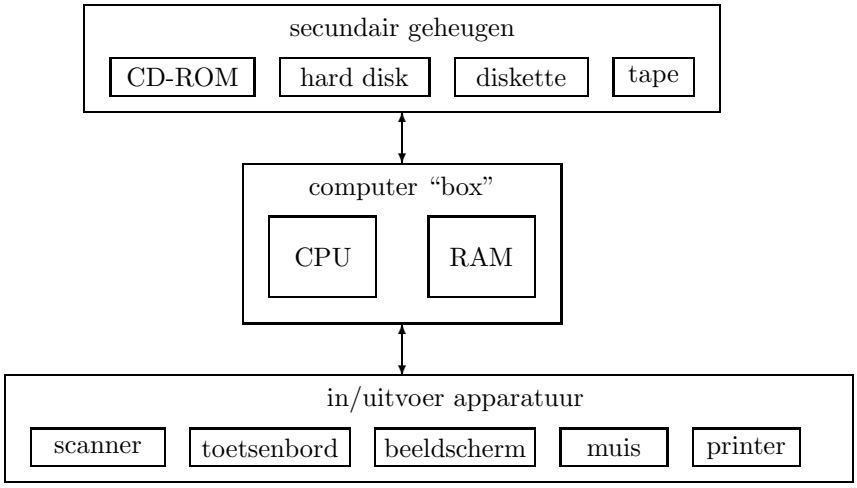
Inhoudsopgave

1 Algoritmes en computerprogramma's

1.1 Basiselementen van een computer

Een computer is opgebouwd uit een heleboel verschillende onderdelen. Een schematisch diagram wordt getoond in de volgende figuur.



Er zijn drie belangrijke elementen:

het secundair geheugen voor grote hoeveelheden gegevens op te slaan;

de computer box samengesteld uit de Central Processing Unit (CPU) en werkgeheugen of Random Access Memory (RAM);

de in/uitvoer (I/O) apparatuur voor de interactie tussen de gebruiker en de computer.

Deze fysische onderdelen van de machine noemt men de *hardware*. *Software* zijn verzamelingen instructies die er voor zorgen dat de hardware operaties en berekeningen uitvoert. Software wordt geschreven in een *programmeertaal*. Zo'n programmeertaal moet een aantal elementen bevatten

- om de processor berekeningen te laten uitvoeren;
- om informatie efficiënt beschikbaar te hebben in het werkgeheugen;
- om de communicatie te verzekeren met de gebruiker via toetsenbord, muis, beeldscherm, ...;
- om grote hoeveelheden gegevens te kunnen *lezen* van secundaire geheugens en om resultaten te kunnen *schrijven* naar die secundaire geheugens voor latere raadpleging.

1.2 Algoritmes

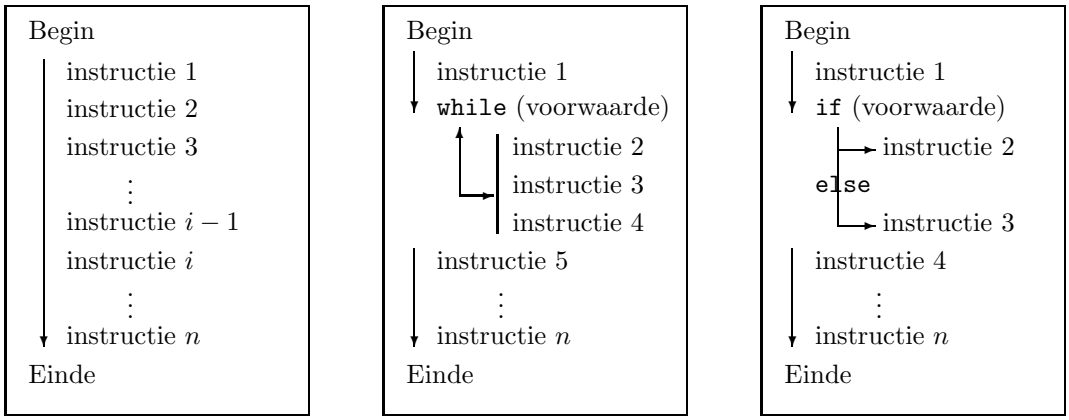
Computers worden gebruikt om problemen op te lossen. Om tot een oplossing te komen, moeten berekeningen uitgevoerd worden:

- de som van een aantal getallen;
- een tabel met x-waarden en bijhorende functiewaarden;
- een nulpunt van een functie;
- de bepaalde integraal van een functie tussen twee grenzen;
- de oplossing van een stelsel lineaire vergelijkingen;
- ...

Sommige van deze berekeningen zijn heel eenvoudig, voor andere bestaat de berekening uit een heleboel stappen. Een berekening kan beschreven worden in de vorm van een *algoritme*. Een algoritme is een *duidelijke en preciese procedure (een opeenvolging van goed-gedefinieerde stappen) om een gegeven probleem op te lossen*. Een voorbeeld is het *algoritme van Gauss* voor het oplossen van een stelsel lineaire vergelijkingen.
Voor heel veel problemen bestaan (nog) geen algoritmes die het probleem exact oplossen. In die gevallen wordt dikwijls een beroep gedaan op een procedure die een benaderende oplossing genereert, bijvoorbeeld de *trapeziumregel* voor het berekenen van een bepaalde integraal.
Niettegenstaande een algoritme een precieze beschrijving van de oplossingsprocdure is, is dit meestal niet door een computer interpreteerbaar. Het algoritme moet omgezet worden in een *programma*. Een programma is geschreven in een *programmeertaal*.
Een computerprogramma is een lijst van instructies welke door de processor interpreteerbaar en uitvoerbaar zijn. Een programma kan ook gezien worden als een verzameling van data en routines om deze data te verwerken. Deze routines (procedures of functies) worden neergeschreven met behulp van instructies. De processor voert deze instructies *sequentiëel* (één na één) uit. Een computerprogramma moet dus (in het algemeen) zoals een tekst gelezen worden: lijn per lijn van boven naar onder. Elke lijn bevat een aantal instructies voor de processor. Een programmeertaal bevat een aantal speciale *keywords* om aan te geven dat een aantal instructies in *herhaling* moeten uitgevoerd worden of dat een beslissing moet gemaakt worden omtrent welke set instructies moeten uitgevoerd worden.
Naast een sequentiële uitvoering van een programma is er dus de mogelijkheid voor een

lus : de mogelijkheid om een verzameling instructies een aantal maal na elkaar uit te voeren;

keuze : de mogelijkheid om te beslissen welke verzameling instructies uitgevoerd worden.



sequentie herhaling keuze

Deze drie elementen, *sequentie*, *herhaling* (iteratie) en *keuze*, bepalen de *controle stroom* doorheen het programma.

1.3 Som van twee getallen

1.3.1 Een eerste poging

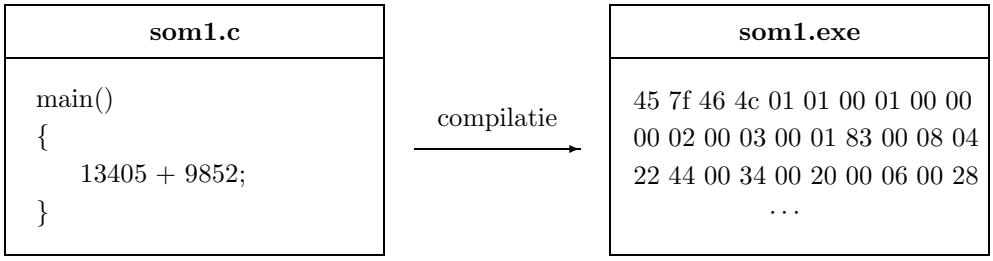
De som van twee getallen maken is vrij eenvoudig:

```
13405 + 9852
```

In de CPU is namelijk een elementaire machine-instructie aanwezig om een optelling uit te voeren. Om tot een programma te komen, moet deze *rekenkundige expressie* opgenomen worden in een *functie* die door de processor kan uitgevoerd worden:

```
/*
 * som1.c : een eerste poging
 */
main()
{
     13405 + 9852;
}
```

De eerste drie lijnen geven uitleg over het programma dat volgt. Dit is voor de (menselijke) lezer van het programma bestemd en wordt *commentaar* genoemd. Alles wat zich tussen `/*` en `*/` bevindt, is commentaar en heeft geen betekenis voor de computer.
`main` is de naam van de functie. De twee haakjes (`()`) geven aan dat het vorige woord de naam van een functie is. Dit geheel wordt de *hoofding* van een functie genoemd. De `main()` functie moet in elk C-programma voorzien worden. Bij het uitvoeren van het programma, zal het systeem steeds met deze functie beginnen.
Wat in de functie moet uitgevoerd worden, wordt in de vorm van *statements* geschreven tussen de open accolade (`{`) en de sluit accolade (`}`). Dit geheel noemt men een *blok*.
Een *statement* is een *expressie* gevolgd door een punt-komma (`;`). Het programma bevat één expressie: het optellen van de twee getallen.
Dit geheel kan in een bestand ingetikt worden, bijvoorbeeld het bestand met naam `som1.c`. Het programma kan dan *gecompileerd* worden. De *compiler* zet het *bronbestand* (C-programma) om in een *doel-bestand* (machine-code) voor een specifieke processor. Het doelbestand kan dan uitgevoerd worden op de computer. Het doelbestand wordt daarom ook dikwijls *executable* genoemd.



bron-bestand doel-bestand

De vorm waarin het programma ingetikt wordt, is voor de compiler van geen belang. Een syntactisch even correcte vorm is `main() { 13405 + 9852; }` . Maar voor de programmeur (schrijver en/of lezer) is dit niet erg duidelijk. Daarom is het wenselijk het programma in een vorm in te tikken zodat de blok-structuur heel duidelijk weergegeven wordt.

1.3.2 Een zinvolle versie

Het programma is compleet nutteloos. De computer heeft wel de som van twee getallen berekend, maar het resultaat zou moeten zichtbaar gemaakt worden. Daarnaast kan het programma maar de som van twee specifieke getallen berekenen. Wanneer de som van twee andere getallen moet berekend worden, moet het programma aangepast worden.
Om het programma meer zinvol te maken, zijn variabelen nodig en communicatie met de gebruiker via toetsenbord en beeldscherm.

```
1   /*
     * som2.c : een meer zinvolle poging
3    */
    #include <stdio.h>
5   void main(void)
    {
7       int a;      /* eerste getal */
```

```
    int b;          /* tweede getal */
 9  int som;        /* som */

11  scanf("%d%d%*c", &a, &b);                    /* invoer */
    som = a + b;                                 /* rekengedeelte */
13  printf("%d + %d -> %d\n", a, b, som);        /* uitvoer */
}
```

De functie `main` wordt door het systeem opgeroepen. Normaal heeft een functie één of meer argumenten en geeft de functie een resultaat. Om aan te geven dat `main` geen argumenten heeft en ook geen resultaatswaarde berekent, wordt tweemaal `void` gespecificeerd.
De eerste drie statements zijn *declaraties* van *variabelen*. Een *variabele* is een symbolische naam van een geheugenplaats in het werkgeheugen. Deze naam wordt gebruikt om de geheugenplaats aan te duiden (*lvalue*). Men had evengoed een nummer kunnen gebruiken: geheugenplaats nummer 1, geheugenplaats nummer 2, ... Maar in een programma met veel variabelen, zou dit leiden tot een vrij moeilijk leesbare code. Men geeft dus een symbolische naam (bijv. `a`, `b`, `som`) aan zo'n geheugenplaats. De waarde van de variabele (*rvalue*) is de inhoud van zo'n geheugenplaats, dus een getalwaarde waarmee in het programma gerekend wordt.
Een declaratie bevat naast de naam van de variabele ook een *type*. Het type geeft aan de compiler aan hoe groot de geheugenplaats moet zijn, of hoeveel bytes voor de desbetreffende variabele moet voorzien worden. In dit voorbeeld wordt het type `int` gebruikt. Dit type geeft aan dat de variabele waarden kan hebben die behoren tot de verzameling van de gehele getallen (integers). Afhankelijk van de processor wordt door de compiler twee of vier bytes voorzien voor een variabele van type `int`.
Het type van een variabele bepaalt ook welke operaties (bewerkingen) toegelaten zijn. Op variabelen van het type `int` kunnen de verschillende rekenkundige bewerkingen uitgevoerd worden: `+`, `-`, `*`, `/` en `%`. Voor de deling zijn twee operaties voorzien, namelijk voor de berekening van het quotiënt en de rest:

```
34 / 6  -->  5
34 % 6  -->  4
```

Het *statement* waarin de berekening gebeurt (de som van twee getallen) werkt nu met variabelen:

- neem de inhoud van variabele `a` (rvalue) ;
- tel hierbij de inhoud van variabele `b` (rvalue) op;
- plaats het resultaat in de variabele `som` (lvalue).

Het `=` teken is het symbool voor de operatie *toekenning*. Langs de rechterkant van dit symbool worden de rvalues van variabelen (de numerieke waarden) gebruikt; langs de linkerkant de lvalue van een variabele (aanduiding van een geheugenplaats).
De communicatie met de gebruiker is een vrij moeilijk probleem. Maar hiervoor zijn standaard functies voorzien, `scanf` voor invoer van het toetsenbord en `printf` voor uitvoer naar het beeldscherm. De declaratie van deze functies kan teruggevonden worden in het *header*bestand `<stdio.h>`. De definitie van de functies wordt bewaard in de *standaard C-bibliotheek*. De compiler (linker) zal deze bibliotheek raadplegen en de definities van de gebruikte (opgeroepen) functies opnemen in de *executable*.
Tussen de haakjes na de functienaam staan een aantal *argumenten*. Het eerste argument (de formaatstring) geeft aan welke conversies moeten gebeuren. De volgende argumenten geven de variabelen aan die bij de in/uitvoer betrokken zijn.
Het formaat `%d + %d -> %d\n`
geeft aan dat `printf` drie getalwaarden moet weergeven op het beeldscherm. Deze weergave moet de decimale voorstelling zijn. Tussen de drie getallen worden een aantal tekens weergegeven en na de drie getallen wordt naar een nieuwe lijn (`\n`) gegaan. Het tweede argument geeft de naam van de variabele die als eerste waarde zal weergegeven worden. Door de naam van de variabele

als argument te gebruiken, wordt de inhoud (rvalue) van de geheugenplaats doorgegeven aan de functie. In de functie `printf` wordt de waarde (interne voorstelling) geconverteerd naar een decimale vorm (externe voorstelling) en dan weergegeven op het scherm. Analoog voor het derde en het vierde argument.
De `printf` lijn kan vervangen worden door een aantal `printf`'s:

```
printf("%d", a);
printf(" + %d", b);
printf(" -> %d", som);
printf("\n");
```

Het formaat `%d%d%*c`
geeft aan dat `scanf` twee decimale getallen van het toetsenbord moet lezen, deze getallen moet omvormen naar een interne voorstelling en een extra teken.
Het tweede argument is niet de naam van een variabele, maar een expressie `&a`. Wanneer gewoon de naam zou gebruikt worden, zou aan `scanf` de inhoud (rvalue) van de variabele doorgegeven worden. Deze inhoud is echter nog niet gekend. Het is juist de bedoeling dat `scanf` voor een inhoud (een waarde) gaat zorgen door een decimaal getal van het toetsenbord in te lezen en dat te converteren naar een inwendige voorstelling. De functie `scanf` moet dus een aanduiding krijgen waar de waarde in het werkgeheugen moet gestockeerd worden. De expressie `&a` berekent deze aanduiding (of het *adres* van de geheugenplaats) en dit adres wordt doorgegeven aan `scanf`.
Analoog geeft het derde argument `&b` het adres van de variabele `b` door aan de functie `scanf`.
Het laatste deel in de formaatstring geeft aan dat nog een extra teken moet ingelezen worden (`%c`) maar dat dit teken niet moet toegekend worden aan een variabele (`*`).

1.4 Omzetting mijl naar kilometer

Het algoritme om een gegeven aantal mijl om te zetten in een aantal kilometers is vrij eenvoudig: *vermenigvuldig het aantal mijl met de factor 1.609.*

Opgave. Schrijf een programma dat 10 mijl omzet naar km.

```
/*
 *    mijl1km.c : omzetting: mijl -> km
 */
#include <stdio.h>

void main(void)
{
    double km, mijl;

    mijl = 10.0;
    km = mijl * 1.609;
    printf("%f mijl is %f kilometer\n", mijl, km);
}
```

Opgave. Schrijf een programma dat een afstand in mijl inleest en het overeenkomstig aantal kilometer afdrukt.

```
/*
 *    mijl2km.c : omzetting: mijl -> km
 */
#include <stdio.h>

void main(void)
```

```
7   {
        double km, mijl;
9
        printf("Geef het aantal mijl: ");
11      scanf("%lf%*c", &mijl);
        km = mijl * 1.609;
13      printf("%f mijl is %f kilometer\n", mijl, km);
    }
```

Merk op dat voor het inlezen van een variabele van type `double` een formaat `%lf` gespecificeerd wordt. Dit is nodig omdat het tweede argument een verwijzing is naar een *dubbele* (of **l***ange*) plaats in het werkgeheugen. Deze `l` specificatie mag ook bij het formaat in de `printf` gebruikt worden, maar daar hoeft dat niet.

1.5 Berekening van de grootste gemene deler

Opgave. Lees twee gehele getallen en bereken de grootste gemene deler (ggd) van deze twee getallen.
In dit geval (ggd), kan beroep gedaan worden op het *algoritme van Euclides*:

1. Noem de twee getallen x en y met $x < y$.
2. Bereken de rest r bij deling van y door x.
3. Vervang y door x, en x door r.
4. Herhaal stap 2 en 3 totdat r nul is.
5. De grootste gemene deler is gelijk aan y.

Voorbeeld. Gevraagd: ggd van 16966 en 34017.
Oplossing:

- de rest van de deling 34017/16966 is gelijk aan 85
- de rest van de deling 16966/85 is gelijk aan 51
- de rest van de deling 85/51 is gelijk aan 34
- de rest van de deling 51/34 is gelijk aan 17
- de rest van de deling 34/17 is gelijk aan 0
- de grootste gemene deler van 16966 en 34017 is gelijk aan 17.

Om dit algoritme door een computer te laten uitvoeren, moet dit omgezet worden naar een computerprogramma, bijv. in C.

```
/*
2    *    ggd.c : berekening grootste gemene deler
     */
4   #include <stdio.h>
    void main(void)
6   {
        int a;        /* eerste getal */
8       int b;        /* tweede getal */
        int x, y, r;
10
        scanf("%d%d%*c", &a, &b);
```

```
12      /*   bepaal x en y zodat x < y */
        if ( a > b )
14      {
            x = b;
16          y = a;
        }
18      else
        {
20          x = a;
            y = b;
22      }
        while ( x != 0 )            /* herhaal tot rest gelijk aan nul */
24      {
            r = y % x;              /* bereken rest */
26          y = x;                  /* y krijgt de waarde van x */
            x = r;                  /* x krijgt de waarde van r */
28      }
        printf("ggd(%d, %d) = %d\n", a, b, y);
30  }
```

Naast statements met toekenningen en in/uitvoer functies, bevat dit programma twee nieuwe statements: een `if` en een `while` statement. Dit zijn voorbeelden van *controle*-statements om de sequentiële uitvoering van het programma om te kunnen buigen.
Het `if` statement (*keuze*) wordt gebruikt om de variabelen `x` en `y` de getalwaarden van de twee ingelezen getallen te geven zodat $x < y$. Het `while` statement (*herhaling*) is een mogelijk C-vertaling van de "Herhaal" stap in het algoritme van Euclides. Deze twee statements zijn *compound* of *blok* statements omdat onderdelen ervan zelf statements zijn. Omwille van de leesbaarheid wordt zo'n statement met behulp van indentatie als een blok weergegeven in het bron-bestand.
In de beschrijving van het algoritme gebeurt de test of de rest gelijk is aan 0 op het einde. Deze test gebeurt in het programma in het begin. In het programma is deze aanpassing nodig voor een goede afloop van het programma wanneer één van de ingelezen getallen gelijk is aan nul (ggd(0,1) = 1 !).

1.6 Begrippen

- Een algoritme.
- Een programma: compilatie en uitvoering.
- De `main` functie.
- Declaraties en variabelen; lvalue en rvalue van een variabele.
- Statements: rekenkundige expressies, toekenningsexpressie, invoer en uitvoer functies.

Een overzicht van een aantal *C interpreter/compiler*s kan gevonden worden op

`http://www.thefreecountry.com/compilers/cpp.shtml`

Pelles C IDE, die gebruikt wordt tijdens de praktijkzittingen:

`http://www.smorgasbordet.com/pellesc/`

2 Numerieke data en expressies

2.1 Constanten

In het werkgeheugen kunnen volgende soorten data voorgesteld worden. Deze data moet door de CPU kunnen verwerkt worden.

Integers : een exacte voorstelling van gehele getallen over een beperkt bereik (eventuele overflow/underflow problemen).

Floating point numbers : een niet-exacte voorstelling van reële getallen omdat een eindige binaire string niet om het even welk reëel getal kan voorstellen; het bereik is veel groter dan bij integers.

Characters en text strings : zie een volgend hoofdstuk.

