C

De programmeertaal C – een overzicht

Dictaat

Versie 0.1

J.Z.M. Broeders

Dictaat De programmeertaal C – een overzicht

Versie 0.1

J.Z.M. Broeders

De LaTeX-code van dit dictaat kun je vinden op: `https://bitbucket.org/HR_ELEKTRO/cprog/`.

Inhoudsopgave

Inleiding

Dit dictaat geeft een overzicht van de programmeertaal C. Daarbij wordt er vanuit gegaan dat je de basisbeginselen van programmeren in C al kent. Details betreffende de programmeertaal C worden in dit dictaat niet gegeven. Die zijn prima te vinden op het internet. Indien nodig verwijzen we hiernaar door middel van een link waarmee je, als je dit dictaat leest op een device met internettoegang, meteen de detailinformatie kunt openen.

Er zijn verschillende versies van de C-standaard. In dit dictaat wordt gebruik gemaakt van C11[3].

Alle programmacode in dit dictaat is getest met de open source C-compiler GCC[1] versie 5.3.0. Op https://bitbucket.org/HR_ELEKTRO/cprog/src/master/progs/ vind je de source code van alle in dit dictaat besproken programma's. Als je dit dictaat leest op een device met internettoegang, dan kun je op de bestandsnamen in dit dictaat klikken om de programma's te downloaden.

Veel literatuur over C-programmeren is geschreven in het Engels. Om er voor te zorgen dat je deze literatuur tijdens het lezen van dit dictaat eenvoudig kunt gebruiken heb ik veel van het jargon dat in deze literatuur wordt gebruikt niet vertaald naar het Nederlands. De keuze om een term wel of niet te vertalen is arbitraal. Zo heb ik bijvoorbeeld het Engelse begrip "return type" niet vertaalt in het Nederlandse "terugkeertype" of "retourtype", maar heb ik het Engelse begrip "parameter type" wel vertaald naar het Nederlandse "parametertype".

Op- en aanmerkingen zijn altijd welkom. Je kunt me mailen op `mailto:J.Z.M.Broeders@hr.nl` of nog beter: maak een Issue aan op `https://bitbucket.org/HR_ELEKTRO/cprog/issues`.

[1] De GNU Compiler Collection (GCC) bevat een zeer veel gebruikte open source C-compiler. Zie `http://gcc.gnu.org/`

1

Onthouden

In een computerprogramma moeten gegevens (Engels: data) vaak tijdelijk opgeslagen worden. Als in een programma bijvoorbeeld iets geteld moet worden, dan moet de waarde van de teller ergens opgeslagen worden. Je kunt in de programmeertaal C data tijdelijk bewaren in een *variabele*. De waarde van zo'n variabele wordt opgeslagen in het zogenoemde *werkgeheugen*. Bij de meeste computers is het werkgeheugen uitgevoerd als zogenoemd RAM[2]-geheugen waarbij elk groepje van 8 bits (een byte) een adres heeft gekregen. De plaats (het adres) waar bepaalde data wordt opgeslagen hoef je niet te weten en wil je ook niet weten. In plaats van het adres gebruik je daarom een naam (Engels: *identifier*) om aan te geven welke variabele je bedoelt. Zo'n identifier is in de programmeertaal C gebonden aan een aantal regels[3]:

- Een identifier mag alleen uit letters, cijfers en underscores (_) bestaan.
- Een identifier mag niet beginnen met een cijfer.
- Bepaalde sleutelwoorden (Engels: keywords) hebben een specifieke betekenis in de programmeertaal C en die mogen niet als identifier gebruikt worden[4]. Bijvoorbeeld: `if` en `else`.

Een variabele heeft in de programmeertaal C een *naam*, een *type* en een *waarde*. Het type van een variabele geeft aan wat voor soort data in de variabele kan worden opgeslagen.

1.1 Basistypes

De programmeertaal C kent een aantal zogenoemde *basistypes*. In dit dictaat maak je gebruik van de volgende basistypes[5]:

[2] De afkorting RAM staat voor Random Access Memory. Dit betekent dat elk gegeven dat in dit geheugen is opgeslagen even snel kan worden benaderd.

[3] De gedetailleerde regels kun je vinden op: `http://en.cppreference.com/w/c/language/identifier`.

[4] De complete lijst kun je vinden op: `http://en.cppreference.com/w/c/keyword`.

[5] Een complete lijst met beschikbare basistypes kun je vinden op: `http://en.cppreference.com/w/c/language/arithmetic_types`.

- `bool`
  In een variabele van dit type kan een Booleaanse waarde **`true`** of **`false`** worden opgeslagen. Dit type is toegevoegd aan de programmeertaal C in de C99 standaard. Om deze reden moet je in een programma waar je dit type wilt gebruiken de regel:
  `#include <stdbool.h>`
  opnemen.
- **`char`**
  In een variabele van dit type kan een karakter worden opgeslagen. Zo'n variabele wordt opgeslagen in één byte. Een constante waarde van dit type wordt opgegeven tussen enkele aanhalingstekens. Bijvoorbeeld `'Y'` of `'7'`. Er zijn ook een aantal speciale karakters gedefinieerd. Deze worden voorafgegaan door een backslash. Bijvoorbeeld `'\n'` staat voor het nieuweregelteken (enter), `'\t'` staat voor het tabulatieteken (tab) en `'\''` staat voor het enkele aanhalingsteken[6].
- **`int`**
  In een variabele van dit type kan een geheel getal (Engels: integer) worden opgeslagen. In de C standaard is niet gedefinieerd hoeveel bytes zo'n variabele in beslag neemt. Compilers voor 32- of 64-bit processors gebruiken vrijwel altijd vier bytes (32 bits) voor een variabele van het type **`int`**. Zo'n variabele heeft een minimale waarde van $-2^{31} = -2147483648$ en een maximale waarde van $2^{31} - 1 = 2147483647$. Compilers voor 8- of 16-bit processors daarentegen gebruiken vrijwel altijd twee bytes (16 bits). Zo'n variabele heeft een minimale waarde van $-2^{15} = -32768$ en een maximale waarde van $2^{15} - 1 = 32767$. De programmeertaal C heeft geen beveiliging tegen over- en underflow. Als je bij een variabele van het type **`int`** met de maximaal mogelijke waarde één optelt, dan krijgt deze variabele de minimaal mogelijke waarde.
- **`float`** en **`double`**
  In een variabele van een van deze types kan een floating-point getal[7] worden opgeslagen. Floating-point getallen gebruiken een vast aantal significante bits (de significant) die worden geschaald door een exponent. Het getal 1234.56789 kan worden genoteerd als 1.23456789×10^3 en ook als 123456789×10^{-5}. Zoals je kunt zien, kan de positie van de punt binnen het getal 'zweven' door de waarde van de exponent aan te passen. In computersystemen wordt de IEEE754-standaard [2] voor floating-point nummers bijna altijd gebruikt. Deze standaard definieert verschillende formaten[8]. Bijvoorbeeld enkele precisie (die wordt gebruikt om het type **`float`** in de C-programmertaal te implementeren) en dubbele precisie (die wordt gebruikt om het type **`double`** te implementeren in C). Een variabele van het type **`float`** wordt opgeslagen in vier bytes (32 bits) en een variabele van het type **`double`** wordt, zoals de naam al aangeeft, opgeslagen in het dubbele aantal bytes (acht bytes oftewel 64 bits). Een aantal van deze bits worden gebruikt voor de significant en een aantal voor de exponent. Het

[6] In het Engels wordt dit een escape sequence genoemd. De complete lijst kun je vinden op: `https://en.cppreference.com/w/c/language/escape`.

[7] In correct Nederlands: drijvendekommagetal of zwevendekommagetal. In het Nederlands wordt namelijk een komma gebruikt om de eenheden te scheiden van de tienden. In het Engels en ook in de programmeertaal C wordt hiervoor, in plaats van de komma, de punt gebruikt. Om deze reden wordt in dit dictaat de Engelse term floating-point gebruikt.

[8] Een volledige lijst kun je vinden op: `https://en.wikipedia.org/wiki/IEEE_754`.

bereik van het type **float** is $\pm 3.4 \times 10^{\pm 38}$. Dit type heeft 24 significante bits, wat overeenkomt met ongeveer 7 significante decimale cijfers. Een **double** heeft een bereik van $\pm 1.7 \times 10^{\pm 308}$ en heeft 53 significante bits, wat overeenkomt met ongeveer 15 significante decimale cijfers[9]. Omdat het aantal bits waarin een floating-point getal wordt opgeslagen beperkt is, kan voor vele reële getallen zoals $\frac{1}{2}$ en $\sqrt{2}$ alleen een benadering in het geheugen worden opgeslagen. De floating-point weergave maakt het mogelijk om een breed dynamisch bereik van waarden bestrijken met een constant aantal significante bits.

Verderop in dit dictaat zal in hoofdstuk 8 worden uitgelegd hoe je van deze basistypes ingewikkeldere types kunt afleiden (Engels: derived types).

Op dit moment willen we je eerst enkele voorbeeldprogramma's laten zien waarin variabelen van de zojuist beschreven basistypes gebruikt worden. Omdat een programma waarin alleen variabelen aangemaakt worden niet erg spannend zijn, maken we in deze programma's ook gebruik van de functie `printf` om de uitvoer van het programma weer te geven (zie paragraaf 2.1) en van operatoren om bewerkingen op de variabelen uit te voeren (zie paragraaf 3.1).

1.1.1 Een voorbeeldprogramma met `int`'s

De somformule van Gauss[10] is een formule om de som van de eerste n opeenvolgende natuurlijke getallen te bepalen:

$$1+2+3+4+\cdots+n=\sum_{k=1}^{n}k=\frac{n(n+1)}{2}. \tag{1.1}$$

In listing 1.1 is een programma gegeven dat de som van 1 tot en met 100 stap voor stap (met een **for**-loop, zie paragraaf 4.3) en met behulp van de somformule van Gauss berekent.

De uitvoer van dit programma is zoals verwacht:

```
som = 5050
formule = 5050
```

Programmeren is een vaardigheid waarbij de details vaak erg belangrijk zijn. In het vervolg van deze paragraaf geven we hier enkele voorbeelden van.

Als je in listing 1.1 de waarde `100` vervangt door de waarde `50000` (zie `int_error1.c`), dan geeft het programma de volgende uitvoer:

[9] See: `https://en.cppreference.com/w/c/language/arithmetic_types`.

[10] Deze somformule is al heel lang bekend, maar is op basis van een anekdote naar de beroemde Duitse wiskundige Carl Friedrich Gauss genoemd. Diens leraar op de basisschool zou zijn leerlingen een tijdje bezig willen hebben houden door hen de gehele getallen van 1 tot en met 100 te laten optellen. De jonge Gauss zou het juiste antwoord echter binnen een paar seconden hebben gegeven, dit tot verbazing van zijn leraar. Gauss besefte, ervan uitgaand dat de op te tellen gehele getallen van 1 tot en met 100 liepen, dat paarsgewijze optelling van 'tegenoverliggende' getallen identieke tussenresultaten oplevert: $1+100=101$, $2+99=101$, $3+98=101$ enzovoort, de totale som bedraagt dan $50\times 101=5050$. Bron: Wikipedia.

```c
#include <stdio.h>

// toon aan dat som van 1 + 2 + 3 + 4 + ... + n gelijk is aan (n + 1) * n / 2
int main(void)
{
    int n = 100;
    int som = 0;
    for (int getal = 1; getal <= n; getal = getal + 1)
    {
        som = som + getal;
    }
    printf("som = %d\n", som);
    int formule = (n + 1) * n / 2;
    printf("formule = %d\n", formule);
    return 0;
}
```

Listing 1.1: Het op twee manieren berekenen van de som van 1 tot en met 100, zie `int.c`.

```
som = 1250025000
formule = -897458648
```

Wat is hier aan de hand? Is de somformule van Gauss niet correct? Op pagina 4 heb je gelezen dat de programmeertaal C geen beveiliging heeft tegen overflow. De maximale waarde die in een variabele van het type **int** kan worden opgeslagen is in dit geval[11] blijkbaar $2^{31} - 1 = 2147483647$. Als je de som van alle getallen van 1 tot en met 50000 stap voor stap bepaalt, dan wordt de maximale waarde van de variabele `som` gelijk aan `1250025000`. Deze waarde past in een 32-bit integer variabele, want $1250025000 \leq 2147483647$. Als je de somformule van Gauss uitrekent dan bereken je `(n + 1)* n`. Als `n` de waarde `50000` heeft dan is dit product gelijk aan `50001 * 50000 = 2500050000`. Deze waarde past echter *niet* in een 32-bit integer variabele, want $2500050000 > 2147483647$. Het tekenbit van het resultaat wordt overschreven bij het berekenen van het product en daarom is het resultaat negatief, namelijk -1794917296[12]. Er is een zogenoemde overflow opgetreden, maar de programmeertaal C geeft hier geen foutmelding of waarschuwing voor.

Als programmeur moet je er dus zelf voor zorgen dat de variabelen die je gebruikt genoeg bits bevatten om de groots mogelijke waarde van die variabele in op te slaan. In paragraaf 1.4 zullen je zien hoe je de grootte van een integer variabele kunt bepalen door het juiste type te gebruiken.

[11] Compilers voor 32- of 64-bit processors gebruiken vrijwel altijd vier bytes (32 bits) voor een variabele van het type **int**. Compilers voor 8- of 16-bit processors daarentegen gebruiken vrijwel altijd twee bytes (16 bits).

[12] Het antwoord is $10010101000000111011110001010000_2$. In het two's complement stelsel is dit getal negatief (bit 31 is gelijk aan 1) en kun je de waarde bepalen door het te inverteren en er één bij op te tellen: $-01101010111111000100001110110000_2 = -1794917296_2$.

Als je het programma uit listing 1.1 aanpast zodat het de som van 1 tot en met 100000 berekent, dan kan het programma het correcte antwoord (5000050000) *niet* weergeven omdat dit antwoord groter is dan $2^{32}-1$ en dus niet in een variabele van het type `int` past.

Als je in listing 1.1 de waarde `100` vervangt door de waarde `100000`, dan geeft het programma (zie `int_error2.c`) de volgende foutieve[13] uitvoer:

```
som = 705082704
formule = 705082704
```

Een programmeur die het programma toch correct wil laten werken als n gelijk is aan 50000, komt op het idee om de berekening te wijzigen in `(n + 1)* (n / 2)`. Als je in listing 1.1 de waarde `100` vervangt door de waarde `50000` en de berekening `(n + 1)* n / 2` vervangt door `(n + 1)* (n / 2)`, dan geeft het programma de volgende uitvoer:

```
som = 1250025000
formule = 1250025000
```

Dit lijkt een goede aanpassing van het programma. Maar pas op, als je in dit programma de waarde `50000` vervangt door de waarde 7, dan geeft het programma (zie `int_error3.c`) de volgende foutieve uitvoer:

```
som = 28
formule = 24
```

Wat is hier aan de hand? In hoofdstuk 3 zul je leren dat een integer deling een integer resultaat oplevert. Dus de berekening `7 / 2` levert de waarde `3` als resultaat (en dus niet `3.5`). De berekening `(7 + 1)* (7 / 2)` levert dus de waarde `8 * 3` = `24` en dat is niet gelijk aan de som van 1 tot en met 7 (want dat is `28`).

Dit soort details zorgt ervoor dat het correct schrijven van een programma lastig is. Het is dus erg belangrijk om elk programma wat je schrijft grondig te testen!

1.1.2 Een voorbeeldprogramma met `double`'s

Waarschijnlijk heb je wel eens gehoord van de Fibonacci-getallen[14]. De eerste twee Fibonacci-getallen F_0 en F_1 zijn respectievelijk 0 en 1. Vervolgens is elk volgend Fibonacci-getal steeds de som van de twee voorgaande Fibonacci-getallen. Dus $F_2 = F_1 + F_0 = 1 + 0 = 1$, $F_3 = F_2 + F_1 = 1 + 1 = 2$, $F_3 = F_2 + F_1 = 2 + 1 = 3$, enz. De definitie voor het bereken van F_n is:

$$F_n = \begin{cases} 0 \text{ voor } n = 0; \\ 1 \text{ voor } n = 1; \\ F_{n-1} + F_{n-2} \text{ voor } n > 1. \end{cases} \tag{1.2}$$

[13] Het juiste antwoord is $5000050000_{10} = 1001010100000011010110101010100000_2$. Maar dit getal past niet in 32 bits. De laagste 32 bits worden opgeslagen $00101010000001101011010101010000_2 = 705082704_{10}$.

[14] Zie: `https://nl.wikipedia.org/wiki/Rij_van_Fibonacci`.

Het n^{de} Fibonacci getal kan ook uitgerekend worden met de formule van Binet[15]:

$$F_n = \frac{(1+\sqrt{5})^n - (1-\sqrt{5})^n}{2^n\sqrt{5}}. \tag{1.3}$$

Dit is opmerkelijk omdat F_n een geheel getal is maar in de formule van Binet komt het (niet gehele) getal $\sqrt{5}$ voor.

In listing 1.1 is een programma gegeven dat F_n stap voor stap (met een **while**-loop, zie paragraaf 4.1) en met behulp van de formule van Binet berekent.

Om de formule van Binet te berekenen heb je een variabele nodig waarin een floating-point getal kan worden opgeslagen. In dit geval is gekozen voor het type **double**, zie pagina 4. Om de wortels en de machten uit te rekenen wordt gebruik gemaakt van twee functies `sqrt` en pow uit de standaard C library, zie paragraaf 3.2.

```
#include <stdio.h>
#include <math.h>

// toon aan dat Fi (het i-de Fibonacci getal) gelijk is aan ↩
 ↪ formule van Binet
int main(void)
{
    int n = 40;
    int i = 2;
    int Fi_min_2 = 0, Fi_min_1 = 1, Fi;
    while (i <= n)
    {
        Fi = Fi_min_1 + Fi_min_2;
        Fi_min_2 = Fi_min_1;
        Fi_min_1 = Fi;
        i = i + 1;
    }
    printf("F%d = %d\n", n, Fi);
    double formule = (pow((1 + sqrt(5)) / 2, n) - pow((1 - ↩
     ↪ sqrt(5)) / 2, n)) / sqrt(5);
    printf("formule = %.0f\n", formule);
    return 0;
}
```

Listing 1.2: Het op twee manieren berekenen van F_{40}, zie `double.c`.

De uitvoer van dit programma is zoals verwacht:

```
F40 = 102334155
formule = 102334155
```

[15] Zie: `https://en.wikipedia.org/wiki/Fibonacci_number`

1.1.3 Een voorbeeldprogramma met bool's

De wetten van De Morgan[16] zijn bekende wetten uit de booleaanse algebra:

$$\overline{a} + \overline{b} = \overline{a \cdot b} \tag{1.4}$$

$$\overline{a} \cdot \overline{b} = \overline{a + b} \tag{1.5}$$

In deze formules zijn a en b booleaanse variabelen en staat de operator $+$ voor de logische OF-operatie en staat de operator $\cdot$ voor de logische EN-operatie. Een streep boven een expressie staat voor de logische NIET-operatie.

