negativ sein!";
      cout << endl << "Neue Eingabe: ";
      cin >> zahl;
    }
    cout << "Wurzel aus " << zahl
         << " = " << sqrt(zahl) << endl;
}
```

Das Programm liefert etwa folgenden Dialog:

```
Zahl eingeben: -43.567
Zahl darf nicht negativ sein!
Neue Eingabe: 43.567
Wurzel aus 43.567 = 6.60053
```

Anweisungen

Die `do-while`-Schleife

Bei einer `do-while`-Schleife wird jeweils *am Ende* der Schleife überprüft, ob sie zu wiederholen ist.

```
do
   Anweisung
while ( Bedingung ) ;
```

Nach der Abarbeitung der Anweisung wird die Bedingung ausgewertet; sie muss vom Typ `bool` sein oder in `bool` umgewandelt werden können. .

Wenn sie *wahr* ist, d.h. den Wert `true` oder einen Wert ungleich `0` liefert, wird die Anweisung wiederholt.

Ist die Bedingung *falsch*, wird die Schleife abgebrochen und die Abarbeitung des Programms hinter der `do-while`-Schleife fortgesetzt.

Anweisungen

Die `do-while`-Schleife

Eine `do-while`- Schleife kann stets in eine `while`-Schleife umgeformt werden (und umgekehrt).

```
do
{
  Anweisungen
} while ( Bedingung ) ;
```

entspricht

```
Anweisungen
while ( Bedingung )
{
  Anweisungen
}
```

Anweisungen

Die `for`-Schleife

Will man eine Schleife mit einem Zähler steuern, der von einem Anfangswert mit einer bestimmten Schrittweite bis zu einem Endwert läuft, wird die `for`-Schleife verwendet, die man oft auch *Zählschleife* nennt.

Eine Schleife der Form:

```
for ( Initialisierungs-Ausdruck ;
      Bedingung ;
      Inkrement-Ausdruck )
  Anweisung
```

ist gleichwertig zu folgendem Programmstück (solange nicht `continue` vorkommt):

```
Initialisierungs-Ausdruck ;
while ( Bedingung )
{
  Anweisung
  Inkrement-Ausdruck ;
}
```

Anweisungen

Die `for`-Schleife

Beispiel: Fakultät berechnen

```
#include <iostream>
using namespace std;

int main()
{
    for (int n = 1; n <= 12; n++)
    {
        int produkt = 1;

        for (int i = 1; i <= n; i++)
            produkt *= i;

        cout << n << "! = "
             << produkt << endl;
    }
}
```

Im Initialisierungsteil der `for`-Schleife kann eine Schleifenvariable deklariert werden; sie ist nur in der Schleife gültig.

Anweisungen

Die `for`-Schleife

Der Initialisierungs- und der Inkrement-Ausdruck einer `for`-Schleife können eine durch Komma getrennte Liste von Ausdrücken sein.

Die durch Komma getrennten Ausdrücke werden von links nach rechts ausgewertet.

Beispiel:

Die Reihenfolge der Feldelemente wird umgekehrt. (Aus `{1, 2, 3, 4, 5}` wird `{5, 4, 3, 2, 1}`.)

```
int feld [] = {1, 2, 3, 4, 5};
int length = sizeof(feld) / sizeof(feld[0]);
int temp, i, j;

for(i=0, j=length-1; j>i; i++, j--)
{
  temp = feld[i];
  feld[i] = feld[j];
  feld[j] = temp;
}
```

Anweisungen

Die `for`-Schleife

Jeder der drei-Ausdrücke in einer `for`-Schleife kann auch fehlen; die Semikolons müssen aber trotzdem gesetzt werden.

Werden der *Initialisierungs-Ausdruck* oder der *Inkrement-Ausdruck* ausgelassen, bleibt ihr Platz in der Schleife leer.

Beispiel: zwei (fast) gleichwertige `for`-Schleifen

```
for (int i = 1; i <= n; i++)
  produkt *= i;
```

```
int i = 1;
for ( ; i <= n ; )
{
  produkt *= i;
  i++;
}
```

Fehlt die Bedingung, so wird an ihrer Stelle `true` angenommen.

Beispiel: eine Endlosschleife

```
for(;;) { ... }
```

Anweisungen

Schleifen

Beispiele: Berechnung der Summe 1 + 2 + 3 + ... + n