Voorbeelden:

gehele constanten		reële constanten
123	(decimaal)	12.3
-123		0.5
0123	(octaal)	.5
0x123	(hexidecimaal)	5.
		5e4

2.2 Variabelen

Een variabele is een symbolische naam voor een geheugenplaats. Een variabele geeft normaal een plaats (*lvalue*) aan en heeft een waarde (*rvalue*).
De naam van de variabele wordt ook aangeduid met de term *identifier*. Het is een rij letters en/of cijfers; tevens mag de underscore (`_`) in de naam voorkomen. Een naam mag niet beginnen met een cijfer. Hoofdletters en kleine letters worden als verschillend beschouwd. Er is geen beperking op de lengte, maar in sommige implementaties zijn bijvoorbeeld alleen de eerste acht tekens significant; de overige worden dan eenvoudig genegeerd (`dit_is_een_variabele_voor_de_som`).
Een naam wordt door de programmeur gekozen. In de mate van het mogelijke moet de naam betekenisvol zijn: de naam moet iets vertellen over de functie van de variabele in het programma.
Elke variabele moet gedeclareerd worden als zijnde van een bepaald *type*. Dit type bepaalt hoeveel bytes in het werkgeheugen nodig zijn voor zo'n variabele. Het geeft ook aan welke operatoren toegelaten zijn op de variabele.

int	2 of 4 bytes	gehele getallen
double	8 bytes	floating point getallen

In volgende hoofdstukken zullen nog andere types aan bod komen.

```
/*
 *  varid.c : een voorbeeld van een int en een double
 */
#include <stdio.h>

void main(void)
{
    int         i, j;
    double      ff;

    i = 2;
    j = 5/3;
    ff = 5.0/3;
```

```
14          printf("i  = %d\n", i);
            printf("j  = %d\n", j);
16          printf("ff = %20.17f\n", ff);
        }
```

De output van dit programma is:

```
i  = 2
j  = 1
ff =  1.66666666666666670
```

Merk op dat de deling van twee gehele getallen (variabele `j`) het geheeltallig quotiënt oplevert. Het formaat `%20.17f` geeft aan dat er 20 plaatsen moeten voorzien worden op het scherm (veldbreedte) om de waarde van `ff` weer te geven en dat er 17 cijfers na de komma moeten getoond worden.

2.3 Expressies

De algemene vorm van een eenvoudige expressie is:

```
operand_1   operator   operand_2
```

zoals bijvoorbeeld `a + b`.

2.3.1 Rekenkundige expressies

Rekenkundige operatoren:

`+`	optelling	
`-`	aftrekking	
`*`	vermenigvuldiging	
`/`	deling	het quotiënt bij geheeltallige deling
`%`	modulus	de rest bij geheeltallige deling

De operatoren `*`, `/` en `%` hebben een hogere prioriteit dan `+` en `-`. Naast deze *binaire* operatoren kan ook het *unaire* minteken (`-`) vermeld worden, met een hogere prioriteit dan `*`, `/` en `%`.

```
i = -6;             j = -i;
```

(In sommige C-versies is ook een unaire `+` operator gekend.) Er is echter geen operator voor machtsverheffing.

Wanneer beide operands van hetzelfde type zijn, is het resultaat van de expressie ook van dat type:

```
1 / 2       heeft als resultaat    0
1.0 / 2.0   heeft als resultaat    0.5
5 % 3       heeft als resultaat    2
```

Wanneer één operand `integer` is en de ander `double`, wordt de `integer` operand omgezet naar een `double` en dan wordt de operator uitgevoerd. Het resultaat van de expressie is `double`:

```
1 / 2.0   heeft als resultaat    0.5
1.0 / 2   heeft als resultaat    0.5
```

In zeer vele toepassingen worden tellers gebruikt die frequent met 1 verhoogd of met 1 verlaagd moeten worden. Hiervoor zijn speciale *unaire* operatoren voorzien: de *increment* operator `++` en de *decrement* operator `--`. Deze operatoren bestaan in *pre* en *post* versie:

```
int h = 4;  int k = 4;  int m = 4;  int n = 4;
int i, j;

i = ++h;  /* h wordt verhoogd voordat de waarde toegekend wordt aan i
             na uitvoering hebben i en h beide dezelfde waarde, nl. 5 */
j = k++;  /* k wordt verhoogd nadat de waarde toegekend is aan j
             na uitvoering heeft j de waarde 4 en  k de waarde 5 */
i = --m;  /* m wordt verlaagd voordat de waarde toegekend wordt aan i */
j = n--;  /* n wordt verlaagd nadat de waarde toegekend is aan j */
```

2.3.2 Logische expressies

Een logische expressie heeft twee mogelijke uitkomsten:

waar of true : alle niet nul-waarden;

onwaar of false : een nul-waarde

In een eenvoudige logische expressie worden de waarden van twee variabelen (of expressies) met elkaar vergeleken, met behulp van *relationele* operatoren. Om eenvoudige logische expressies te combineren, gebruikt men *logische* operatoren:

`<`	kleiner dan
`<=`	kleiner dan of gelijk aan
`>`	groter dan
`>=`	groter dan of gelijk aan
`==`	gelijk aan
`!=`	verschillend van
`!`	logische *niet* (unair)
`&&`	logische *en*
`||`	logische *of*

Voorbeelden:

```
x <= 40.0             /* heeft x een waarde kleiner dan of gelijk aan 40.0 */
2.0 < x && x < 5.2    /* heeft x een waarde gelegen tussen 2.0 en 5.2 */
```

De expressie `2.0 < x < 5.2` is syntactisch correct maar heeft niet de semantische betekenis die waarschijnlijk bedoeld was.
Merk ook het verschil op tussen `=` (toekenning) en `==` (test op gelijkheid).

```
x = 5    /* is een toekenning en altijd waar, want niet 0 */
x == 5   /* is een test en alleen waar als de variabele x
            als inhoud (rvalue) de waarde 5 heeft        */
```

Combinatie van eenvoudige logische expressies:

- `exp1 && exp2` is waar wanneer zowel `exp1` als `exp2` waar is;
- `exp1 || exp2` is waar wanneer één van beide (`exp1` en/of `exp2`) waar is;
- `!exp1` is waar wanneer `exp1` niet waar is.

Dit wordt meestal weergegeven in waarheidstabellen:

	0	1
0	0	0
1	0	1

e1 && e2

	0	1
0	0	1
1	1	1

e1 || e2

0	1
1	0

! e1

Kortsluitingsprincipe: bij een logische operator wordt de tweede operand niet geëvalueerd tenzij dit nodig is. Dus, een operand na `&&` of `||` wordt niet meer berekend als de operand ervoor al uitsluitsel over het resultaat geeft. (De waarde van de tweede operand is van geen belang.)

```
0 && ...              /* heeft de waarde 0 (false)  */
1 || ...              /* heeft de waarde 1 (true)   */
```

Toepassing:

```
n!=0 && t/n>1  /* wanneer n gelijk is aan nul wordt t/n niet berekend */
n==0 || t/n>1  /* wanneer n gelijk is aan nul wordt t/n niet berekend */
```

In tegenstelling tot sommige andere talen, geven deze expressies in C geen "Run time error".

2.3.3 Toekenningsoperatoren

Het resultaat van een expressie wordt meestal toegekend aan een variabele. Hiervoor wordt de *toekenningsoperator* (`=`) gebruikt.

```
x = y + z
```

Dit is een expressie: eerst wordt de waarde van het rechterdeel, de expressie `y+z`, berekend: hiervoor worden de *rvalue*s van de variabelen `y` en `z` gebruikt. De resulterende waarde wordt dan toegewezen aan het linkerdeel, de variabele `x` (hiervoor is de *lvalue* van `x` nodig).
Wanneer achter deze expressie een puntkomma (`;`) geplaatst wordt, heeft men een *statement*. Een opeenvolging van statements geeft een programma. Men kan echter de expressie `x=y+z` ook als rechterdeel gebruiken in de expressie:

```
a =  x = y + z
```

De waarde van `y+z` wordt berekend, toegekend aan `x` (lvalue) en dan wordt de (nieuwe) waarde van `x` (rvalue) toegekend aan de variabele `a` (lvalue). In deze expressie zijn drie operatoren aanwezig. De volgorde waarin deze operatoren uitgevoerd worden, wordt bepaald door de *prioriteit* en de *associativiteit* (zie verder).
Langs de linkerkant van het toekenningsteken moet steeds een variabele met een lvalue staan: er moet een plaats in het werkgeheugen voorzien zijn waar het resultaat gestockeerd kan worden. (Langs links kan dus geen constante, bijvoorbeeld 6, staan want een constante heeft **geen** *lvalue*, wel een rvalue.) Langs de rechterkant van het `=` teken worden de rvalues van de verschillende variabelen gebruikt. Met deze waarden wordt een resultaat berekend.
Ook het resultaat van een logische expressie kan toegekend worden aan een variabele:

```
1  void main(void)
   {
3      int a, b, r;

5      scanf("%d%d%*c", &a, &b );
       r = a <= 0 || b >= 10;
7      printf("r = %d\n", r);
       r = a > 0 && b < 10;
9      printf("r = %d\n", r);
   }
```

Wanneer het resultaat van een operatie op een variabele terug toegewezen wordt aan deze variabele, kan dit korter geschreven worden. Bij deze operatoren wordt een rekenbewerking gecombineerd met een toekenning. Ze worden ook *toekenningsoperatoren* genoemd.

`i += 4`	`i = i + 4`
`z -= x`	`z = z - x`
`k *= j`	`k = k * j`
`d /= 3`	`d = d / 3`
`z %= y`	`z = z % y`

Bijvoorbeeld, bij `i += 4` wordt eerst de rvalue van i berekend. Bij deze waarde wordt 4 bijgeteld en het resultaat wordt toegewezen aan i, via zijn lvalue.

2.3.4 Prioriteit en associativiteit

Wanneer verschillende operatoren in één expressie gebruikt worden, gelden prioriteitsregels om te bepalen welke operator eerst moet uitgevoerd worden. Deze regels kunnen natuurlijk opgeheven worden door gebruik te maken van haakjes (`()`).
Overzicht van de reeds geziene operatoren gerangschikt van hoge naar lage prioriteit:

`!`	`++`	`--`	`-`			rechts
`*`	`/`	`%`				links
`+`	`-`					links
`<`	`<=`	`>`	`>=`			links
`==`	`!=`					links
`&&`						links
`||`						links
`=`	`+=`	`-=`	`*=`	`/=`	`%=`	rechts

Het unaire min-teken (voor negatieve getallen) is terug te vinden op de eerste lijn en heeft dus een hogere prioriteit dan de andere rekenkundige binaire operatoren. Wanneer twee operatoren van dezelfde prioriteit in een expressie voorkomen, dan wordt gekeken naar de *associativiteit*.
De meeste operatoren associëren van links naar rechts: $a+b+c$ wordt geïnterpreteerd als $(a+b)+c$.
De unaire operatoren en de toekenningsoperatoren worden uitgevoerd van rechts naar links: `a=b=c` wordt geïnterpreteerd als `a=(b=c)`.

Oefening.

```
int  i=8,  j=4,  r=0,  s=0;
r = i++ - ++j;
r = ( s = i / j ) == i % j;
r = --i < j++ || i++ > --j;
```

2.4 Enkele voorgedefinieerde functies

Naast constanten en variabelen kunnen in expressies ook *functie-oproepen* voorkomen. In vorige voorbeelden werden reeds twee functies gebruikt, namelijk `scanf` en `printf`. In die voorbeelden werd het resultaat van de functie niet gebruikt. Alleen het neveneffect (iets lezen van toetsenbord of iets schrijven naar beeldscherm) was nuttig.
De meeste functies in C hebben wel een resultaat dat nuttig kan gebruikt worden. De meest voor de hand liggende voorbeelden zijn wiskundige functies.

```
double x, y;
x = 1.0;
y = cos(x);
```

Voor het argument `x` wordt de cosinus-waarde berekend en deze functiewaarde wordt toegekend aan de variabele `y`.
Het algoritme voor de berekening van deze cosinus-waarde is terug te vinden in de *functie-definitie*.
Als gebruiker van deze functie hoeft men niet de details hiervan te weten. Wat wel belangrijk is, is hoe de functie kan gebruikt worden: het type van de argumenten en het type van de terugkeerwaarde. Deze informatie is weergegeven in de *functie-declaratie* of *functie-prototype*.
De functie-declaraties staan telkens in de aangegeven header files. De functie-definities worden door de *linker* in de gepaste bibliotheek (library) gezocht.
Enkele standaard functies (`<stdlib.h>`):

exit(int status);	beëindigen van een programma
int abs(int n);	absolute waarde (integer argument)

Wiskundige standaardfuncties (`<math.h>`):

double cos(double x);	cosinus
double sin(double x);	sinus
double tan(double x);	tangens
double exp(double x);	exponentiële waarde
double log(double x);	natuurlijk logaritme
double pow(double x, double y);	x tot de macht y
double sqrt(double x);	vierkantswortel
double fabs(double x);	absolute waarde (floating point argument)
double acos(double x);	boogcosinus
double asin(double x);	boogsinus
double atan(double x);	boogtangens

Opgave. Schrijf een programma dat een tabel met cosinuswaarden afdrukt voor hoeken van 0 tot 360 met stappen van 30 graden.

```
/*
 *  costab.c : een tabel met cosinuswaarden
 */
#include <stdio.h>
#include <math.h>

void main(void)
{
    double graden;
    double radialen,y;

    printf("   x     |   cos(x)\n");
    printf("-----------------\n");
    graden = 0.0;
    while ( graden <= 360.0 )
    {
        radialen = graden*M_PI/180.0;
        y = cos(radialen);
        printf(" %5.1f   |   %7.4f\n", graden, y);
        graden += 30.0 ;
    }
}
```

Het resultaat van dit programma:

```
   x     |   cos(x)
-----------------
   0.0   |   1.0000
  30.0   |   0.8660
  60.0   |   0.5000
  90.0   |   0.0000
 120.0   |  -0.5000
 150.0   |  -0.8660
 180.0   |  -1.0000
 210.0   |  -0.8660
 240.0   |  -0.5000
 270.0   |  -0.0000
 300.0   |   0.5000
 330.0   |   0.8660
 360.0   |   1.0000
```

Omdat de voorgedefinieerde functie een argument in radialen verwacht, moet de hoek gegeven in graden, omgezet worden ($\pi x/180$). `M_PI` is een constante die gedefinieerd is in `<math.h>` met een waarde (benaderend) gelijk aan π.

2.5 Begrippen

- Constanten, variabelen, identifiers, types.
- Expressies: rekenkundig, vergelijkend, logisch.
- Toekenning: lvalue en rvalue van een variabele.
- Operatoren: unair en binair, prioriteit en associativiteit.
- Voorgedefinieerde functies: functie-prototypes.

3 Controle structuren

3.1 Keuze statements

3.1.1 Het `if` statement

Om te kunnen laten beslissen of een statement al dan niet moet worden uitgevoerd, kan gebruik gemaakt worden van een keuze- of conditioneel statement.

```
if ( expressie )
     statement1;
else
     statement2;
```

Als de `expressie` een waarde verschillend van 0 (d.i. *true*) heeft, dan wordt `statement1` uitgevoerd en `statement2` niet. Wanneer de waarde van de `expressie` gelijk is aan 0 (d.i. *false*), dan wordt `statement2` uitgevoerd en `statement1` wordt overgeslagen.
De expressie waarvan de waarde getest wordt, is meestal een *logische expressie*.
Zowel `statement1` als `statement2` kunnen blok-statements zijn, zodat de algemene vorm ook als volgt weergegeven wordt:

```
if ( expressie )
{
        statement;
        statement;
        ...
}
else
{
        statement;
        statement;
        ...
}
```

In de volgende vorm is het `else` gedeelte weggelaten:

```
if ( expressie )
{
        statement;
        statement;
        ...
}
```

Opgave. Lees een geheel getal en druk af of het even of oneven is.

```
/*
 * even1.c : test of een ingelezen geheel getal even of oneven is
 */
#include <stdio.h>

void main(void)
{
    int getal;

    printf("Geef een geheel getal: ");
    scanf("%d%*c",&getal);
    if (getal%2 == 0)
    {
        printf("%d is even\n",getal);
    }
    else
    {
        printf("%d is oneven\n",getal);
    }
```

```
20  }
```

De expressie in de `if` test hoeft niet een logische expressie te zijn. De waarde van een rekenkundige expressie kan ook gebruikt worden:

```
/*
 * even2.c : test of een ingelezen geheel getal even of oneven is
 */
#include <stdio.h>

void main(void)
{
    int getal;

    printf("Geef een geheel getal: ");
    scanf("%d%*c",&getal);
    if ( getal%2 )
        printf("%d is oneven\n",getal);
    else
        printf("%d is even\n",getal);
}
```

Opgave. Schrijf een C-programma dat twee tijden in seconden inleest en de som afdrukt. Indien de som één minuut of meer is, moet dit afgedrukt worden. Indien de som één uur of meer is, moet dit afgedrukt worden.

```
/*
 * tijd.c :  twee tijden in seconden inlezen en de som afdrukken
 */
#include <stdio.h>

void main(void)
{
    int tijd1,tijd2,som, factor;

    printf("Geef tijd 1: ");
    scanf("%d%*c",&tijd1);
    printf("Geef tijd 2: ");
    scanf("%d%*c",&tijd2);
    som = tijd1+tijd2;
    if (som < 60)
    {
        printf("De som is %d seconden\n",som);
    }
    else
    {
        factor = som/60;
        if ( factor < 60)
            printf("De som is %d minuten en %d seconden\n",
                                    factor, som%60);
        else
            printf("De som is %d uren %d minuten en %d seconden\n",
                                    factor/60, factor%60, som%60);
    }
}
```

Wanneer een `if` statement gebruikt wordt binnen een `if` statement, moet duidelijk gemaakt worden waar de `else` bijhoort.

```
if ( a < b )
    if ( a < c )
        statement1
    else
        statement2
```

```
if ( a < b )
    if ( a < c )
        statement1
else
    statement2
```

Indentatie is hier niet voldoende; de blokstructuur moet aangegeven worden met behulp van de accolades `{` en `}` :

```
if ( a < b )
{
    if ( a < c )
        statement1
    else
        statement2
}
```

```
if ( a < b )
{
    if ( a < c )
        statement1
}
else
    statement2
```

Tikfouten kunnen er voor zorgen dat een *syntactisch* correct programma geproduceerd wordt, dat echter *semantisch* niet doet wat gewenst is.

```
if ( getal = 5 )
{
  /*
   * dit wordt altijd uitgevoerd
   * getal = 5 is een toekenning
   *
en waar, want niet 0
   */
}
```

```
if ( getal == 5 );
{
  /*
   * dit wordt altijd uitgevoerd
   * de puntkomma (;) geeft
   *     het einde van de if aan
   */
}
```

De expressie waarvan de waarde getest wordt, kan natuurlijk bestaan uit een aantal relationele expressies die met behulp van logische operatoren gecombineerd zijn:

```
if ( getal >= 5  && getal <= 10 )
    /* in het interval [5,10] */
```

```
if ( getal < 5
|| getal > 10 )
    /* buiten interval [5,10] */
```

Dikwijls moet de verwerking opgesplitst worden in *meerdere* onderling exclusieve acties. Dit kan gebeuren met de `else if` constructie.

Opgave. Schrijf een programma dat een aantal gehele getallen leest en telt hoeveel er kleiner zijn dan 10, hoeveel er gelegen zijn tussen 10 en 99, hoeveel tussen 100 en 999 en hoeveel tussen 1000 en 9999. Als de rij getallen ingegeven zijn, wordt 11111 ingegeven om het einde van de ingave aan te geven.

```
/*
 *  freq.c : frequenties van ingelezen getallen
 */
#include <stdio.h>

void main(void)
```

```
{
    int     getal;
    int     cijfer = 0;
    int     tiental = 0;
    int     honderdtal = 0;
    int     duizendtal = 0;

    printf("Geef een aantal gehele getallen, eindig met 11111\n");
    scanf("%d%*c", &getal);
    while (getal != 11111)
    {
        if (getal < 10)
            cijfer++;
        else if (getal < 100)
            tiental++;
        else if (getal < 1000)
            honderdtal++;
        else if (getal < 10000)
            duizendtal++;
        else
            ;    /* niet in geinteresseerd */
        scanf("%d%*c", &getal);
    }
    printf("%d %d %d %d \n",
                cijfer, tiental, honderdtal, duizendtal);
}
```

Merk op dat de laatste `else` gevolgd door het leeg statement, evengoed kan weggelaten worden.

3.1.2 Het `switch` statement

Wanneer een keuze moet gemaakt worden tussen meer dan twee mogelijkheden, kan dit gebeuren met een aantal `if` statements. Soms is het mogelijk een meer overzichtelijke structuur te gebruiken.

```
switch ( integer expressie )
{
    case waarde_1:
        nul of meerdere statements;
    case waarde_2:
        nul of meerdere statements;
    ...
    default:
        nul of meerdere statements;
}
volgend_statement;
```

De `integer expressie` wordt geëvalueerd en de resulterende waarde wordt vergeleken met de mogelijke constante waarden bij elke `case`. Bij de eerste gelijke waarde die gevonden wordt, worden de bijhorende statements uitgevoerd. Wanneer het laatste statement in zo'n `case` het `break` statement is, wordt de `switch` verlaten en wordt de uitvoering verder gezet bij `volgend_statement`. Wanneer geen enkele gelijke waarde gevonden wordt, worden de statements behorend bij `default` uitgevoerd. Indien de `default` case weggelaten is, wordt helemaal niets gedaan.

Opgave. Schrijf een programma dat een cijfer inleest. Wanneer dit cijfer groter is dan nul en kleiner is dan 4, moet de bijhorende woordwaarde afgedrukt worden. Bij een andere waarde kan gewoon de melding "cijfer" afgedrukt worden.

```
/*
 * cijfer.c : inlezen en afdrukken van een cijfer
 */
#include <stdio.h>

void main(void)
{
    int symbool;

    printf("Geef een cijfer: ");
    scanf("%d%*c", &symbool);
    switch( symbool )
    {
        case 1 :
            printf("een\n");
            break;
        case 2 :
            printf("twee\n");
            break;
        case 3 :
            printf("drie\n");
            break;
        case 0 :
        case 4 :
        case 5 :
        case 6 :
        case 7 :
        case 8 :
        case 9 :
            printf("cijfer\n");
            break;
        default  :
            printf("Dit is geen cijfer %d\n", symbool);
            break;
    }
}
```

Merk op dat er bij **case** 1, **case** 2 en **case** 3 een **break** staat en bij **case** 0, **case** 4, **case** 5, **case** 6, **case** 7 en **case** 8 niet.