In listing 1.3 is een programma gegeven dat een waarheidstabel produceert waarmee de wetten van De Morgan bewezen worden. In dit programma wordt gebruik gemaakt van twee **do**-loops, zie paragraaf 4.2, om de verschillende combinaties van de waarden van a en b af te lopen.

Het programma berekent voor elke mogelijke waarde van a en b de volgende booleaanse variabelen:

$$c = \overline{a} + \overline{b} \tag{1.6}$$

$$d = \overline{a \cdot b} \tag{1.7}$$

$$e = \overline{a} \cdot \overline{b} \tag{1.8}$$

$$f = \overline{a + b} \tag{1.9}$$

Om de booleaanse waarden op te slaan hebben je variabelen nodig van het type `bool`, zie pagina 4.

Om booleaanse formules vergelijkingen (1.6) tot (1.9) in een C-programma te berekenen heb je booleaanse operatoren nodig, zie paragraaf 3.1. De OF-operatie wordt in C aangegeven met de `||`-operator. De EN-operatie wordt aangegeven met de `&&`-operator en de NIET-operatie wordt aangegeven met de `!`-operator.

De booleaanse variabelen worden geprint met behulp van `printf` door gebruik te maken van `%d`. De booleaanse waarde **`false`** wordt dan omgezet naar de integer waarde `0` en **`true`** wordt omgezet naar `1`.

De uitvoer van dit programma is zoals verwacht:

```
a b |!a+!b|!(a.b)|!a.!b|!(a+b)
----+-----+------+-----+------
0 0 |  1  |  1   |  1  |  1
0 1 |  1  |  1   |  0  |  0
1 0 |  1  |  1   |  0  |  0
1 1 |  0  |  0   |  0  |  0
```

[16] Zie: `https://nl.wikipedia.org/wiki/Wetten_van_De_Morgan`.

```c
#include <stdio.h>
#include <stdbool.h>

// print waarheidstabel voor de wetten van De Morgan
int main(void)
{
    printf("a b |!a+!b|!(a.b)|!a.!b|!(a+b)\n");
    printf("----+-----+------+-----+------\n");
    bool a = false;
    do
    {
        bool b = false;
        do
        {
            bool c = (!a || !b);
            bool d = !(a && b);
            bool e = (!a && !b);
            bool f = !(a || b);
            printf("%d %d |  %d  |  %d   |  %d  |  %d\n", a, b, c, ↩
             ↪ d, e, f);
            b = !b;
        } while (b != false);
        a = !a;
    } while (a != false);
    return 0;
}
```

Listing 1.3: Print een waarheidstabel, zie `bool.c`.

1.2 Type modifiers

Bij het aanmaken van een variabele in C moet je, zoals je weet, het type van de variabele opgeven, zie paragraaf 1.1. Hierbij kun je ook een zogenoemde *type modifier* gebruiken om een soort variant van het gebruikte type te specificeren. De volgende type modifiers worden in deze paragraaf besproken: **signed**, **unsigned**, **short** en **long**.

1.2.1 signed en unsigned

Deze modifiers zijn het meest zinvol om te gebruiken bij het basistype **int**, zie pagina 4. De modifier **signed** is de default voor het type **int**. Dat wil zeggen dat het type **signed int** hetzelfde is als het type **int**. Compilers voor 32- of 64-bit processors gebruiken vrijwel altijd vier bytes (32 bits) voor een variabele van het type **int**. Zo'n variabele heeft een minimale waarde van $-2^{31} = -2147483648$ en een maximale waarde van $2^{31} - 1 = 2147483647$. Compilers voor 8- of 16-bit processors daarentegen gebruiken vrijwel altijd twee bytes (16 bits). Zo'n variabele heeft een minimale waarde van $-2^{15} = -32768$ en een maximale waarde van $2^{15} - 1 = 32767$. Het type **unsigned int** kan alleen positieve waarde bevatten.

32-bit unsigned integers hebben een minimale waarde van 0 en een maximale waarde van $2^{32} - 1 = 4294967295$. 16-bit unsigned integers hebben een minimale waarde van 0 en een maximale waarde van $2^{16} - 1 = 65535$.

Als je zeker weet dat een variabele altijd positieve gehele getallen zal bevatten, dan kun je gebruik maken van het type **unsigned int**. In paragraaf 1.1.1 heb je een programma gezien waarin de som van alle getallen van 1 tot 50000 werd berekent. Omdat in dat programma het type **int** werd gebruikt, trad er een overflow op bij het berekenen van de somformule van Gauss. Omdat in dit programma alleen positieve getallen gebruikt worden, kan er misschien beter gebruik gemaakt worden van het type **unsigned int**. In listing 1.4 is een programma gegeven dat de som van 1 tot en met 50000 stap voor stap en met behulp van de somformule van Gauss berekent.

```c
#include <stdio.h>

// toon aan dat som van 1 + 2 + 3 + 4 + ... + n gelijk is aan (n + ↩
 ↪ 1) * n / 2
int main(void)
{
    unsigned int n = 50000;
    unsigned int som = 0;
    for (unsigned int getal = 1; getal <= n; getal = getal + 1)
    {
        som = som + getal;
    }
    printf("som = %u\n", som);
    unsigned int formule = (n + 1) * n / 2;
    printf("formule = %u\n", formule);
    return 0;
}
```

Listing 1.4: Het op twee manieren berekenen van de som van 1 tot en met 50000, zie `uint.c`.

De uitvoer van dit programma is zoals verwacht:

```
som = 1250025000
formule = 1250025000
```

Als je de somformule van Gauss uitrekent, dan berekenen je `(n + 1)* n`. Als `n` de waarde `50000` heeft dan is dit product gelijk aan `50001 * 50000` = `2500050000`. Deze waarde past nog net in een 32-bit unsigned integer variabele, want 2500050000 < 4294967295.

Als je het programma uit listing 1.4 aanpast zodat het de som van 1 tot en met 100000 berekent, dan kan het programma het correcte antwoord (5000050000) *niet* weergeven omdat dit antwoord groter is dan $2^{32}-1$ en dus niet in een variabele van het type **`unsigned int`** past.

Als je in listing 1.4 de waarde `50000` vervangt door de waarde `100000`, dan geeft het programma (zie `int_error1.c`) de volgende foutieve[17] uitvoer:

```
som = 705082704
formule = 705082704
```

Omdat de maximale waarde van een **`unsigned int`** ($2^{32}-1$) maar één bit groter is dan de maximale waarde van een **`int`** ($2^{31}-1$), wordt het type **`unsigned int`** in de praktijk minder gebruikt, dan je misschien zou denken.

Het onnodig gebruik van de modifier **`unsigned`** kan ook tot fouten leiden.

In listing 1.5 is een programma gegeven dat aftelt van 10 tot en met 0.

```
#include <stdio.h>

// tel af van 10 tot en met 0
int main(void)
{
    for (unsigned int getal = 10; getal >= 0; getal = getal - 1)
    {
        printf("%u\n", getal);
    }
    return 0;
}
```

Listing 1.5: Een programma dat aftelt van 10 tot en met 0, zie `uint_error2.c`. Dit programma werkt niet correct!

De uitvoer van dit programma is anders dan de auteur van dit programma dacht:

```
10
9
8
7
```

[17] Het juiste antwoord is $5000050000_{10} = 1001010100000011010110101010100000_2$. Maar dit getal past niet in 32 bits. De laagste 32 bits worden opgeslagen $00101010000001101011010101010000_2 = 705082704_{10}$.

```
6
5
4
3
2
1
0
4294967295
4294967294
4294967293
4294967292
...
```

De voorwaarde `getal >= 0` is namelijk altijd **`true`** als de variabele `getal` van het type **`unsigned int`** is. Als `getal = getal - 1` uitgerekend wordt (als getal een **`unsigned int`** is en) als `getal` 0 is dan wordt `getal` gelijk aan `4294967295`[18]. Het programma zal dus niet vanzelf stoppen.

De meeste compilers waarschuwen ons ook voor dit probleem: *Warning: comparison of unsigned expression >= 0 is always true*.

Als je het programma uit listing 1.5 aanpast zodat het type **`int`** wordt gebruikt in plaats van **`unsigned int`**, dan werkt het programma wel correct, zie `int2`.

1.2.2 `short` en `long`

Wordt nog aan gewerkt!

1.3 Type qualifiers

Wordt nog aan gewerkt!

`const`, **`volatile`** en `restrict`

Zie `https://harrybroeders.bitbucket.io/KOF01/volatile.htm`.

1.4 Integer variabelen met een vaste grootte

Wordt nog aan gewerkt!

C11 integer types.

[18] Als van het 32-bit unsigned getal met de waarde $0_{10} = 00000000000000000000000000000000_2$ 1 wordt afgetrokken, dan krijg je de waarde $11111111111111111111111111111111_2 = 4294967295_{10}$.

1.5 Pointers

Wordt nog aan gewerkt!

1.6 Globale en lokale variabelen

2

In- en uitvoeren

Wordt nog aan gewerkt!

2.1 `printf`

3

Bewerken

Wordt nog aan gewerkt!

3.1 Operatoren

Wordt nog aan gewerkt!

De processoren die tegenwoordig in pc's worden gebruikt hebben speciale hardware om floating-point operaties uit te kunnen voeren. Ze kunnen dan net zo snel werken met variabelen van het type **`double`** als met variabelen van het type **`float`**, zie `https://bitbucket.org/HR_ELEKTRO/cpl01/wiki/double.md`. Veel microprocessoren hebben geen speciale hardware om floating-point operaties uit te voeren. Deze berekeningen moeten dan met integer bewerkingen uitgevoerd worden. In dit geval is een operatie met variabelen van het type **`float`** sneller dan een operatie met variabelen van het type **`double`**. Er zijn ook C-compilers die het type **`double`** opslaan in 32 bits. In dat geval zijn de types **`float`** en **`double`** dus gelijk.

3.2 Standaard functies

Wordt nog aan gewerkt!

4

Herhalen

Als je aan het programmeren bent dan wil je een bepaald stukje code vaak een aantal keren herhalen. Stel bijvoorbeeld dat je een stappenmotor 100 stapjes naar rechts wilt laten draaien. Je moet dan de code om de stappenmotor één stapje naar rechts te laten draaien 100 maal herhalen. Dit kun je natuurlijk doen door deze code 100 maal achter elkaar te kopiëren in je programma, maar dit is geen goed idee. Je programma wordt op deze manier lang, onoverzichtelijk en slecht aanpasbaar. Een betere manier is om de computer zelf de herhaling te laten uitvoeren en de herhaling met behulp van een herhalingsstatement in je programma op te nemen. Soms is het aantal benodigde herhalingen bekend op het moment dat de herhaling moet worden uitgevoerd, zoals in het bovenstaande voorbeeld. Maar in andere gevallen is dat niet zo. Stel bijvoorbeeld dat je een stappenmotor net zo lang naar rechts wilt laten draaien totdat een eindstop is bereikt. In dit geval moet het programma de code om de stappenmotor één stapje naar rechts te laten draaien net zolang herhalen totdat het eindstopsignaal actief wordt. Op het moment dat deze herhaling wordt uitgevoerd is dus nog niet bekend hoe vaak de betreffende code herhaald moet worden. Misschien moet de code wel helemaal niet uitgevoerd worden als de eindstop al actief is.

Er zijn in de programmeertaal C drie herhalingsstatements:

- `while`. Gebruik een `while` als het aantal benodigde herhalingen bij het uitvoeren van de herhaling *onbekend* is.
- `do while`. Gebruik een `do while` als het aantal benodigde herhalingen *groter dan één* maar verder onbekend is.
- `for`. Gebruik een `for` als het aantal benodigde herhalingen bij het uitvoeren van de herhaling *bekend* is.

4.1 `while`

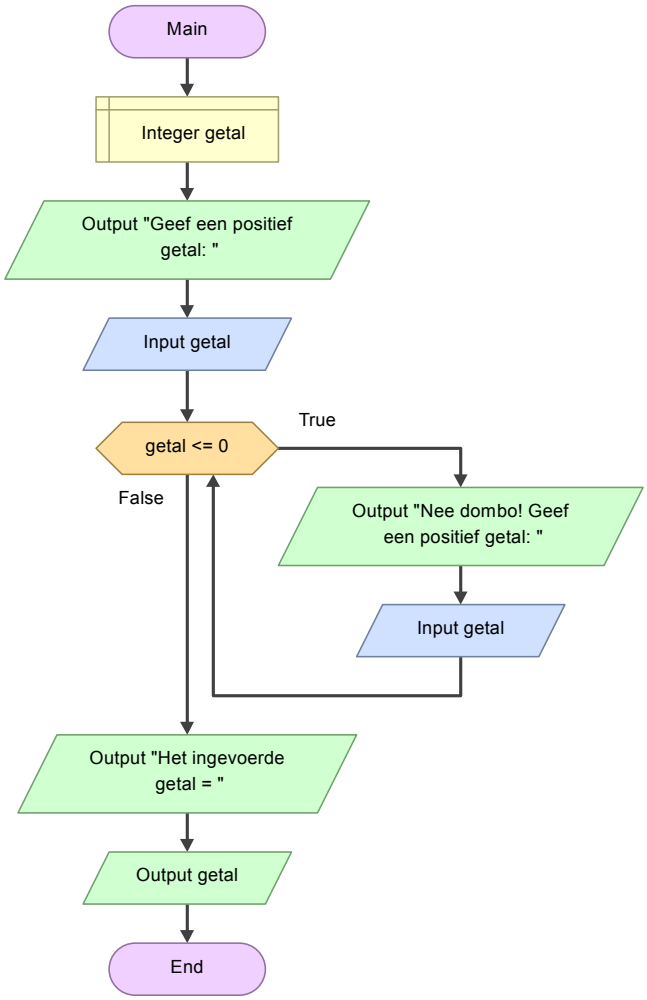
Je gebruikt een `while` als het aantal benodigde herhalingen bij het uitvoeren van de herhaling onbekend is. Als voorbeeld gebruiken we hier het inlezen van een positief geheel getal. Als de gebruiker een negatief getal invoert dan geeft het programma een foutmelding en wordt het getal opnieuw ingelezen. Deze code wordt net zolang herhaald totdat de gebruiker een

positief getal heeft ingevoerd. De code van dit programma is gegeven in listing 4.1 en een flowchart van dit programma is gegeven in figuur 4.1.

```c
#include <stdio.h>

int main(void)
{
    int getal;
    printf("Geef een positief getal: ");
    scanf("%d", &getal);
    while (getal <= 0)
    {
        printf("Nee dombo! Geef een positief getal: ");
        scanf("%d", &getal);
    }
    printf("Het ingevoerde getal = %d\n", getal);
    return 0;
}
```

Listing 4.1: Een voorbeeld van het gebruik van een **while**, zie `while.c`.



Figuur 4.1: De flowchart van het programma uit listing 4.1.

4.2 do while

Je gebruikt een **do while** als het aantal benodigde herhalingen groter dan één maar verder onbekend is. Als voorbeeld gebruiken we hier opnieuw het inlezen van een positief geheel getal. Dit programma vraagt aan de gebruiker om een positief getal in te voren en vervolgens wordt een getal ingelezen. Deze code wordt herhaald zolang de gebruiker negatieve getallen blijft invoeren. De betreffende code moet dus minstens één maal uitgevoerd worden. Er wordt in dit geval bij ongeldige invoer geen foutmelding gegeven, maar de originele vraag wordt herhaald. De code van dit programma is gegeven in listing 4.2 en een flowchart van dit programma is gegeven in figuur 4.2.

```c
#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 4.2: Een voorbeeld van het gebruik van een **do while**, zie dowhile.c.

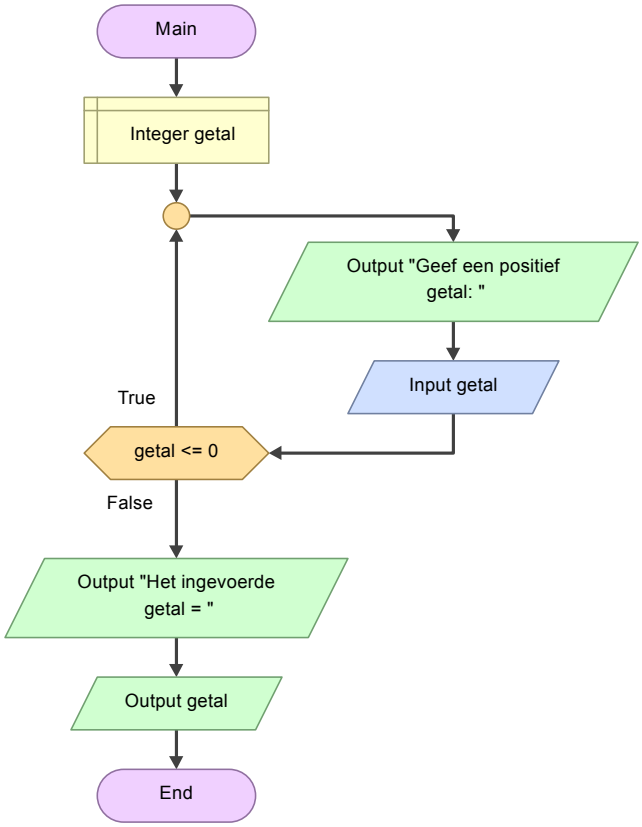
4.3 for

Je gebruikt een **for** als het aantal benodigde herhalingen bij het uitvoeren van de herhaling bekend is. Het **for**-statement heeft een enigszins ingewikkeld formaat.

