Inhoudsopgave

3 Beslissen en herhalen **1**
3.1 Blok 1
3.2 Beslissen met het `if`-statement 2
3.3 Beslissen met het `if else`-statement 3
3.4 Beslissen met het `switch`-statement 4
3.5 Herhalen met het `while` statement 5
3.6 Herhalen met het `for` statement 6
3.7 Herhalen met het `do while` statement 7
3.8 Extra lusbewerkingen: de `break` en `continue`-statements 8
3.9 Het `goto`-statement 9

Index **11**

3

Beslissen en herhalen

De statements in een C-programma staan onder elkaar geschreven. Na vertaling door de C-compiler worden de statement na elkaar uitgevoerd zoals ze in het C-programma voorkomen. We noemen dat *sequentiële verwerking*. Soms willen we echter dat een reeks van statements wordt uitgevoerd aan de hand van een bepaalde voorwaarde. C biedt een aantal mogelijkheden om dat te beschrijven.

Het is ook mogelijk om een aantal statements herhaaldelijk uit te voeren aan de hand van een bepaalde voorwaarde. C biedt een drietal herhalingsstatements, elk met zijn eigen opzet. Het herhalen zorgt ervoor dat de sequentiële verwerking wordt onderbroken om de serie van statements opnieuw uit te voeren. Binnen een herhaling worden de statement sequentieel uitgevoerd, maar het is natuurlijk mogelijk binnen een herhaling te beslissen of weer te herhalen. Het herhalen van statements wordt een *lus* genoemd (Engels: loop).

3.1 Blok

Een *blok* of *blokstructuur*, in C-terminologie ook wel een *compound statement* genoemd, is een reeks van statements tussen accolades. Voor de C-compiler wordt een blok als één statement gezien. In listing 3.1 is te zien dat binnen het blok een lokale variabele `i` wordt gedeclareerd. Alleen binnen het blok is deze variabele te gebruiken. Buiten het blok bestaat de variabele niet, tenzij de variabele nog op andere plaatsen is gedeclareerd.

```
1 // i does not exist
2 {
3     int i=0;
4
5     i = i + 1;
6     i = i << 2;
7 }
8 // i does not exist
```

Listing 3.1: *Een blok met een lokale variabele.*

3.2 Beslissen met het `if`-statement

Met behulp van het `if`-statement kunnen we een reeks statement uitvoeren aan de hand van een expressie die waar. In listing 3.2 is de algemene gedaante van het `if`-statement te zien.

```
1 if (expressie)
2 {
3     statements
4 }
```

Listing 3.2: *Algemene opzet van het `if`-statement.*

Tussen de haakjes moet een expressie staan. Als de expressie waar is, dan worden de statements die bij de `if` horen uitgevoerd. Een expressie is waar als de uitkomst *ongelijk* aan 0 is. Een expressie is onwaar als de uitkomst gelijk aan 0 is. Voorbeelden van expressies zijn (zonder de haakjes van de `if`):

```
hoogte < 10
(lengte > 2) && (lengte < 80)
(getal < 1) || (getal > 10)
(year % 4 == 0 && year % 100 != 0) || year % 400 == 0
```

Let erop dat een expressie met relationele operator gelijk aan 1 is als de expressie waar is en gelijk aan 0 is als de expressie onwaar is. Dus

```
getal == 25
```

levert een 1 op als de expressie waar is, anders levert de vergelijking een 0 op. Een veel gemaakte fout bij beginnende programmeurs is om de == te schrijven met =. Dit is een toekenning, geen vergelijking.