3.2 Iteratie statements

Opgave. Schrijf een programma dat 10 gehele getallen leest en daarvan de som afdrukt.

```
/*
 * intsom1.c : 10 gehele getallen inlezen en de som berekenen
 *             op een langdradige manier
 */
#include <stdio.h>

void main(void)
{
    int getal,som;

    som = 0;
```

```
12          printf("Geef een geheel getal: ");
            scanf("%d%*c",&getal);
14          som += getal;
            printf("Geef een geheel getal: ");
16          scanf("%d%*c",&getal);
            som += getal;
18          printf("Geef een geheel getal: ");
            scanf("%d%*c",&getal);
20          som += getal;
            printf("Geef een geheel getal: ");
22          scanf("%d%*c",&getal);
            som += getal;
24          printf("Geef een geheel getal: ");
            scanf("%d%*c",&getal);
26          som += getal;
            printf("Geef een geheel getal: ");
28          scanf("%d%*c",&getal);
            som += getal;
30          printf("Geef een geheel getal: ");
            scanf("%d%*c",&getal);
32          som += getal;
            printf("Geef een geheel getal: ");
34          scanf("%d%*c",&getal);
            som += getal;
36          printf("Geef een geheel getal: ");
            scanf("%d%*c",&getal);
38          som += getal;
            printf("Geef een geheel getal: ");
40          scanf("%d%*c",&getal);
            som += getal;
42
            printf("De som van deze getallen is %d\n",som);
44      }
```

Dit is duidelijk geen elegante oplossing. Het wordt nog veel erger wanneer bijvoorbeeld 100 getallen moeten gesommeerd worden.
De meeste programmeertalen hebben hiervoor een aantal controle structuren voorzien waarmee het mogelijk is een herhaling (een iteratie) compact neer te schrijven. In deze contekst spreekt men dikwijls over een *programma-lus*. C heeft drie iteratie-structuren.

3.2.1 Het `while` statement

Een eerste structuur is reeds gebruikt geweest in het programma van het algoritme van Euclides (ggd.c). De algemene vorm is:

```
while ( expressie )
      statement;
```

Zolang de `expressie` een waarde *true* (d.i. een waarde verschillend van 0) oplevert, wordt `statement` uitgevoerd. Dit `statement` is meestal een blok-statement:

```
while ( expressie )
{
      statement1;
      statement2;
      ...
```

```
        }
        volgend_statement;
```

In één van deze statements gebeuren normaal aanpassingen aan verschillende variabelen zodat na een tijd de `expressie` de waarde *false* oplevert. Op dat moment wordt de lus verlaten en gaat het programma verder met de uitvoering van `volgend_statement`.

```
1   /*
     *   wsom2.c : berekenen van de som van 10 getallen
3    */
    # include <stdio.h>
5   # define AANTAL 10

7   void main(void)
    {
9       int getal, teller, som;

11      som = 0;
        teller = 0;                     /* initialisatie */
13      while ( teller < AANTAL )       /* test op einde */
        {
15          printf("Geef een geheel getal: ");
            scanf("%d%*c",&getal);
17          som += getal;
            ++teller;                   /* de stap */
19      }
        printf("De som van deze getallen is %d\n", som);
21      printf("en het gemiddelde is %d\n", som/AANTAL);
    }
```

De drie basiselementen die in een herhalingsstructuur voorkomen, zijn:

1. de initialisatie: de variabele `teller` krijgt een beginwaarde;
2. de test op het beëindigen van de lus: de variabele `teller` wordt vergeleken met een eindwaarde (`AANTAL`);
3. de stap: de variabele `teller` wordt aangepast, in dit geval verhoogd.

In de `while` constructie ligt alleen de "test op einde" syntactisch vast. Omdat deze test gebeurt voordat enig statement in de lus uitgevoerd wordt, kan het zijn dat deze test reeds de eerste maal `false` oplevert, zodat de statements in de lus nooit zullen uitgevoerd worden.
In dit voorbeeld wordt ook de `define` preprocessor constructie gebruikt. De bedoeling is een *symbolische naam* aan een *constante* te geven, zodat in de rest van het programma met deze naam kan gewerkt worden. De constante zelf (`10`) komt slechts eenmaal in het programma voor, namelijk bij de `define`. Men spreekt van een *naamconstante*.
Wanneer voor de logische expressie bij de `while` de waarde 1 (`true`) gebruikt wordt, krijgt men een *oneindige lus*:

```
int teller = 0;
while ( 1 )
    ++teller;
```

Omwille van een tikfout krijgt men soms een niet gewenste oneindige lus:

```
int n = 0;
while ( n = 1 )
    ++n;
```

Na de `while` moet minstens één statement in de lus staan. Eventueel kan dit een leeg statement (gewoon een `;`) zijn. Volgend voorbeeld doet waarschijnlijk niet wat bedoeld is:

```
while(teller < AANTAL);
{
    printf("Geef een geheel getal: ");
    scanf("%d%*c", &getal);
    som += getal;
    ++teller;
}
```

3.2.2 Het `for` statement

In een tweede iteratie-structuur is syntactisch vastgelegd hoe de drie basiselementen van een lus (initialisatie, test op einde en de stap) moeten neergeschreven worden. De semantiek is volledig identiek aan het `while` statement.

```
for (init_expressie; test_expressie; stap_expressie)
    statement;
```

De drie basiselementen:

1. de initialisatie: de `init_expressie`: één of meerdere lusvariabelen krijgen elk een beginwaarde;
2. de test op het beëindigen van de lus: `test_expressie`: logische expressie met de lusvariabelen;
3. de stap: de `stap_expressie`: aanpassing van de lusvariabelen.

Ook hier kan `statement` een blok-statement zijn.

```
/*
 * fsom2.c : berekenen van de som van 10 getallen
 */
# include <stdio.h>
# define AANTAL 10

void main(void)
{
    int getal, teller, som;

    som = 0;
    for (teller=0; teller < AANTAL; teller++)
    {
        printf("Geef een geheel getal: ");
        scanf("%d%*c", &getal);
        som += getal;
    }
    printf("De som van deze getallen is %d\n",som);
}
```

De variabele `teller` is in dit voorbeeld de lusvariabele.

De drie expressies in de `for` kunnen weggelaten worden, bijvoorbeeld:

```
for (        ; teller < AANTAL; teller++)
for (teller=0;                ; teller++)
for (        ; teller < AANTAL;         )
for (        ;                ;         )
```

Na de `for` moet weer minstens één statement in de lus staan. Een eventueel leeg statement kan ongewenste resultaten opleveren.

```
for (teller=0; teller < AANTAL; teller++);
{
    printf("Geef een geheel getal: ");
    scanf("%d%*c", &getal);
    som += getal;
}
```

Opmerking. De begin- en eindwaarden van de lusvariabele kunnen zelf variabelen zijn. In de body van de lus mogen deze variabelen wel *niet* van waarde veranderen.

3.2.3 Het `do while` statement

Algemene vorm:

```
do
{
    statement1;
    statement2;
     ...
}
while( expressie );
```

De statements in het blok worden uitgevoerd. Zolang de `expressie` de waarde *true* oplevert, worden de statements in het blok herhaald. Omdat hier de test op het einde van de lus gebeurt, zullen de statements in het blok minstens eenmaal uitgevoerd worden.

```
/*
 *   dsom2.c : berekenen van de som van 10 getallen
 */
# include <stdio.h>
# define AANTAL 10

void main(void)
{
    int getal, teller, som;

    som = 0;
    teller =0;
    do
    {
        printf("Geef een geheel getal: ");
        scanf("%d%*c", &getal);
        som += getal;
        ++teller;
    }
    while(teller < AANTAL);
    printf("De som van deze getallen is %d\n",som);
}
```

Opgave. In een programma moet een cijfer 1, 2, 3 of 4 ingelezen worden (mogelijke keuzen uit een menu). Indien een ander symbool ingegeven wordt, moet de keuze opnieuw opgevraagd worden. Schrijf dit programmadeel.

```
/*
 *  keuze.c : inlezen en controle van een keuze
 */
```

```
4  # include <stdio.h>

6  void main(void)
   {
8      int keuze;

10     do
       {
12         printf("Geef uw keuze [1, 2, 3 of 4]: ");
           scanf("%d%*c", &keuze);
14     }
       while ( keuze < 1 || keuze > 4 );
16 }
```

3.2.4 Het `break` en `continue` statement

Bij het `switch` statement wordt `break` gebruikt om de statements bij een bepaalde `case` af te sluiten zodat verder gegaan wordt met het statement na het `switch` statement.
In de drie lus-constructies is ook een `break` statement mogelijk. Het effect is dat de lus (voortijdig) verlaten wordt en dat de uitvoering verder gezet wordt na het lus statement. Hiermee kan een lus beëindigd worden met behulp van een test die niet in het begin of het einde van de lus staat maar ergens tussenin.
Daarnaast is er ook het `continue` statement. Er wordt dan meteen de test op beëindiging die bij de `while` of `do while` hoort, uitgevoerd. Bij `for` wordt eerst de `stap_expressie` uitgevoerd en dan de test op beëindiging.

Opgave. Schrijf een programma dat een aantal gehele getallen leest; dit aantal is ten hoogste 100. De positieve getallen moeten gesommeerd worden, negatieve getallen worden overgeslagen en wanneer een nul ingelezen wordt, moet gestopt worden.

```
/*
2  * sompos.c : sommeren van positieve getallen
   */
4 #include <stdio.h>
  #define AANTAL 100
6
  void main(void)
8 {
      int i, getal, som;
10
      som = 0;
12    for ( i=1; i<=AANTAL; i++ )
      {
14        scanf("%d%*c", &getal);
          if ( getal < 0 )
16            continue;
          if ( getal == 0 )
18            break;
          som += getal;
20    }
      printf("het aantal ingelezen getallen is %d\n", i);
22    printf("de som van de positieve getallen is %d\n", som);
  }
```

Merk op. Wanneer de `for`-lus in bovenstaand voorbeeld vervangen wordt door een `while`-lus is er een klein verschil in wat het programma doet.

3.3 Nog enkele voorbeelden.

Opgave. Schrijf een programma dat een aantal gehele getallen leest en daarvan het gemiddelde en het grootste bepaalt en afdrukt. Als de rij getallen ingegeven zijn, wordt 999 ingegeven om het einde van de ingave aan te geven. Die 999 mag dus niet meegerekend worden voor gemiddelde of grootste.

```
1  /*
    *  gemid.c : het gemiddelde van een aantal ingelezen getallen
3   */
   #include <stdio.h>
5
   void main(void)
7  {
       int    getal, teller, grootste;
9      double som;
       double gemiddelde;
11
       som=0.0;
13     teller=0;
       printf("Geef een aantal gehele getallen, eindig met 999\n");
15     scanf("%d%*c", &getal);
       grootste = getal;
17     while (getal != 999)
       {
19         som +=getal;                        /* gemengde expressie */
           teller++;
21         if (getal > grootste)
               grootste = getal;
23         scanf("%d%*c", &getal);
       }
25     if (teller == 0)
       {
27         printf("Er zijn geen getallen ingegeven.\n");
       }
29     else
       {
31         printf("Het grootste getal is %d\n", grootste);
           gemiddelde = som/teller;            /* gemengde expressie */
33         printf("Het gemiddelde is %8.2f\n", gemiddelde);
       }
35 }
```

Opgave. Schrijf een programma dat een willekeurig geheel getal, in het interval [1,100] kiest, dat dan moet geraden worden. Als de gok groter is, dan meldt het programma "te hoog". Als de gok kleiner is, dan meldt het programma "te laag". Als de gok gelijk is aan het gekozen getal, dan moet het programma het aantal gedane pogingen afdrukken.

Hulp. De functie-oproep `random(100)` geeft een willekeurig geheel getal in het interval [0,99]. De expressie `random(100)+1` geeft dus een willekeurig geheel getal in [1,100]. Om bij elke uitvoering van het programma een ander random-getal te genereren, moet een functie `randomize()`

opgeroepen worden. De declaratie van deze twee functies (prototypes) is te vinden in het header bestand `<stdlib.h>`.

```
/*
 *   spel.c : raden van een getal tussen 1 en 100
 */
#include <stdio.h>
#include <stdlib.h>

void main(void)
{
    int zoekgetal;
    int getal;
    int teller;
    int gevonden = 0;

    randomize();
    teller=0;
    zoekgetal = random(100)+1;
    printf("Zoeken van een getal in het interval [1,100]\n");
    do
    {
        printf("Doe een gok: ");
        scanf("%d%*c", &getal);
        teller++;
        if (getal == zoekgetal)
            gevonden = 1;
        else if (getal > zoekgetal)
            printf("Uw gok is te hoog\n");
        else
            printf("Uw gok is te laag\n");
    }
    while ( gevonden == 0 );
    printf("Juist gegokt in %d pogingen",teller);
}
```

Opgave. Bereken een benaderende waarde voor de bepaalde integraal van $\cos^2(x)$ tussen $a = 0.0$ en $b = 1.0$. Gebruik voor de benadering de trapezium-regel:

$$\begin{aligned} I &= \sum_{i=1}^{N}(x_{i+1} - x_i)\frac{f(x_i)+f(x_{i+1})}{2} \\ &= h * (\frac{f(a)}{2} + \sum_{i=2}^{N} f(x_i) + \frac{f(b)}{2}) \qquad \text{met } h = (b-a)/N \end{aligned}$$

```
/*
 * trap.c : benadering bepaalde integraal
 */
#include <stdio.h>
#include <math.h>
#define EPS  1.0E-4

void main(void)
{
```

```
10      int     i, n;
        double a, b;
12      double stap, x;
        double integraal, vorig;
14
        printf("   n   |      vorig      integraal\n");
16      printf("-----------------------------------\n");
        n = 1;
18      a = 0.0;
        b = 1.0;
20      vorig = 0.0;
        integraal = (b-a)*(cos(a) * cos(a) + cos(b) * cos(b))/2.0;
22      while ( fabs( (vorig-integraal)/integraal ) > EPS )
        {
24          vorig = integraal;
            n *= 2;
26          stap = (b-a)/n;
            x = a;
28          integraal = (cos(a) * cos(a) + cos(b) * cos(b))/2.0;
            for (i=2; i<=n; i++)
30          {
                x += stap;
32              integraal += (cos(x) * cos(x));
            }
34          integraal *= stap;
            printf("  %3d   |  %10.6f %10.6f \n", n, vorig, integraal);
36      }
        printf("  %3d   |  %10.6f %10.6f \n", n, vorig, integraal);
38  }
```

Het resultaat van dit programma:

```
  n   |     vorig     integraal
--------------------------------
   2  |    0.645963   0.708057
   4  |    0.708057   0.722569
   8  |    0.722569   0.726139
  16  |    0.726139   0.727028
  32  |    0.727028   0.727250
  64  |    0.727250   0.727306
  64  |    0.727250   0.727306
```

Exact:

$$\begin{aligned}\int_0^1 \cos^2 x\,dx &= \left[\tfrac{x}{2} + \tfrac{\cos x \sin x}{2}\right]_0^1 \\ &= 0.727324357\end{aligned}$$

3.4 Begrippen

- Keuze-controle-structuren: `if`, `if-else` en `switch`.
- Lus-controle-structuren: `while`, `for` en `do while`.
- Voortijdig afbreken: `break` en `continue`.

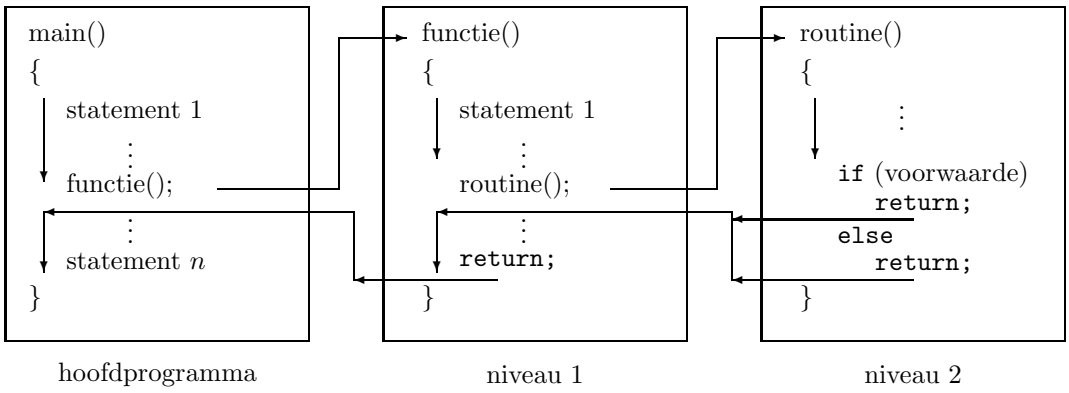
4 Zelfgeprogrammeerde functies

4.1 Waarom zelfgedefinieerde functies

1. om programmagedeelten die meerdere keren voorkomen slechts éénmaal te schrijven, bijv. $\cos^2 x$ in vorig voorbeeld;
2. om zelf een aantal veelgebruikte functies ter beschikking te hebben;
3. om een probleem in kleinere deelproblemen op te splitsen. Deelproblemen zijn meer handelbaar en men hoeft alleen aan de oplossing van dat deelprobleem te denken.

Een vuistregel. Een functie (ook `main`) mag niet langer dan een 50-tal lijnen zijn (een afgedrukte versie op een blad papier geeft dan een goed overzicht). Volgens sommigen mag een functie slechts een 25-tal lijnen lang zijn zodat de volledige functie op één scherm te bekijken is.

Naast *herhalingen* en *beslissingen* verstoren *functieoproepen* het normale lineair verloop van de controle stroom doorheen een programma:



4.2 `void`-functies zonder parameters

De meeste functies in C hebben een terugkeerwaarde. Bij de declaratie en de definitie van de functie wordt dit type vermeld.

```
double cos(double);      /* declaratie in <math.h> */
```

In sommige gevallen worden toch functies gedefinieerd die geen terugkeerwaarde hebben. In andere talen wordt zo'n deelprogramma *procedure* of *subroutine* genoemd.
Wanneer bij de declaratie in C het type gewoon zou weggelaten worden, wordt toch impliciet verondersteld dat het een functie is met een terugkeerwaarde van type `int`. Wanneer men wil aangeven dat er helemaal geen terugkeerwaarde is, kan men het type `void` gebruiken. Dit type `void` wordt ook gebruikt om aan te geven dat er geen parameters zijn (cfr. de `main` functie).

Opgave. Op verschillende plaatsen in een programma moet een lijn bestaande uit 60 '-' tekens op het scherm komen.

```
#include <stdio.h>
#define LIJNLEN 60
void main(void)
{
    void lijn(void);          /* functie declaratie of prototype */
    ...
    lijn();                   /* functie oproep     */
    ...
```

```
    lijn();
    ...
    lijn();
    ...
}

void lijn(void)                 /* functie definitie   */
{
    int i;

    for(i=0; i<LIJNLEN; i++)
    {
        printf("-");
    }
    printf("\n");
}
```

De variabele `i` die in de functie gedefinieerd wordt, is een *lokale* variabele. Zo'n variabele kan alleen binnen de functie gebruikt worden.

4.3 `void`-functies met parameters

Wanneer er toch parameters zijn, worden deze tussen de haakjes bij de functie definitie vermeld. Men noemt dit een *formele parameter*. Zo'n parameter krijgt een naam en er wordt ook een type gespecificeerd. Bij de functie declaratie worden tussen de haakjes de types van de verschillende parameters aangegeven.
Bij de functie oproep worden tussen de haakjes de *actuele argumenten* gegeven. Elk van deze argumenten is een expressie. Deze wordt geëvalueerd tijdens de oproep en de resulterende waarde wordt gebruikt om de corresponderende formele parameter te initialiseren.

Opgave. Zelfde probleem als hiervoor maar de lijn is niet overal even lang.

```
#include <stdio.h>
void main(void)
{
    void lijn(int n);           /* functie declaratie */
    int l = 60;

    ...
    lijn(40);                   /* functie oproep     */
    ...
    lijn(l);
    ...
    lijn(10);
    ...
}

void lijn(int lengte)           /* functie definitie   */
{
    int i;

    for(i=0; i<lengte; i++)
    {
        printf("-");
    }
```

```
        printf("\n");
    }
```

Net zoals de lokale variabele `i` binnen de functie `lijn`, is de parameter `lengte` ook *lokaal* binnen de functie en is dus alleen binnen de functie gekend.

4.4 Functies met een functie-(return)-waarde

Zoals hierboven reeds vermeld hebben de meeste functies in C een terugkeerwaarde. Bij de declaratie en de definitie van de functie wordt het type van deze *terugkeerwaarde* vermeld.
Wanneer een functieoproep in een expressie voorkomt, zal de resulterende waarde van de functie gebruikt worden om de expressie te evalueren. Men kan de `()` van de oproep zien als een unaire operator die toegepast wordt op de functienaam. Deze operator heeft een zeer hoge prioriteit.

Opgave. Schrijf een programma dat een grondtal en een (niet negatieve) exponent inleest en aan de hand van een functie de macht (grondtal tot de macht exponent) berekent.

```
1   /*
     * macht.c : bereken grondtal tot de macht exponent
3    */
    #include <stdio.h>
5   double macht(double a, int n);

7   void main(void)
    {
9       double  grondtal;
        double  resultaat;
11      int     exponent;

13      printf("Machtsberekening\n");
        printf("Geef grondtal: ");
15      scanf("%lf%*c", &grondtal);
        printf("Geef (niet negatieve) exponent: ");
17      scanf("%d%*c", &exponent);
        resultaat = macht(grondtal,exponent);
19      printf("De macht is %8.2lf\n", resultaat);
    }
21
    double macht(double grondtal, int exponent)
23  {
        double functiewaarde = 1.0;
25      int    i;

27      for (i=0; i<exponent; i++)
        {
29          functiewaarde *= grondtal;
        }
31      return functiewaarde;
    }
```

In een expressie `berekening = 5.0 + 7.0 * macht(2.4, 5)` zal eerst de functie `macht` uitgevoerd worden. Het resultaat hiervan zal vermenigvuldigd worden met `7.0` en hierbij zal `5.0` opgeteld worden. Deze uiteindelijke waarde zal dan toegekend worden aan de variabele `berekening`. De operator `()`, d.i. de oproep van de functie, heeft dus een hogere prioriteit dan `*` en `+`.