```c
for (statement1; expressie; statement2)
{
    // code die herhaald moet worden
}
```

De code `statement1` wordt één maal uitgevoerd als de **for** begint. Vervolgens wordt de code `expressie` uitgevoerd. Als deze expressie **false** oplevert, dan wordt de **for** meteen beëindigd en wordt de code die herhaald moet worden dus helemaal niet uitgevoerd. Als de expressie **true** oplevert, dan wordt de code die herhaald moet worden één maal uitgevoerd. Vervolgens wordt code `statement2` uitgevoerd. Nu wordt code `expressie` weer uitgevoerd enzovoorts. Dit patroon herhaalt zich zolang de expressie **true** oplevert.

Als voorbeeld gebruiken we hier het berekenen van 12!. De faculteit van 12 is gedefinieerd als $12! = 1 \times 2 \times 3 \times 4 \times \cdots \times 12$. Het programma beginnen met het aanmaken van de variabele



Figuur 4.2: De flowchart van het programma uit listing 4.2.

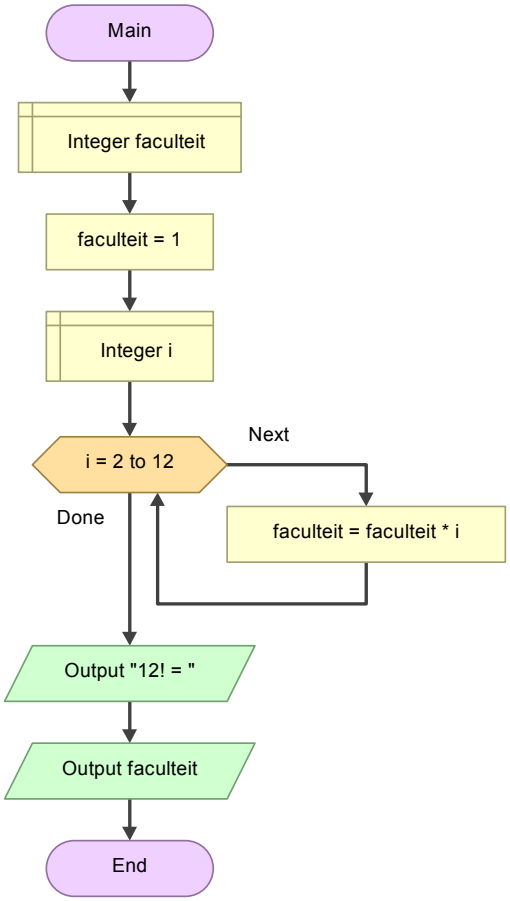
`faculteit` die geïnitialiseerd wordt met de waarde 1. Vervolgens wordt de code `faculteit = faculteit * i` elf keer herhaald uitgevoerd voor $i = 2, 3, 4, \ldots, 12$. Als `statement1` van de **for** is **int** `i = 2` gebruikt, om de variabele `i` te definiëren en te initialiseren[19]. Als `expressie` is `i <= 12` gebruikt. Deze expressie levert **true** op zolang de variabele `i` kleiner dan of gelijk is aan 12. Als `statement2` is `i = i + 1` gebruikt zodat de variabele `i`, nadat de code `faculteit = faculteit * i` is uitgevoerd, telkens met één verhoogd wordt.

```
#include <stdio.h>

int main(void)
{
    int faculteit = 1;
    for (int i = 2; i <= 12; i = i + 1)
    {
        faculteit = faculteit * i;
    }
    printf("12! = %d\n", faculteit);
    return 0;
}
```

Listing 4.3: Een voorbeeld van het gebruik van een **for**, zie `for.c`.

[19] Het is pas sinds de C99-standaard mogelijk om de variabele `i` op deze plek te definiëren. De variabele `i` is nu alleen bekend tot aan de accolade sluiten } van het **for**-statement. In voorgaande versies van C moest de variabele voor het **for**-statement worden gedefinieerd.



Figuur 4.3: De flowchart van het programma uit listing 4.3.

Voorspel de uitvoer van het programma gegeven in listing 4.4 om te kijken of je het `for`-statement goed begrijpt. Controleer je voorspelling door het programma uit te voeren.

```c
#include <stdio.h>

int main(void)
{
    for (int i = 0; i < 10; i = i + 2)
    {
        printf("i = %d\n", i);
    }
    for (int k = 5; k > 0; k--)
    {
        printf("k = %d\n", k);
    }
    for (int m = 1024; m < 4000; m = m * 2)
    {
        for (int n =  m; n > 0; n = n / 2)
        {
            printf("m = %d en n = %d\n", m, n);
        }
    }
    return 0;
}
```

Listing 4.4: Een programma met meerdere **for**-statements, zie `for2.c`.

5

Beslissen

Als je aan het programmeren bent dan wil je een bepaald stukje code vaak alleen onder een bepaalde voorwaarde uitvoeren. Het programma moet als het ware beslissen of de betreffende code wel of niet uitgevoerd moet worden. Stel bijvoorbeeld dat je een verwarming wilt aanschakelen als de temperatuur onder de 18 °C zakt. In dat geval moet de code om de verwarming aan te schakelen alleen uitgevoerd worden als de temperatuur lager dan 18 °C is. De programmeertaal C kent twee statements waarmee beslissingen kunnen worden genomen: `if` (eventueel gevolgd door `else`) en `switch`.

5.1 `if`

Je gebruikt een `if` als je een bepaald stukje code alleen onder een bepaalde voorwaarde wilt uitvoeren. Als voorbeeld gebruiken we hier het berekenen van de absolute waarde van een geheel getal. De absolute waarde van een getal kun je bepalen door het getal positief te maken als dit negatief is. Als het getal al positief is, dan hoef je niets te doen. De code van dit programma is gegeven in listing 5.1 en een flowchart van dit programma is gegeven in figuur 5.1. De code `getal = -getal;` zal alleen worden uitgevoerd als de expressie `getal < 0` `true` oplevert.

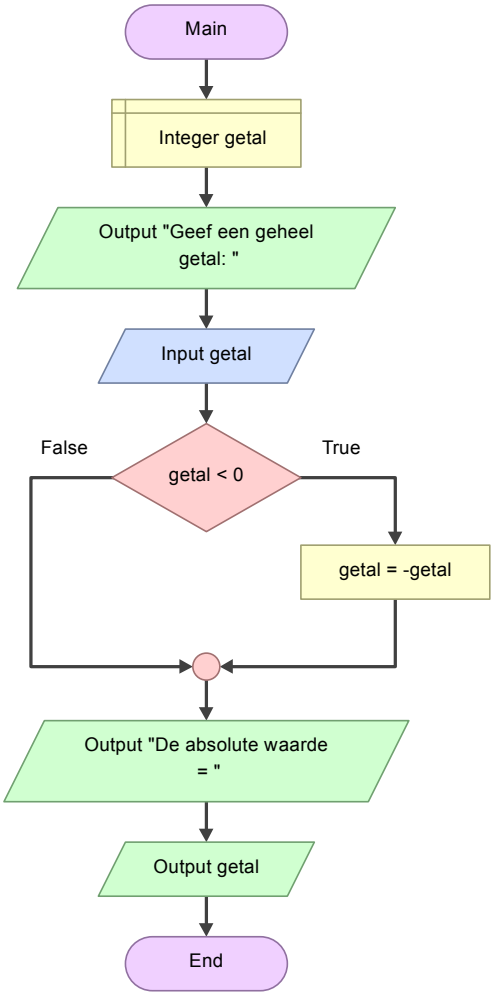
5.2 `else`

In het programma dat gegeven is in listing 5.1 hoeft geen specifieke code te worden uitgevoerd als *niet* aan de voorwaarde wordt voldaan. Als dit wel het geval is, dan kun je na het `if`-statement een `else` gebruiken. Als voorbeeld gebruiken we het bepalen van het maximum van twee getallen. Als het eerste getal groter is dan het tweede getal, dan is het eerste getal het maximum van de twee. Maar anders (als het eerste getal *niet* groter is dan het tweede getal), is het tweede getal het maximum van de twee. De code van dit programma is gegeven in listing 5.2 en een flowchart van dit programma is gegeven in figuur 5.2.

```c
#include <stdio.h>

int main(void)
{
    int getal;
    printf("Geef een geheel getal: ");
    scanf("%d", &getal);
    if (getal < 0)
    {
        getal = -getal;
    }
    printf("De absolute waarde = %d\n", getal);
    return 0;
}
```

Listing 5.1: Een voorbeeld van het gebruik van een **if**, zie `if.c`.



Figuur 5.1: De flowchart van het programma uit listing 5.1.

5.3 switch

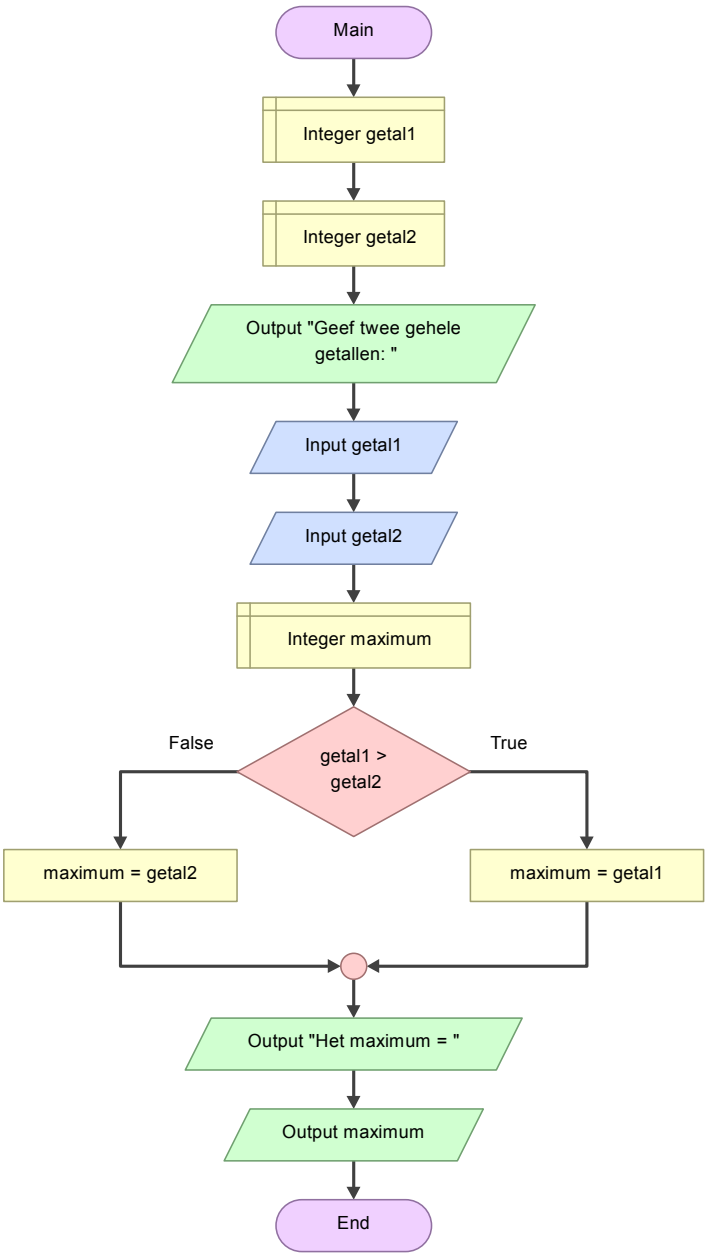
Als voorbeeld bekijken we nu een programma waarmee een afstand die wordt ingevoerd in mm, cm of dm kan worden omgezet naar m. De code van dit programma is gegeven

```c
#include <stdio.h>

int main(void)
{
    int getal1, getal2;
    printf("Geef twee gehele getallen: ");
    scanf("%d%d", &getal1, &getal2);
    int maximum;
    if (getal1 > getal2)
    {
        maximum = getal1;
    }
    else
    {
        maximum = getal2;
    }
    printf("Het maximum = %d\n", maximum);
    return 0;
}
```

Listing 5.2: Een voorbeeld van het gebruik van een **else**, zie `ifelse.c`.

in listing 5.3. In dit programma worden een aantal geneste **if**-**else**-statements gebruikt die het programma enigszins overzichtelijk maken. In dit voorbeeld zijn er slechts drie mogelijke waarden van de variabel `prefix` die we willen onderscheiden, maar je kunt je voorstellen wat er gebeurt als dit aantal toeneemt. In dit soort gevallen is het beter om een **switch**-statement te gebruiken. De code van het programma waarin de **switch** is toegepast, is gegeven in listing 5.3. Je ziet dat voor elke waarde van de variabel `prefix` die we willen onderscheiden een **case**-label binnen de **switch** is opgenomen. Het **break**-statement aan het einde van elke **case** is erg belangrijk. Als dit **break**-statement ontbreekt dan wordt na de huidige **case** de volgende **case** uitgevoerd. Behalve een **case** kan binnen de switch ook het label **default**: worden gebruikt. De **switch** springt naar dit label als de waarde van de variabele die in het **switch**-statement is gebruikt bij geen van de **case**-labels vermeld is.



Figuur 5.2: De flowchart van het programma uit listing 5.2.

```c
#include <stdio.h>

int main(void)
{
    double getal;
    char space, prefix, unit;
    printf("Geef een afstand in mm, cm of dm (bijvoorbeeld 2.6 ↩
     ↪ dm): ");
    scanf("%lf%c%c%c", &getal, &space, &prefix, &unit);
    double prefixvalue = 0;
    if (space == ' ' && unit == 'm')
    {
        if (prefix == 'm')
        {
            prefixvalue = 0.001;
        }
        else
        {
            if (prefix == 'c')
            {
                prefixvalue = 0.01;
            }
            else
            {
                if (prefix == 'd')
                {
                    prefixvalue = 0.1;
                }
            }
        }
    }
    if (prefixvalue == 0)
    {
        printf("De invoer is niet correct!\n");
    }
    else
    {
        printf("Dit is %f meter\n", getal * prefixvalue);
    }
    return 0;
}
```

Listing 5.3: Een programma met heel veel **if**-statements, zie convertprefix.c.

```c
#include <stdio.h>

int main(void)
{
    double getal;
    char space, prefix, unit;
    printf("Geef een afstand in mm, cm of dm (bijvoorbeeld 2.6 ↩
     ↪ dm): ");
    scanf("%lf%c%c%c", &getal, &space, &prefix, &unit);
    double prefixvalue = 0;
    if (space == ' ' && unit == 'm')
    {
        switch (prefix)
        {
        case 'm':
            prefixvalue = 0.001;
            break;
        case 'c':
            prefixvalue = 0.01;
            break;
        case 'd':
            prefixvalue = 0.1;
            break;
        }
    }
    if (prefixvalue == 0)
    {
        printf("De invoer is ongeldig!\n");
    }
    else
    {
        printf("Dit is %f meter\n", getal * prefixvalue);
    }
    return 0;
}
```

Listing 5.4: Een programma met een **switch**-statement, zie convertprefix2.c.

6

Programmeren, hoe doe je dat nu eigenlijk?

Wordt nog aan gewerkt!

Zie `https://bitbucket.org/HR_ELEKTRO/cpl01/wiki/tafels_stap_voor_stap.md`.

Zie `https://bitbucket.org/HR_ELEKTRO/cpl01/wiki/cosinus_stap_voor_stap.md`.

7

Structureren van code

Als een programma groter wordt, is het nodig om er structuur in aan te brengen zodat je het overzicht kan behouden. De programmeercode kun je structureren door gebruik te maken van functies. Je kunt ook structuur aanbrengen door je code te verdelen over meerdere bestanden en ook door bepaalde delen op te nemen in een herbruikbare bibliotheek (Engels: library). Ook de data die je gebruikt in je programma kun je structureren, dit wordt besproken in het volgende hoofdstuk.

7.1 Functies

Als een programma langer wordt, dan wordt het al snel onoverzichtelijk. Sommige stukken code komen misschien meerdere keren in het programma voor. Bijvoorbeeld het inlezen van een positief getal. Dit kopiëren van code zorgt ervoor dat het programma slecht onderhoudbaar wordt. Als er namelijk een wijziging in deze code moet worden doorgevoerd dan moeten ook alle kopietjes worden aangepast. Ook zijn delen uit zo'n lang programma niet eenvoudig te gebruiken in een ander programma. De code is slecht herbruikbaar.

Je kunt deze problemen in de programmeertaal C oplossen door het gebruik van *functies*. Een logisch bij elkaar behorend stuk code wordt dan ergens apart (in een functie) geplaatst. Deze functie kun je vervolgens naar believen aanroepen om de code van de functie uit te voeren.

7.1.1 Functies zonder parameters en zonder returnwaarde[20]

Als sterk versimpeld voorbeeld beschouwen we het programma dat gegeven is in listing 7.1

Op twee plaatsen in dit programma worden, in de uitvoer van het programma, drie regels overgeslagen. In plaats van deze code te dupliceren kun je deze code ook opnemen in een functie. Het is belangrijk om de functie een duidelijke naam te geven die aangeeft wat de

[20] De begrippen parameter en returnwaarde worden verderop in dit hoofdstuk uitgelegd.

```c
#include <stdio.h>

int main(void)
{
//  ...
    printf("\n"); // sla 3 regels over
    printf("\n");
    printf("\n");
//  ...
    printf("\n"); // sla 3 regels over
    printf("\n");
    printf("\n");
//  ...
    return 0;
}
```

Listing 7.1: Een programma waar op twee plaatsen drie regels overgeslagen worden, zie `slaregelsover1.c`.

functie doet. In dit geval is gekozen voor de naam `sla_3_regels_over`. Het programma waarbij gebruikt gemaakt wordt van een functie is gegeven in listing 7.2.

```c
#include <stdio.h>

void sla_3_regels_over(void)
{
    printf("\n");
    printf("\n");
    printf("\n");
}

int main(void)
{
//  ...
    sla_3_regels_over();
//  ...
    sla_3_regels_over();
    //  ...
    return 0;
}
```

Listing 7.2: Een programma waar op twee plaatsen drie regels overgeslagen worden met behulp van de functie `sla_3_regels_over`, zie `slaregelsover2.c`.

Bovenin het programma wordt de functie `sla_3_regels_over` gedefinieerd. Het keyword **void** betekent leeg. Het gebruik van **void** vóór de functienaam geeft aan dat deze functie niets teruggeeft. Verderop in dit hoofdstuk zul je zien hoe een functie indien gewenst wel iets kan teruggeven. Het gebruik van **void** na de functienaam, tussen de haken, geeft aan dat aan deze functie niets meegegeven kan worden bij aanroep. Verderop in dit hoofdstuk zul je zien hoe je aan een functie indien gewenst wel iets kan meegeven bij aanroep. Vervolgens

wordt deze functie in `main` twee maal aangeroepen met de code `sla_3_regels_over()`. De haakjes `()` achter de functienaam geven aan dat de functie aangeroepen moet worden.

Als dit programma gestart wordt, begint de uitvoering bij `main`. Als de aanroep `sla_3_regels_over()` moet worden uitgevoerd, wordt naar de code van deze functie gesprongen. Aan het einde van de functie wordt teruggekeerd achter de plaats waar de functie is aangeroepen en wordt de uitvoering van het programma daar voortgezet. Bij het aanroepen van een functie 'onthoud' de processor dus waarvandaan de functie aangeroepen wordt, zodat het programma na afloop van de functie achter deze aanroep kan worden vervolgd.

De functie moet 'gezien' zijn voordat de functie aangeroepen kan worden. Vandaar dat de definitie van de functie `sla_3_regels_over()` boven `main` geplaatst is. Het is echter niet nodig om de volledige code van de functie voor `main` te definiëren. Het is voldoende om alleen de eerste regel van de functie voor `main` te definiëren. Dit wordt dan een functiedeclaratie of functieprototype genoemd. De volledige code van de functie moet natuurlijk nog wel worden gedefinieerd. Dit kan bijvoorbeeld achter de `main` maar kan ook in een apart bestand. We komen hier verderop in deze paragraaf op terug. In listing 7.3 kun je zien hoe je een functiedeclaratie kunt gebruiken[21].