```
int n = ... ;

int i = 1, sum = 0;
while (i <= n)
{
  sum += i;
  i++;
}

int i = 1, sum = 0;
do
{
  sum += i;
  i++;
} while (i <= n);

int i, sum = 0;
for (i = 1; i <= n; i++)
  sum += i;

int sum = n * (n+1) / 2;
```

Anweisungen

Sprunganweisungen

Anweisungen können mit **Marken** (engl. *label*) versehen werden. Eine Marke ist ein Bezeichner und geht einer Anweisung voran:

Marke **`:`** ***Anweisung***

In C++ gibt es den unbedingten Sprung:

`goto`

Außerdem gibt es die folgenden strukturierten Sprunganweisungen:

- **`break`**
- **`continue`**
- **`return`**

Anweisungen

Sprunganweisungen

`break`

Die **`break`**-Anweisung kann nur innerhalb von Schleifen und Fallunterscheidungen (`switch`) stehen.

Ein `break` beendet die innerste Schleife bzw. Fallunterscheidung, in der das `break` enthalten ist; d. h., es wird an die schließende geschweifte Klammer `}` dieser Kontrollstruktur verzweigt.

`continue`

Die **`continue`**-Anweisung kann nur innerhalb von Schleifen auftreten.

Ein `continue` beendet den aktuellen Schleifendurchlauf der innersten Schleife, in der das `continue` enthalten ist, und springt an die *Wiederholungsbedingung* der Schleife.

In einer `for`-Schleife springt `continue` zum *Inkrementausdruck*.

Anweisungen

Sprunganweisungen

Beispiel: Menü-Steuerung mit `break` und `continue`

```
#include <iostream>
using namespace std;

int main()
{
  char c;
  while(true)
  {
    cout << "Wählen Sie: a, b, x=Ende : ";
    cin >> c;

    if (c == 'a')
    {
      cout << "Programm a\n";
      continue;                // zurück zur Auswahl
    }

    if (c == 'b')
    {
      cout << "Programm b\n";
      continue;                // zurück zur Auswahl
    }

    if (c == 'x')
      break;                   // Schleife verlassen

    cout << "Falsche Eingabe! Bitte wiederholen!\n";
  }
  cout << "\n Programmende mit break\n";
}
```

Anweisungen

Sprunganweisungen

Beispiel: Menü-Steuerung mit Hilfsvariable (Pascal-Stil)

```
#include <iostream>
using namespace std;

int main()
{
    char c;
    bool zuEnde = false;
    while(!zuEnde)
    {
        cout << "Wählen Sie: a, b, x=Ende : ";
        cin >> c;

        switch(c)
        {
          case 'a': cout << "Programm a\n";
                    break;
          case 'b': cout << "Programm b\n";
                    break;
          case 'x': zuEnde = true;
                    break;
          default : cout << "Falsche Eingabe! "
                         << "Bitte wiederholen!\n";
                    break;
        }
    }
    cout << "\n Programmende\n";
}
```

Anweisungen

Sprunganweisungen

Die Sprunganweisung

`goto` *`Marke;`*

verzweigt an diejenige Anweisung, die mit dieser Marke versehen ist.

`goto` sollte sehr sparsam eingesetzt werden.

Es kann z. B. sinnvoll verwendet werden, um eine tief geschachtelte Anweisung zu verlassen. Im folgenden Beispiel können wir `break` nicht verwenden, weil es nur die *innerste* Schleife verlässt.

Beispiel:

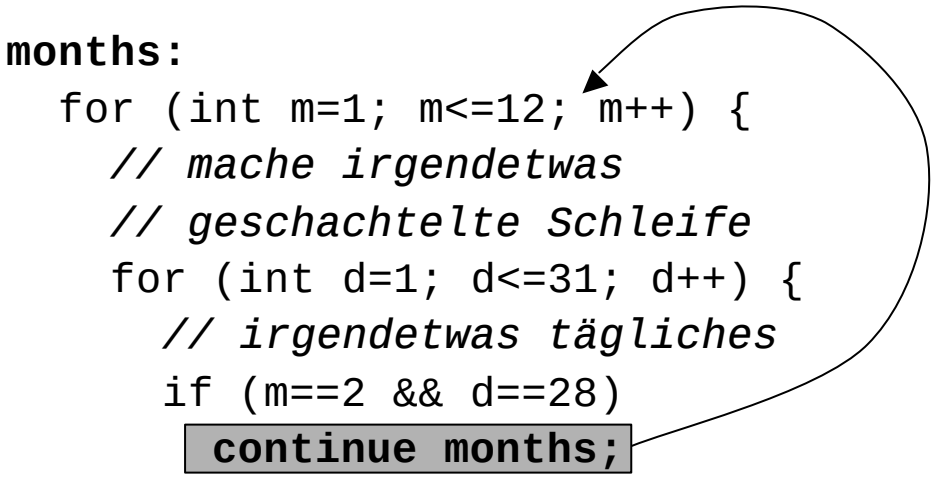
```
for (...)
{
  for (...)
    {
      ...
      if (desaster)
        goto fehler;
    }
}