De terugkeer vanuit een functie naar de oproep hoeft niet op het einde van de functie te gebeuren. Het `return` statement kan om het even waar in de functie staan.
Merk op dat de prototype declaratie zowel *lokaal* binnen het `main` blok kan gebeuren als *globaal* buiten het `main` blok.

4.5 Voorbeelden van gebruik van functies

Een standaard functie:

```
/*
 *  fsin.c : voorbeeld van gebruik van een standaardfunctie
 */
#include <stdio.h>
#include <math.h>    /* bevat (o.a.) de functie declaraties */

void main(void)
{
    double x, y;

    x = M_PI / 6 ;     /* dertig graden */
    y = sin(x);        /* functie oproep */
    printf("De sinus van dertig graden is %8.4lf\n", y);
}
```

Een zelfgedefinieerde functie:

```
/*
 * fbtw.c : voorbeeld van gebruik van een zelf-gedefinieerde functie
 */
#include <stdio.h>
int btw21(int x);    /* functie-prototype (functie-declaratie) */

void main(void)
{
    int faktuurbedrag, btw, inclusief;

    printf("Geef het faktuurbedrag: ");
    scanf("%d%*c", &faktuurbedrag);
    btw = btw21(faktuurbedrag);              /* functie-oproep */
    inclusief = faktuurbedrag + btw;
    printf("BTW           = %5d\n", btw);
    printf("inclusief btw = %5d\n", inclusief);
}

int btw21(int a)                          /* functie-definitie */
{
    int b;

    b = a*0.21 + 0.5;          /* de 0.5 is voor de afronding  */
    return b;
}
```

Merk op. De naam van de formele parameter in het prototype (`x`) hoeft niet dezelfde te zijn als de naam bij de functie-definitie (`a`). Het is echter wel aan te raden voor beginnende programmeurs om dezelfde naam te gebruiken.

4.6 Oude stijl

In de oorspronkelijke taaldefinitie van Kernighan en Ritchie werden functies op een iets andere manier gedefinieerd en gedeclareerd.

```
1   /*
     * oudmacht.c : bereken grondtal tot de macht exponent
3    */
    #include <stdio.h>
5
    main()
7   {
        double  macht();                        /* functie declaratie */
9
        double  grondtal;
11      double  resultaat;
        int     exponent;
13
        grondtal = 7.4;
15      exponent = 4;
        resultaat = macht(grondtal,exponent);       /* functie oproep */
17      printf("De macht is %8.2f\n", resultaat);
    }
19
    double macht(grondtal, exponent)             /* functie definitie */
21  double grondtal;
    int    exponent;
23  {
        double functiewaarde = 1.0;
25      int    i;

27      for (i=0; i<exponent; i++)
            functiewaarde *= grondtal;
29      return functiewaarde;
    }
```

In de functiedeclaratie wordt geen informatie gegeven over het type van de parameters. Alleen het type van de terugkeerwaarde wordt vermeld. Wanneer dit type `int` is, kan de hele declaratie weggelaten worden. Daarnaast wordt bij de functiedefinitie tussen de haakjes alleen de namen van de formele parameters vermeld. De types van deze parameters worden op de volgende lijnen gedeclareerd.
Omdat bij de functiedeclaratie geen types voor de parameters gegeven zijn, kan bij de oproep van de functie niet gecontroleerd worden of wel argumenten van het juiste type gebruikt worden. Een oproep `macht(4.5, 6.7)` zou dus mogelijk zijn. Tijdens complilatie kan niet nagegaan worden of `6.7` van het juiste type is. Het resultaat dat tijdens runtime berekend wordt, zal echter niet juist zijn.

4.7 Geheugenklassen

Lokaal. Variabelen die in een functie gedeclareerd zijn, zijn "by default" *lokale* variabelen. Bij intrede in het functie-blok wijst het systeem automatisch geheugenruimte toe voor die variabelen. Zo'n variabele krijgt pas een betekenisvolle waarde na de eerste toekenning. Deze variabelen zijn alleen binnen dat blok gedefinieerd en gekend. Na het verlaten van dat blok gaan de waarden van die variabelen verloren en wordt de bijhorende geheugenruimte automatisch terug vrijgegeven.

Globaal. Om informatie over de grenzen van blokken of functies heen te kunnen uitwisselen maakt men gebruik van functieparameters. In uitzonderlijke gevallen kan men gebruik maken van van *globale* variabelen. Zo'n variabele wordt buiten een functie gedeclareerd, en er wordt permanent geheugenruimte aan toegewezen. Wanneer het programma start, wordt een globale variabele op nul geïnitialiseerd.
Voorbeeld:

```
/*
 *  scope.c : programma met globale en lokale variabelen
 *  demonstratie van "scope" of het bereik van een variabele
 */
#include <stdio.h>
int   a, b;         /* globale variabelen */
void  f(void);      /* functie-prototype */

void main(void)
{
    a = 5;
    b = 7;
    printf("in het hoofdprogramma is a = %d    b = %d\n", a, b);
    f();
    printf("na aanroep van de functie f is a = %d   b = %d\n", a, b);
}

void f(void)
{
    int a;    /* lokale variabele,  de globale a is onzichtbaar */

    a = 15;
    b = 17;
    printf("in de functie is a = %d    b = %d\n", a, b);
}
```

Dit programma geeft de volgende output:

```
in het hoofdprogramma is a = 5    b = 7
in de functie is a = 15    b = 17
na aanroep van de functie f is a = 5   b = 17
```

Argumenten en parameters. Tijdens de oproep van een functie worden de formele parameters in de functie-definitie geïnitialiseerd met de waarde van de argumenten die in het oproep-statement staan.

```
/*
 *  pararg.c :  een functie met parameters en lokale variabelen
 */
#include <stdio.h>
int f(int x, int y);                /* functie-prototype  */

void main(void)
{
    int a, b, c;       /* lokale variabelen */

    a=5; b=6; c=7;
    printf("Voor aanroep van de functie zijn"
```

```
13              "␣a␣=␣%d,␣␣b␣=␣%d␣␣en␣␣c␣=␣%d\n\n", a, b, c);
        c = f(a,b);                          /* functie-oproep  */
15      printf("Na␣aanroep␣van␣de␣functie␣zijn"
                "␣a␣=␣%d,␣␣b␣=␣%d␣␣en␣␣c␣=␣%d\n\n", a, b, c);
17      a = f(b,c);                          /* functie-oproep  */
        printf("Na␣aanroep␣van␣de␣functie␣zijn"
19              "␣a␣=␣%d,␣␣b␣=␣%d␣␣en␣␣c␣=␣%d\n\n", a, b, c);
    }
21
    int f(int a, int b)                    /* functie-definitie */
23  {
        int c,d;
25
        d = a + b;
27      a = 1;
        b = 2;
29      c = 3;
        printf("In␣de␣functie␣zijn"
31              "␣a␣=␣%d,␣␣b␣=␣%d␣␣en␣␣c␣=␣%d\n\n", a, b, c);
        return d;
33  }
```

Dit programma geeft als output:

```
Voor aanroep van de functie zijn a = 5,  b = 6  en  c = 7
In de functie zijn a = 1,  b = 2  en  c = 3
Na aanroep van de functie zijn a = 5,  b = 6  en  c = 11
In de functie zijn a = 1,  b = 2  en  c = 3
Na aanroep van de functie zijn a = 17,  b = 6  en  c = 11
```

Static. Een `static` variabele is (meestal) een lokale variabele die zijn vorige waarde bij herintrede in het blok (de functie) nog steeds heeft.
Voorbeeld:

```
1   /*
     *  vbstat.c :  voorbeeld met een static-variabele
3    */
    #include <stdio.h>
5   void f(void);

7   void main(void)
    {
9       int i = 14;

11      f();
        printf("main␣␣␣␣:␣i␣=␣%d\n", i);
13      f();
        printf("main␣␣␣␣:␣i␣=␣%d\n", i);
15      f();
    }
17
    void f(void)
19  {
        static int i = 5;       /* statische variabele  */
```

```
21
        printf("functie : i = %d\n", i);
23      i++;
    }
```

Dit programma geeft als output:

```
functie : i = 5
main    : i = 14
functie : i = 6
main    : i = 14
functie : i = 7
```

4.8 Een probleem in deelproblemen opsplitsen

Opgave. Schrijf een programma dat de coëfficiënten van een vierkantsvergelijking inleest en de oplossingen van deze vierkantsvergelijking afdrukt. Beschouw alle gevallen!

```
    /*
2    *  vkv.c :  oplossen van een vierkantsvergelijking
     */
4   #include <stdio.h>
    #include <math.h>
6   void lineair(double a, double b);
    void vkv(double a, double b, double c);
8
    void main(void)
10  {
        double a, b, c;
12
        printf("Oplossen van een vierkantsvergelijking\n");
14      printf("Geef de coefficienten a, b en c ");
        scanf("%lf%lf%lf%*c", &a, &b, &c);
16      if (a)
        {
18          vkv(a,b,c);
        }
20      else
        {
22          lineair(b,c);
        }
24  }

26  /*
     *  oplossen van de lineare vergelijking ax+b=0
28   */
    void lineair(double a, double b)
30  {
        if ( a )
32      {
            printf("lineare vergelijking met wortel: %f\n", -b/a);
34      }
        else
36      {
```

```
            if ( b )
38              printf("Valse vergelijking\n");
            else
40              printf("Identieke vergelijking\n");
        }
42  }

44  /*
     *    oplossen van een "echte" vierkantsvergelijking
46   */
    double  rat(double a, double b);
48  double  irrat(double a, double d);

50  void vkv(double a, double b, double c)
    {
52      double d;
        double x1, x2;
54
        x1 = rat(a,b);
56      d = b*b-4*a*c;
        if ( d==0 )
58      {
            printf("Een dubbele wortel %f\n", x1);
60      }
        else if ( d>0 )
62      {
            x2 = irrat(a,d);
64          printf("Twee reele wortels: %f en %f\n", x1+x2, x1-x2);
        }
66      else if ( d<0 )
        {
68          x2 = irrat(a,-d);
            printf("Twee complexe wortels: %f + %fi  en  %f - %fi\n",
70                  x1, x2, x1, x2);
        }
72  }

74  double rat(double a, double b)
    {
76      return  -b/2/a;                         /* links associatief !! */
    }
78
    double irrat(double a, double d)
80  {
        return sqrt(d)/2/a;
82  }
```

4.9 Leesoefening.

Opgave. Leg de werking van volgend programma uit; wat wordt er uitgeschreven?

```
    /*  funpri.c : oproepen van een functie in een expressie */
2   #include <stdio.h>
    int fun(int x, int y);                  /* functie-prototype  */
```

```
4   void main(void)
    {
6       int a = 7;
        int b = 2;
8       int r;

10      r = fun(a,b);
        printf("␣a␣=␣%d,␣␣b␣=␣%d␣␣en␣␣r␣=␣%d\n", a, b, r);
12      r = --a + fun(a,b) - b++;
        printf("␣a␣=␣%d,␣␣b␣=␣%d␣␣en␣␣r␣=␣%d\n", a, b, r);
14  }
    int fun(int x, int y)                    /* functie-definitie */
16  {
        int h;
18
        h = x;
20      x = y;
        y = h;
22      printf("␣␣␣␣␣␣x␣=␣%d,␣␣y␣=␣%d␣\n", x, y);
        return x + y + h;
24  }
```

4.10 Begrippen

- Functies:

oproep	definitie	prototype

f();	void f(void)
	{
	...
	return;
	}
		void f(void);

f(n);	void f(int m)
	{
	...
	... m ...;
	...
	return;
	}
		void f(int m);

y = f(x);	double f(double y)
	{
	double r;
	...
	r = ...y...;
	...
	return r;
	}
		double f(double z);

- Returnwaarde, parameters en argumenten.
- Geheugenklassen: lokaal, globaal, static.
- Belangrijke opmerking. Opdat een functie algemeen bruikbaar zou zijn, mag een functie, tenzij in uitzonderlijke gevallen, van geen globale variabelen gebruik maken. D.w.z. alle variabelen in een functie moeten lokaal in die functie gedeclareerd zijn of via de parameterlijst doorgegeven zijn.

5 Arrays

5.1 Waarom arrays

Opgave. Schrijf een programma dat 5 gehele getallen leest en in omgekeerde volgorde terug afdrukt.

```
/*
 *  keerom1.c
 */
#include <stdio.h>

void main(void)
{
    int a,b,c,d,e;

    printf("Geef 5 gehele getallen: ");
    scanf("%d%d%d%d%d%*c", &a, &b, &c, &d, &e);
    printf("In omgekeerde volgorde is dit: %d %d %d %d %d\n",
            e, d, c, b, a);
}
```

Het bovenstaand programma is geen stijlvol programma. Er worden vijf verschillende variabelen gebruikt voor iets dat vanuit het probleemdomein als één rij getallen bestempeld wordt. Daarnaast is de oplossing ook onbruikbaar wanneer het over meer getallen (bijvoorbeeld 100) gaat.

5.2 Een array

Gerelateerde variabelen van hetzelfde data-type kunnen samengevoegd worden in een *array*. Een array is een *samengesteld data-type* en kan gezien worden als een rij waarvan de elementen allemaal met dezelfde naam maar elk met een eigen volgnummer worden aangeduid.
Een programma met een array voor voorgaand probleem:

```
/*
 *   keerom2.c
 */
#include <stdio.h>
#define AANTAL 100

void main(void)
{
    int a[AANTAL];
    int i;

    printf("Geef %d gehele getallen:\n", AANTAL);
    for(i=0; i<AANTAL; i++)
    {
        printf("Geef getal %3d: ", i+1);
        scanf("%d%*c", &a[i]);                    /* onder elkaar */
    }
    printf("In omgekeerde volgorde:\n");
    for (i=AANTAL-1; i>=0; i--)
        printf("%8d", a[i]);
    printf("\n");
}
```

De declaratie `int a[100]` geeft aan dat `a` de naam is van een rij van honderd gehele (int) getallen. Een declaratie van een array bevat dus drie elementen:

data type : het type (int, double, ...) van elk van de elementen in de array;

naam : zoals bij elke variabele;

grootte : het aantal elementen in de array; dit moet een *constante-expressie* zijn, zodat de compiler de juiste hoeveelheid geheugenplaatsen in het werkgeheugen kan reserveren.

In plaats van een getal voor de grootte van de array te gebruiken, wordt meestal met een *naam-constante* (symbolische naam) gewerkt:

```
#define  AANTAL   100
```

Wanneer het programma moet aangepast worden om bijvoorbeeld rijen met 1000 elementen om te keren, moet dit slechts op één plaats gebeuren, bij de `define`.
De lengte of grootte van een array wordt ook wel *dimensie* genoemd. Een rij is een een-dimensionale array en wordt soms ook *vector* genoemd.
Bij de declaratie kunnen de elementen van de array geïnitialiseerd worden:

```
int a[5] = { 17, 19, 23, 29, 31 };
```

Bij tegelijk declareren en initialiseren, hoeft de lengte niet gespecificeerd te worden:

```
int a[] = { 17, 19, 23, 29, 31 };
```

Dit is nu echter geen goede manier van werken meer. Eén reden hiervoor is het niet gebruiken van de `define` constructie om de lengte een symbolische naam te geven, die dan overal in het programma kan gebruikt worden. Wanneer er minder initialiserende waarden zijn dan opgegeven in de lengte, worden de eerste elementen geïnitialiseerd met de gegeven waarden en de volgende elementen krijgen de waarde nul (afhankelijk van de gebruikte compiler).

```
int a[12] = { 17, 19, 23, 29, 31 };
```

De verschillende elementen in de array `a` kunnen aangesproken worden met de `[]` operator:

```
a[0],  a[1],  a[2],  ..., a[98],  a[99]
```

Honderd geeft het aantal elementen aan, maar de rangnummer begint bij 0 en eindigt dus bij 99. Voor het element `a[100]` is dus GEEN plaats voorzien.
Bij het gebruik van een array-element `a[i]` wordt i de *index* of *subscript* genoemd. Deze index kan een constante, een variabele of een expressie zijn. De waarde van de index moet wel een geheel getal zijn, groter of gelijk aan nul en kleiner dan het aantal elementen dat bij de declaratie is opgegeven.
De naam van de array zelf kan niet zomaar in om het even welke expressie gebruikt worden. Deze naam duidt namelijk geen enkelvoudige geheugenplaats aan maar een rij van geheugenplaatsen. De rekenkundige operator `+` bijvoorbeeld kan wel gebruikt worden om de inhoud van twee geheugenplaatsen op te tellen. Het resultaat kan dan met de toekenningsoperator (`=`) toegewezen worden aan een derde variabele (geheugenplaats).

```
double a, b, c;

a = b + c;
```

Wanneer `a`, `b` en `c` arrays zijn, elk bijvoorbeeld van lengte 10, is bovenstaande rekenkundige operatie NIET mogelijk. Een variabele van het type array heeft geen *lvalue*. De optelling moet element per element gebeuren:

```
/*
 * arsom.c : optellen van twee arrays
 */
#include <stdio.h>
#define AANTAL 5

void main(void)
{
    double a[AANTAL], b[AANTAL], c[AANTAL];
    int   i;

    /* inlezen van vectoren b en c */
    for (i=0; i<AANTAL; i++)
        a[i] = b[i] + c[i];
    /* afdrukken van vector a  */
}
```

De expressie `a == b` is syntactisch wel correct maar wordt semantisch anders geïnterpreteerd dan waarschijnlijk bedoeld is. Deze expressie gaat niet na of de twee arrays dezelfde elementen bevatten. De gelijkheid nagaan moet weer element per element gebeuren:

```
/*
 * arvgl.c : vergelijken van twee arrays
 */
#include <stdio.h>
#define AANTAL 5

void main(void)
{
    int a[AANTAL], b[AANTAL];
    int   i;
    int   gelijk = 1;

    /* inlezen van vectoren a en b */
    for (i=0; i<AANTAL; i++)
        if ( a[i] != b[i] )
        {
            gelijk = 0;
            break;
        }
}
```

Na de lus kan de variabele `gelijk` getest worden. Wanneer deze gelijk is aan 1, zijn de twee arrays gelijk, in de betekenis dat de waarden in de corresponderende elementen gelijk zijn.
Merk op dat in het programma een *kortsluitingsprincipe* kan ingebouwd worden. Na het vinden van een eerste index waarbij de twee elementen niet gelijk zijn, hoeven de volgende elementen niet meer vergeleken te worden.

Belangrijke vaststelling. De naam van een array duidt dus een rij van geheugenplaatsen aan. De *rvalue* van deze variabele is het *beginadres* van de rij. Wanneer deze naam als actuele parameter in een functie-oproep gebruikt wordt, dan wordt de waarde van de formele parameter in de functie-definitie gelijk aan dit beginadres. Resultaat: de formele parameter duidt dezelfde rij van geheugenplaatsen aan! Dit wil dus zeggen dat wanneer een arraynaam als actueel argument gebruikt wordt, deze array niet element per element in de formele parameter gecopieerd wordt (zoals wel gebeurt bij een enkelvoudig data-type).

Omdat de formele parameter wijst naar de originele rij, zal elke wijziging die in de functie door middel van de formele parameter gebeurt, uitgevoerd worden op deze originele rij. Dit wordt geïllustreerd in volgend voorbeeld waarin met behulp van functies, een rij getallen in omgekeerde volgorde uitgeschreven wordt.

```
/*
 * keerom3.c : (met functies)
 */
#include <stdio.h>
#define AANTAL 100

void leesrij(int x[], int n);                          /* prototype */
void drukrij(int x[], int n);

void main(void)
{
    int m;
    int a[AANTAL];

    scanf("%d%*c", &m);
    leesrij(a, m);                                  /* functie oproep */
    drukrij(a, m);
}

void leesrij(int x[], int n)                      /* functie definitie */
{
    int i;

    printf("Geef %d gehele getallen: ", n);
    for(i=0; i<n; i++)
    {
        scanf("%d", &x[i]);                          /* naast elkaar */
    }
    scanf("%*c");
}

void drukrij(int r[], int m)
{
    int i;

    printf("In omgekeerde volgorde:\n");
    for (i=0; i<m; i++)
    {
        printf("%6d", r[m-1-i]);
    }
    printf("\n");
}
```

Bij de definitie van de functie wordt bij de formele parameter de lengte van de array niet gegeven. Deze lengte is van geen belang omdat in de formele parameter alleen een beginadres zal komen bij een oproep en niet de volledige rij.

5.3 Meer-dimensionale arrays

In veel toepassingen heeft men te maken met een tabel, soms ook *matrix* genoemd. Dit kan in C neergeschreven worden als een array met twee dimensies:

```
int  a[MAXRIJ][MAXKOLOM];
```

De eerste dimensie geeft aan hoeveel rijen er zijn en de tweede dimensie het aantal kolommen. Wanneer MAXRIJ gelijk is aan 3 en MAXKOLOM gelijk aan 2 geeft dit de volgende matrix

a[0][0]	a[0][1]
a[1][0]	a[1][1]
a[2][0]	a[2][1]

Deze matrix wordt in het werkgeheugen echter gestockeerd als één lange rij van opeenvolgende elementen:

a[0][0]	a[0][1]	a[1][0]	a[1][1]	a[2][0]	a[2][1]

Dit kan gezien worden als een rij van elementen waarin elk element op zich een rij is.
Bij de definitie van de variabele kan deze geïnitialiseerd worden:

```
int a[3][2] = { {1, 2}, {2, 3}, {3, 4} };
```

Rekening houdend met het feit dat deze matrix als een rij gestockeerd wordt, kan men ook schrijven

```
int a[3][2] = { 1, 2, 2, 3, 3, 4 };
```

Dit is echter veel minder duidelijk.