```c
#include <stdio.h>

void sla_3_regels_over(void);

int main(void)
{
//   ...
     sla_3_regels_over();
//   ...
     sla_3_regels_over();
     //  ...
     return 0;
}

void sla_3_regels_over(void)
{
     printf("\n");
     printf("\n");
     printf("\n");
}
```

Listing 7.3: Een programma waarin een functiedeclaratie gebruikt is, zie `slaregelsover3.c`.

[21] Als je de functiedeclaratie vergeet zal het programma wel compileren, al zal de compiler wel een warning geven. De compiler is in dit geval namelijk niet in staat om te controleren of de functie correct wordt aangeroepen. Het is dus sterk aan te raden om als een functie niet boven `main` gedefinieerd is een functiedeclaratie te gebruiken.

7.1.2 Functies met parameters

Als nog steeds sterk versimpeld voorbeeld beschouwen we nu het programma dat gegeven is in listing 7.4

```
#include <stdio.h>

int main(void)
{
//   ...
     printf("\n"); // sla 3 regels over
     printf("\n");
     printf("\n");
//   ...
     printf("\n"); // sla 4 regels over
     printf("\n");
     printf("\n");
     printf("\n");
//   ...
     return 0;
}
```

Listing 7.4: Een programma waar op twee plaatsen een aantal regels overgeslagen wordt, zie `slaregelsover4.c`.

Op twee plaatsen in dit programma worden, in de uitvoer van het programma, regels overgeslagen. De eerste keer worden drie regels overgeslagen en de tweede keer worden vier regels overgeslagen. Je zou nu een functie kunnen definiëren om drie regels over te slaan en nog een andere functie om vier regels over te slaan. Maar het zou natuurlijk veel handiger zijn als je een functie zou kunnen definiëren waarmee je een variabel aantal regels kunt overslaan. Dit is mogelijk door een functie met een zogenoemde parameter te definiëren. Bij aanroep van de functie moet dan een zogenoemd argument worden meegegeven. De waarde van het argument wordt dan bij aanroep van de functie naar de parameter gekopieerd.

In listing 7.5 kun je zien hoe de functie `sla_regel_over` is gedefinieerd. De functie heeft een parameter van het type **`int`**. Bij de eerste aanroep van de functie wordt het argument `3` meegegeven. Deze waarde wordt bij aanroep gekopieerd naar de parameter `aantal`. Een parameter is in feite niets anders dan een variabele die bij het aanroepen van de functie geïnitialiseerd wordt met de waarde van het bij aanroep meegegeven argument. De code van de functie wordt vervolgens uitgevoerd. Doordat de parameter `aantal` is geïnitialiseerd met de waarde `3` wordt de code in het **`for`**-statement drie maal herhaald en daardoor worden in de uitvoer drie regels overgeslagen. De parameter `aantal` is alleen in de functie `sla_regels_over` te gebruiken. Buiten de functie is de parameter onbekend. Bij de tweede aanroep van de functie wordt het argument `4` meegegeven. Deze waarde wordt bij aanroep gekopieerd naar de parameter `aantal`. De code van de functie wordt vervolgens weer uitgevoerd. Doordat de parameter `aantal` nu is geïnitialiseerd met de waarde `4` wordt de code in het **`for`**-statement vier maal herhaald en daardoor worden in de uitvoer vier regels overgeslagen.

```c
#include <stdio.h>

void sla_regels_over(int aantal)
{
    for (int teller = 0; teller < aantal; teller++)
    {
        printf("\n");
    }
}

int main(void)
{
//  ...
    sla_regels_over(3);
//  ...
    sla_regels_over(4);
    //  ...
    return 0;
}
```

Listing 7.5: Een programma waar op twee plaatsen een aantal regels overgeslagen wordt met behulp van de functie `sla_regels_over`, zie `slaregelsover5.c`.

Een functie kan meerdere parameters hebben. De verschillende parameters worden van elkaar gescheiden door een komma. Elke parameter heeft zijn eigen typeaanduiding. Bij aanroep moet het aantal argumenten overeenkomen met het aantal parameters. De waarde van het eerste argument wordt gekopieerd naar de eerste parameter en de waarde van het tweede argument wordt gekopieerd naar de tweede parameter enzovoort.

In listing 7.6 is een voorbeeld gegeven van een functie met twee parameters. De functie `print_rechthoek` print een rechthoek met een bepaalde breedte en hoogte. De uitvoer van het in listing 7.6 gegeven programma is te zien in figuur 7.1.

```
++++++++++++++++++++
+                  +
+                  +
+                  +
+                  +
+                  +
+                  +
++++++++++++++++++++
```

Figuur 7.1: De uitvoer van het programma uit listing 7.6.

De functie is bedoeld voor het tekenen van een rechthoek met een breedte groter dan 2 en kleiner dan 80 en een hoogte van groter dan 2 en kleiner dan 40. De standaardfunctie `assert` wordt gebruikt om te controleren of de meegegeven argumenten aan deze voorwaarden voldoen. Als de expressie die in de `assert` wordt gedefinieerd **`false`** oplevert, dan wordt het programma afgebroken en wordt een foutmelding gegeven.

```c
#include <stdio.h>
#include <assert.h>

void print_lijn(int lengte)
{
    assert (lengte > 2 && lengte < 80);
    for (int teller = 0; teller < lengte; teller++)
    {
        printf("+");
    }
    printf("\n");
}

void print_rechthoek(int breedte, int hoogte)
{
    assert (hoogte > 2 && hoogte < 40);
    print_lijn(breedte);
    for (int regel = 0; regel < hoogte - 2; regel++)
    {
        printf("+");
        for (int teller = 0; teller < breedte - 2; teller++)
        {
            printf(" ");
        }
        printf("+\n");
    }
    print_lijn(breedte);
}

int main(void)
{
    print_rechthoek(20, 8);
    return 0;
}
```

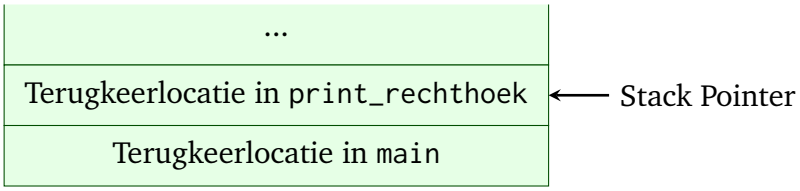
Listing 7.6: Een voorbeeld van een functie met twee parameters, zie `print_rechthoek.c`.

Merk op dat de functie `print_rechthoek` gebruik maakt van de functie `print_lijn` om de boven- en onderkant van de rechthoek te printen. De uitvoering van het programma start in `main`. Vervolgens wordt de functie `print_rechthoek` aangeroepen. De terugkeerlocatie wordt 'onthouden' door de processor door het ergens op te slaan. Vervolgens wordt vanuit de functie `print_rechthoek` de functie `print_lijn` aangeroepen. Ook deze terugkeerlocatie wordt opgeslagen door de processor.

Na afloop van de functie `print_lijn` wordt het programma vervolgt op de terugkeerlocatie die als laatste is opgeslagen. Vervolgens wordt het **for**-statement in `print_rechthoek` uitgevoerd en daarna wordt `print_lijn` nogmaals aangeroepen. Ook deze terugkeerlocatie wordt weer opgeslagen door de processor. Na afloop van de functie `print_lijn` wordt het programma vervolgt op de terugkeerlocatie die zojuist is opgeslagen. Na afloop van

de functie `print_rechthoek` wordt het programma vervolgt op de als eerste opgeslagen terugkeerlocatie. De volgorde waarin terugkeerlocaties worden opgeslagen en weer worden opgeroepen wordt LIFO (Last In First Out) genoemd.

De processor slaat terugkeerlocaties op, op de zogenoemde stack. De bovenkant van de stack wordt aangewezen door een speciaal daarvoor bestemd register in de processor: de stack pointer. In figuur 7.2 is de stack getekend op het moment dat processor bezig is met het uitvoeren van de functie `print_lijn`.



Figuur 7.2: De stack op het moment dat de functie `print_lijn` wordt uitgevoerd.

7.1.3 Functies met een returnwaarde

Tot nu toe heb je functies geschreven die iets afdrukken met behulp van `printf`. Als je kijkt naar de standaard C-functie `sin` dan zie je dat deze functie niets afdrukt maar de sinus van het argument teruggeeft. Een goed ontworpen functie moet in meerdere programma's gebruikt kunnen worden. Functies die het berekende resultaat meteen afdrukken zijn eigenlijk helemaal niet handig om te gebruiken in verschillende programma's. Stel je voor dat de functie `sin` de sinus van het argument meteen zou afdrukken in plaats van het resultaat terug te geven. Deze functie zou dan slechts in een beperkt aantal gevallen te gebruiken zijn. De versie die het resultaat teruggeeft is in veel meer gevallen te gebruiken. Soms zal de waarde van de sinus afgedrukt moeten worden, door de returnwaarde van de `sin`-functie als argument door te geven aan de `printf`-functie:

```
print(sin(arg));
```

Maar meestal zal de waarde van de sinus gebruikt worden in een berekening:

```
overstaande_rechthoekszijde = sin(hoek) * schuine_zijde
```

De definitie van een C-functie die een waarde teruggeeft begint niet met **`void`** maar met het type van de waarde die wordt teruggegeven. Dit wordt het returntype van de functie genoemd. Vanuit de functie kan een waarde van het returntype worden teruggegeven met behulp van het **`return`**-statement. Er kan slechts één waarde worden teruggegeven via het **`return`**-statement.

In listing 7.7 is een voorbeeld gegeven van een functie die een waarde van het type **`double`** teruggeeft. Deze functie berekent het gemiddelde van 3 als argumenten meegegeven gehele getallen.

De definitie van de functie `gemiddelde` begint met de typeaanduiding **`double`** waarmee aangegeven wordt dat deze functie een waarde van het type **`double`** teruggeeft. In de functie wordt een **`return`**-statement gebruikt. De waarde van de expressie achter **`return`** wordt

```c
#include <stdio.h>

double gemiddelde(int getal1, int getal2, int getal3)
{
    double resultaat = (getal1 + getal2 + getal3) / 3.0;
    return resultaat;
}

int main(void)
{
    double gem = gemiddelde(27, 29, 33);
    printf("%f\n", gem);
    return 0;
}
```

Listing 7.7: Een eenvoudig voorbeeld van een functie met een returntype, zie `gemiddelde1.c`.

gekopieerd naar de plaats waar de functie wordt aangeroepen. De aanroep van de functie wordt als het ware vervangen door de returnwaarde. In `main` wordt de waarde die wordt teruggegeven door de functie `gemiddelde` opgeslagen in de variabele `gem` en vervolgens afgedrukt met behulp van `printf`.

Er is in het voorbeeld dat is weergegeven in listing 7.7 gebruik gemaakt van de variabelen `resultaat` en `gem`, maar beide variabelen zijn in feite niet nodig. In listing 7.8 is een compactere versie van dit voorbeeld weergegeven.

```c
#include <stdio.h>

double gemiddelde(int getal1, int getal2, int getal3)
{
    return (getal1 + getal2 + getal3) / 3.0;
}

int main(void)
{
    printf("%f\n", gemiddelde(27, 29, 33));
    return 0;
}
```

Listing 7.8: Een compactere versie van het programma uit listing 7.7, zie `gemiddelde2.c`.

Een functie kan meerdere **`return`**-statements bevatten. Zodra een **`return`**-instructies wordt uitgevoerd wordt de functie beëindigd. Als eenvoudig voorbeeld is in listing 7.9 een functie gegeven die de maximale waarde van twee, als argumenten meegegeven, gehele getallen teruggeeft. Merk op dat je deze functie ook kan gebruiken om de maximale waarde van 3 getallen te bepalen door de returnwaarde van de ene aanroep te gebruiken als argument van de volgende aanroep.

Omdat de functie meteen wordt beëindigd als de return wordt uitgevoerd, is de **`else`** in de functie `max` overbodig. Zie listing 7.10 voor een compactere versie van deze functie.

```c
#include <stdio.h>

int max(int getal1, int getal2)
{
    if (getal1 > getal2)
    {
        return getal1;
    }
    else
    {
        return getal2;
    }
}

int main(void)
{
    int i1, i2, i3;
    scanf("%d%d%d", &i1, &i2, &i3);
    printf("De maximale waarde is: %d\n", max(i1, max(i2, i3)));
    return 0;
}
```

Listing 7.9: Een programma dat drie gehele getallen inleest en de maximale waarde afdrukt, zie max1.c.

```c
int max(int getal1, int getal2)
{
    if (getal1 > getal2)
    {
        return getal1;
    }
    return getal2;
}
```

Listing 7.10: Een compactere versie van de functie max, zie max2.c.

Tot nu toe waren de functies die we als voorbeeld hebben bekeken nog niet zo heel goed bruikbaar in de praktijk. Daarom sluiten we deze paragraaf af met een functie die je wel degelijk in de praktijk zou kunnen toepassen. De uitvoer van het in listing 7.11 gegeven programma is te zien in figuur 7.3. De door de gebruiker ingetypte invoer is groen en onderstreept weergegeven. In listing 7.11 is een functie gegeven die een geheel getal inleest, controleert of de ingevoerde waarde een getal is en of deze waarde tussen een als argument meegegeven minimale en maximale waarde ligt (inclusief). Zolang dit niet zo is, wordt een foutmelding gegeven en kan de gebruiker het opnieuw proberen,

```c
#include <stdio.h>

int lees_geheel_getal(int min, int max)
{
    int getal;
    printf("Geef een geheel getal [%d..%d]: ", min, max);
    while (scanf("%d", &getal) != 1 || getal < min || getal > max)
    {
        char karakter;
        do
        {
            scanf("%c", &karakter);
        }
        while (karakter != '\n');
        printf("Onjuiste invoer. Probeer het opnieuw!\n");
        printf("Geef een geheel getal [%d..%d]: ", min, max);
    }
    return getal;
}

int main(void)
{
    printf("Het ingelezen toetscijfer is %d.\n", ↩
     ↪ lees_geheel_getal(1, 10));
    return 0;
}
```

Listing 7.11: Een functie om een geheel getal in te lezen, zie `lees_geheel_getal.c`.

```
Geef een geheel getal [1..10]: 0
Onjuiste invoer. Probeer het opnieuw!
Geef een geheel getal [1..10]: 11
Onjuiste invoer. Probeer het opnieuw!
Geef een geheel getal [1..10]: zeven
Onjuiste invoer. Probeer het opnieuw!
Geef een geheel getal [1..10]: 7
Het ingelezen toetscijfer is 7.
```

Figuur 7.3: Een mogelijke uitvoer van het programma uit listing 7.11.

7.1.4 Functies die meer dan één waarde teruggeven

Met behulp van het **return**-statement kun je in C vanuit een functie één waarde teruggeven. Als je meerdere waarden wilt teruggeven dan kan dat op twee manieren.

Je kunt:

- de waarden 'inpakken' in een structure, dit wordt verderop in dit dictaat behandeld, zie paragraaf 8.2;
- de waarden teruggeven via zogenoemde *call by reference* parameters.

Bij het gebruik van normale parameters wordt de *waarde* van elk argument gekopieerd naar de overeenkomstige parameter. Deze parameters worden ook wel *call by value* parameters genoemd. Call by reference parameters kunnen in C geimplementeerd worden door pointers, zie paragraaf 1.5, te gebruiken. Beschouw het programma dat gegeven is in listing 7.12. Het is de bedoeling dat de waarden van de variabelen x en y verwisseld zijn na aanroep van de functie `wissel(x, y)`. Zo'n functie die twee getallen verwisseld kan bijvoorbeeld handig zijn bij het sorteren van getallen. De uitvoer van dit programma is gegeven in figuur 7.4.

Zoals je ziet zijn de waarden van de variabelen x en y *niet* verwisseld na aanroep van de functie `wissel(x, y)`. Dit had je natuurlijk kunnen zien aankomen. Bij aanroep van de functie `wissel` wordt de waarde van de als argument meegegeven variabele `x` gekopieerd naar de parameter `a` en wordt de waarde van de als argument meegegeven variabele `y` gekopieerd naar de parameter `b`. Vervolgens worden de waarden van `a` en `b` keurig verwisseld in de functie `wissel`, maar bij terugkeer in `main` hebben `x` en `y` nog steeds hun oorspronkelijke waarden. De standaard parameteroverdacht in C is namelijk call by value. Bij aanroep wordt de waarde van elk argument gekopieerd naar de bijbehorende parameter.

```c
#include <stdio.h>

void wissel(int a, int b)
{
    int hulpje = a;
    a = b;
    b = hulpje;
}

int main(void)
{
    int x = 7, y = 8;
    printf("x = %d en y = %d\n", x, y);
    wissel(x, y);
    printf("x = %d en y = %d\n", x, y);
    return 0;
}
```

Listing 7.12: Een *niet werkende* functie om twee getallen te verwisselen, zie `foute_wissel.c`.

```
x = 7 en y = 8
x = 7 en y = 8
```

Figuur 7.4: De uitvoer van het programma uit listing 7.12.

Om de functie correct te laten functioneren moet je gebruik maken van pointers, zie listing 7.13. De uitvoer van dit programma is gegeven in figuur 7.5. Als parameters van de functie `wissel` zijn nu twee pointers naar **`int`**'s gedefinieerd. Bij aanroep van de functie

wordt het adres van de variabele x als eerste argument meegegeven. De waarde van dit adres wordt gekopieerd naar de parameter ptr_a. Deze pointer 'wijst' of 'refereert' dus naar de variabele x. Het adres van de variabele y wordt als tweede argument meegegeven. De waarde van dit adres wordt gekopieerd naar de parameter ptr_b. Deze pointer wijst dus naar de variabele y.

Bij aanvang van de functie wissel wordt de waarde van de variabele hulpje gelijk aan waarde waar de pointer a_ptr naar wijst. Deze pointer wijst naar de variabele x en die heeft de waarde 7, dus hulpje krijgt de waarde 7. Vervolgens wordt de variabele waar de pointer a_ptr naar wijst gelijk gemaakt aan de waarde waar de pointer b_ptr naar wijst. a_ptr wijst naar de variabele x en b_ptr wijst naar de variabele y, die de waarde 8 heeft. Dus x krijgt de waarde 8. Tot slot wordt de variabele waar de pointer b_ptr naar wijst gelijk gemaakt aan de waarde van de variabele hulpje. a_ptr wijst naar de variabele y en hulpje heeft de waarde 7. Dus y krijgt de waarde 7. Bij terugkeer in main zijn waarden van de variabelen x en y dus inderdaad verwisseld, zoals de bedoeling was.

Het doorgeven van adressen als argumenten in plaats van waarden wordt call by reference genoemd. Een adres refereert (verwijst) namelijk naar een bepaalde variabele in het geheugen en je geeft de referentie naar de variabele door in plaats van de waarde van de variabele.