Een veel gebruikte operator is de negatie-operator `!`. We kunnen deze operator gebruiken zoals te zien is in listing 3.3. De functie `is_date_valid` geeft een 1 als er een geldige datum wordt meegegeven, anders geeft de functie een 0. De negatie-operator draait de waarde van een expressie om, dus een waarde *ongelijk* aan 0 wordt een 0 en een waarde gelijk aan 0 wordt een 1.

```
1  //..
2
3  valid = is_date_valid("2020/03/21");
4
5  if (!valid)     // if date is invalid
6  {
7      printf("Ongeldige datum\n");
8  }
9
10 // ...
```

Listing 3.3: *Gebruik van de negatie-operator.*

3.3 Beslissen met het `if else`-statement

Een `if`-statement mag optioneel een `else`-gedeelte bevatten, zie listing 3.4. Hier worden de $statements_1$ direct onder de `if` uitgevoerd als de expressie waar is en de $statements_2$ onder de `else` worden uitgevoerd als de expressie niet waar is.

```
1  if (expressie)
2  {
3      statements_1
4  }
5  else
6  {
7      statements_2
8  }
```

Listing 3.4: *Algemene opzet van het `if else`-statement.*

Een voorbeeld van het gebruik van `if else` is te zien in listing 3.5. We lezen in regel 9 twee gehele getallen in met behulp van de functie `scanf`. Daarna wordt met behulp van het `if else`-statement bepaald welk getal het grootste is. Als `getal1` groter is dan `getal2` dat kennen we `getal1` toe aan `maximum`. Is `getal1` kleiner dan of gelijk aan `getal2` dan kennen we `getal2` toe aan `maximum`. Daarna drukken we `maximum` af. Hierbij moet worden opgemerkt dat als de variabele gelijk zijn aan elkaar, dat `getal2` aan `maximum` wordt toegekend. Dat is geen probleem om dat de variabelen gelijk zijn aan elkaar.

```
1  #include <stdio.h>
2
3  int main(void)
4  {
5      int getal1, getal2;
6      int maximum;
7
8      printf("Geef␣twee␣gehele␣getallen:␣");
9      scanf("%d␣%d", &getal1, &getal2);
10
11     if (getal1 > getal2)
12     {
13         maximum = getal1;
14     }
15     else
16     {
17         maximum = getal2;
18     }
19
20     printf("Het␣maximum␣=␣%d\n", maximum);
21     return 0;
22 }
```

Listing 3.5: *Voorbeeld van een `if-else`-statement..*

3.4 Beslissen met het `switch`-statement

Het `switch`-statement wordt gebruikt om een expressie te testen op meerdere constante waarden. De *expressie* na `switch` is meestal een variabele maar er mag ook een expressie staan. Binnen de `switch` wordt de expressie vergeleken met één of meerdere constante expressies met behulp van het `case`-statement. Is de expressie gelijk aan een constante expressie, dan wordt het statement dat bij de `case` hoort uitgevoerd. Als geen van de constante expressies gelijk is aan de expressie na de `switch`, wordt het statement dat bij `default` hoort uitgevoerd. Het `default`-statement mag achterwege blijven. De algemene opzet van het `switch`-statement is te zien in listing 3.6. Het statement achter `case` hoeft niet tussen accolades gezet te worden.

```
1 switch (expressie)
2 {
3     case constante-expressie: statement
4     case constante-expressie: statement
5     case constante-expressie: statement
6     // ...
7     default: statement
8 }
```

Listing 3.6: *Opzet van het `switch`-statment.*

De waarden bij meerdere `case`-statement kunnen worden samengevoegd, zodat dezelfde statements worden uitgevoerd bij meerdere mogelijkheden. We noemen dit een *fall through*. Een voorbeeld is te zien in listing 3.7. In het `switch`-statement wordt `getal` vergeleken met verschillende waarden. Als `getal` de waarde 1, 2 of 3 heeft, wordt variabele `letter` op `'a'` gezet. Het `break`-statement in regel 6 zorgt ervoor dat het `switch`-statement direct wordt verlaten. Als we die zouden weglaten, worden de statements die horen bij de volgende `case` uitgevoerd. In regel 9 worden de statements uitgevoerd als geen van de constante expressies achter de `case`-statement gelijk is aan `getal`. Het `break`-statement in regel 10 is logisch gezien overbodig, maar het is *good practice* om deze wel op te nemen. De volgorde van `case`s en `default` is namelijk willekeurig, dus na `default` zou nog een `case` mogen komen. Het is wederom *good practice* om alle `case`s voor de `default` te plaatsen.

```
1  switch (getal)
2  {
3      case 1:
4      case 2:
5      case 3:  letter = 'a';
6               break;
7      case 4:  letter = 'b';
8               break;
9      default: letter = 'c';
10              break;
11 }
```