Opgave. Construeer een matrix waarin elk element gelijk is aan de som van de rij-index en de kolom-index en druk deze matrix af.

```
/*
 *  matrix.c :  met mat[i][j] = i+j
 */
#include <stdio.h>
#define MAXRIJ    3
#define MAXKOLOM 5

void drukmat(int mat[][MAXKOLOM], int m, int n);

void main(void)
{
    int   mat[MAXRIJ][MAXKOLOM];
    int   rij, kolom;

    for (rij=0; rij<MAXRIJ; rij++)
            for (kolom=0; kolom<MAXKOLOM; kolom++)
            mat[rij][kolom] = rij + kolom;
    drukmat(mat, MAXRIJ, MAXKOLOM);
}

void drukmat(int mat[][MAXKOLOM], int m, int n)
{
    int i, j;

    for (i=0; i<m; i++)
```

```
26      {
                for (j=0; j<n; j++)
28              {
                printf("%4d", mat[i][j]);
30              }
                printf("\n");
32      }
    }
```

De geconstrueerde matrix heeft volgende inhoud:

0	1	2	3	4
1	2	3	4	5
2	3	4	5	6

en wordt in het werkgeheugen als één lange rij gestockeerd:

0	1	2	3	4	1	2	3	4	5	2	3	4	5	6

In de formele parameter zal bij een oproep alleen het beginadres komen net zoals bij een eendimensionale array. Toch wordt in de definitie van de functie bij de formele parameter de waarde van de tweede dimensie van de array opgegeven. Deze lengte is in de functie nodig om de positie te kunnen berekenen van het element in de lange rij waarin de matrix in het werkgeheugen gestockeerd is. Het element `mat[i][j]` kan gevonden worden op positie $i * \text{MAXKOLOM} + j$. De grootte van de eerste dimensie is hiervoor niet nodig en moet dus ook niet gespecificeerd worden in de declaratie van de formele parameter.

5.4 Uitgewerkte voorbeelden

5.4.1 Priemgetallen

Opgave. Schrijf een programma met functies dat met behulp van de "zeef van Eratosthenes" alle priemgetallen kleiner dan 1000 afdrukt (alle niet-priemgetallen worden weggezeefd).

```
1   /*
     *  zeef.c : alle priemgetallen < MAXAANTAL
3    */
    #include <stdio.h>
5   #define MAXAANTAL 1000

7   void vulzeef(int zeef[]);
    void schrap(int i, int zeef[]);
9
    void main(void)
11  {
        int zeef[MAXAANTAL];
13      int i;

15      vulzeef(zeef);
        printf("Priemgetallen:\n");
17      for (i=2; i<MAXAANTAL; i++)
        {
19          if ( zeef[i] == 1 )
            {
21              printf("%4d",i);
                schrap(i,zeef);
```

```
23              }
        }
25      printf("\n");
    }
27
    void vulzeef(int zeef[])
29  {
        int i;
31
        zeef[0]=zeef[1]=0;
33      for (i=2; i<MAXAANTAL; i++)
        {
35          zeef[i] = 1;
        }
37  }

39  void schrap(int getal, int zeef[])
    {
41      int i;

43      for (i=2*getal; i<MAXAANTAL; i+=getal)
        {
45          zeef[i] = 0;
        }
47  }
```

5.4.2 Driehoek van Pascal

Opgave. Schrijf een programma met functies dat de driehoek van Pascal afdrukt tot op rij n, waarbij n in te lezen is. Een eerste functie moet n inlezen, een tweede functie moet de driehoek maken en een derde functie moet de driehoek afdrukken.

$$(a+b)^n = \binom{n}{0}a^n b^0 + \binom{n}{1}a^{n-1}b^1 + \binom{n}{2}a^{n-2}b^2 + \ldots + \binom{n}{n-1}a^1 b^{n-1} + \binom{n}{n}a^0 b^n$$

$$\binom{n+1}{k} = \binom{n}{k} + \binom{n}{k-1}$$

```
1   /*
     *  pascal.c : zijn driehoek
3    */
    #include <stdio.h>
5   #define MAX 20

7   int leesdimensie(void);
    void maakdriehoek(int a[][MAX], int n);
9   void printdriehoek(int a[][MAX], int n);

11  void main(void)
    {
13      int a[MAX][MAX];
        int n;
15
        printf("Driehoek van Pascal\n");
```

```
17      n = leesdimensie();
        maakdriehoek(a, n);
19      printdriehoek(a, n);
    }
21
    int leesdimensie(void)
23  {
        int d;
25
        do
27      {
            printf("geef dimensie (in [0 .. %2d]): ", MAX);
29          scanf("%d%*c", &d);
        } while ( d>=MAX );
31      return d;
    }
33
    void maakdriehoek(int a[][MAX], int n)
35  {
        int i,j;
37
        for (i=0; i<=n; i++)
39      {
            a[i][0] = 1;
41          a[i][i] = 1;
        }
43      for (i=1; i<=n; i++)
        {
45          for (j=1; j<i; j++)
            {
47              a[i][j] = a[i-1][j-1] + a[i-1][j];
            }
49      }
    }
51
    void printdriehoek(int a[][MAX], int n)
53  {
        int i,j;
55
        for (i=0; i<=n; i++)
57      {
            printf("  %2d: ", i);
59          for (j=0; j<=i; j++)
            {
61              printf("%6d", a[i][j]);
            }
63          printf("\n");
        }
65  }
```

Het resultaat voor $n = 6$:

0:	1						
1:	1	1					
2:	1	2	1				
3:	1	3	3	1			
4:	1	4	6	4	1		
5:	1	5	10	10	5	1	
6:	1	6	15	20	15	6	1

5.4.3 Matrix-product

Opgave. Schrijf een programma dat twee matrices inleest en het matrix product berekent en afdrukt.

```
1   /*
     *  matprod.c :  matrix product  C = A x B
3    */
    #include <stdio.h>
5   #define MDIM 10

7   void leesmatrix(double x[][MDIM], int ar, int ak);
    void drukmatrix( double x[][MDIM], int ar, int ak);
9   double inwprod(double a[][MDIM], double b[][MDIM], int r, int k, int l);
    void maakproduct( double a[][MDIM], double b[][MDIM],
11                    double c[][MDIM], int ar1, int ak1, int ak2);
    void main(void)
13  {
        double a[MDIM][MDIM];
15      double b[MDIM][MDIM];
        double c[MDIM][MDIM];
17      int ar1, ak1, ar2, ak2;

19      printf("Geef het aantal rijen van de eerste matrix ");
        scanf("%d%*c", &ar1);
21      printf("Geef het aantal kolommen van de eerste matrix ");
        scanf("%d%*c", &ak1);
23      ar2=ak1;
        printf("geef het aantal kolommen van de tweede matrix ");
25      scanf("%d%*c", &ak2);
        printf("Geef de eerste matrix\n");
27      leesmatrix(a, ar1, ak1);
        printf("Geef de tweede matrix\n");
29      leesmatrix(b, ar2, ak2);
        maakproduct(a, b, c, ar1, ak1, ak2);
31      printf("\nDe productmatrix is:\n");
        drukmatrix(c, ar1, ak2);
33  }
    void leesmatrix(double x[][MDIM], int ar, int ak)
35  {
        int i, j;
37      double y; /* nodig om een bug in C++ op te vangen */

39      for (i=0; i<ar; i++)
        {
41          printf("Geef rij %d : ", i+1);
            for (j=0; j<ak; j++)
```

```
43              {
                    scanf("%lf%*c", &y);
45                  x[i][j] = y;
                }
47              printf("\n");
            }
49      }
        void drukmatrix( double x[][MDIM], int ar, int ak)
51      {
            int i,j;
53
            for (i=0; i<ar; i++)
55          {
                for (j=0; j<ak; j++)
57              {
                    printf("%8.2f", x[i][j]);
59              }
                printf("\n");
61          }
        }
63      double inwprod(double a[][MDIM], double b[][MDIM], int r, int k, int l)
        {
65          int j;
            double som;
67
            som = 0.0;
69          for(j=0; j<l; j++)
            {
71              som = som + a[r][j]*b[j][k];
            }
73          return som;
        }
75      void maakproduct(double a[][MDIM], double b[][MDIM],
                         double c[][MDIM], int ar1, int ak1, int ak2)
77      {
            int     r, k;
79
            for (r=0; r<ar1; r++)
81          {
                for (k=0; k<ak2; k++)
83              {
                    c[r][k] = inwprod(a, b, r, k, ak1);
85              }
            }
87      }
```

5.5 Begrippen

- Array: declaratie en initialisatie.
- De selectie-operator `[]`.
- Een arraynaam als argument bij een functieoproep.
- Meer-dimensionale arrays.

6 Pointers

6.1 Operatoren

Als argument aan `scanf` wordt niet de naam van de variabele (bijv. `a`) doorgegeven, maar de expressie `&a`. De unaire operator `&` toegepast op een variabele heeft als resultaat het adres van deze variabele in het werkgeheugen. De waarde van dit adres is voor de gebruiker een nietsbetekenend getal, maar `scanf` gebruikt deze waarde om een verwijzing te hebben naar de geheugenplaats waar het ingelezen getal moet gestockeerd worden.
In plaats van de waarde `&a` door te geven als een argument in een functie, kan die ook toegekend worden aan een andere variabele:

```
p = &a
```

Zo'n variabele wordt een *pointer* variabele genoemd, omdat de inhoud een verwijzing is naar een andere variabele: de waarde is dus het adres van die andere variabele.
De omgekeerde operator bestaat ook:

```
b = *p
```

In dit geval wordt de verwijzing die in de pointer `p` zit, *gevolgd*. Men komt dus uit bij een andere geheugenplaats en de inhoud van die geheugenplaats is het resultaat. Het is ook mogelijk `*p` als lvalue te gebruiken:

```
*p = 8
```

Op de plaats waar de pointer `p` naar wijst, wordt de waarde 8 gestockeerd.
Bij de declaratie van pointervariabelen wordt aangegeven naar wat voor data-type de pointer wijst:

```
int *p;
```

Het object waar `p` naar wijst (`*p`) is van het type `int`. Of, `p` is dus een pointer naar een integer. Noteer wel dat na deze declaratie, alleen maar plaats voor de pointer (bijv. 4 bytes) gereserveerd is in het werkgeheugen. De inhoud van `p` is nog niet geïnitialiseerd, dus `p` wijst nog naar *nergens*.

```
1  /*
   *    pointer.c
3  */
   #include <stdio.h>
5
   void main(void)
7  {
       int a, b;
9      int *p;

11     scanf("%d%*c", &a);
       p = &a;
13     *p = 5;

15     p = &b;
       printf("Geef een geheel getal: ");
17     scanf("%d%*c", p);

19     printf("a = %d,     b = %d\n", a, b);
   }
```

Soms wordt een iets andere notatie gebruikt:

```
int* p;
```

Men zou dit dan kunnen lezen als de variabele `p` is een "pointer naar een int".
Deze notatie is niet zo duidelijk wanneer verschillende variabelen op dezelfde lijn gedeclareerd worden:

```
int* p, i;
```

Niettegenstaande de waarschijnlijke intentie van de schrijver, is de variabele `i` geen pointer maar gewoon een `int`.

6.2 Toepassing: het wijzigen van variabelen via parameters

6.2.1 Call by value

Wanneer in C een functie met parameters opgeroepen wordt, worden de waarden van de actuele argumenten doorgegeven naar de formele parameters: de waarden van de actuele argumenten worden gebruikt als initialisatie voor de formele parameters.
In de functie zelf worden de formele parameters gebruikt als operands in de expressies. Deze formele parameters kunnen daarbij van waarde veranderen. Maar wanneer de functie uitgevoerd is en verlaten wordt (via `return`), worden deze eventuele aangepaste waarden van de formele parameters NIET teruggecopieerd naar de actuele argumenten in de oproep.

```
1  /*
    *    param.c :  het gedrag van parameters
3   */
   #include <stdio.h>
5  #define AANTAL 10

7  int kwadar(int a, int b[]);

9  void main(void)
   {
11     int i, n, res;
       int b[AANTAL];
13
       n = 5;
15     for (i=0; i<AANTAL; i++)
           b[i] = i;
17     printf("Voor aanroep van de functie is n = %d\n", n);
       printf("En de rij b bevat : ");
19     for(i=0; i<AANTAL; i++)
           printf("%4d", b[i]);
21     printf("\n");
       res = kwadar(n, b);
23     printf("Na aanroep van de functie is n = %d\n", n);
       printf("    de rij b bevat : ");
25     for (i=0; i<AANTAL; i++)
           printf("%4d", b[i]);
27     printf("\n");
       printf("en het resultaat is %d\n", res);
29 }

31 int kwadar(int x, int y[])
   {
33     int i;

35     x = x*x;
```

```
    for (i=0; i<AANTAL; i++)
37              y[i] = i*i;
    return x;
39  }
```

Dit programma drukt af:

```
Voor aanroep van de functie is n = 5
En de rij b bevat :    0   1   2   3   4   5   6   7   8   9
Na aanroep van de functie is n = 5
   de rij b bevat :    0   1   4   9  16  25  36  49  64  81
en het resultaat is 25
```

De waarde van `n` wordt doorgegeven naar de formele parameter `x`. In de functie `kwadar` wordt deze `x` gewijzigd. Maar na terugkeer naar de functie `main` is `n` niet gewijzigd. Zoals in vorig hoofdstuk uitgelegd, worden arrays op een andere manier doorgegeven. In dit geval worden niet alle elementen van de array gecopieerd naar de formele parameter, maar wordt alleen het beginadres van de array (via de arraynaam) doorgegeven naar de formele parameter. Dus in de functie `kwadar` wijst deze formele parameter `y` naar de array `b` van de `main` functie.

6.2.2 Call by address

Om wijzigingen die in de functie gebeuren op de formele parameters, toch effect te laten hebben in de oproepende routine op de actuele argumenten, moet iets analoogs gebeuren als met het doorgeven van een array naar een functie. In plaats van de waarde zelf door te geven naar de functie, kan het adres naar de variabele doorgegeven worden. De formele parameter is dan van het pointer-type. In de functie wijst de waarde van de pointer naar de variabele zelf, zodat de waarde ervan kan gewijzigd worden.

```
1   /*
     * wissel.c :   wijzigen van variabelen via parameters
3    */
    #include <stdio.h>
5   void wissel(int *pi, int *pj);

7   void main(void)
    {
9       int a,b;

11      a = 5;
        b = 8;
13
        printf("Voor de functieoproep is a = %d    b = %d\n", a, b);
15      wissel(&a, &b);
        printf("  Na de functieoproep is a = %d    b = %d\n", a, b);
17  }

19  void wissel(int *x, int *y)
    {
21      int hulp;

23      hulp = *x;
        *x = *y;
25      *y = hulp;
    }
```

Dit programma drukt af:

```
Voor de functieoproep is a = 5   b = 8
  Na de functieoproep is a = 8   b = 5
```

In de functie `scanf` wordt dus door middel van de expressie `&a` als argument een *call by address* gedaan. In volgend voorbeeld gebeurt de adresberekening reeds in de oproepende functie:

```
/*
 *   flees.c : inlezen van variabelen met een functie
 */
#include <stdio.h>
void lees(int *pi, int *qi);

void main(void)
{
    int a, b;

    lees(&a, &b);
    printf("Na de functieoproep is a = %d   b = %d \n", a, b);
}

void lees(int *p, int *q)
{
    printf("Geef twee gehele getallen: ");
    scanf("%d%d%*c", p, q);
}
```

6.3 Arrays en pointers

Door middel van de declaratie `int x[5]` wordt in het werkgeheugen plaats voor 5 gehele getallen gereserveerd. Deze plaatsen kunnen aangesproken worden als `x[0], x[1], ... x[4]`. De naam van de array `x` is een aanduiding voor het beginadres van deze gereserveerde zone. De grootte van deze gereserveerde zone wordt vastgelegd tijdens compilatie. De naam van de array is een expressie met een waarde gelijk aan het adres van het eerste element van de array.

```
int   x[5];
int   *p;
int   *q;

p = &x[0] ;
q = x;
```

Het resultaat van deze twee toekenningen is identiek: zowel `p` als `q` zijn pointers die wijzen naar het begin van de plaats waar de array `x` gestockeerd is. Ook de naam `x` wijst naar dit begin, maar hier is een klein onderscheid. `p` en `q` zijn variabelen die door de toekenningen een waarde gekregen hebben; deze variabelen kunnen een andere waarde krijgen. `x` is de naam van een array en kan niet van waarde veranderen! Of `p` en `q` hebben een *lvalue*: er is plaats in het werkgeheugen voorzien; terwijl voor `x` zelf geen plaats voorzien is: `x` heeft geen *lvalue*.

6.3.1 De grenzen van een array

Door middel van een declaratie `int x[5]` wordt een gebied van 5 integers gereserveerd in het werkgeheugen. Deze elementen kunnen aangesproken worden met `x[0], x[1], x[2], x[3]` en `x[4]`. Maar een standaard C compiler zal geen problemen maken wanneer `x[5]` of `x[-1]` gespecificeerd wordt. Het is zo dat elke index mag gebruikt worden, tijdens run-time wordt geen controle gedaan of de index wel binnen de gedefinieerde grenzen ligt.

Om een element met index **i** uit een array van integers op te halen, wordt gekeken naar de integer die gestockeerd is op een offset van $i \times$ **sizeof(int)** bytes vanaf de start van de array. Deze index i kan positief of negatief zijn. Wanneer deze index buiten de gedefinieerde grenzen van de array valt, zal een element in het werkgeheugen aangesproken worden dat waarschijnlijk voor iets anders in gebruik is. Wanneer zo'n element in een expressie gebruikt wordt, zal dit element een waarde geven die waarschijnlijk niet erg zinvol is. Wanneer een toekenning aan zo'n element gebeurt, zal andere nuttige informatie verloren zijn. Het kan ook zijn dat de toekenning leidt tot een run-time fout omdat een gedeelte van het werkgeheugen wordt aangesproken waar door het programma niet mag gestockeerd worden. Dit kan tot zeer moeilijk te vinden fouten leiden, die eventueel pas opduiken nadat het programma al jaren in gebruik is.
Deze flexibele omgang met arrays is één van de grote ergernissen bij mensen die van een andere taal overstappen naar C. Maar een echte C-programmeur weet wat hij aan het doen is en heeft daarbij geen run-time controles nodig.

6.3.2 Rekenen met pointers

Een pointer is een data type gelijkaardig aan een integer. Er kunnen een beperkte set van bewerkingen met pointers gebeuren:

- De optelling of de aftrekking van een integer bij/van een pointer geeft een nieuw adres.
- Pointers kunnen met elkaar vergeleken worden (`==` en `!=`) om na te gaan of ze naar hetzelfde element in het geheugen wijzen; of ze dus hetzelfde adres bevatten.
- De aftrekking van twee pointers geeft het aantal elementen tussen de twee adressen.

```
/*
 * arp.c : rekenen met pointers
 */
#include <stdio.h>
#define AANTAL 5

void main(void)
{
    int     a[AANTAL];
    int     *pi;
    int     *qi;
    int     i;

    for (i=0; i<AANTAL; i++)
        a[i] = i;
    pi = &a[1];
    qi = &a[3];
    printf("pi %8x   qi %8x   pi+2 %8x   *(pi+2) %4d   qi-pi %d\n",
                pi, qi, pi+2, *(pi+2), qi-pi);
    pi = &a[0];
    qi = &a[AANTAL-1];
    for (   ; pi<=qi; pi++)
    {
        i = pi - a;
        printf("        a[%1d] = %4d %4d   (%.8x)\n", i, *pi, a[i], pi);
    }
}
```

De uitvoer van dit programma:

```
pi 7f7f124c   qi 7f7f1254   pi+2 7f7f1254   *(pi+2)    3   qi-pi 2
      a[0] =    0    0   (7f7f1248)
      a[1] =    1    1   (7f7f124c)
      a[2] =    2    2   (7f7f1250)
      a[3] =    3    3   (7f7f1254)
      a[4] =    4    4   (7f7f1258)
```

Een element in een array `a` kan aangesproken worden met `a[i]` (de array voorstelling) en met `*(a+i)` (de pointer voorstelling).
Wanneer een array als actueel argument gebruikt wordt in een functie dan wordt in de functiedefinitie de formele parameter getypeerd als `int []`. In deze formele parameter wordt bij een oproep een adres naar een gebied (de array) gestockeerd. Het type kan dus evengoed gespecificeerd worden als `int *`. Nochtans krijgt `int []` de voorkeur omdat hiermee beter aangegeven wordt dat het over een array gaat. Voor de compiler is er echter geen enkel verschil.

6.3.3 Meerdimensionale arrays

Een 2-dimensionale array is in C een 1-dimensionale array waarbij elk element zelf een 1-dimensionale array is. De definitie

```
int  a[3][5];
```

reserveert in het werkgeheugen een zone van 15 opeenvolgende integers. Deze zone wordt gezien als drie rijen waarbij elke rij bestaat uit vijf elementen. Het element op de i-de rij en de j-de kolom kan in array notatie, in pointer notatie of in een gemengde notatie weergegeven worden:

```
a[i][j]
*(a[i]+j)
(*(a+i))[j]
*(*(a+i)+j)
```

Hierin kan `a[i]` of `*(a+i)` gezien worden als een pointer naar de i-de rij. Hiervan moet dan het j-de element genomen worden.

6.4 Begrippen

- De pointer-operatoren `&` en `*`.
- Parameteroverdracht: call by value en call by address.
- Rekenen met pointers.