fehler:
  //aufraeumen
```

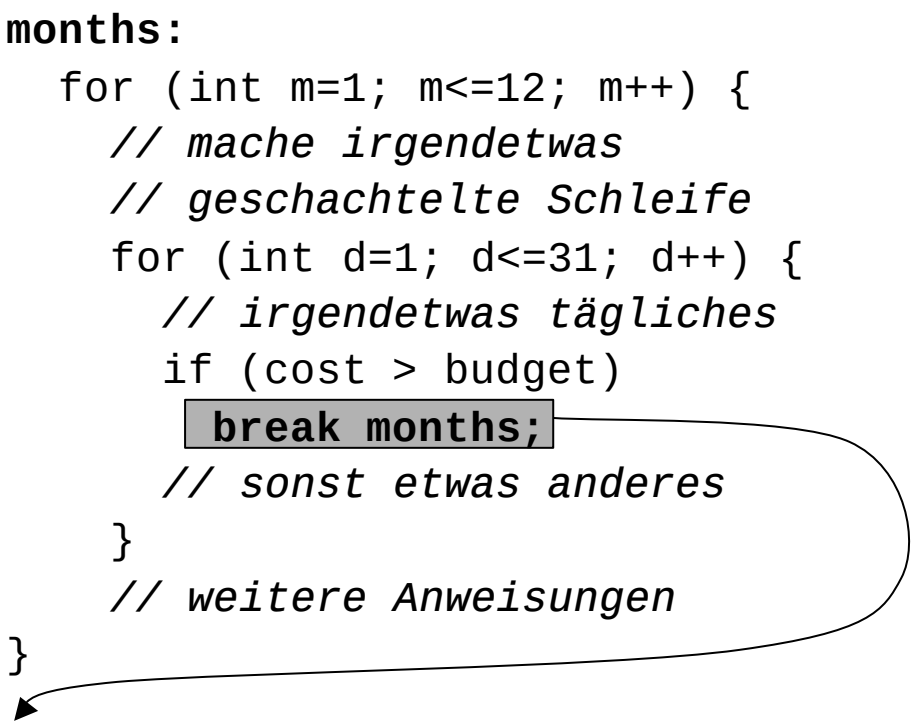
Anweisungen

Sprunganweisungen in Java

continue- und break-Anweisungen mit Marke



```
months:
  for (int m=1; m<=12; m++) {
    // mache irgendetwas
    // geschachtelte Schleife
    for (int d=1; d<=31; d++) {
      // irgendetwas tägliches
      if (m==2 && d==28)
        continue months;
      // sonst etwas anderes
    }
    // weitere Anweisungen
}
```



```
months:
  for (int m=1; m<=12; m++) {
    // mache irgendetwas
    // geschachtelte Schleife
    for (int d=1; d<=31; d++) {
      // irgendetwas tägliches
      if (cost > budget)
        break months;
      // sonst etwas anderes
    }
    // weitere Anweisungen
}
cost = 0;
```

Anweisungen

Sprunganweisungen

`return`

Die **`return`**-Anweisung beendet die Ausführung einer Methode und kehrt zum Aufrufer zurück. Wenn die Methode keinen Wert zurückgibt, wird die einfache Rückgabeanweisung verwendet:

```
return;
```

Verfügt die Methode über einen Rückgabewert, folgt auf `return` ein Ausdruck, der den Rückgabewert festlegt. Falls der Typ des Ausdrucks nicht mit dem Rückgabetyp der Methode übereinstimmt, dann wird eine implizite Typanpassung vorgenommen:

```
return Ausdruck;
```

Beispiel (*nicht* zur Nachahmung empfohlen):

```
char nichtNegativ(double wert)
{
  if(wert < 0)
    return 0;      // eine 'int'- Konstante
  else
    return wert;   // ein 'double'
}
```

Anweisungen

Sprunganweisungen

`return`

Beispiel: Hexadezimalziffer nach `int` konvertieren

```
int hexValue(char ch)
{
  switch(ch)
  {
    case '0': case '1': case '2': case '3':
    case '4': case '5': case '6': case '7':
    case '8': case '9':
      return ch - '0';

    case 'a': case 'b': case 'c':
    case 'd': case 'e': case 'f':
      return ch - 'a' + 10;

    case 'A': case 'B': case 'C':
    case 'D': case 'E': case 'F':
      return ch - 'A' + 10;

    default:
      cout << ch
           << " ist keine Hexadezimalziffer."
           << endl;
      return -1;
  }
}
```