```c
#include <stdio.h>

void wissel(int *ptr_a, int *ptr_b)
{
    int hulpje = *ptr_a;
    *ptr_a = *ptr_b;
    *ptr_b = hulpje;
}

int main(void)
{
    int x = 7, y = 8;
    printf("x = %d en y = %d\n", x, y);
    wissel(&x, &y);
    printf("x = %d en y = %d\n", x, y);
    return 0;
}
```

Listing 7.13: Een *werkende* functie om twee getallen te verwisselen, zie wissel.c.

```
x = 7 en y = 8
x = 8 en y = 7
```

Figuur 7.5: De uitvoer van het programma uit listing 7.13.

7.1.5 Zichtbaarheid en levensduur van lokale variabelen

Wordt nog aan gewerkt!

static

7.1.6 Recursieve functies

Een functie die *zichzelf* aanroept wordt een recursieve functie genoemd. Een veel gebruikt voorbeeld van een toepassing van een recursieve functie is het berekenen van de faculteit van een natuurlijk getal n. De faculteit van n is gedefinieerd zoals gegeven in vergelijking (7.1).

$$n! = \prod_{i=1}^{n} i = 1 \cdot 2 \cdot 3 \cdot \cdots \cdot n \tag{7.1}$$

Dus bijvoorbeeld $4! = 1 \cdot 2 \cdot 3 \cdot 4 = 24$. Verder is afgesproken: $0! = 1$. Deze wiskundige functie is ook recursief te definiëren[22], zie vergelijking (7.2).

$$n! = \begin{cases} 1, & \text{als } n <= 1 \\ n \cdot (n-1)!, & \text{als } n > 1 \end{cases} \tag{7.2}$$

Dus 4! kan ook als volgt berekend worden: $4! = 4 \cdot 3! = 4 \cdot 3 \cdot 2! = 4 \cdot 3 \cdot 2 \cdot 1! = 4 \cdot 3 \cdot 2 \cdot 1 = 24$.

Als je deze recursieve definitie codeert in C, krijg je een recursieve functie, zie listing 7.14.

```c
#include <stdio.h>

double faculteit(unsigned int n)
{
    if (n <= 1)
    {
        return 1;
    }
    return n * faculteit(n - 1);
}

int main(void)
{
    printf("20! = %.0f\n", faculteit(20));
    return 0;
}
```

Listing 7.14: Een *recursieve* functie om de faculteit van een natuurlijk getal te berekenen, zie `faculteit_recursief.c`.

Omdat de faculteit gedefinieerd is voor natuurlijke getallen (gehele getallen groter dan of gelijk aan nul) is als type van de parameter **unsigned int** gekozen (een geheel getal zonder teken). Omdat de waarde van de facuteit van grotere getallen al snel erg groot wordt, is als returntype het type **double** gekozen. De waarde van 13! past bijvoorbeeld al niet meer in een (32-bits) **unsigned int**. De waarde van 171! past ook al niet meer in een **double**. Het

[22] In een recursieve definitie wordt hetgeen gedefinieerd wordt in de definitie zelf gebruikt.

nadeel van het gebruik van **double** als returntype is dat de nauwkeurigheid van een **double** beperkt is tot ongeveer 17 significante cijfers[23].

Het is in dit geval echter helemaal niet nodig om een recursieve functie te gebruiken. Je kunt ook 'gewoon' een herhalings-statement gebruiken om $n!$ te berekenen, zie listing 7.15. Een functie die een herhalings-statement gebruikt wordt, met een mooi woord, een *iteratieve* functie genoemd. De meeste mensen zullen de code uit listing 7.15 eenvoudiger te begrijpen vinden dan de code uit listing 7.14.

```
#include <stdio.h>

double faculteit(unsigned int n)
{
    double resultaat = 1;
    for (unsigned int i = 2; i <= n; i++)
    {
        resultaat = resultaat * i;
    }
    return resultaat;
}

int main(void)
{
    printf("23! = %.0f\n", faculteit(23));
    return 0;
}
```

Listing 7.15: Een *iteratieve* functie om de faculteit van een natuurlijk getal te berekenen, zie `faculteit_iteratief.c`.

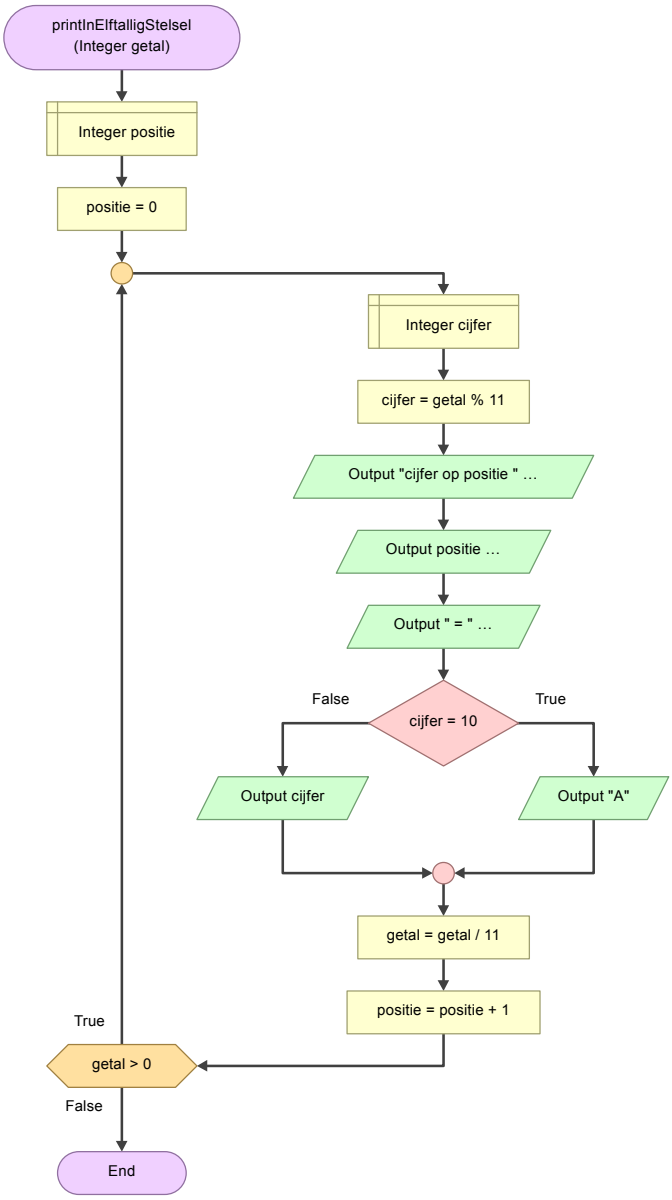
Een recursieve aanpak is vooral handig als er nog code wordt uitgevoerd *na* de recursieve aanroep. Stel dat je een natuurlijk getal wilt afdrukken in het elftallig stelsel[24], dan kun je gebruik maken van het iteratieve algoritme dat gegeven is in figuur 7.6.

Als je het getal 361 met behulp van dit algoritme omzet naar het elftallig stelsel, dan produceert dit algoritme de uitvoer die is gegeven in figuur 7.7.

Als je echter het elftallige resultaat op de normale manier wilt afdrukken als `2A9`, dan is dat lastig omdat de cijfers in de verkeerde volgorde worden gegenereerd door het iteratieve algoritme. Je moet dus de gegenereerde cijfers ergens opslaan en ze aan het einde in de omgekeerde volgorde afdrukken. Dit zou je kunnen doen door een zogenoemde array te gebruiken, zie paragraaf 8.1 verderop in dit dictaat. Maar je zou ook gebruik kunnen maken van het recursieve algoritme dat gegeven is in figuur 7.8

[23] De exacte waarde van $23! = 25852016738884976640000$ maar de functies uit listing 7.14 geeft als resultaat 25852016738884978212864. Je ziet dat de eerste 16 cijfers correct zijn, de laatste 7 cijfers zijn echter niet juist. De functies geven dus slecht een benadering van $n!$ voor grotere waarden van n.

[24] Het elftallig stelsel is een positioneel talstelsel met elf als grondtal. Dit stelsel kent de cijfers 0 t/m 9 en A. Het getal 2A9 in het elftallig stelsel komt overeen met het getal $2 \times 11^2 + 10 \times 11 + 9 = 361$ in het tientallig stelsel. Het elftallig stelsel zou met Carnaval van pas kunnen komen, zie `https://www.telegraaf.nl/nieuws/1037460/gek-getal`.

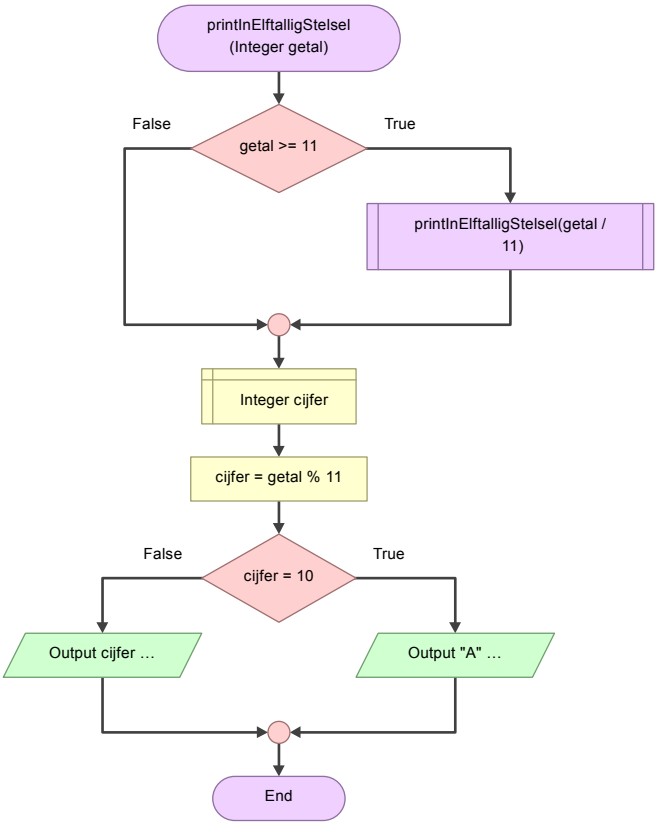


Figuur 7.6: Een iteratief algoritme om een getal om te zetten naar het elftallig stelsel.

```
cijfer op positie 0 = 9
cijfer op positie 1 = A
cijfer op positie 2 = 2
```

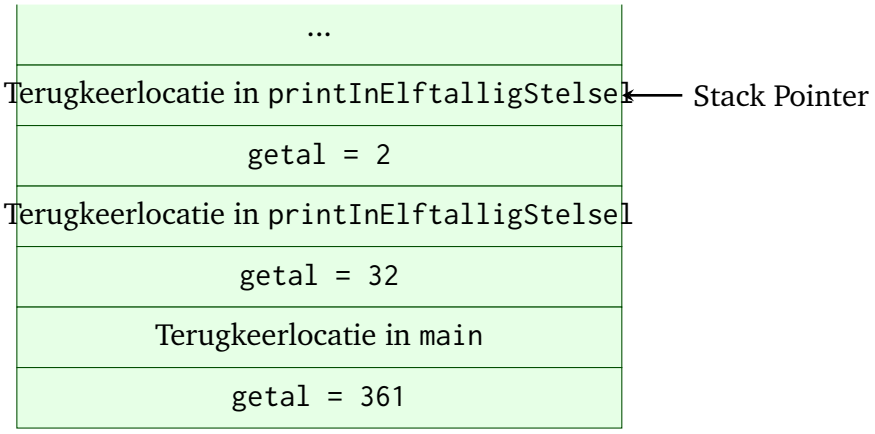
Figuur 7.7: De uitvoer van het algoritme uit figuur 7.6 als getal gelijk is aan 361.

Dit programma maakt slim gebruik van het feit dat functies die achtereenvolgens aangeroepen worden in de omgekeerde volgorde terugkeren (LIFO, zie pagina 39). Als we ervan uitgaan dat het argument dat wordt doorgegeven aan de functie `printInElftalligStel-`



Figuur 7.8: Een recursief algoritme om een getal om te zetten naar het elftallig stelsel.

sel[25] voorafgaande aan de functieaanroep op de stack wordt gezet, dan ziet de stack er nadat de functie voor de derde keer is aangeroepen uit zoals gegeven in figuur 7.9.



Figuur 7.9: De stack op het moment dat de functie `printInElftalligStelsel` drie maal is aangeroepen.

De implementatie in de programmeertaal C van het algoritme gegeven in figuur 7.8 is gegeven in listing 7.16

[25] In dit voorbeeld worden in de functienaam in plaats van lage streepjes tussen de woorden hoofdletters gebruikt om de scheiding tussen de woorden aan te geven. Dus `printInElftalligStelsel` in plaats van `print_in_elftallig_stelsel`. De notatie met hoofdletters als scheiding wordt camelCase genoemd, de notatie met lage streepjes als scheiding wordt snake_case genoemd. In dit dictaat wordt bij voorkeur snake_case gebruikt omdat dit in C gebruikelijk is, het wordt bijvoorbeeld aangeraden in de *Embedded C Coding Standard*[1]. In dit voorbeeld wordt echter canelCase gebruikt om de eenvoudige reden dat de tool Flowgorithm waarmee de flowcharts uit figuren 7.6 en 7.8 zijn gemaakt geen snake_case ondersteund.

```c
#include <stdio.h>

void printInElftalligStelsel(unsigned int getal)
{
    if (getal >= 11)
    {
        printInElftalligStelsel(getal / 11);
    }
    int cijfer = getal % 11;
    if (cijfer < 10)
    {
        printf("%d", cijfer);
    }
    else
    {
        printf("A");
    }
}

int main(void)
{
    printInElftalligStelsel(361);
    return 0;
}
```

Listing 7.16: De implementatie in de programmeertaal C van het in figuur 7.8 gegeven algoritme, zie `elftallig_recursief.c`.

Omdat het erg belangrijk is dat je goed begrijpt hoe dit programma door de processor wordt uitgevoerd, wordt dit hier stap voor stap besproken. In de functie `main` wordt de functie `printInElftalligStelsel` aangeroepen met als argument `361`. Deze waarde wordt op de stack geplaatst en daarna wordt naar de code van de functie gesprongen. Het terugkeeradres in `main` wordt op de stack geplaatst (bovenop de waarde `361`). In de functie wordt de plaats op de stack waar de waarde van het argument is geplaatst aangeduid met de variabelenaam `getal`. Omdat de waarde van `getal` groter is dan 11 wordt de functie `printInElftallig-Stelsel` nogmaals aangeroepen, nu moet `getal / 11` als argument. `getal` heeft de waarde `361` dus de waarde van het argument is `361 / 11` = `32`. Deze waarde wordt op de stack geplaatst en daarna wordt naar de code van de functie gesprongen. Het terugkeeradres in `printInElftalligStelsel` wordt op de stack geplaatst (bovenop de waarde `32`). In de functie wordt de plaats op de stack waar de waarde van het argument is geplaatst aangeduid met de variabelenaam `getal`. Omdat de waarde van `getal` groter is dan 11 wordt de functie `printInElftalligStelsel` nogmaals aangeroepen, nu moet `getal / 11` als argument. `getal` heeft de waarde `32` dus de waarde van het argument is `32 / 11` = `2`. Het terugkeeradres in `printInElftalligStelsel` wordt op de stack geplaatst (bovenop de waarde `2`). De inhoud van de stack is nu weergegeven in figuur 7.9.

De functie `printInElftalligStelsel` wordt nu niet meer aangeroepen. De variabele `cijfer` wordt aangemaakt en krijgt de waarde `getal % 11`. `getal` heeft de waarde `2` dus cijfer krijgt

de waarde 2, dit cijfer wordt vervolgens afgedrukt. Aan het einde van de functie `printInElftalligStelsel` wordt het terugkeeradres en het argument van de stack verwijderd. We keren dus terug naar de plaats waar `printInElftalligStelsel` in `printInElftalligStelsel` is aangeroepen en dat is het einde van het **`if`**-statement. De variabele `getal` heeft in deze functie de waarde 32. De variabele `cijfer` wordt aangemaakt en krijgt de waarde `getal % 11`. `getal` heeft de waarde 32 dus cijfer krijgt de waarde 10, vervolgens wordt het 'cijfer' A afgedrukt. Aan het einde van de functie `printInElftalligStelsel` wordt het terugkeeradres en het argument van de stack verwijderd. We keren dus terug naar de plaats waar `printInElftalligStelsel` in `printInElftalligStelsel` is aangeroepen en dat is het einde van het **`if`**-statement. De variabele `getal` heeft in deze functie de waarde 361. De variabele `cijfer` wordt aangemaakt en krijgt de waarde `getal % 11`. `getal` heeft de waarde 361 dus cijfer krijgt de waarde 9, dit cijfer wordt vervolgens afgedrukt. Aan het einde van de functie `printInElftalligStelsel` wordt het terugkeeradres en het argument van de stack verwijderd. We keren dus terug naar de plaats waar `printInElftalligStelsel` in `main` is aangeroepen en het programma wordt beëindigd.

In listing 7.17 is een soortgelijk voorbeeld gegeven van een recursieve functie met twee parameters. De functie `print_in_talstelsel` zal de integer (zonder teken) die als eerste argument wordt meegegeven afdrukken in het als tweede argument meegegeven talstelsel. De uitvoer van het in listing 7.17 gegeven programma is te zien in figuur 7.10.

```
1111101000
2626
1000
3E8
CAFE
```

Figuur 7.10: De uitvoer van het programma uit listing 7.17.

Twee andere leerzame toepassingen van recursieve functies zijn:

- Een programma dat sudoku's op kan lossen: sudoku.pdf
- Een programma dat Boter Kaas en Eieren kan spelen: Tic-Tac-Toe.pdf

7.1.7 Functie als parameter

Wordt nog aan gewerkt!

7.2 Programma verdelen over meerdere bestanden

Wordt nog aan gewerkt!

`static`

```c
#include <stdio.h>
#include <assert.h>

void print_in_talstelsel(unsigned int getal, int grondtal)
{
    assert(grondtal > 1 && grondtal < 38);
    if (getal >= grondtal)
    {
        print_in_talstelsel(getal / grondtal, grondtal);
    }
    int cijfer = getal % grondtal;
    if (cijfer < 10)
    {
        printf("%d", cijfer);
    }
    else
    {
        printf("%c", 'A' - 10 + cijfer);
    }
}

int main(void)
{
    print_in_talstelsel(1000, 2);
    printf("\n");
    print_in_talstelsel(1000, 7);
    printf("\n");
    print_in_talstelsel(1000, 10);
    printf("\n");
    print_in_talstelsel(1000, 16);
    printf("\n");
    print_in_talstelsel(51966, 16);
    printf("\n");
    return 0;
}
```

Listing 7.17: Een voorbeeld van een recursieve functie, zie `print_in_talstelsel.c`.

7.3 Deel van een programma opnemen in een library

Wordt nog aan gewerkt!

8

Structuren van data

De data die je gebruikt in je programma kun je structureren door zogenoemde *arrays* of *structuren* (Engels: stuctures) te gebruiken. Variabelen die bij elkaar horen, kunnen met behulp van arrays en structs gegroepeerd worden. Deze samengestelde types worden in de C-standaard derived types genoemd.

8.1 Arrays

In een array moeten alle elementen van hetzelfde type zijn (omdat het anders niet mogelijk is om snel het adres van het n^{de} element te berekenen). Een element uit de array wordt geselecteerd met behulp van een *index*. Het eerste element heeft index 0, het tweede element index 1 enz. In een array genaamd `rij` kun je het derde element selecteren met de expressie `rij[2]`.

8.1.1 Definiëren van een array

Als voorbeeld gebruiken we een windmeter die elke seconde de windkracht in km/h meet. De nauwkeurigheid van de meting is ± 0,5 km/h. De meetresultaten kunnen dus als gehele getallen worden opgeslagen. Alle metingen gedurende een minuut worden opgeslagen in een array van 60 elementen. In plaats van een array zou je ook 60 losse variabelen kunnen gebruiken. Maar dat is natuurlijk veel minder handig. De array `windmeting_per_seconde` waarin 60 metingen kunnen worden opgeslagen kan als volgt gedefinieerd worden:

```
int windmeting_per_seconde[60];
```

De array bevat 60 **`int`**'s, het eerste element heeft index 0 en het laatste element heeft index 59.

Het is ook mogelijk om een array meteen te initialiseren. Een array met alle perfecte getallen[26] die in 32-bits passen kan als volgt gedefinieerd worden:

[26] Een perfect getal is een positief natuurlijk getal dat gelijk is aan de som van zijn echte delers, zie eventueel `https://nl.wikipedia.org/wiki/Perfect_getal`.