7 Interne voorstelling, types en conversies

7.1 Voorstelling van gegevens: bits en bytes

Een (digitale) computer werkt steeds met binaire grootheden. De reden daartoe is het feit dat de computer bestaat uit een aantal onderdelen die slechts twee toestanden kunnen aannemen (bistabiele elementen; vb: flip-flop-schakelingen). De ene toestand wordt conventioneel voorgesteld als 1 en de andere als 0. Gewoonlijk wordt dit aangeduid door de technische term BIT (BInary digiT = binair cijfer). Een *bit* is dus de kleinst mogelijke informatie-eenheid in een computersysteem. Bits dienen als bouwstenen voor grotere betekenisvolle informatiehoeveelheden. Elke vorm van informatie wordt gemanipuleerd in de vorm van een combinatie van enen en nullen.
Het geheugen wordt ingedeeld in een aantal gebieden van gelijke grootte. Een gebied van 8 bits noemt men een *byte*. (Een *nible* is een groep van 4 bits). Een *(computer)woord* bestaat, afhankelijk van de computer architectuur, uit een 2 bytes (16 bits), 4 bytes (32 bits) of 8 bytes (64 bits).
Het geheugen kan men voorstellen als een opeenvolging van vakken die elk uit 8 bits of één byte bestaan. Iedere geheugenplaats heeft een welbepaalde unieke identificatie (volgnummer) die men adres noemt. In het geheugen moeten programma en data gestockeerd worden. De grootte van een geheugen wordt meestal aangeduid in Kb, Mb of Gb.

$$\begin{array}{rcrcrcr} 1\,Kb & = & 1 \text{ kilobyte} & = & 2^{10} \text{ bytes} & = & 1024 \text{ bytes} \\ 1\,Mb & = & 1 \text{ megabyte} & = & 2^{20} \text{ bytes} & = & 1048576 \text{ bytes} \\ 1\,Gb & = & 1 \text{ gigabyte} & = & 2^{30} \text{ bytes} & = & 1073741824 \text{ bytes} \end{array}$$

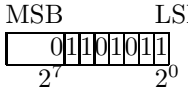
7.2 Binaire en andere talstelsels

In het ons vertrouwde *decimale* talstelsel wordt als grondtal de waarde 10 genomen. Afhankelijk van de positie van een cijfer in een getal, wordt de waarde ervan bepaald.

$$1063 = 1 \times 10^3 + 0 \times 10^2 + 6 \times 10^1 + 3 \times 10^0$$

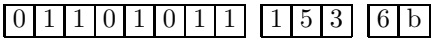
In dit talstelsel zijn tien verschillende cijfers nodig.
In het binaire talstelsel is het grondtal 2 en zijn er maar twee cijfers nodig: de 0 en de 1 (bit).

$$01101011 = 0 \times 2^7 + 1 \times 2^6 + 1 \times 2^5 + 0 \times 2^4 + 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0$$



getal 107 teken k

Een getal in binaire voorstelling is meestal erg lang. Om een wat handiger voorstelling te hebben, kan men gebruik maken van het octale talstelsel met grondtal 8 of het hexadecimale talstelsel met grondtal 16. De omzetting van een binair getal naar een octaal getal kan vrij eenvoudig gebeuren door vanaf de LSB (least significant bit) de bits samen te nemen in groepjes van drie. Bij een hexadecimaal getal zijn dit groepjes van vier bits.



In het octale talstelsel gebruikt men de cijfers van 0 tot 7. Bij het hexadecimale talstelsel moeten naast de 10 decimale cijfers nog 6 extra symbolen ingevoerd worden. Men gebruikt hiervoor de letters van 'a' ('A') tot en met 'f' ('F').

$$\begin{array}{rcl} 0153 & = & 1 \times 8^2 + 5 \times 8^1 + 3 \times 8^0 \\ 0x6b & = & 6 \times 16^1 + 11 \times 16^0 \end{array}$$

Opgave. Schrijf een functie die een positief geheel getal omzet in binaire voorstelling. Schrijf een functie die de binaire voorstelling van een positief geheel getal omzet in decimale vorm.

```
int d2b(int n, int bits[])
{
    int i=0, l=0;
    int a[LEN];

    while ( n > 0 )
    {
        if ( n % 2 )
            a[l++] = 1;
        else
            a[l++] = 0;
        n /= 2;
    }
    for (i=0; i<l; i++)
        bits[i] = a[l-1-i];
    return l;
}
```

De oproep `d2b(75,bits)` heeft als resultaat **7** met in bits:

1	0	0	1	0	1	1

```
int b2d(int l, int bits[])
{
    int r=bits[0];
    int i;

    for (i=1; i<l; i++)
        r = r * 2 + bits[i];
    return r;
}
```

De oproep `b2d(7,bits)` met in bits

1	0	0	1	0	1	1

heeft als resultaat **75**.

Een overzicht van enkele getallen in de verschillende talstelsels:

binair	octaal	decimaal	hexadecimaal
0 0000	0	0	0
0 0001	1	1	1
0 0010	2	2	2
0 0011	3	3	3
0 0100	4	4	4
0 0101	5	5	5
0 0110	6	6	6
0 0111	7	7	7
0 1000	10	8	8
0 1001	11	9	9
0 1010	12	10	a
0 1011	13	11	b
0 1100	14	12	c
0 1101	15	13	d
0 1110	16	14	e
0 1111	17	15	f
1 0000	20	16	10

Probleem. Een bit kan maar twee mogelijke waarden hebben: 0 of 1. Hoe moet het minteken van een geheel getal of de komma van een reëel getal voorgesteld worden?

7.3 Gehele getallen

Een geheel getal kan in twee bytes (16 bits) gestockeerd worden. Eén bit is voorbehouden als tekenbit. Het grootste positieve getal is dus $2^{15} - 1 = 32767$.

getal	voorstelling
0	0000 0000 0000 0000
1	0000 0000 0000 0001
2	0000 0000 0000 0010
3	0000 0000 0000 0011
4	0000 0000 0000 0100
...	...
32766	0111 1111 1111 1110
32767	0111 1111 1111 1111

Negatieve gehele getallen worden voorgesteld in de twee-complement-vorm. De hoogste bit wordt gebruikt als tekenbit en stelt de kleinst mogelijke negatieve waarde voor. Wanneer 16 bits gebruikt worden is dit gelijk aan $-2^{15} = -32768$. De waarde van de binaire voorstelling van een andere negatief getal wordt berekend door de absolute waarde van het 15 bits-getal bij -32768 op te tellen.

getal	voorstelling	verklaring
-1	1111 1111 1111 1111	-32768 + 32767 = -1
-2	1111 1111 1111 1110	-32768 + 32766 = -2
-3	1111 1111 1111 1101	-32768 + 32765 = -3
-4	1111 1111 1111 1100	-32768 + 32764 = -4
...	...	...
-32767	1000 0000 0000 0001	-32768 + 1 = -32767
-32768	1000 0000 0000 0000	-32768 + 0 = -32768

Een gemakkelijke methode om de 2-complement voorstelling te berekenen is de volgende: bepaal de binaire voorstelling van de absolute waarde van het getal; complementeer elke bit en tel bij dit complement 1 op; het resultaat is de 2-complement voorstelling.
Een voorbeeld: de voorstelling van -2

binaire voorstelling van 2	0000 0000 0000 0010
complement	1111 1111 1111 1101
1 bijtellen	0000 0000 0000 0001
resultaat	1111 1111 1111 1110

Om de decimale waarde van een 2-complement voorstelling te berekenen, kan dezelfde methode gebruikt worden:

decimale voorstelling van	1111 1111 1111 0001	
complement	0000 0000 0000 1110	
1 bijtellen	0000 0000 0000 0001	
resultaat	0000 0000 0000 1111	dus **-15**

7.4 Reële getallen

Een reëel getal kan (benaderend) voorgesteld worden in een wetenschappelijke notatie in de vorm

$$0.5 \times 10^{6} \qquad -0.999 \times 10^{16} \qquad 0.642 \times 10^{-23}$$

Het gedeelte voor het $\times$ teken wordt *mantisse* (of significand) genoemd, daarna komt de *basis* gevolgd door de *exponent*. Normaal wordt als basis 10 gebruikt.
In deze floating point vorm kan dus elk reëel getal geschreven worden met behulp van twee gehele getallen.

$$\begin{array}{rcll} 123.456 & = & 0.123456 \times 10^{3} & 123456 \text{ en } 3 \\ 123456 & = & 0.123456 \times 10^{6} & 123456 \text{ en } 6 \\ 0.000123456 & = & 0.123456 \times 10^{-3} & 123456 \text{ en } -3 \end{array}$$

Op deze manier kan een reëel getal dus op verschillende manieren voorgesteld worden. Om tot een uniforme manier te komen, bestaan er standaards. Tegenwoordig wordt de ANSI/IEEE 754-1984 norm gebruikt. Er wordt met een basis 2 gewerkt en door de mantisse te beperken tot het interval $[1.0, 2.0[$ is ook de exponent uniek gedefinieerd.

S	exponent (E)	mantisse (M)

of $(-1)^S \times M \times 2^E$ bijv. $10.0 = (-1)^0 \times 1.25 \times 2^3$

De mantisse M is steeds van de vorm $1.f$ en dus hoeft alleen het fractiegedeelte f gestockeerd te worden. Bijvoorbeeld

$$0.25 = 0 \times 2^{-1} + 1 \times 2^{-2} + 0 \times 2^{-3} + 0 \times 2^{-4} + \ldots$$

Er worden slechts een beperkt aantal bytes voorzien om de mantisse en de exponent te stockeren. Met zo'n eindige binaire string kan niet om het even welk decimaal getal exact voorgesteld worden. Meestal is in deze binaire string de eerste bit het teken van de mantisse, de volgende bits geven zowel de tekenbit als de waardebits van de exponent en de resterende bits zijn de waardebits van het fractiegedeelte (f) van de mantisse.
Beide delen moeten ook negatieve getallen kunnen voorstellen. Voor de mantisse wordt met een tekenbit (S) gewerkt. Voor de exponent (met 8 bits) wordt echter gebruik gemaakt *excess 127 mode*.

bit-patroon	exponent	verklaring
0000 0000	-127	0-127
...	...	...
0111 1110	-1	126-127
0111 1111	0	127-127
1000 0000	1	128-127
1000 0001	2	129-127
1000 0010	3	130-127
...	...	...
1111 1111	128	255-127

Bij het type `float` worden 4 bytes voorzien: 1 tekenbit, 8 bits voor de exponent en 23 bits voor de fractie van de mantisse. De interne voorstelling van het reële getal 10.0 is dus:

0	100 0001 0	010 0000 0000 0000 0000 0000

Een kortere schrijfwijze mbv. hexadecimale voorstelling: $0x41200000$.

7.5 Elementaire types

De C-taal kent de volgende elementaire types:

char	1 byte	$-2^7 ... 2^7 - 1$	127
short int	2 bytes	$-2^{15} ..., -1, 0, 1, ... 2^{15} - 1$	32767
int			
long int	4 bytes	$-2^{31} ..., -1, 0, 1, ... 2^{31} - 1$	2147483647
float	4 bytes		
double	8 bytes		

Het aantal bytes dat voor een `int` gebruikt wordt, is afhankelijk van de processor: 2 of 4 bytes. Om *overdraagbare* programma's te schrijven is het dus beter te werken met `short` en `long`.
Bij deze binaire voorstellingen wordt telkens één bit gebruikt als *tekenbit*. Bij sommige toepassingen wordt alleen met positieve getallen gewerkt. In die gevallen is geen tekenbit nodig en kunnen grotere getallen voorgesteld worden.

unsigned char	1 byte	$0, 1, 2, ...2^8 - 1$	255
unsigned short int	2 bytes	$0, 1, 2, ...2^{16} - 1$	65535
unsigned int			
unsigned long int	4 bytes	$0, 1, 2, ...2^{32} - 1$	4294967295

```
1   /*
     *  types.c : en illustratie van sizeof
3    */
    #include <stdio.h>
5
    void main(void)
7   {
        int           i, j;
9       float         x, f;
        double        ff;
11      short int     si;
        long int      li;
13      unsigned short int  usi;
        unsigned long int   uli;
15
        i = 2;
17      j = 5/3;
        x = 5/3;
19      f = 5.0/3;
        ff = 5.0/3;
21      si = 40000;              usi = 40000;
        li = 3000000000;         uli = 3000000000;
23
        printf("i  = %d\n", i);
25      printf("j  = %d\n", j);
        printf("x  = %20.17f\n", x);
27      printf("f  = %20.17f\n", f);
        printf("ff = %20.17f\n", ff);
29      printf("si = %12d     usi = %12u\n", si, usi);
        printf("li = %12d     uli = %12u\n", li, uli);
31      printf("\n");

33      printf("Een short neemt %d bytes in beslag\n", sizeof(short));
        printf("Een int neemt %d bytes in beslag\n", sizeof(int));
35      printf("Een long neemt %d bytes in beslag\n", sizeof(long));
        printf("Een float neemt %d bytes in beslag\n", sizeof(float));
37      printf("Een double neemt %d bytes in beslag\n", sizeof(double));
        printf("Een char neemt %d bytes in beslag\n", sizeof(char));
39  }
```

De output van dit programma is:

```
i  = 2
j  = 1
x  =  1.00000000000000000
f  =  1.66666662693023680
ff =  1.66666666666666670
si =       -25536    usi =        40000
li =  -1294967296    uli =   3000000000
```

```
Een short neemt 2 bytes in beslag
Een int neemt 4 bytes in beslag
Een long neemt 4 bytes in beslag
Een float neemt 4 bytes in beslag
Een double neemt 8 bytes in beslag
Een char neemt 1 bytes in beslag
```

Merk op dat de deling van twee gehele getallen (variabele `j`) het geheeltallig quotiënt oplevert. Noteer ook het verschil tussen *signed* en *unsigned* getallen. In een declaratie wordt `unsigned` expliciet vermeld. De default waarde is signed; dit kan eventueel vermeld worden maar hoeft niet en wordt dus niet veel gedaan.

```
unsigned int  aantal;
signed int    waarde;
```

Het aantal bytes dat een bepaald data type gebruikt in het werkgeheugen kan opgevraagd worden met de `sizeof` operator. De operand van deze operator is de naam van het type tussen haakjes. Daarnaast is het ook mogelijk deze operator toe te passen op een variabele, deze hoeft niet tussen haakjes geplaatst te worden.

```
unsigned    len;
double      ff;

len = sizeof(double);
len = sizeof ff;
len = sizeof(ff);
```

7.6 Nauwkeurigheid

Gehele getallen. Met gehele getallen (`int` en aanverwanten) kan exact gerekend worden. Omdat slechts een beperkt aantal bytes voorzien worden om een geheel getal te stockeren kan er *overflow* of *underflow* optreden.

```
int tegroot = 1234567890123456 ;
int teklein = -9876543210987654;
```

Overflow treedt op wanneer het getal te groot en positief is om het te stockeren in de variabele. In het bovenstaande voorbeeld zal de variabele `tegroot` niet de gespecificeerde waarde hebben. Analoog treedt underflow op wanneer het getal te groot en negatief is om het te stockeren in de variabele.
Bij het rekenen met gehele getallen kan dus overflow ontstaan. De meeste programmeertalen signaleren dit niet maar de resultaten zijn fout! Op processor-niveau kan het via de overflow-vlag gecontroleerd worden.

```
short a = 20000 ;                    0100 1110 0010 0000
short b = 20000 ;                    0100 1110 0010 0000
       de som a + b :                1001 1100 0100 0000
```

Reële getallen. Aangezien de mantisse uit een beperkt aantal bits bestaat, zijn er ook slechts een beperkt aantal beduidende cijfers te stockeren. Dit impliceert eventueel verlies van nauwkeurigheid.
Hoeveel bits voorzien worden voor exponent en mantisse is systeemafhanelijk. Een mogelijke verdeling is de volgende. Van de 4 bytes die voor een `float` voorzien zijn, wordt één byte gebruikt voor de *exponent* en de overige drie voor de *mantisse*. Met deze drie bytes kan men een nauwkeurigheid bereiken van 6 decimale cijfers. De exponent in basis 10 ligt in het interval $[-38, 38]$.

Wanneer deze precisie niet volstaat, kan het `double` type gebruikt worden. De 64 bits worden opgedeeld in een 11 bits voor de exponent en de rest voor de mantisse. Dit geeft een nauwkeurigheid van tenminste 15 decimale cijfers met een bereik voor de exponent (basis 10) gelijk aan $[-308, 308]$.
Op het eerste zicht lijkt de floating point voorstelling zeer nauwkeurig. Maar sommige numerieke berekeningen zijn inherent *numeriek instabiel* zodat zeer kleine fouten zeer snel kunnen groeien.

Opgave. Schrijf een programma dat een vierkante matrix a genereert met $a_{ij} = 1/(i + j - 1)$ voor $1 \leq i \leq n$ en $1 \leq j \leq n$ (dit is een Hilbert matrix van orde n). Laat het programma voor verschillende n waarden de determinant berekenen en afdrukken.

$$\begin{bmatrix} 1 & \frac{1}{2} & \frac{1}{3} & \frac{1}{4} \\ \frac{1}{2} & \frac{1}{3} & \frac{1}{4} & \frac{1}{5} \\ \frac{1}{3} & \frac{1}{4} & \frac{1}{5} & \frac{1}{6} \\ \frac{1}{4} & \frac{1}{5} & \frac{1}{6} & \frac{1}{7} \end{bmatrix}$$

```
/*
 *    determ.c : berekenen determinant met a[i][j] = 1/(i+j-1)
 */
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#define MAXDIM 9
#define EPS    1.0E-8

typedef float reeel;
void genmatrix(reeel x[][MAXDIM], int n);
void drukmatrix(reeel x[][MAXDIM], int n);
reeel determinant(reeel x[][MAXDIM], int n);

void main(void)
{
    reeel a[MAXDIM][MAXDIM];
    int n;

    for (n=2; n<MAXDIM; n++)
    {
        genmatrix(a,n);
        printf("De determinant van de matrix met n=%d\n", n);
        drukmatrix(a,n);
        printf("is %e\n", determinant(a,n));
    }
}

void genmatrix(reeel x[][MAXDIM], int n)
{
    int i, j;

    for (i=0; i<n; i++)
    {
        for(j=0; j<n; j++)
        {
            x[i][j]=1.0/(i+j+1.0);
        }
    }
```

```
40  }

42  void drukmatrix(reeel x[][MAXDIM], int n)
    {
44      int i, j;

46      for (i=0; i<n; i++)
        {
48          for(j=0; j<n; j++)
            {
50              printf("%8.5f", x[i][j]);
            }
52          printf("\n");
        }
54      printf("\n");
    }
56
    reeel determinant(reeel x[][MAXDIM], int n)
58  {
        reeel det, faktor, temp;
60      int   i, r, k, rmax;
        int   wissel = 0;
62
        /* eliminatie van Gauss */
64      for (i=0; i<n-1; i++)
        {
66          rmax = i;
            for (r=i+1; r<n; r++)
68          {
                if ( fabs(x[r][i]) > fabs(x[rmax][i]) )
70                  rmax = r;
            }
72          if ( rmax > i )
            {
74              wissel++;
                for(k=i; k<n; k++)
76              {
                    temp = x[i][k];
78                  x[i][k] = x[rmax][k];
                    x[rmax][k] = temp;
80              }
            }
82          for (r=i+1; r<n; r++)
            {
84              if (fabs(x[i][i]) > EPS)
                {
86                  faktor = x[r][i]/x[i][i];
                }
88              else
                {
90                  printf("PANIEK\n\n");
                    printf("n=␣%d␣␣i=␣%d␣␣r=␣%d␣␣\n",n,i,r);
92                  exit(1);
                }
```

```
94                  for(k=i; k<n; k++)
                    {
96                      x[r][k]=x[r][k]-faktor*x[i][k];
                    }
98              }
        }
100     /* determinant is product van de hoofddiagonaal */
        if ( wissel % 2 )
102         det = -1.0;
        else
104         det = 1.0;
        for (i=0; i<n; i++)
106     {
            det = det*x[i][i];
108     }
        return det;
110 }
```

Het resultaat voor verschillende n waarden wanneer met `float` (*enkelvoudige precisie*) en met `double` (*dubbele precisie*) gerekend wordt:

n	float	double	exact
2	8.333334e-02	8.333333e-02	$12^{-1} = 8.333333\text{e-}02$
3	4.629642e-04	4.629630e-04	$2160^{-1} = 4.629630\text{e-}04$
4	1.653463e-07	1.653439e-07	$6048000^{-1} = 1.653439\text{e-}07$
5	3.751022e-12	3.749295e-12	$266716800000^{-1} = 3.749295\text{e-}12$
6	5.456915e-18	5.367300e-18	
7	7.336620e-25	4.835803e-25	
8	2.532158e-32	2.737050e-33	

Merk op dat in het programma slechts op één plaats een aanpassing moet gebeuren voor de berekening in enkelvoudige of dubbele precisie:

```
typedef float  reeel;
typedef double reeel;
```

Met behulp van `typedef` kunnen nieuwe types gedefinieerd worden. Dit wordt in een volgend hoofdstuk uitgebreider behandeld.