Listing 3.7: *Opzet van het `switch`-statment.*

3.5 Herhalen met het `while` statement

Een `while`-statement wordt gebruikt als een reeks statements 0 of meer keer moet worden uitgevoerd. De opzet van het `while`-statement is te zien in listing 3.8. Als *expressie* waar is (een uitkomst ongelijk aan 0) dan worden de statements binnen de accolades uitgevoerd. Nadat de statements zijn uitgevoerd, wordt weer opnieuw begonnen met het uitrekenen van *expressie*. Is *expressie* (weer) waar, dan worden de statements nogmaals uitgevoerd. Is *expressie* niet waar (de uitkomst is gelijk aan 0) dan wordt gesprongen naar het statement na de accolade-sluiten. Het kan zijn dat *expressie* bij de allereerste keer niet waar is. Dan worden de statements binnen de `while` niet uitgevoerd.

```
1 while (expressie)
2 {
3     statement
4 }
```

Listing 3.8: *Opzet `while`-statement.*

We leggen de werking van het `while`-statement uit aan de hand van een voorbeeld, te zien in listing . In regel 8 wordt gevraagd om een positief getal in te voeren. In regel 10 wordt het `while`-statement uitgevoerd door de expressie `getal <= 0` uit te rekenen. Is de expressie niet waar dan wordt direct naar regel 16 gesprongen en de statement binnen de `while` worden niet uitgevoerd. Is het getal kleiner dan of gelijk aan 0 dan worden de statements binnen de `while` uitgevoerd. Daar wordt weer gevraagd om een positief getal in te voeren. Vervolgens wordt de expressie opnieuw uitgerekend. Zolang de expressie niet waar is worden de statements binnen de `while` uitgevoerd.

```
1  #include <stdio.h>
2
3  int main(void)
4  {
5      int getal;
6
7      printf("Geef␣een␣positief␣getal:␣");
8      scanf("%d", &getal);
9
10     while (getal <= 0)
11     {
12         printf("Helaas!␣Geef␣een␣positief␣getal:␣");
13         scanf("%d", &getal);
14     }
15
16     printf("Het␣ingevoerde␣getal␣=␣%d\n", getal);
17     return 0;
18 }
```

Listing 3.9: *Voorbeeld van een `while`-statement..*

3.6 Herhalen met het `for` statement

Het `for`-statement gebruiken we als het aantal herhalingen bekend en *eindig* is. Het `for`-statement heeft de gedaante zoals te zien in in listing 3.10. Vóór de lus wordt $statement_1$ uitgevoerd. Bij het herhalen van de lus wordt *expressie* uitgevoerd. Als dat waar is worden de statements in $statements_3$ uitgevoerd. Voor het einde van de lus, net voor de accolade-sluiten, wordt $statement_2$ uitgevoerd. Let erop dat $statement_2$ ná $statement_3$ wordt uitgevoerd. Verder mag vanaf C99 $statement_1$ een declaratie van een variabele bevatten. De variabele is dan alleen te gebruiken binnen het `for`-statement. Let erop dat $statement_1$, *expressie* en $statement_2$ van elkaar gescheiden zijn met een punt-komma.

```
for (statement1; expressie; statement2)
{
    statement3
}
```

Listing 3.10: *Opzet van het `for`-statement.*

Normaal gesproken wordt in $statement_1$ een variabele op een beginwaarde gezet. De *expressie* wordt berekend zoals dat in het `while`-statement ook gebeurt, en $statement_2$ wordt normaal gesproken gebruikt om een variabele aan te passen voor de volgende lusdoorgang. Meestal betekent dit dat de variabele wordt verhoogd of verlaagd.