```
int perfect_getal[5] = {6, 28, 496, 8128, 33550336};
```

In dit geval kan de compiler het aantal elementen in de array zelf bepalen. Je kunt dus ook de volgende code gebruiken:

```
int perfect_getal[] = {6, 28, 496, 8128, 33550336};
```

Als de lijst korter is dan het aantal opgegeven elementen, dan worden de overige elementen met nullen geïnitialiseerd:

```
int windmeting_per_seconde[60] = {10, 11, 10, 5}; // element 4 ↩
 ↪ t/m 59 zijn 0
```

Wil je een array met nullen initialiseren, dan kun je volstaan met één nul:

```
int windmeting_per_seconde[60] = {0}; // alle elementen zijn 0
```

Een globale array wordt overigens automatisch met nullen gevuld.

Een *constante array* moet altijd geïnitialiseerd worden en de elementen mogen niet gewijzigd worden:

```
const int perfect_getal[] = {6, 28, 496, 8128, 33550336}; // ↩
 ↪ read-only
```

8.1.2 Gebruiken van een array

Je kunt een element selecteren door gebruik te maken van blokhaken `[]` met daartussen het elementnummer. Dit elementnummer mag ook berekend worden met een expressie. Om het derde perfecte getal toe te kennen aan de variabele `i` gebruik je de toekenning:

```
i = perfect_getal[2];
```

Let erop dat het derde getal uit de array index `2` heeft, omdat de elementen genummerd worden vanaf `0`.

Om de waarde van de variabele `gemeten_wind` toe te kennen aan het tiende element van de array `windmeting_per_seconde` gebruik je de toekenning:

```
windmeting_per_seconde[9] = gemeten_wind;
```

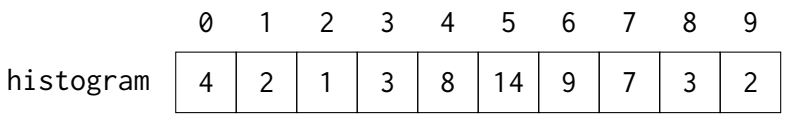
Je kunt, uiteraard alleen de elementen gebruiken die gedefinieerd zijn, dus de expressies

```
windmeting_per_seconde[-1] = gemeten_wind;
windmeting_per_seconde[60] = gemeten_wind;
windmeting_per_seconde[10000] = gemeten_wind;
```

zijn niet correct. Vreemd genoeg geeft de compiler bij het vertalen van deze expressies geen foutmelding. Bij het uitvoeren van het programma wordt de waarde van de variabele `gemeten_wind` weggeschreven op de plaats in het geheugen waar het niet bestaande array element zich zou bevinden als de array wel groot genoeg zou zijn. Maar op deze plaats in het geheugen bevindt zich misschien waardevolle informatie, die dan onbedoeld overschreven wordt. Dit kan tot gevolg hebben dat het programma zich ‘vreemd’ gaat gedragen omdat

een andere variabele van het programma overschreven wordt. Ook kan het gevolg zijn dat het programma door het operating systeem beëindigd wordt omdat de plaats waar het programma probeert te schrijven niet toegankelijk is voor het betreffende programma. De programmeur moet dus zelf goed in de gaten houden dat de indexen die gebruikt worden correct zijn.

Een prettige manier om een array voor te stellen is om de array als een rij vierkante hokjes te tekenen. Boven de hokjes schrijven we de elementnummers. In de hokjes schrijven we de waarden (of inhouden) van de elementen. Een voorbeeld is te zien in figuur 8.1.



Figuur 8.1: Uitbeelding van de array `histogram` met tien elementen.

We gaan nu een aantal voorbeelden bekijken waarbij we ervan uitgaan dat in de array `histogram` de resultaten van een toets zijn opgeslagen. In het eerste element (met index `0`) is het aantal studenten opgeslagen dat het cijfer één heeft behaald. In figuur 8.1 kun je bijvoorbeeld zien dat twee studenten een tien hebben behaald.

In listing 8.1 is een programma gegeven dat de verdeling van de toetsresultaten grafisch weergeeft.

```
#include <stdio.h>

int main()
{
    int histogram[] = {4, 2, 1, 3, 8, 14, 9, 7, 3, 2};
    for (int i = 0; i < 10; i++)
    {
        printf("%2d ", i + 1);
        for (int j = 0; j < histogram[i]; j++)
        {
            printf("*");
        }
        printf("\n");
    }
    return 0;
}
```

Listing 8.1: Een programma om het histogram weer te geven, zie `array1.c`.

De uitvoer van dit programma is gegeven in figuur 8.2.

Het is natuurlijk een beetje vreemd dat de compiler bij het definiëren van de array `histogram` zelf kan bepalen hoeveel elementen deze array heeft, namelijk tien, maar dat je dat in het **`for`**-statement toch zelf weer moet tellen en specificeren. Het is mogelijk om de compiler zelf te laten uitrekenen wat het aantal elementen van een array is. Dit doe je met behulp van de operator **`sizeof`**. Deze operator kun je gebruiken om de grootte van een variabele op te vragen. Als je de expressie **`sizeof`** `histogram` gebruikt, dan zal dit door de compiler

```
 1 ****
 2 **
 3 *
 4 ***
 5 ********
 6 ***************
 7 *********
 8 *******
 9 ***
10 **
```

Figuur 8.2: De uitvoer van het programma uit listing 8.1.

vervangen worden door het aantal *bytes* dat de variabele `histogram` inneemt in het geheugen. Omdat je niet het aantal bytes maar het aantal *elementen* van de array wilt weten moet je het aantal bytes dat de array groot is nog delen door het aantal bytes dat één element (bijvoorbeeld het eerste element) groot is. Je kunt het aantal elementen van de array `histogram` dus door de compiler laten bepalen met behulp van de expressie:

```
sizeof histogram / sizeof histogram[0]
```

De operator **`sizeof`** geeft een waarde terug van het type `size_t`. Dit is een **`unsigned`** type dat alleen positieve gehele getallen kan bevatten. Het bevat gegarandeerd voldoende bits om de grootst mogelijke grootte in op te slaan. Om deze reden wordt het type `size_t` ook vaak gebruikt om de index van een array in op te slaan. In listing 8.2 is een verbeterde versie gegeven van het programma dat weergegeven is in listing 8.1. Dit programma is beter aanpasbaar omdat het correct blijft werken als het aantal getallen dat gebruikt wordt in de array `histogram` wijzigt. In listing 8.1 zou in dat geval ook de constante `10`, die gebruikt is in het **`for`**-statement, aangepast moeten worden.

```
#include <stdio.h>

int main()
{
    int histogram[] = {4, 2, 1, 3, 8, 14, 9, 7, 3, 2};
    for (size_t i = 0; i < sizeof histogram / sizeof histogram[0]; ↩
     ↪ i++)
    {
        printf("%2zu ", i + 1);
        for (int j = 0; j < histogram[i]; j++)
        {
            printf("*");
        }
        printf("\n");
    }
    return 0;
}
```

Listing 8.2: Verbeterd programma om het histogram weer te geven, zie `array2.c`.

In listing 8.3 is een programma gegeven waarmee het gemiddelde voor de toets behaalde resultaat met behulp van de gegeven uit het histogram berekend kan worden. De uitvoer van dit programma is:

```
Het gemiddeld behaalde resultaat = 5.905660
```

```c
#include <stdio.h>

int main()
{
    int histogram[] = {4, 2, 1, 3, 8, 14, 9, 7, 3, 2};
    int totaal = 0, aantal_resultaten = 0;
    for (size_t i = 0; i < sizeof histogram / sizeof histogram[0]; ↩
     ↪ i++)
    {
        totaal += histogram[i] * (i + 1);
        aantal_resultaten += histogram[i];
    }
    printf("Het gemiddeld behaalde resultaat = %f\n", (double) ↩
     ↪ totaal / aantal_resultaten);
    return 0;
}
```

Listing 8.3: Een programma om het gemiddeld behaalde resultaat te berekenen, zie `array3.c`.

Merk op dat de variabele `totaal` met een cast wordt omgezet naar een floating-point getal. Als je dit niet doet, dan levert de deling een geheel getal als resultaat, in dit geval 5.

8.1.3 Array als parameter van een functie

Wordt nog aan gewerkt!

8.1.4 Definiëren van een string

Wordt nog aan gewerkt!

8.1.5 Gebruiken van een string

Wordt nog aan gewerkt!

8.1.6 Definiëren van een multidimensionale array

Wordt nog aan gewerkt!

8.1.7 Gebruiken van een multidimensionale array

Wordt nog aan gewerkt!

8.1.8 Definiëren van een variable-length array

In versies van C vóór C99 moest het aantal elementen van een array een compile-time constante zijn. Dat wil zeggen dat het aantal elementen van de array tijdens het compileren van het programma bekend moet zijn. Sinds C99 is het echter ook mogelijk om een array te definiëren waarbij het aantal elementen pas tijdens run-time (dus tijdens het uitvoeren van de code) bekend hoeft te zijn. Zo'n array wordt een *variable-length array* of kortweg VLA genoemd[27]. Deze naam is een beetje verwarrend want de lengte (het aantal elementen) van een VLA kan niet wijzigen nadat de VLA is aangemaakt. Bedoeld wordt dat de lengte (het aantal elementen) van de array door middel van een variabele gedefinieerd kan worden. In listing 8.4 is een programma gegeven waarin een VLA wordt aangemaakt. De gebruiker van het programma kan het gewenste aantal elementen invoeren. Nadat de array is aangemaakt wordt het aantal bytes waaruit de array bestaat geprint[28]. Een mogelijke uitvoer van dit programma voor een processor waarbij een **int** 32-bits is, is gegeven in figuur 8.3. De invoer van de gebruiker is hierin groen en onderstreept weergegeven.

```
#include <stdio.h>

int main(void)
{
    int aantal;
    do
    {
        printf("Hoeveel elementen moet de array bevatten? ");
        scanf("%d", &aantal);

    }
    while (aantal <= 0);
    int rij[aantal];
    printf("De array rij is %zu bytes groot.\n", sizeof rij);
    return 0;
}
```

Listing 8.4: Een programma waarin een VAR wordt aangemaakt, zie `vla1.c`.

Een VLA kan niet als globale variabele worden gedefinieerd. Een VLA kan niet geïnitialiseerd worden. Een VLA kan niet als element van een structure (zie paragraaf 8.2 verderop in dit dictaat) worden gedefinieerd.

[27] Vreemd genoeg zijn VLA's in C11 optioneel. Als de constante `__STDC_NO_VLA__` is gedefinieerd als `1`, dan ondersteund de C11-compiler geen VLA's.

[28] Het aantal bytes waaruit een variabele bestaat kan worden opgevraagd met behulp van de operator **sizeof**. Deze operator geeft een waarde terug van het type `size_t` die kan worden afgedrukt door de functie `printf` met behulp van de format specifier `%zu`, zie eventueel `https://en.wikipedia.org/wiki/Printf_format_string`.

```
Hoeveel elementen moet de array bevatten? 100
De array rij is 400 bytes groot.
```

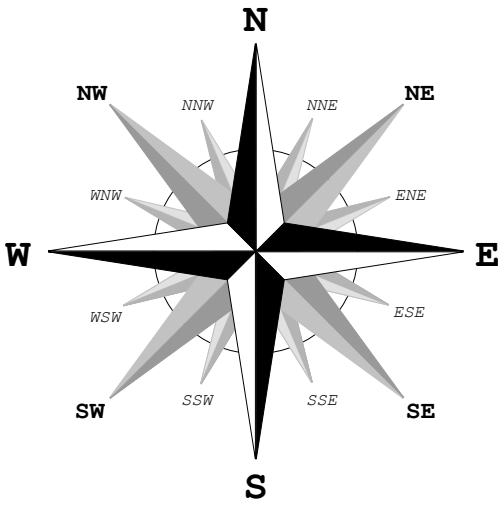
Figuur 8.3: Een mogelijke uitvoer van het programma uit listing 8.4.

8.2 Structures

In een structure kunnen de elementen van verschillende typen zijn. Een element uit de structure wordt geselecteerd met behulp van een naam (*veldnaam*, in het Engels: member name). In een structure genaamd `windmeting` kun je het element genaamd `snelheid` selecteren met de expressie `windmeting.snelheid`.

8.2.1 Definiëren van een structure

Als voorbeeld gebruiken we een windsensor die de windsnelheid en de windrichting meet. Deze sensor geeft de windsnelheid in km/h als geheel getal. De windrichting wordt aangegeven met 1 (N, E, S, W), 2 (NE, SE, SW, NW) of 3 letters (NNE, ENE, ESE, SSE, SSE, WSW, WNW, NNW), zie figuur 8.4.



Figuur 8.4: De verschillende windrichtingen worden door de sensor met 1, 2 of 3 letters weergegeven. Bron `https://en.wikipedia.org/wiki/Cardinal_direction`.

Een structure waarin de door deze windsensor gemeten data kan worden opgeslagen, kan gedefinieerd worden door middel van het keyword **`struct`**[29]:

```
struct wind
{
    int snelheid;
    char richting[4];
};
```

[29] Er is een array van 4 karakters gedefinieerd om de windrichting in op te slaan. Het extra karakter is nodig om het nul-karakter in op te slaan. Een string wordt in C namelijk afgesloten met het karakter `'\0'`

Vervolgens kunnen de variabelen `windmeting1` en `windmeting2` van dit **`struct`**-type worden aangemaakt en gevuld met data, zie: `struct1.c`:

```
int main(void)
{
    struct wind windmeting1, windmeting2;
    windmeting1.snelheid = 9;
    strcpy(windmeting1.richting, "W");
    windmeting2.snelheid = 25;
    strcpy(windmeting2.richting, "NNW");
```

Als je de **`struct`** `wind` alleen in de functie `main` gebruikt kun je deze **`struct`** ook lokaal definiëren, zie `struct2.c`:

```
int main(void)
{
    struct wind
    {
        int snelheid;
        char richting[4];
    };
    struct wind windmeting1, windmeting2;
```

In dit geval mag je de declaratie van de **`struct`** en de definitie van de variabelen combineren, zie `struct3.c`:

```
int main(void)
{
    struct wind
    {
        int snelheid;
        char richting[4];
    } windmeting1, windmeting2;
```

Omdat je nu de naam van de **`struct`** (in de C-standaard de *structure tag* genoemd) niet meer nodig hebt kun je die ook weglaten, zie `struct4.c`:

```
int main(void)
{
    struct
    {
        int snelheid;
        char richting[4];
    } windmeting1, windmeting2;
```

Je kunt een **`struct`**-variabele (net zoals een normale variabele) bij het aanmaken meteen vullen, zie `struct5.c`:

```
int main(void)
{
    struct
    {
```

```
        int snelheid;
        char richting[4];
    } windmeting1 = {9, "W"}, windmeting2 = {25, "NNW"};
```

Sinds C99 kun je om een stuct-variabele meteen te vullen ook gebruik maken van een zogenoemde *designated initializer*, zie `struct6.c`:

```
int main(void)
{
    struct
    {
        int snelheid;
        char richting[4];
    } windmeting1 = {.snelheid = 9, .richting = "W"},
      windmeting2 = {.richting = "NNW", .snelheid = 25};
```

Deze manier van initialiseren heeft, vergeleken met de traditionele methode, als voordeel dat de volgorde waarin de velden van de **struct** worden geïnitialiseerd vrij te kiezen is. Een nadeel is dat bij een wijziging van een veldnaam ook de initialisatie aangepast moet worden.

In de praktijk wordt een structure vaak globaal gedeclareerd en zijn er functies waarbij dit **struct**-type als parameter wordt gebruikt, zie `struct7.c`:

```
struct wind
{
    int snelheid;
    char richting[4];
};

struct wind meet_wind(void);
int bepaal_gemiddelde_windsnelheid(struct wind meting[], size_t ↩
 ↪ aantal_metingen);
void print_wind(struct wind w);
```

In dit geval is het handig om een *typedefinitie* te gebruiken. Met een typedefinitie kun je, met behulp van het keyword **typedef**, een alias voor een bestaand type definiëren, zie `struct8.c`:

```
struct wind
{
    int snelheid;
    char richting[4];
};
typedef struct wind wind_t;

wind_t meet_wind(void);
int bepaal_gemiddelde_windsnelheid(wind_t meting[], size_t ↩
 ↪ aantal_metingen);
void print_wind(wind_t w);
```

De typenaam `wind_t`[30] kan nu als een alias (andere naam) voor het type **`struct`** `wind` gebruikt geworden. De declaratie van de **`struct`** en de definitie van de typenaam kunnen ook gecombineerd worden:

```
typedef struct wind
{
    int snelheid;
    char richting[4];
} wind_t;
```

Omdat je nu de naam (tag) van de **`struct`** niet meer nodig hebt, kun je die ook weglaten, zie `struct10.c`:

```
typedef struct
{
    int snelheid;
    char richting[4];
} wind_t;
```

Stel dat je de windsensor gebruikt in een weerstation waar naast de wind ook de temperatuur gemeten wordt. De temperatuursensor meet de temperatuur in °C en geeft een **`float`**-waarde. Het type `wind_t` kan dan gebruikt worden als veld in een nieuw type genaamd `Weerdata`, zie `struct11.c`:

```
typedef struct
{
    int snelheid;
    char richting[4];
} wind_t;

typedef struct
{
    float temperatuur;
    wind_t wind;
} Weerdata;
```

Als het type `wind_t` niet als zelfstandig type wordt gebruikt, dan kan het type `Weerdata` ook als volgt gedefinieerd worden, zie `struct11a.c`:

```
typedef struct
{
    float temperatuur;
    struct {
        int snelheid;
        char richting[4];
    } wind;
} Weerdata;
```

[30] In dit dictaat is ervoor gekozen om namen van zelfgedefinieerde types te laten eindigen op `_t`. Dit wordt aangeraden in *Embedded C Coding Standard*[1].

8.2.2 Gebruiken van een stucture

Een variabele van een **struct**-type kan gebruikt worden net zoals elke andere variabele. Zo'n variabele kan toegekend worden aan een andere variabele van hetzelfde type met behulp van de **=**-operator. Als de variabelen `w1` en `w2` beide van het hierboven gedefinieerde type `wind_t` zijn dan zorgt de toekenning `w1 = w2;` ervoor dat alle data uit de variabele `w2` gekopieerd wordt naar `w1` Een variabele van een **struct**-type kan ook als parametertype en als return type van een functie gebruikt worden.

Met behulp van de **==**-operator kun je bepalen of twee variabelen van hetzelfde **struct**-type gelijk zijn. De Booleaanse expressie `w1 == w2` geeft de waarde **true** als `w1` en `w2` exact dezelfde data bevatten en **false** als dit niet zo is.

Bij het gebruik van een **struct**-variabele kun je een veld benaderen door de `.`-operator te gebruiken: `variabele_naam.veld_naam`. De hierboven gedeclareerde functies `print+wind` en `bepaal_gemiddelde_windsnelheid` kun je bijvoorbeeld als volgt definiëren, zie `struct10.c`:

```
int bepaal_gemiddelde_windsnelheid(wind_t meting[], size_t ↩
 ↪ aantal_metingen)
{
    int totaal = 0;
    for (size_t i = 0; i < aantal_metingen; i++)
    {
        totaal += meting[i].snelheid;
    }
    return totaal / aantal_metingen;
}

void print_wind(wind_t w)
{
    printf("%d km/h %s", w.snelheid, w.richting);
}
```

Een variabele van het hierboven gedefinieerde type `Weerdata` kun je als volgt initialiseren en afdrukken, zie `struct11.c`:

```
    Weerdata weermeting = {17.3, {25, "WSW"}};
    printf("Weermeting: %4.1f C %3d km/h %-3s\n", ↩
     ↪ weermeting.temperatuur, weermeting.wind.snelheid, ↩
     ↪ weermeting.wind.richting);
```

Omdat het veld genaamd `wind` in het **struct**-type `Weerdata` zelf ook een **struct**-type is, moet je bij het initialiseren van het veld `wind` extra accolades gebruiken. Als je de `snelheid` van de `wind` die is opgeslagen in de variabele `weermeting` wilt benaderen moet je de `.`-operator twee keer achter elkaar gebruiken:

```
    weermeting.wind.snelheid
```

Stel er zijn verschillende windmetingen opgeslagen. Nu blijkt dat de windsensor verkeerd geijkt is en alle gemeten windsnelheden 2 km/h te laag zijn. Je wilt nu deze metingen corrigeren. Daartoe is de volgende functie gedeclareerd, zie `struct12.c`:

```
void corigeer_meting(wind_t *pw);
```

Als parameter is een `wind_t`-pointer gebruikt zodat de functie `corigeerMeting` de aan de functie meegegeven meting kan wijzigen (call by reference). Een testprogramma om deze functie te testen is als volgt geïmplementeerd, zie `struct12.c`. Bij aanroep van de functie `corigeerMeting` wordt het adres van de `meting`, die gecorrigeerd moet worden, meegegeven.

```
int main(void)
{
    wind_t meting = {25, "WNW"};
    printf("%3d km/h %s\n", meting.snelheid, meting.richting);
    corigeer_meting(&meting);
    printf("Na correctie:\n");
    printf("%3d km/h %s\n", meting.snelheid, meting.richting);
    return 0;
}
```

De functie `corigeerMeting` kan als volgt geïmplementeerd worden, zie `struct12.c`:

```
void corigeer_meting(wind_t *pw)
{
    (*pw).snelheid += 2;
}
```

De haakjes in het statement `(*pw).snelheid += 2;` zijn noodzakelijk omdat de `.`-operator een hogere prioriteit heeft dan de `*`-operator. Omdat dit een vervelende syntax is, hebben de ontwerpers van C de `->`-operator aan de taal toegevoegd. De functie `corigeerMeting` kan met behulp van deze operator ook als volgt geïmplementeerd worden, zie `struct13.c`:

```
void corigeer_meting(wind_t *pw)
{
    pw->snelheid += 2;
}
```

9

Bitn**ken

Werktuigbouwkundig ingenieurs worden wel eens oneerbiedig aangesproken met de vakterm: 'fietsenmaker'. Zo bestaat er ook een vakterm voor E-ingenieurs die zich bezighouden met het programmeren van microcontrollers: 'bitn..kers'[31]. Op deze pagina wordt uitgelegd op welke manieren je met bitjes kunt spelen en hoe je dat in C (veilig ;-) moet doen.

De gegeven voorbeelden kunnen uitgevoerd worden op de MSP-EXP430G2 Launchpad met een MSP430G2553 microcontroller. Dit ontwikkelbord heeft een rode led (LED1) pin P1.0, een groene led (LED2) op pin P1.6 en een drukknop (S2) naar aarde op pin P1.3.

9.1 Bitje veranderen

De onderstaande voorbeelden veranderen een bitje in het `P1OUT`-register van de MSP430G2553. Het `P1DIR`-register moet dan wel geladen worden met `0b01000001`[32] om pin 0 en pin 6 van poort P1 op output te zetten (met deze pinnen stuur je de 2 ledjes) .

9.1.1 Bitje setten

Je kunt een bitje setten (1 maken) met behulp van een bitwise-or-operator. Je moet het bitje dat je wilt setten or-en met `1` en de rest met `0`. Om dus bijvoorbeeld pin P1.6 één te maken moet je `P1OUT` or-en met het binaire getal: `01000000`.

```
#include <msp430.h>

int main(void)
{
    WDTCTL = WDTPW | WDTHOLD; // Stop watchdog timer.
```

[31] Zie: `http://ti-aalst.powerhost.be/index.php?show=woordenboek&letter=B&id=178`

[32] De prefix `0b` kan in de C-compiler voor de MSP430 gebruikt worden om een getal in binaire notatie in te voeren. In standaard C is dit echter *niet* mogelijk. Daarom is het eigenlijk beter om altijd de hexadecimale notatie te gebruiken, in dit geval `0x41`, omdat deze notatie door elke standaard C-compiler wordt ondersteund. Om de leesbaarheid te vergroten wordt in dit hoofdstuk toch de binaire notatie gebruikt.

```
    P1DIR = 0b01000001; // Maak P1.0 en P1.6 uitgangen en alle ↩
     ↪ andere pinnen van P1 ingangen.
    P1OUT = P1OUT | 0b01000000; // Alleen pin P1.6 hoog maken, de ↩
     ↪ overige pinnen van poort P1 worden niet gewijzigd.
    while (1);
    return 0;
}
```

De regel:

```
    P1OUT = P1OUT | 0b01000000;
```

kun je ook verkorten tot:

```
    P1OUT |= 0b01000000;
```

Let op! Er zit een groot verschil tussen de bitwise-or-operator `|` en de logical-or-operator `||`. Bij de bitwise-or wordt de or-bewerking bit-voor-bit uitgevoerd. `0b00101100|0b00001001` is dus gelijk aan `0b00101101`. Bij de logical-or wordt het getal omgezet naar een logische (binaire) waarde (**`true`** of **`false`**). Daarna wordt de or-bewerking uitgevoerd met als resultaat **`true`** (`1`) of **`false`** (`0`).

`0b00101100||0b00001001` wordt dus omgezet in **`true||true`** en is dus gelijk aan **`true`** (`0b00000001`). Als in het bovenstaande programma de `|`-operator vervangen wordt door de `||`-operator, dan wordt pin P1.6 (de groene led) *niet* geset, maar pin P1.0 (de rode led) wel! Begrijp je dat?

Er is nog een verschil tussen de bitwise-or-operator `|` en de logical-or-operator `||`. Bij de logical-or-operator worden de operanden van links naar rechts uitgerekend en zodra het antwoord bekend is wordt de berekening gestopt. Dit wordt short-circuit evaluation genoemd. Bij de bitwise-or wordt de expressie altijd helemaal doorgerekend. Voorbeeld: als de expressie `fun1()||fun2()` wordt uitgevoerd en `fun1()` geeft **`true`** terug dan wordt `fun2()` *niet* aangeroepen (het antwoord van de expressie is **`true`**). Als `fun1()|fun2()` wordt uitgevoerd dan worden `fun1()` en `fun2()` altijd beide aangeroepen (ook als `fun1()` allemaal enen teruggeeft).

Er is nog een subtiel verschil. Bij de logical-or-operator ligt de evaluatievolgorde van de operanden vast maar bij de bitwise-or niet. Voorbeeld: als de expressie `fun1()||fun2()` wordt uitgevoerd wordt `fun1()` als eerste aangeroepen (`fun2()` wordt mogelijk helemaal niet aangeroepen). Als `fun1()|fun2()` wordt uitgevoerd dan is het afhankelijk van de compiler of eerst `fun1()` of eerst `fun2()` wordt aangeroepen (ze worden wel gegarandeerd beide aangeroepen).

9.1.2 Bitje clearen

Je kunt een bitje clearen (`0` maken) met behulp van een bitwise-and-operator. Je moet het bitje dat je wilt clearen and-en met `0` en de rest met `1`. Om dus bijvoorbeeld pin P1.6 nul te maken moet je `P1OUT` and-en met het binaire getal: `10111111`.

```
#include <msp430.h>

int main(void)
{
    WDTCTL = WDTPW | WDTHOLD; // Stop watchdog timer.
```

```
    P1DIR = 0b01000001; // Maak P1.0 en P1.6 uitgangen en alle ↩
     ↪ andere pinnen van P1 ingangen.
    P1OUT = P1OUT & 0b10111111; // Alleen pin P1.6 laag maken, de ↩
     ↪ overige pinnen van poort P1 worden niet gewijzigd.
    while (1);
    return 0;
}
```

De regel:

```
P1OUT = P1OUT & 0b10111111;
```

kun je ook verkorten tot:

```
P1OUT &= 0b10111111;
```

Het is ook mogelijk om bij het clearen hetzelfde bitpatroon te gebruiken als bij het setten. Je moet dan de compiler zelf de inverse laten uitrekenen door middel van de bitwise-not-operator ~:

```
P1OUT &= ~0b01000000;
```

Let op! Er zit een groot verschil tussen de bitwise-and-operator & en de logical-and-operator &&. Bij de bitwise-and wordt de and-bewerking bit-voor-bit uitgevoerd. `0b00101100&0b00001001` is dus gelijk aan `0b00001000`. Bij de logical-and wordt het getal omgezet naar een logische (binaire) waarde (**`true`** of **`false`**). Daarna wordt de and-bewerking uitgevoerd met als resultaat **`true`** (1) of **`false`** (0). `0b00101100&&0b00001001` wordt dus omgezet in **`true`**`&&`**`true`** en is dus gelijk aan **`true`** (`0b00000001`). Als in het bovenstaande programma de &-operator vervangen wordt door de &&-operator, dan wordt pin P1.6 (de groene led) toevallig *wel* nul gemaakt, maar wordt pin P1.0 (de rode led) bovendien geset! Begrijp je dat?

Er is nog een verschil tussen de bitwise-and-operator & en de logical-and-operator &&. Bij de logical-and-operator worden de operanden van links naar rechts uitgerekend en zodra het antwoord bekend is wordt de berekening gestopt. Dit wordt short-circuit evaluation genoemd. Bij de bitwise-and wordt de expressie altijd helemaal doorgerekend. Voorbeeld: als de expressie `fun1()&&fun2()` wordt uitgevoerd en `fun1()` geeft **`false`** terug dan wordt `fun2()` *niet* aangeroepen (het antwoord van de expressie is **`false`**). Als `fun1()&fun2()` wordt uitgevoerd dan worden `fun1()` en `fun2()` altijd beide aangeroepen (ook als `fun1()` allemaal nullen teruggeeft).

Er is nog een subtiel verschil. Bij de logical-and-operator ligt de evaluatievolgorde van de operanden vast maar bij de bitwise-and niet. Voorbeeld: als de expressie `fun1()&&fun2()` wordt uitgevoerd wordt `fun1()` als eerste aangeroepen (`fun2()` wordt mogelijk helemaal niet aangeroepen). Als `fun1()&fun2()` wordt uitgevoerd dan is het afhankelijk van de compiler of eerst `fun1()` of eerst `fun2()` wordt aangeroepen (ze worden wel gegarandeerd beide aangeroepen).

Let op! Er zit een groot verschil tussen de bitwise-not-operator ~ en de logical-not-operator !. Bij de bitwise-not wordt de not-bewerking bit-voor-bit uitgevoerd. `~0b00101100` is dus gelijk aan `0b11010011`. Bij de logical-not wordt het getal omgezet naar een logische (binaire) waarde (**`true`** of **`false`**). Daarna wordt de not bewerking uitgevoerd met als resultaat **`true`** (1) of **`false`** (0). `!0b00101100` is dus gelijk aan `0b00000000`.

9.1.3 Bitje flippen

Je kunt een bitje flippen (inverteren) met behulp van een bitwise-exor-operator. Je moet het bitje dat je wilt flippen exor-en met 1 en de rest met `0`. Om dus bijvoorbeeld pin P1.6 te inverteren moet je `P1OUT` exor-en met het binaire getal: `01000000`.

```
#include <msp430.h>

int main(void)
{
    WDTCTL = WDTPW | WDTHOLD; // Stop watchdog timer.
    P1DIR = 0b01000001; // Maak P1.0 en P1.6 uitgangen en alle ↩
     ↪ andere pinnen van P1 ingangen.
    P1OUT = P1OUT ^ 0b01000000; // Alleen pin P1.6 inverteren, de ↩
     ↪ overige pinnen van poort P1 worden niet gewijzigd.
    while (1);
    return 0;
}
```

De regel:

```
    P1OUT = P1OUT ^ 0b01000000;
```

kun je ook verkorten tot:

```
    P1OUT ^= 0b01000000;
```

Er bestaat in C vreemd genoeg geen logical-exor-operator.

9.1.4 Meerdere bitjes veranderen

Als je meerdere bitjes wilt setten, meerdere bitjes wilt clearen of meerdere bitjes wilt inverteren, dan kun je dat doen door in het bitpatroon waarmee je respectievelijk de bitwise-or, bitwise-and of bitwise-exor uitvoert meerdere bitjes te setten.

In deze paragraaf gaan we ervan uit dat alle pinnen van poort P1 uitgangen zijn. In het onderstaande voorbeeld worden P1.4 en P1.2 hoog gemaakt, P1.5 en P1.1 laag gemaakt en P1.7, P1.6 en P1.0 geïnverteerd:

```
#include <msp430.h>

int main(void)
{
    WDTCTL = WDTPW | WDTHOLD; // Stop watchdog timer.
    P1DIR = 0b11111111; // Maak alle pinnen van poort P1 uitgangen.
    P1OUT = 0b01011010; // Willekeurige test waarde op poort P1.
    P1OUT |= 0b00010100; // Alleen P1.4 en P1.2 hoog maken.
    P1OUT &= ~0b00100010; // Alleen P1.5 en P1.1 laag maken.
    P1OUT ^= 0b11000001; // Alleen P1.7, P1.6 en P1.0 inverteren.
    while (1);
```

```
    return 0;
}
```

9.2 Bitje testen

De onderstaande voorbeelden testen een bitje in het P1IN-register van de MSP430G2553. Op de MSP-EXP430G2 Launchpad kan pin P1.3 met aarde verbonden worden door drukknop S2 in te drukken. Er is geen pull-up weerstand aangesloten op deze pin. Dus als de drukknop S2 niet wordt ingedrukt, dan is pin P1.3 nergens mee verbonden. In de onderstaande programma's gebruik je P1.3 zodat je het gedrag van het programma met S2 kunt testen. Als de test **true** oplevert, wordt pin P1.6 hoog gemaakt (het groene ledje wordt aangezet) en als de test **false** oplevert, wordt pin P1.6 laag gemaakt (het groene ledje wordt uitgezet) zodat je meteen het resultaat van de test kunt zien. Het P1DIR-register moet dan wel geladen worden met 0b01000001 om de pinnen van poort P1 die met de leds zijn verbonden op output te zetten en de overige pinnen van poort P1, waaronder degene die verbonden is met S2, op input te zetten.

9.2.1 Is het bitje laag?

Je kunt testen of een bitje laag is door dit bitje te 'isoleren' van de andere bitjes in de betreffende variabele. De overige bits worden gemaskeerd. Je kunt een bitje isoleren door een bitwise-and-bewerking. Het volgende voorbeeld zal als schakelelaar S2 ingedrukt is (pin P1.3 is dan laag) alleen het groene ledje laten branden (P1.6 wordt hoog gemaakt) en anders (S2 niet ingedrukt) het groene ledje niet laten branden (P1.6 wordt laag gemaakt):

```
#include <msp430.h>

int main(void)
{
    WDTCTL = WDTPW | WDTHOLD; // Stop watchdog timer.
    P1DIR = 0b01000001; // Maak P1.0 en P1.6 uitgangen en alle ↩
     ↪ andere pinnen van P1 ingangen
    P1REN |= 0b00001000; // Zet pull-up weerstand bij P1.3 aan
    P1OUT |= 0b00001000;
    while (1)
    {
        if ((P1IN & 0b00001000) == 0b00001000)
        {
            P1OUT |= 0b01000000;
        }
        else
        {
            P1OUT &= ~0b01000000;
        }
    }
```

```
    return 0;
}
```

Om er voor te zorgen dat de P1.3 hoog is als de schakelaar S2 niet is ingedrukt, wordt intern in de microcontroller een pull-up weerstand aangezet die verbonden is met pin P1.3. Dit gebeurt door de volgende twee regels:

```
    P1REN |= 0b00001000; // Zet pull-up weerstand bij P1.3 aan
    P1OUT |= 0b00001000;
```

De extra haakjes in het **if**-statement zijn noodzakelijk omdat de bitwise-and-operator & een lagere prioriteit heeft dan de vergelijkingsoperator ==.

De regel:

```
        if ((P1IN & 0b00001000) == 0b00000000)
```

kun je ook verkorten tot:

```
        if (!(P1IN & 0b00001000))
```

De expressie `(P1IN & 0b00001000)` geeft namelijk als resultaat `0b00001000` als pin P1.3 hoog is en `0b00000000` als pin P1.3 laag is. `0b00001000` is ongelijk aan nul en wordt dus gezien als de logische waarde **true** en `0b00000000` is gelijk aan nul en wordt dus gezien als de logische waarde **false**. Als je deze logische waarde met een logical-not-operator ! inverteert, krijg je de waarde **true** als pin P1.3 laag is en **false** als P1.3 hoog is. Ook hier zijn extra haakjes in het **if**-statement noodzakelijk omdat de bitwise-and-operator & een lagere prioriteit heeft dan de logical-not-operator !.

9.2.2 Is het bitje hoog?

Je kunt testen of een bitje hoog is door dit bitje te 'isoleren' van de andere bitjes in de betreffende variabele. De overige bits worden gemaskeerd. Je kunt een bitje isoleren door een bitwise-and-bewerking. Het volgende voorbeeld zal als schakelelaar S2 *niet* ingedrukt is (pin P1.3 is dan hoog) alleen het groene ledje laten branden (P1.6 wordt hoog gemaakt) en anders (S2 wel ingedrukt) het groene ledje niet laten branden (P1.6 wordt laag gemaakt):

```
#include <msp430.h>

int main(void)
{
    WDTCTL = WDTPW | WDTHOLD; // Stop watchdog timer.
    P1DIR = 0b01000001; // Maak P1.0 en P1.6 uitgangen en alle ↩
     ↪ andere pinnen van P1 ingangen.
    P1REN |= 0b00001000; // Zet pull-up weerstand bij P1.3 aan.
    P1OUT |= 0b00001000;
    while (1)
    {
        if ((P1IN & 0b00001000) == 0b00000000)
        {
```

```
            P1OUT |= 0b01000000;
        }
        else
        {
            P1OUT &= ~0b01000000;
        }
    }
    return 0;
}
```

De extra haakjes in het **if**-statement zijn noodzakelijk omdat de bitwise-and-operator & een lagere prioriteit heeft dan de vergelijkingsoperator ==.

De regel:

```
if ((P1IN & 0b00001000) == 0b00001000)
```

kun je ook verkorten tot:

```
if (P1IN & 0b00001000)
```

De expressie `P1IN & 0b00001000` geeft namelijk als resultaat `0b00001000` als pin P1.3 hoog is en `0b00000000` als pin P1.3 laag is. `0b00001000` is ongelijk aan nul en wordt dus gezien als de logische waarde **true** en `0b00000000` is gelijk aan nul en wordt dus gezien als de logische waarde **false**. De expressie krijgt dus de waarde **true** als pin P1.3 hoog is en **false** als P1.3 laag is.

9.2.3 Meerdere bitjes testen

Je kunt vaak meerdere bitjes met één bewerking testen door meerdere bitjes te isoleren.

In het onderstaande voorbeeld wordt het groene ledje aangezet zolang P1.3 laag is en P1.5 hoog is (dus als S2 ingedrukt is en ingangspin P1.5 met een draadje met de V_{cc} is verbonden). Als dit niet het geval is (S2 is niet ingedrukt of P1.5 is niet verbonden met V_{cc}) dan wordt het groene ledje uitgezet. Om er voor te zorgen dat pin P1.5 laag is als er geen verbinding is met V_{cc}, wordt intern in de microcontroller een pull-down weerstand aangezet die verbonden is met pin P1.5.

```
#include <msp430.h>
// Ingangspin P1.5 kan hoog gemaakt worden door deze pin via een ↩
 ↪ draadje met de Vcc te verbinden.

int main(void)
{
    WDTCTL = WDTPW | WDTHOLD; // Stop watchdog timer.
    P1DIR = 0b01000001; // Maak P1.0 en P1.6 uitgangen en alle ↩
     ↪ andere pinnen van P1 ingangen.
    P1REN |= 0b00101000; // Zet pull-up weerstand bij P1.3 aan en
    P1OUT |= 0b00001000; // zet pull-down weerstand bij P1.5 aan.
    while (1)
```

```
    {
        if ((P1IN & 0b00101000) == 0b00100000)
        {
            P1OUT |= 0b01000000;
        }
        else
        {
            P1OUT &= ~0b01000000;
        }
    }
    return 0;
```

De onderstaande waarheidstabel kan helpen bij het doorgronden van de werking van het bovenstaande programma. Om ruimte te besparen zijn de constanten in hexadecimale notatie weergegeven: `0b00101000` = `0x28` en `0xb00100000` = `0x20`.

S2	P1.3	P1.5	P1IN&0x28	(PINB&0x28)==0x20	P1.6	led
niet ingedrukt	1	0	0x08	false	0	uit
niet ingedrukt	1	1	0x28	false	0	uit
wel ingedrukt	0	0	0x00	false	0	uit
wel ingedrukt	0	1	0x20	true	1	aan

Let op! De regel:

```
        if((PINB & 0b00101000) == 0b00100000) {
```

kun je nu niet verkorten!

De regel:

```
        if(PINB & 0b00101000) {
```

geeft namelijk een heel ander resultaat. De expressie `PINB & 0b00101000` levert namelijk altijd `true` op als de knop (S2) niet wordt ingedrukt.

De regel:

```
        if(!(PINB & 0b00101000)) {
```

geeft ook een heel ander resultaat. De expressie `!(PINB & 0b00101000)` levert namelijk alleen `true` op als de knop (S2) is ingedrukt en als P1.5 *niet* verbonden is met V_{cc}.

In het onderstaande voorbeeld wordt het rode ledje aangezet zolang P1.3 laag is of P1.5 hoog is (dus als S2 ingedrukt is of als ingangspin P1.5 met een draadje met de V_{cc} is verbonden). Als dit niet het geval is (S2 is niet ingedrukt en P1.5 is niet verbonden met V_{cc}) dan wordt het rode ledje uitgezet. Om er voor te zorgen dat pin P1.5 laag is als er geen verbinding is met V_{cc}, wordt intern in de microcontroller een pull-down weerstand aangezet die verbonden is met pin P1.5.

```
#include <msp430.h>
```

```
// Ingangspin P1.5 kan hoog gemaakt worden door deze pin via een ↩
 ↪ draadje met de Vcc te verbinden.

int main(void)
{
    WDTCTL = WDTPW | WDTHOLD; // Stop watchdog timer.
    P1DIR = 0b01000001; // Maak P1.0 en P1.6 uitgangen en alle ↩
     ↪ andere pinnen van P1 ingangen.
    P1REN |= 0b00101000; // Zet pull-up weerstand bij P1.3 aan en
    P1OUT |= 0b00001000; // zet pull-down weerstand bij P1.5 aan.
    while (1)
    {
        if ((P1IN & 0b00101000) != 0b00001000)
        {
            P1OUT |= 0b00000001;
        }
        else
        {
            P1OUT &= ~0b00000001;
        }
    }
    return 0;
```

De onderstaande waarheidstabel kan helpen bij het doorgronden van de werking van het bovenstaande programma. Om ruimte te besparen zijn de constanten in hexadecimale notatie weergegeven: `0b00101000` = `0x28` en `0xb00001000` = `0x08`.

S2	P1.3	P1.5	P1IN&0x28	(PINB&0x28)!=0x08	P1.0	led
niet ingedrukt	1	0	0x08	false	0	uit
niet ingedrukt	1	1	0x28	true	1	aan
wel ingedrukt	0	0	0x00	true	1	aan
wel ingedrukt	0	1	0x20	true	1	aan

Let op! De regel:

```
if((PINB & 0b00101000) != 0b00001000) {
```

kun je nu niet verkorten!

9.3 Schuiven met bitjes

In C zijn ook operatoren gedefinieerd waarmee je een bitpatroon kunt schuiven. Deze operatoren worden shift-operators genoemd en het zijn binaire operatoren (er zijn 2 operanden). De operator << schuift naar links en de operator >> naar rechts. Bij het schuiven naar links worden er altijd nullen ingeschoven. Wat er gebeurt bij schuiven naar rechts wordt

in paragraaf 9.3.2 uitgelegd. Aan de linkerkant van de shift-operator staat het patroon dat verschoven moet worden en aan de rechterkant staat het aantal plaatsen wat geschoven moet worden, de zogenaamde shift-count.

In het onderstaande voorbeeld wordt in het bitpatroon van register P1IN bit 3 geïsoleerd dit bitbatroon wordt vervolgens drie plaatsen naar links geschoven en tot slot ge-ord met P1OUT. Op deze wijze wordt de waarde van de knop S2 (P1.3) gekopieerd naar de groene led (P1.6). De led brand als de knop *niet* is ingedrukt (P1.3 is hoog) en dooft als de knop wordt ingedrukt (P1.3 is laag).

```
#include <msp430.h>

int main(void)
{
    WDTCTL = WDTPW | WDTHOLD; // Stop watchdog timer.
    P1DIR = 0b01000001; // Maak P1.0 en P1.6 uitgangen en alle ↩
     ↪ andere pinnen van P1 ingangen.
    P1REN |= 0b00001000; // Zet pull-up weerstand bij P1.3 aan.
    P1OUT |= 0b00001000;
    while (1)
    {
        uint8_t bitpatroon = P1IN & 0b00001000;
        bitpatroon = bitpatroon << 3;
        P1OUT |= bitpatroon;
    }
    return 0;
}
```

De regel:

```
        uint8_t bitpatroon = P1IN & 0b00001000;
```

leest de waarde van P1IN en maakt alle bits `0` behalve bit 3 en zet deze waarde in de variabele `bitpatroon`. Dus als S2 niet is ingedrukt, heeft de variabel `bitpatroon` de waarde `0b00001000` en als S2 is ingedrukt, heeft `bitpatroon` de waarde `0b00000000`. De regel:

```
        bitpatroon = bitpatroon << 3;
```

schuift deze waarde drie plaatsen naar links. Dus als S2 niet was ingedrukt, dan heeft de variabel `bitpatroon` na het schuiven de waarde `0b01000000` en als S2 was ingedrukt, heeft `bitpatroon` na het schuiven de waarde `0b00000000`. De regel:

```
        P1OUT |= bitpatroon;
```

Dus als S2 niet was ingedrukt, zal P1.6 hoog worden en zal de groene led branden maar als S2 was ingedrukt, zal P1.6 laag worden en zal de groene led niet branden.

De regel:

```
        bitpatroon = bitpatroon << 3;
```

kun je ook verkorten tot:

```
bitpatroon <<= 3;
```

Het is ook niet nodig om de variabele `bitpatroon` te gebruiken. De regels:

```
uint8_t bitpatroon = P1IN & 0b00001000;
bitpatroon = bitpatroon << 3;
P1OUT |= bitpatroon;
```

kun je ook verkorten tot:

```
P1OUT |= (P1IN & 0b00001000) << 3;
```

De haakjes zijn noodzakelijk omdat de <<-operator een hogere prioriteit heeft dan de &-operator.

9.3.1 Schuiven naar links is hetzelfde als vermenigvuldigen met een macht van 2

Bij het schuiven naar *links* worden er altijd nullen ingeschoven. Schuiven van x plaatsen naar links komt overeen met vermenigvuldigen met 2^x. De schuifoperatie:

```
a = b << 2;
```

geeft exact hetzelfde resultaat als de vermenigvuldiging:

```
a = b * 4;
```

9.3.2 Schuiven naar rechts is bijna hetzelfde als delen door een macht van 2

Bij schuiven naar *rechts* is het wat ingewikkelder. Als het patroon *unsigned* is, worden er ook nullen ingeschoven. Als de unsigned 8 bits waarde `0b10111101` twee plaatsen naar rechts wordt geschoven, dan levert dat de waarde `0b00101111` op. Bij unsigned getallen komt x plaatsen schuiven naar rechts overeen met delen door 2^x. Als gegeven is dat de variabelen `a` en `b` gedefinieerd zijn van het type `uint8_t`, dan levert de schuifoperatie:

```
a = b >> 2;
```

exact hetzelfde resultaat als de deling:

```
a = b / 4;
```

Als het patroon *signed* is, dan wordt bij het inschuiven de tekenbit (de meest significante bit) gekopieerd. Als de 8 bits signed waarde `0b10111101` twee plaatsen naar rechts wordt geschoven, dan levert dat de waarde `0b11101111` op. Bij negatieve signed getallen komt x plaatsen schuiven naar rechts ook overeen met delen door 2^x maar is het resultaat vreemd genoeg niet hetzelfde als het resultaat van de `/`-operator. Als gegeven is dat de variabelen `a` en `b` gedefinieerd zijn van het type `int8_t`, dan levert de schuifoperatie:

```
a = b >> 2;
```

niet hetzelfde resultaat als de deling:

```
a = b / 4;
```

Als b de waarde `0b10111101` heeft, dan krijgt a na de schuifoperator de waarde `0b11101111`. Als b de waarde `0b10111101` heeft, dan krijgt a na de deling de waarde `0b11110000`.

Bij signed schuiven naar rechts is de rest (wat er wordt uitgeschoven) altijd positief bij signed delen is de rest negatief als het deeltal negatief is.

Bij delen door vier met behulp van de >>-operator: `0b10111101 >> 2` = `0b11101111` rest `0b01` (rest is wat er wordt uitgeschoven). In het signed two's complement talstelsel is dit dus decimaal: -67 gedeeld door 4 = -17 rest 1. Deze vorm van delen wordt 'Euclidean division' genoemd[33]. Bij delen met behulp van de /-operator: `0b10111101 / 4` = `0b11110000` rest `0b11111101` (de rest kun je bepalen met de %-operator). In het signed two's complement talstelsel is dit dus decimaal: -67 gedeeld door 4 = -16 rest -3. Deze manier van delen wordt 'truncated division' genoemd[34].

Beide antwoorden zijn wiskundig correct. Want $-17 \times 4 + 1 = -67$ en $-16 \times 4 + -3 = -67$[35].

9.3.3 Maskers en patronen samenstellen door een 1 naar links te schuiven

Bij het manipuleren en testen van afzonderlijke bits wordt vaak gebruik gemaakt van bitpatronen of maskers waarin op slecht één positie een 1 voorkomt. Om bijvoorbeeld pin P1.6 één te maken moet je het P1OUT-register or-en met het binaire patroon: 01000000.

```
#include <msp430.h>

int main(void)
{
    WDTCTL = WDTPW | WDTHOLD;
    P1DIR = 0b01000001;
    P1OUT |= 0b01000000;
    while (1);
    return 0;
}
```

Je kunt het benodigde patroon ook uit laten rekenen door de compiler door de constante 1 zes plaatsen naar links te schuiven:

```
#include <msp430.h>

int main(void)
{
    WDTCTL = WDTPW | WDTHOLD;
```

[33] Zie http://en.wikipedia.org/wiki/Euclidean_division.

[34] Zie http://en.wikipedia.org/wiki/Modulo_operation.

[35] Zie eventueel http://en.wikipedia.org/wiki/Remainder.

```
    P1DIR = 0b01000001;
    P1OUT |= 1<<6;
    while (1);
    return 0;
}
```

De expressie 1<<3 wordt door de compiler uitgerekend en levert de waarde 0b01000000 op zodat beide programma's exact dezelfde machinecode opleveren. De meeste mensen vinden het tweede programma beter leesbaar omdat je meteen ziet dat bit 6 van poort P1 geset wordt.

Als in het benodigde patroon meer dan 1 bit geset moeten worden dan kan dit door verschillende schuifexpressies met een bitwise-or met elkaar te combineren. In het onderstaande programma worden de bits 6 en 0 van het P1DIR-register één gemaakt met behulp van de <<- en |-operator:

De regel:

```
    P1DIR = 1<<6 | 1<<0;
```

kan natuurlijk ook vervangen worden door:

```
    P1DIR = 0x41;
```

Dit is misschien minder duidelijk maar wel minder typewerk.

9.4 Voorgedefinieerde maskers in msp430.h

In de headerfile msp430.h zijn met **#define** constanten gedefinieerd voor het bitmasker benodigd voor elk bit: BIT0 tot en met BIT7. Het in paragraaf 9.1.1 gegeven programma om de groene led aan te zetten kan dus ook als volgt geïmplementeerd worden:

```
#include <msp430.h>

int main(void)
{
    WDTCTL = WDTPW | WDTHOLD;
    P1DIR = BIT6 | BIT0;
    P1OUT |= BIT6;
    while (1);
    return 0;
}
```

In de headerfile msp430.h zijn alle namen van de verschillende bitjes in de I/O-registers van de MSP430 met **#define** gekoppeld aan hun bitmasker. Op deze manier kun je dus bitjes manipuleren en testen zonder dat je het bitnummer hoeft te weten (je moet dan natuurlijk wel de naam van het bitje weten).

Als de analoog naar digitaal converter ADC10 in de MSP430G2553 in de zogenoemde single channel single-conversion mode[36] gebruikt wordt, dan kan (volgens de datasheet[37]) een conversie gestart worden door bit 0 genaamd `ADC10SC` in register `ADC10CTL0` hoog te maken. Dit kun je als volgt programmeren:

```
ADC10CTL0 |= 0b0000000000000001;
```

In de headerfile `msp430.h` is de constante `ADC10SC` gedefinieerd zodat je dit ook als volgt kunt programmeren:

```
ADC10CTL0 |= ADC10SC;
```

Dit is veel beter leesbaar.

Als de conversie klaar is zal de ADC10 bit 2 genaamd `ADC10IFG` in register `ADC10CTL0` hoog maken. Als je *geen* gebruik maakt van de ADC10-interrupt dan kun je een programma laten wachten tot de conversie klaar is met de volgende **`while`**-loop:

```
while ((ADC10CTL0 & 0b0000000000000100) == 0);
```

In de headerfile `msp430.h` is de constante `ADC10IFG` gedefinieerd zodat je dit ook als volgt kunt programmeren:

```
while ((ADC10CTL0 & ADC10IFG) == 0);
```

Dit is veel beter leesbaar.

In de file `msp430.h` wordt gekeken naar het ingestelde type MSP430 microcontroller om te bepalen welke bitnamen aan welke bitmaskers moeten worden gekoppeld. Het is dus van groot belang om bij de projectopties het juiste MSP430 type, in ons geval de MSP430G2553, te selecteren.

[36] Hoe je deze mode kunt instellen is voor dit voorbeeld niet relevant.

[37] Zie eventueel `http://www.ti.com/lit/ug/slau144j/slau144j.pdf#page=553`.

10

Meer leren

Wordt nog aan gewerkt! Lijstje met onderwerpen die niet zijn behandeld:

- bitfields
- `union`
- `enum`
- Dynamisch geheugen `malloc` en `free`
- Layout van een struct en padding
- C11 threads

Bibliografie

[1] Michael Barr. *Embedded C Coding Standard*. BARR group, 2018. ISBN: 978-1-72112-798-6. URL: https://barrgroup.com/sites/default/files/barr_c_coding_standard_2018.pdf (geciteerd op pp. 48, 62).

[2] "IEEE Standard for Floating-Point Arithmetic". In: *IEEE Std 754-2008* (2008), p. 1–70 (geciteerd op p. 4).

[3] ISO. *ISO/IEC 9899:2011 Programming languages — C*. Geneva, Switzerland: International Organization for Standardization, 2011. URL: https://www.iso.org/standard/57853.html (geciteerd op p. 1).