7.7 Type conversie

Als operanden in een expressie van verschillend type zijn, dan worden deze operands naar een gemeenschappelijk type geconverteerd.

operand 1	operand 2	gemeenschappelijk type
char	int	int
short	int	int
int	long	long
int	float	float
int	double	double
float	double	double

Er wordt dus telkens impliciet geconverteerd naar het “grotere” type. Daardoor zal zo weinig mogelijk informatie omwille van afkapping verloren gaan.
Bij een toekenningsexpressie kan echter een conversie van een “groter” type naar een “kleiner” type nodig zijn.

```
int     i, res;
float   f;

i = 31;          /*  geen conversie  */
f = 4;           /*  impliciete conversie van 4 naar 4.0  */
res = i / f;     /*  deling geeft 7.75; en bij de toekenning: 7  */
```

In de tweede toekenning gebeurt een impliciete conversie van de integer constante 4 naar een float (4.0). Bij de deling wordt de inhoud van de variabele `i` omgezet naar een float, dan wordt de deling uitgevoerd met als resultaat `7.75`. Dit float resultaat moet nu toegekend worden aan een integer variabele. Omdat het type van de variabele `res` niet kan aangepast worden (deze variabele is als `int` gedeclareerd), moet het resultaat omgevormd worden naar een integer. Dit gebeurt door afkappen, er is dus informatie verlies.

linkerkant	rechterkant	opmerking
char	int	
short	int	
int	long	
int	float	
float	double	afronding: alleen precisie verlies

7.8 Type casting

Soms is het nodig expliciet een conversie op te leggen. Dit kan door het vermelden van het gewenste type tussen haakjes voor de operand die moet geconverteerd worden. Een type tussen haakjes voor een operand kan gezien worden als een unaire operator: de *cast-operator*.

```
/*
 * tcast.c : typecasting
 */
#include <stdio.h>

void main(void)
{
    int i,j;
    float x,y;

    i = 1;
    j = 2;
    x = i/j;
    y = (float)i/j;
    printf("x = %8.4f        y = %8.4f\n", x, y);
}
```

Het resultaat van dit programma:

```
x =   0.0000        y =   0.5000
```

7.9 Begrippen

- Talstelsels.
- Interne voorstelling van gehele en reële getallen.
- Elementaire types.
- Conversie operaties: impliciet en expliciet.
- Expliciete conversie: de type cast-operator.

8 Niet-numerieke data-types

Een computer kan naast numerieke gegevens ook niet-numerieke informatie verwerken. Tot nu toe is dat alleen gebleken bij de `printf`. Het eerste argument is een opeenvolging van letters tussen dubbel quotes (").

8.1 Het type char

Het elementaire type is een `char`, een *karakter*, ook *teken* of *symbool* genoemd. Declaratie, definitie en initialisatie van een variabele:

```
char  c;

c = 'a';              /* c = a;  dubbelzinnig */
```

Door de toekenning heeft de variabele `c` de karakter waarde van de letter `a` gekregen. Door middel van de enkel quotes (') wordt aangegeven dat het over een karakter-constante gaat. Zo'n constante of variabele neemt in het werkgeheugen één byte in beslag. In één byte (acht bits) kunnen $2^8 = 256$ mogelijke waarden gestopt worden. Dit is ruim voldoende voor alle letters van het alfabet en nog heel wat andere tekens.
Intern wordt dus een karakter voorgesteld als een rij van acht bits, bijvoorbeeld de letter 'a':

```
01100001          0110 0001      0x61
```

Deze waarde kan ook als een getal geïnterpreteerd worden, namelijk (decimaal) de waarde 97. De programmeertaal C laat toe dat de inhoud van karakter variabelen ook als gehele getallen beschouwd kunnen worden. Bij interpretatie van de inhoud van een byte als een geheel getal, geldt natuurlijk dat de hoogste bit het teken van het getal aangeeft. Het interval van mogelijke waarden is dus $[-128, 127]$.
Oorspronkelijk werd beslist dat 127 tekens voldoende waren. Er werd een tabel opgesteld die voor elk teken de bijhorende numerieke waarde weergeeft en dus de interne (binaire) voorstelling definieert. Zoals gewoonlijk zijn er verschillende tabellen opgesteld geweest. Het meest gebruikte schema is ASCII (American Standard Code for Information Interchange).
Omdat de waarde van een karakter variabele dus ook als geheel getal beschouwd kan worden, zijn hierop rekenkundige operaties, zoals optelling en aftrekking, en vergelijkingsoperatoren, zoals `<` en `==`, mogelijk.

Opgave. Maak een tabel met karakterwaarden en bijhorende hexidecimale codering. Laat de tabel beginnen met de waarde `' '` (spatie) en eindigen met `'~'`.

```
1  /*
    * prascii.c : tabel met afdrukbare karakters
3   */
   #include <stdio.h>
5  void main(void)
   {
7      char c;

9      for (c=' '; c<='~'; c++)
       {
11         printf ("  %c  %2x %c", c, c, '|' );
           if ( (c+1)%8 == 0 )
13             printf("\n");
       }
15     printf("\n");
   }
```

Het resultaat:

```
    20 |  !  21 |  "  22 |  #  23 |  $  24 |  %  25 |  &  26 |  '  27 |
 (  28 |  )  29 |  *  2a |  +  2b |  ,  2c |  -  2d |  .  2e |  /  2f |
 0  30 |  1  31 |  2  32 |  3  33 |  4  34 |  5  35 |  6  36 |  7  37 |
 8  38 |  9  39 |  :  3a |  ;  3b |  <  3c |  =  3d |  >  3e |  ?  3f |
 @  40 |  A  41 |  B  42 |  C  43 |  D  44 |  E  45 |  F  46 |  G  47 |
 H  48 |  I  49 |  J  4a |  K  4b |  L  4c |  M  4d |  N  4e |  O  4f |
 P  50 |  Q  51 |  R  52 |  S  53 |  T  54 |  U  55 |  V  56 |  W  57 |
 X  58 |  Y  59 |  Z  5a |  [  5b |  \  5c |  ]  5d |  ^  5e |  _  5f |
 `  60 |  a  61 |  b  62 |  c  63 |  d  64 |  e  65 |  f  66 |  g  67 |
 h  68 |  i  69 |  j  6a |  k  6b |  l  6c |  m  6d |  n  6e |  o  6f |
 p  70 |  q  71 |  r  72 |  s  73 |  t  74 |  u  75 |  v  76 |  w  77 |
 x  78 |  y  79 |  z  7a |  {  7b |  |  7c |  }  7d |  ~  7e |
```

Merk op dat de codes voor de letters van het alfabet van klein naar groot gaan. Deze eigenschap van de ASCII tabel wordt gebruikt bij het sorteren van namen. Hoofdletters zijn alfabetisch "kleiner" dan kleine letters.
De tabel start bij een numerieke waarde 0x20 of 32. De getallen tussen 0 en 31 worden gebruikt voor een aantal speciale tekens, die een specifieke betekenis binnen de informatica (+communicatie) wereld gekregen hebben. Een paar voorbeelden:

hex-waarde	dec-waarde	C-notatie	betekenis
0	0	`'\0'`	het null karakter
8	8	`'\b'`	backspace
9	9	`'\t'`	horizontal tab
a	10	`'\n'`	line feed, newline
c	12	`'\f'`	form feed
d	13	`'\r'`	carriage return

Voor de C-notatie wordt een \ gebruikt, gevolgd door een teken. Om het backslash teken zelf voor te stellen wordt `'\\'` gebruikt. Analoog wordt de quote in C genoteerd als `'\''`.
Uit bovenstaande tabellen blijkt dat er een verschil is tussen de karakterwaarde '0' en het numeriek cijfer 0. De karakterwaarde '0' heeft een numerieke tegenwaarde gelijk aan 0x30.

Opgave. Maak een tabel met karakter- of tekenwaarden van '0' tot '9' en bereken de bijhorende decimale cijferwaarde.

```
/*
 * c2d.c : omzetting van karakterwaarde naar cijferwaarde
 */
#include <stdio.h>
void main(void)
{
    char c;
    int  i;

    for (c='0'; c<='9'; c++)
    {
        i = c - '0';
        printf ("%c : (%d of 0x%.2x)  %d\n",  c, c, c, i);
    }
}
```

In `<ctype.h>` vindt men een aantal macro's om het soort van karakter te testen. In sommige C-implementaties wordt gebruikt gemaakt van functies:

int isalnum(int ch)	alfanumeriek
int isalpha(int ch)	alfabetisch
int isdigit(int ch)	numeriek
int isupper(int ch)	hoofdletter
int islower(int ch)	kleine letter
int isprint(int ch)	afdrukbaar-karakter
int isspace(int ch)	spatie-karakter
int tolower(int ch)	omzetting naar kleine letter
int toupper(int ch)	omzetting naar hoofdletter

Opgave. Schrijf een programma dat een tekst symbool per symbool leest (niet stockeert) en het aantal symbolen telt alsook het aantal letters 'e' of 'E' in de tekst. De tekst wordt afgesloten met Ctrl-Z Enter. Ctrl-Z is het EOF (End Of File) symbool bij klassieke PC-besturingssystemen.

Hulp. Gebruik de functie `getchar()` die één symbool van het toetsenbord leest en de ASCII-waarde van dat symbool als functiewaarde heeft. Indien Ctrl-Z ingegeven wordt, dan geeft getchar() EOF als functiewaarde. EOF is in `<stdio.h>` gedefinieerd als -1.

```
/*
 *    symbtel.c : symbolen tellen
 */
#include <stdio.h>

void main(void)
{
    int c;
    int aantal_symbolen = 0;
    int aantal_e = 0;

    printf("Geef een tekst, eindig met <ctrl-Z><Enter>\n");
    c = getchar();
    while (c != EOF)
    {
        aantal_symbolen++;
        if (c == 'e' || c == 'E')  /* of (c == 0x65 || c == 0x45) */
            aantal_e++;
        c = getchar();
    }
    printf("\n\n");
    printf("Er zijn %d symbolen gelezen, ", aantal_symbolen);
    printf("waarvan %d \'e\' of \'E\'\n", aantal_e);
}
```

Opgave. Schrijf een programma dat een tekst symbool per symbool leest (niet stockeert) en het aantal woorden telt. De tekst wordt afgesloten met het EOF teken.

```
/*
 *   woorden.c : tellen van het aantal woorden in een tekst
 */
#include <stdio.h>

void main(void)
{
    int c;
```

```
    int v = ' ';
10  int woorden = 0;

12  printf("Geef een tekst , eindig met <Ctrl-Z><Enter>\n\n");
    while ((c=getchar()) != EOF)
14  {
        if ( !isalpha(c) && isalpha(v) )
16          ++woorden;
        v = c;
18  }
    if ( c == EOF && isalpha(v) )
20      ++woorden;
    printf("\nHet aantal woorden in de tekst is %d\n", woorden);
22 }
```

Omdat de toekenningsoperator (`=`) een lagere prioriteit heeft dan de vergelijkingsoperator (`!=`), staan er haakjes rond `c=getchar()`.

8.2 Strings

8.2.1 Constanten

Enkelvoudige karakterconstanten worden aangeduid met behulp van enkel quotes, bijvoorbeeld `'a'`, `'5'`, `'\n'`.
Een rij van karakters tussen dubbel quotes is een string:

```
"hoeveel getallen"
"a5\n"
"a"
"naam: Jos\nwoonplaats: Duffel\n"
```

In een stringconstante zijn een aantal karakterconstanten *samengenomen*. Het type is dus een *array van karakters*. De interne voorstelling is een opeenvolging van een aantal ASCII codes. Om het einde van de rij codes aan te geven wordt het karakter `'\0'` gebruikt. Dus `"a5"` wordt intern voorgesteld als | 0x61 | 0x35 | 0x00 |.

8.2.2 Variabelen

Om met strings bewerkingen uit te voeren, zijn variabelen nodig die geconstrueerd worden met behulp van het `array` type, d.i. een samengesteld type van karaktervariabelen.

```
char woord[20];
```

De gedeclareerde variabele `woord` is een array waarin maximaal 20 karakters kunnen gestopt worden. In het werkgeheugen worden dus 20 bytes voorzien voor deze variabele. Een array van karakters wordt dikwijls aangeduid met de naam *string*.
De initialisatie moet zoals bij elke variabele van het array-type gebeuren, namelijk element per element.

```
woord[0] = 's';
woord[1] = 'a';
woord[2] = 'p';
woord[3] = '\0';        /* of (char)0   */
```

Handiger zou zijn: `woord = "sap";` maar dit is niet mogelijk. De reden hiervoor is dat de naam `woord` geen atomaire variabele aanduidt maar iets van het type `array`, en dus geen lvalue heeft. Dit wil zeggen dat bij compilatie de variabele `woord` omgezet wordt naar een constant adres,

wijzend naar het eerste element van de array. Door de toekenning zou dit constant adres moeten wijzigen en dit kan niet.
Nochtans is het mogelijk string-variabelen te initialiseren bij de definitie-deklaratie:

```
char woord[20] = "sap";
```

Tijdens compilatie zal een array van 20 bytes voorzien worden. In de eerste drie elementen worden letters 's', 'a' en 'p' ingevuld. Het vierde element krijgt de waarde `(char)0`. De rest wordt ook opgevuld met de waarden `(char)0`.
String variabelen op zich kunnen alleen gebruikt worden als argumenten bij functies. Net zoals bij arrays van ints of floats, krijgt de formele parameter in de opgeroepen routine de waarde van het adres dat wijst naar het eerste element van de string. Wanneer via deze formele parameter een element in de string gewijzigd wordt, zal deze wijziging na het verlaten van de functie behouden blijven.

Opgave. Lees een string van toetsenbord. Schrijf een functie die alle alfanumerieke karakters omzet naar hoofdletters. Schrijf de originele en de gewijzigde string uit.

```
1  /*
    * k2g.c : omzetting kleine naar grote letters
3   */
   #include <stdio.h>
5  #include <ctype.h>
   void doen(char in[], char uit[]);
7
   void main(void)
9  {
       char   in[32];
11     char   uit[32];
       int    ch;
13
       scanf("%s", in);
15     ch = getchar();
       doen(in, uit);
17     printf("ch = %d\n%s : %s\n", ch, in, uit);
   }
19
   void doen(char in[], char uit[])
21 {
       int i;
23
       i = 0;
25     while ( (uit[i] = in[i]) != '\0' )
       {
27         if ( islower(uit[i]) )
               uit[i] += 'A' - 'a';
29         i++;
       }
31 }
```

Omdat `in` de naam van een array is, moet bij `scanf` geen `&` voor de variabele geschreven worden. De naam `in` is zelf al een verwijzing naar de geheugenplaatsen waar de string moet geplaatst worden.
Merk op dat nadat `scanf` de string ingelezen heeft, er nog één karakter kan ingelezen worden, namelijk de ENTER toets.
In plaats van `scanf("%s", in); ch=getchar();` is ook `scanf("%s%c", in, &ch);` mogelijk.

8.2.3 Bewerkingen door middel van functies

Om toch op een efficiënte manier met strings te kunnen werken zijn een heleboel stringfuncties opgenomen in de standaard C-bibliotheek. Een aantal voorbeelden:

int strlen(char *s)	lengte van de string `s`
int strcmp(char *s, char *t)	vergelijking van string `s` met `t`
int strncmp(char *s, char *t, int n)	vergelijking (maximaal n tekens)
char *strcpy(char *s, char *t)	copiëren van string `t` in `s`
char *strncpy(char *s, char *t, int n)	copiëren (maximaal n tekens)
char *strcat(char *s, char *t)	toevoegen van string `t` aan `s`
char *strncat(char *s, char *t, int n)	toevoegen (maximaal n tekens)

Om deze te gebruiken, moet een include van `<string.h>` gedaan worden.

Opgave. Lees twee strings van toetsenbord. Schrijf deze twee strings in alfabetische volgorde uit.

```
1  /*
    * alfa2.c : alfabetisch afdrukken van twee strings
3   */
   #include <stdio.h>
5  #include <string.h>

7  void main(void)
   {
9      char naam1[32];
       char naam2[32];
11     int vgl;

13     printf("Tik een naam in: ");
       scanf("%s%*c", naam1);
15     printf("Tik een tweede naam in: ");
       scanf("%s%*c", naam2);
17     vgl = strcmp(naam1,naam2);
       if ( vgl == 0 )
19     {
           printf("%s is gelijk aan %s\n", naam1, naam2);
21     }
       else
23     {
           if ( vgl < 0 )
25             printf("%s %s\n", naam1, naam2);
           else
27             printf("%s %s\n", naam2, naam1);
       }
29 }
```

Een eenvoudige implementatie van de functies `strlen`, `strcpy`, `strcat` en `strcmp`.
Bepalen van de lengte van de string `s`:

```
int strlen (char s[])
{
    int i;

    i = 0;
```

```
    while ( s[i] != '\0')
        ++i;
    return i;
}
```

Copiëren van string `from` naar string `to`:

```
char *strcpy(char to[], char from[])
{
    int i;

    i = 0;
    while ( (to[i] = from[i]) != '\0')
        ++i;
    return to;
}
```

Aanplakken van string `from` achteraan aan string `to`:

```
char *strcat(char to[], char from[])
{
    int i, j;

    i = 0;
    while ( to[i] != '\0')
        ++i;
    j = 0;
    while ( (to[i] = from[j]) != '\0')
    {
        ++i;
        ++j;
    }
    return to;
}
```

Vergelijken van string `s` met string `t`, het resultaat is negatief als `s` alfabetisch voor `t` komt en postief in het andere geval; wanneer de twee strings alfabetisch gelijk zijn, is het resultaat 0:

```
int strcmp(char s[], char t[])
{
    int i;

    for (i = 0; s[i] == t[i]; i++)
    {
        if (s[i] == '\0')
            return 0;
    }
    return s[i] - t[i];
}
```

8.2.4 Pointers naar strings

In bovenstaande voorbeelden met stringfuncties werd de declaratie `char s[]` gebruikt. De constructie `char *s` heeft ongeveer dezelfde betekenis: `s` is een variabele met een grootte van vier bytes waarin een adres naar een karakter kan gestopt worden.

```
    char    c;
```

```
char    car[16];
char    *cp;

cp = &c;
*cp = 'd';
cp = &car[0];
*cp = 'e';
cp++;
*cp = 'f';
```

Na de declaratie van `cp`, kan deze variabele nog niet gebruikt worden. Er moet eerst een initialisatie gebeuren. (Dit is analoog aan `int i;  i = 8;`.) Na de eerste toekenning, heeft `cp` een waarde gekregen: de variabele wijst nu naar een karakter. In de volgende toekenning, kan dus de waarde van `cp` gebruikt worden.
Bij de derde toekenning, wordt in `cp` ook het adres naar een karakter ingevuld. Maar vermits op de volgende plaatsen ook karakters staan, wordt hier van een pointer naar een string gesproken. Door de toekenning `*cp = 'e'` wordt in `car[0]` een karakter ingevuld. Door het statement `cp++` uit te voeren, wijst de pointer `cp` naar het volgende element in de array, d.i. `car[1]`. Hierin wordt het teken 'f' ingevuld. Er is nog geen volwaardige string gevormd. Hiervoor moet op een van de volgende plaatsen ([2]..[15]) een null karakter (`'\0'`) ingevuld worden om het einde van de string aan te geven.
Een implementatie van de functies `strlen`, `strcmp` en `strcpy` met behulp van pointer variabelen.

```
/*
 * strlen: geeft de lengte
 *         van de string s
 */
  int strlen (char *s)
  {
        char *p = s;

        while( *p != '\0')
              p++;
        return p-s;
  }
```

```
/*
 *  strcpy:  copieert t naar s
 */
   char *strcpy(char *s, char *t)
   {
        char *r = s;

        while ((*s = *t) != '\0')
        {
             s++;
             t++;
        }
        return r;
   }
```

```
/*    strcmp: geeft <0 als s<t,   0 als s==t,   >0 als s>t   */
int strcmp(char *s, char *t)
{
    for (; *s == *t; s++, t++)
    {
        if (*s == '\0')
            return 0;
    }
    return *s - *t;
}
```

De formele pointer variabelen in deze functie definities krijgen een effectieve waarde (gaan naar één of andere string wijzen) wanneer de functie opgeroepen wordt. De waarde van de actuele argumenten wordt dan in de formele parameters gestopt. Vermits dit adressen zijn, zullen de wijzigingen die in de functies aan de strings gebeuren ook zichtbaar blijven nadat de functie uitgevoerd is.

8.3 Arrays van strings

Opgave. Schrijf een programma dat een naam inleest en daarna een lijst van namen inleest en deze in een array stockeert. De lijst wordt afgesloten met het EOF teken. Daarna wordt nagegaan of de eerst ingelezen naam in de elementen van de lijst voorkomt.

```
/*
 * linzoek.c : lineair zoeken in een lijst van namen
 */
#include <stdio.h>
#include <string.h>
#define MAX 50
#define LEN 20

int leeslijst(char a[][LEN]);
int zoek(char a[][LEN], int n, char b[]);

void main(void)
{
      char tabel[MAX][LEN];
      char naam[LEN];
      int  i, n;

      printf("Te zoeken naam : ");
      scanf("%s%*c", naam);
      n = leeslijst(tabel);
      i = zoek(tabel, n, naam);
      if ( i >= 0 )
            printf("De naam %s is gevonden op plaats %d\n", naam, i);
      else
            printf("De naam %s komt niet voor in de lijst \n", naam);
}

int leeslijst(char a[][LEN])
{
      int i = 0;

      while ( 1 )
      {
            printf("Naam  %2d : ", i);
            if ( scanf("%s%*c", a[i]) == EOF )
                  break;
            i++;
      }
      return i;
}

int zoek(char a[][LEN], int n, char b[])
{
      int i;

      for (i=0; i<n; i++)
      {
            if ( strncmp(a[i], b, LEN) == 0 )
                  return i;
```

```
50          }
            return -1;
52      }
```

8.4 Operaties met bits

Een byte is het kleinste adresseerbare element in het werkgeheugen. Zo'n element kan wel geinterpreteerd worden als een rij van 8 bits. In sommige toepassingen zijn manipulaties op deze kleinere elementen, de bits, nodig. In C zijn hiervoor een aantal operatoren voorzien. Deze operatoren kunnen toegepast worden op variabelen van het type `char`, `short`, `int` en `long`.