Stel dat we de tafels van 1 t/m 10 willen afdrukken. We maken dan gebruikt van twee, in elkaar verweven, `for`-statements. Dit is te zien in listing 3.11. We concentreren ons op de opzet van beide `for`-statements. Er zijn diverse `printf`-statements te zien die de gegevens op een nette manier afdrukken.

```
#include <stdio.h>

int main(void)
{
    printf("  |   1   2   3   4   5   6   7   8   9  10\n");
    printf("--+---------------------------------------\n");

    for (int i = 1; i < 11; i =i + 1)
    {
        printf("%2d|", i);
        for (int j = 1; j < 11; j = j + 1)
        {
            printf("%4d", i*j);
        }
        printf("\n");
    }

    return 0;
}
```

Listing 3.11: *Een for-list.*

De buitenste `for`-statement begint op regel 9, eindigt op regel 17 en declareert variabele `i`. De variabele loopt van 1 tot 11, dus 11 zelf doet *niet* mee. Binnen deze lus gebruiken we een tweede `for`-statement die begint op regel 12 en eindigt op regel 15. Dit keer gebruiken we variabele `j` die in regel 12 wordt gedeclareerd. Ook deze variabele loopt van 1 tot 11 (dus 11 doet niet mee). In regel 14 drukken we het product af van `i` en `j`. De uitvoer van dit programma is te zien in figuur 3.1.

Command Prompt

```
   |   1   2   3   4   5   6   7   8   9  10
---+-----------------------------------------
 1|   1   2   3   4   5   6   7   8   9  10
 2|   2   4   6   8  10  12  14  16  18  20
 3|   3   6   9  12  15  18  21  24  27  30
 4|   4   8  12  16  20  24  28  32  36  40
 5|   5  10  15  20  25  30  35  40  45  50
 6|   6  12  18  24  30  36  42  48  54  60
 7|   7  14  21  28  35  42  49  56  63  70
 8|   8  16  24  32  40  48  56  64  72  80
 9|   9  18  27  36  45  54  63  72  81  90
10|  10  20  30  40  50  60  70  80  90 100
```

Figuur 3.1: *Uitvoer van het programma in listing 3.11.*

Het `for`-statement uit listing 3.10 is *semantisch equivalent*[1] aan:

```
1 statement1
2 while (expressie)
3 {
4     statement3
5     statement2
6 }
```

Listing 3.12: *`while`-statement als `for`-statement.*

We kunnen in het `for`-statement de *expressie* achterwege laten. Dan wordt de lus eeuwigdurend uitgevoerd.

```
for (;;) { ... }   // do forever
```

3.7 Herhalen met het `do while` statement

We gebruiken het `do while`-statement als we statements 1 of meer keer willen uitvoeren. Merk op dat de lus dus minstens één keer wordt uitgevoerd. Voor een opzet, zie listing 3.13. Eerst wordt *statement* uitgevoerd en daarna wordt *expressie* uitgerekend. Als *expressie* waar is, wordt de lus nog een keer uitgevoerd, anders wordt de lus verlaten.

[1] Semantisch equivalent wil zeggen dat de betekenis werkingen gelijk zijn aan elkaar. De twee opzetten zijn *niet* syntactisch equivalent. Ze hebben immers een andere *syntax*.

```
do
{
    statement
}
while (expressie)
```

Listing 3.13: *Opzet `while`-statement.*

We kunnen het programma in listing 3.9 ook schrijven met behulp van een `do while`-statement. Zie listing 3.14. De lus wordt minstens één keer uitgevoerd. Als na invoer van getal blijkt dat de expressie `getal <= 0` waar is, wordt de lus nog een keer uitgevoerd, net zolang totdat de expressie niet waar is.

```
#include <stdio.h>

int main(void)
{
    int getal;

    do
    {
        printf("Geef een positief getal: ");
        scanf("%d", &getal);
    }
    while (getal <= 0);

    printf("Het ingevoerde getal = %d\n", getal);
    return 0;
}
```

Listing 3.14: *Een for-list.*

3.8 Extra lusbewerkingen: de `break` en `continue`-statements

We kunnen een lus voortijdig verlaten met een `break`-statement. Dit is te zien in listing 3.15. De lus wordt vormgegeven door een eeuwig durend `while`-statement (regel 10). We vragen de gebruiker om een positief getal of 0 in te voeren. Als het getal gelijk is aan 0, wordt de lus verlaten met het `break`-statement in regel 16. Alleen de lus waar het `break`-statement bijhoort kan worden verlaten. Als we een lus binnen een lus gebruiken kan dus alleen de binnenste lus worden verlaten.

Met het `continue`-statement kunnen we binnen een lus springen naar de volgende doorloop van de lus. Bij het `while`-statement wordt dus gesprongen naar het begin van de lus, bij het `do while`-statement wordt gesprongen naar het einde van de lus. In beide gevallen wordt de expressie die bij de lus hoort opnieuw uitgerekend.

Bij het `for`-statement werkt het iets anders. Daarvoor moeten we listing 3.10 nog eens bekijken. Aan het einde van het `for`-statement wordt $statement_2$ uitgevoerd, waarin normaal gesproken een variabele wordt aangepast (verhogen of verlagen). Een `continue`

zorgt ervoor dat naar dit statement wordt gesprongen. Overigens kunnen we de code in listing 3.15 geheel herschrijven zonder `break` en `contine`. Zie listing 3.16.

```
#include <stdio.h>

#pragma warning(disable : 4996)

int main(void)
{
    int som = 0;
    int i;

    while (1)
    {
        printf("Geef␣positief␣getal␣of␣0␣om␣te␣stoppen:␣");
        scanf("%d", &i);
        if (i == 0)
        {
            break;
        }
        if (i < 0)
        {
            continue;
        }
        som = som + i;
    }
    printf("De␣som␣is␣%d\n", som);
}
```

Listing 3.15: *Gebruik van het `break`-statement om een lus te verlaten.*

We gebruiken `break` en `continue` als de code binnen de lus complex is met vele in elkaar verweven lussen en testen.

3.9 Het `goto`-statement

Eerst even een opmerking:

> Goto's are the root of all evil.

Er is geen enkele reden om het `goto`-statement te gebruiken. Het is altijd mogelijk om code te schrijven zonder het `goto`-statement. In dit boek maken er ook geen gebruik van. Soms is het echter handig om er gebruik van te maken, bijvoorbeeld als een foutconditie op meerdere plekken in de code kan voorkomen of als we uit een, wat in het Engels *deeply nested structure* genoemd wordt, willen breken, bijvoorbeeld een lus binnen een lus. Een voorbeeld is te zien in listing 3.17.

```c
#include <stdio.h>

#pragma warning(disable : 4996)

int main(void)
{
    int som = 0;
    int i;

    do
    {
        printf("Geef␣positief␣getal␣of␣0␣om␣te␣stoppen:␣");
        scanf("%d", &i);
        if (i > 0)
        {
            som = som + i;
        }
    } while (i != 0);
    printf("De␣som␣is␣%d\n", som);
}
```

Listing 3.16: *Sommeren van positieve getallen.*

```c
for ( ... )
{
    for ( ... )
    {
        if (panic)
        {
            goto error;
        }
    }
}
error:
    // clean up the mess
```

Listing 3.17: *Gebruik van het `goto`-statement.*

Index

B
blok, 1
blokstructuur, 1
`break`, keyword, 4

C
`case`, keyword, 4
compound statement, 1
`continue`, keyword, 8

D
`default`, keyword, 4
`do`, keyword, 7

E
`else`, keyword, 3

F
`for`, keyword, 6

I
`if`, keyword, 2

L
loop, 1
lus, 1

N
negatie-operator, 2

S
`switch`, keyword, 4

W
`while`, keyword, 5