`~`	bitsgewijze inversie of *1-complement*
`&`	bitsgewijze *en*
`^`	bitsgewijze exclusieve *of*
`|`	bitsgewijze inclusieve *of*
`<<`	schuiven naar links
`>>`	schuiven naar rechts

Definitie van de verschillende operatoren:

`~`	
0	1
1	0

`&`	0	1
0	0	0
1	0	1

`|`	0	1
0	0	1
1	1	1

`^`	0	1
0	0	1
1	1	0

Ook de corresponderende *toekenningsoperatoren* (`&=`, `^=`, `|=`, `<<=` en `>>=`) bestaan.

```
/*
2    * bitop.c : bitoperaties
     */
4   #include <stdio.h>
    void main(void)
6   {
        char a, b, c;
8
        a = 0x01;
10      b = 0x02;
        c = 0x03;
12      printf("      ~ %2x  -> %2x\n",     b,   ~b );
        printf("  %2x & %2x  -> %2x\n", a, b, a&b );
14      printf("  %2x & %2x  -> %2x\n", a, c, a&c );
        printf("  %2x ^ %2x  -> %2x\n", a, b, a^b );
16      printf("  %2x ^ %2x  -> %2x\n", a, c, a^c );
        printf("  %2x | %2x  -> %2x\n", a, b, a|b );
18      printf("  %2x | %2x  -> %2x\n", a, c, a|c );
        printf("  %2x << 1  -> %2x\n", b, b<<1   );
20      printf("  %2x >> 1  -> %2x\n", b, b>>1   );
        c = 0xf0;
22      printf(" %3d >> 1  -> %2d\n", c, c>>1   );
        a = 'G';
24      b = a | 0x20;
        printf("        %c  ->  %c\n", a, b   );
26  }
```

De resultaten:

```
  ~  2  -> fffffffd
1 &  2  ->  0
1 &  3  ->  1
1 ^  2  ->  3
1 ^  3  ->  2
1 |  2  ->  3
1 |  3  ->  3
2 << 1  ->  4
2 >> 1  ->  1
-16 >> 1  -> -8
     G  ->  g
```

Bij de `<<` worden de bits naar links geschoven; vooraan verdwijnen bits, achteraan worden nul-bits toegevoegd. Verschuiven naar links over n posities komt overeen met een vermenigvuldiging met 2^n.
Bij de `>>` gebeurt het omgekeerde: er worden bits naar rechts geschoven en achteraan verdwijnen bits. Wat er vooraan toegevoegd wordt, is afhankelijk van het type van de eerste operand. Wanneer dit type `unsigned` is, worden nul-bits toegevoegd. Verschuiven naar rechts over n posities komt overeen met een deling door 2^n.
Bij een `signed` type echter, wordt naar de hoogste bit (dit is de tekenbit) gekeken. Wanneer deze tekenbit gelijk is aan 0, worden nul-bits toegevoegd. Wanneer deze tekenbit gelijk is aan 1, worden één-bits toegevoegd. Dit noemt men *sign-extension*.

8.5 Begrippen

- Het karakter type, de ASCII tabel.
- Strings, stringfuncties, pointers naar strings.
- Bit operatoren: `~`, `&`, `^`, `|`, `<<`, `>>`.
- Sign-extension.

9 Data-types op maat

9.1 Structures

9.1.1 Declaratie en definitie

In een vorig hoofdstuk werden *arrays* behandeld als een samengesteld data-type, namelijk een verzameling van een aantal variabelen (elementen) die elk een zelfde type hebben. Gerelateerde variabelen worden op die manier gegroepeerd. Dit vereenvoudigt het declareren en het gebruiken van zo'n variabelen.
Om ook variabelen van verschillende types te kunnen groeperen, gebruikt men de *structure*. Een *structure* is een samenhangende collectie variabelen, die van verschillend type kunnen zijn; de structure in zijn geheel kan ook als variabele worden opgevat. (In andere talen wordt ook de naam *record* gebruikt.)
In de volgende declaratie wordt een structure beschreven met naam `mens` en drie velden of elementen. Elk van deze drie velden heeft een verschillend type. De naam `mens` wordt weergegeven in het *tag*-veld.

```
struct mens
{
        char  naam[20];
        int   leeftijd;
        float gewicht;
};
```

Zo'n declaratie creëert een nieuw data-type, dat het reële-wereld object dat in het programma zal verwerkt worden, zo goed mogelijk omschrijft. Dit is een elementaire vorm van *data abstractie*.
Om een variabele van dit nieuwe type te definiëren:

```
struct mens a;
struct mens b[100];
```

De variabele `a` bestaat uit 28 bytes, maar die zijn netjes opgedeeld in drie velden. De variabele `b` bevat een array van 100 elementen, elk element is een structure, bestaande uit drie velden.
Declaratie en definitie kunnen samen gebeuren:

```
struct mens
{
        char  naam[20];
        int   leeftijd;
        float gewicht;
} a;
struct mens b[100];
```

Omdat in de eerste declaratie de structure een naam gekregen heeft, moet in de tweede declaratie niet meer de volledige beschrijving gegeven worden, de naam volstaat.
Een structure hoeft geen naam te krijgen; in dit geval is het *tag*-veld leeg:

```
struct
{
        char  naam[20];
        int   leeftijd;
        float gewicht;
} a;
```

In dit geval moet natuurlijk de definitie van de variabele `a` bij de declaratie gebeuren. Het nadeel is dat bij verder gebruik van deze structure telkens de volledige omschrijving moet gegeven worden. Dit kan echter omzeild worden door expliciet een nieuw data-type te definiëren:

```
typedef struct
        {
              char   naam[20];
              int    leeftijd;
              float  gewicht;
        } Mens;
```

De definitie van een variabele van dit nieuwe type:

```
Mens a;
Mens b[100];
```

Bij een type definitie kan de structure zelf ook nog een naam krijgen:

```
typedef struct mens
        {
              char   naam[20];
              int    leeftijd;
              float  gewicht;
        } Mens;
```

Echter, de constructie `struct mens` zal in de rest van het programma waarschijnlijk niet veel meer voorkomen, omdat normaal steeds het nieuwe data type `Mens` zal gebruikt worden.
Naast elementaire data-types of arrays (bijv. strings) kunnen ook structures zelf gebruikt worden om een veld in een (overkoepelende) structure te beschrijven:

```
typedef struct
        {
              Mens   bio;
              float  punten[10];
        } Student;
Student x;
```

De variabele `x` bestaat uit twee velden, het eerste veld is zelf een structure, het tweede veld is een array van 10 floats.

Plaats in het bronbestand. Net zoals andere variabelen kunnen structure variabelen lokaal of globaal gedefinieerd worden. Het is wel de gewoonte om de declaratie van een structure of de definitie van een nieuw type vooraan in een bestand te doen. Wanneer het programma uit meerdere bronbestanden bestaat, is het aangewezen deze type-definities te verzamelen in een apart *header bestand*. Met de `include` constructie kan dit header bestand opgenomen worden in de verschillende bronbestanden.

9.1.2 Gebruik

Net zoals bij arrays kunnen structures als geheel aangeduid worden of kan een specifiek veld gebruikt worden in een expressie. Om een veld aan te spreken wordt de *punt* `.` operator gebruikt:

```
strcpy(a.naam, "jos");
a.leeftijd = 17;
a.gewicht = 54.4;
b[3].gewicht = a.gewicht + 1.0;
x.bio.leeftijd = 22;
x.punten[3] = 64.3;
```

De initialisatie van een structure kan ook bij de definitie gebeuren:

```
Mens aa = { "flup", 27, 83.0 };
Student xx = { {"marie", 21, 62.1 }, { 74.2, 69.3. 77.7 } };
```

In tegenstelling tot een array naam kan de naam van een structure-variabele wel in een toekenning gebruikt worden. Er zijn dus operaties op de volledige structure mogelijk:

```
b[0] = a;
x.bio = a;
b[1] = aanpassen( b[0] );
```

In het laatste statement is `aanpassen` een functie met één argument, namelijk een variabele van type Mens, en het resultaat is ook van type Mens:

```
Mens aanpassen(Mens z)
{
    z.leeftijd++;
    z.gewicht += 0.4;
    return z;
}
```

Bij de oproep van `aanpassen` wordt de waarde van het actuele argument (alle velden van de structure `b[0]`) gecopieerd in de formele parameter `z`. Hetzelfde gebeurt bij de `return`: de velden van de structure `b[1]` krijgen een nieuwe waarde.
Het vergelijken van twee structures kan echter niet met de eenvoudige vergelijkingsoperatoren (bijv. `==`). Hiervoor kan bijvoorbeeld een functie geschreven worden:

```
int gelijk(Mens x, Mens y)
{
    if ( x.leeftijd != y.leeftijd )
        return 0;
    if ( x.gewicht != y.gewicht )
        return 0;
    return !strcmp(x.naam, y.naam);
}
```

Merk op dat het negatieteken (`!`) voor `strcmp` nodig is om een `0` terug te geven als de namen verschillend zijn.
Om de grootte van een structure te berekenen, kan men de `sizeof` operator gebruiken:

```
Mens aa = { "flup", 27, 83.0 };
int  l1, l2;

l1 = sizeof aa;          /* grootte van een variabele */
l2 = sizeof(Mens);       /* grootte van een type */
```

Het resultaat is het aantal bytes dat de structure in beslag neemt. In dit voorbeeld is l1 gelijk aan l2 gelijk aan 28 bytes.

9.2 Het enumeratie type

Een *enum*eratie is een groep symbolische constanten. Eventueel krijgt de groep als geheel ook een naam, door middel van het *tag* veld, net zoals bij structures.

```
enum { maandag, woensdag, vrijdag };
enum tussen {dinsdag, donderdag, zaterdag, zondag };
```

De eerste declaratie geeft gewoon een opsomming van namen. Inwendig worden deze namen gecodeerd als natuurlijke getallen: 0, 1 en 2. Ook bij de tweede declaratie worden deze waarden (0,1 en 2) toegekend en voor zondag de waarde 3.
De definitie van een variabele:

```
enum { maandag, woensdag, vrijdag } dag;
enum tussen ookdag;
```

Omwille van het *tag* veld bij de tweede declaratie, kan de definitie van `ookdag` korter gebeuren. Het gebruik:

```
dag = maandag;
dag++;
ookdag = zondag;
```

Omdat de onderliggende waarden van de variabelen van het type `int` zijn, kunnen minder fraaie toekenningen geschreven worden. De compiler zal hiervoor hoogstens een fout genereren.

```
ookdag = dag;
dag = 2;
dag++;                    /*  ???  */
```

Aan de elementen van de enumeratie wordt geen adresseerbare geheugenruimte toegekend. Het is niet toegestaan de `&` operator (het adres van) toe te passen op een enumeratie-element.

Opmerking. Het gebruik van dit type is een beetje persoonsgebonden. Sommige programma-ontwerpers vinden dit een handige methode om integrale constanten te groeperen die hetzij een programmafunctie gemeen hebben hetzij een set van gelijkaardige waarden definiëren.

```
enum uitzonderingen { e_internal, e_external, e_nuldeling } fout;
enum boolean { onwaar, waar } gevonden;
```

Het gebruik van enumeratie-constanten is een handige manier om het programma te documenteren. Langs de andere kant is het niet aangewezen constanten van uiteenlopende aard onder te brengen in een enumeratie. In dat geval verdienen aparte `#define`s de voorkeur.

Voorbeeld. Gebruik van `enum` om de verschillende toestanden van een variabele te beschrijven.

```
/*
 * dagop.c : voorbeeld van enum
 */
#include <stdio.h>
enum markt { maandag, woensdag, vrijdag };
enum tussen { dinsdag, donderdag, zaterdag, zondag };
void drukmar(enum markt d);
void druktus(enum tussen d);
void main(void)
{
    enum markt dag;
    enum tussen ookdag;

    for ( dag = maandag; dag <=4; dag++ )
    {
        drukmar(dag);
        ookdag = dag-1;
        druktus(ookdag);
    }
}
void drukmar(enum markt d)
{
    switch (d)
    {
        case maandag:
            printf("␣maandag␣(%d)␣␣␣", d);
            break;
```

```
28            case woensdag:
                  printf("woensdag␣(%d)␣␣␣", d);
30                break;
              case vrijdag:
32                printf("␣vrijdag␣(%d)␣␣␣", d);
                  break;
34            default:
                  printf("␣␣␣␣..␣␣␣(%d)␣␣␣", d);
36                break;
          }
38    }
      void druktus(enum tussen d)
40    {
          switch (d)
42        {
              case dinsdag:
44                printf("␣␣dinsdag␣(%d)\n", d);
                  break;
46            case donderdag:
                  printf("donderdag␣(%d)\n", d);
48                break;
              case zaterdag:
50                printf("␣zaterdag␣(%d)\n", d);
                  break;
52            case zondag:
                  printf("␣␣␣zondag␣(%d)\n", d);
54                break;
              default:
56                printf("␣␣␣␣....␣␣(%d)\n", d);
                  break;
58        }
      }
```

Het resultaat:

```
maandag (0)       .... (-1)
woensdag (1)    dinsdag (0)
vrijdag (2)   donderdag (1)
   ..   (3)    zaterdag (2)
   ..   (4)      zondag (3)
```

9.3 Structuren met bitvelden

Om de benodigde ruimte voor een structure zo klein mogelijk te houden, kan men in een structure een veld definiëren dat slechts één of enkele bits groot is:

```
typedef struct
        {
              char   naam[20];
              int    leeftijd;
              float  gewicht;
              unsigned vrouwelijk: 1;
              unsigned gehuwd: 1;
        } Mens;
```

Voor de velden `vrouwelijk` en `gehuwd` zijn er maar twee mogelijke waarden: waar of niet waar. Er is voor elk van deze velden slechts één bit nodig. De hoeveelheid bits wordt aangegeven door het getal achter het dubbelpunt na de veldnaam. Dit aantal ligt tussen 1 en de lengte van een machinewoord (dit is normaal gelijk aan 32).
In dit voorbeeld wordt aangenomen dat een bit gelijk aan 1 overeenkomt met de waarde waar.

```
Mens aa;
aa.vrouwelijk = 1;          /* vrouw */
aa.gehuwd = 0;              /* ongehuwd */
```

Wanneer in plaats van `gehuwd` ook andere toestanden mogelijk zijn, zoals gescheiden of weduwnaar (weduwe), zijn meerdere bits nodig. In de definitie van het type `Mens` wordt het veld `gehuwd` gewijzigd:

```
unsigned status: 2;
```

In dit geval zijn twee bits genoeg; deze hebben nu volgende betekenis:

```
Mens aa;
aa.status = 0;        /* ongehuwd */
aa.status = 1;        /* gehuwd */
aa.status = 2;        /* gescheiden */
aa.status = 3;        /* weduwnaar/weduwe */
```

Deze codering van de verschillende toestanden is niet handig en bevordert ook niet de leesbaarheid van het programma (tenzij men overal in commentaar de betekenis bijvoegt). Om de leesbaarheid te bevorderen, kan men gebruik maken van het `enum` data type:

```
enum state { ongehuwd, gehuwd, gescheiden, wedu };

aa.status = gescheiden;
```

Omdat bitvelden slechts een onderdeel van een machinewoord zijn, hebben zij geen adres: de `&` operator (het adres van) kan dus niet op een bitveld toegepast worden!

Voorbeeld van de combinatie van een bitveld en een enumeratie type.

```
1   /*
     * bitveld.c
3    */
    #include <stdio.h>
5   enum state { ongehuwd, gehuwd, gescheiden, wedu };
    typedef struct
7           {
              enum state status;
9             unsigned rest: 30;
              int     w ;
11          } S;
    void main(void)
13  {
        S a;
15      enum state jos = ongehuwd;

17      a.w = 4;      a.status = wedu;        a.rest = 2;
        printf ("w  %d  s %d  r %d\n", a.w, a.status, a.rest );
19      a.w = 8;      a.status--;             a.rest += 2;
        printf ("w  %d  s %d  r %d\n", a.w, a.status, a.rest );
21      a.w = 16;     a.status = jos;         a.rest <<= 2;
```

```
        printf ("w %d  s %d  r %d\n", a.w, a.status, a.rest );
23  }
```

```
Resultaat:      w  4  s 3  r 2
                w  8  s 2  r 4
                w 16  s 0  r 16
```

9.4 Union

De geheugenruimte die door een *structure* in beslag genomen wordt, is tenminste gelijk aan de som van de ruimte van de componenten. Alle componenten van een structure zijn gelijktijdig fysiek aanwezig. Soms wenst men bepaalde alternatieven van een object te bewerken, die niet gelijktijdig (kunnen) voorkomen. In dat geval moet de benodigde geheugenruimte niet groter zijn dan de grootste van de verschillende varianten. Dit wordt gerealiseerd met een `union`.
Declareren gebeuren zoals bij een `structure`: met of zonder een *tag*-veld; declaratie en definitie van een variabele samen of apart; creatie van een nieuw type via `typedef`:

```
typedef union
        {
              char   naam[16];
              int    leeftijd;
              double gewicht;
        } Eigenschap;
```

Ook het gebruik is volledig analoog aan het gebruik van een structure:

```
Eigenschap u;
int  len;

u.leeftijd = 37;
u.gewicht = 103.5;
len = sizeof(Eigenschap);
```

Opmerking. Bij een union kan een zelfde geheugenplaats met behulp van verschillende namen aangesproken worden. Dit noemt men *aliasing*. Bij ondeskundig gebruik kan dit leiden tot programmeerfouten die in sommige gevallen zeer moeilijk te vinden zijn. Voor de meeste problemen is het gebruik van unions niet nodig en daarom is het aan te raden deze constructie niet te gebruiken. Zowel structures met bitvelden als unions stammen uit de tijd dat RAM geheugens nog zeer duur waren. Elke mogelijke besparing op het vlak van ruimte voor variabelen in een programma, was in die tijd bijzonder welkom.

9.5 Pointers naar structures

Net zoals pointers naar integers of floats kunnen gedeclareerd worden, kan dit ook voor structures:

```
Mens  aa;
Mens *pa;

pa = &aa;
```

In dit voorbeeld wordt een variabele van type Mens gedefinieerd. Hiervoor worden 28 bytes voorzien. Daarnaast is er ook pointer `pa`, die 4 bytes in beslag neemt. De toekenning heeft als effect dat de inhoud van de variabele `pa` het adres bevat van de structure `aa` en dus naar die structure *wijst*.
De afzonderlijke velden van de structure kunnen nu aangesproken worden via de pointer variabele:

```
(*pa).leeftijd = 34;
```

Door `*pa` vindt men de structure zelf terug. Hiervan heeft men een bepaald veld nodig. Dit wordt gevonden door de *punt* operator te gebruiken. Omdat `.` een hogere prioriteit heeft dan `*`, moeten haakjes gebruikt worden.
In de meer gebruikelijke manier maakt men gebruik van het feit dat de variabele `pa` *wijst* naar het begin van de structure:

```
pa->leeftijd = 34;
```

Pointers naar structures waren oorspronkelijk vrij belangrijk omwille van efficiëntie. In het volgende voorbeeld worden een aantal elementen samengebracht. Naast structures met punt operatoren, wordt zoveel mogelijk gebruik gemaakt van pointers. De eerste C-compilers konden voor zo'n programma efficiëntere code genereren, omdat de manier van adresseren beter aansloot bij de beschikbare adresseermechanismen van de processor.

```
1  /*
    * strucop.c : bewerkingen op een structure
3   */
   #include <stdio.h>
5  #include <string.h>
   #define NMLEN  20
7  #define AANTAL 10
   typedef struct
9          {
               char  naam[NMLEN];
11             int   leeftijd;
               float gewicht;
13         } Mens;
   Mens *aanpasp(Mens *z);
15 Mens aanpass(Mens z);
   int gelijk(Mens *px, Mens *py);
17
   void main(void)
19 {
       Mens  b[AANTAL];
21     Mens  *p, *q;

23     p = &b[0];
       strcpy(p->naam, "marieke");
25     p->leeftijd = 17;
       p->gewicht = 48.9;
27     q = &b[1];
       *q = *p;
29     druk( &b[0] );       druk( &b[1] );
       printf("\t: %d\n", gelijk(p, q) );
31     b[2] = aanpass(b[1]);
       druk( &b[1] );       druk( &b[2] );
33     printf("\t: %d\n", gelijk(&b[1], &b[2]) );
       q = aanpasp(p);
35     b[3] = *q;
       druk( &b[0] );       druk( &b[3] );
37     printf("\t: %d\n", gelijk(&b[0], &b[3]) );
   }
39 int gelijk( Mens *px,  Mens *py)
   {
```

```
41      if ( px->leeftijd != py->leeftijd )
            return 0;
43      if ( px->gewicht != py->gewicht )
            return 0;
45      return !strcmp(px->naam, py->naam);
    }
47  Mens aanpass( Mens z)
    {
49      z.leeftijd--;
        z.gewicht -= 0.4;
51      return z;
    }
53  Mens *aanpasp( Mens *z )
    {
55      z->leeftijd++;
        z->gewicht += 0.4;
57      return z;
    }
59  druk( Mens *z )
    {
61      printf("%-9.9s %2d %5.2f   ", z->naam, z->leeftijd , z->gewicht);
    }
```

```
Resultaat:      marieke   17 48.90   marieke   17 48.90    : 1
                marieke   17 48.90   marieke   16 48.50    : 0
                marieke   18 49.30   marieke   18 49.30    : 1
```

Merk op dat de functie `aanpasp` niet volledig hetzelfde doet als de functie `aanpass` waarbij met de structures zelf gewerkt wordt!
Omwille van de beschikbaarheid van zeer krachtige compilers, zou men kunnen besluiten dat pointers naar structures wat aan betekenis verloren hebben. Er is echter nog een andere reden om met pointers te werken, namelijk dynamische geheugenallocatie. In het voorgaande werd telkens bij de definite van de structure voldoende geheugen gealloceerd, bijvoorbeeld bij de array van structures werd aangegeven hoeveel elementen in de array aanwezig zijn. In veel gevallen zou men de allocatie van het benodigde geheugen meer dynamisch willen maken. Afhankelijk van de input zou men willen kunnen beslissen hoe groot de array moet zijn.

9.6 Begrippen

- Samengestelde types: `structure` en `union`.
- De selectie-operatoren: `.` en `->`.
- Het enumeratie type en bitvelden.
- De definitie van nieuwe types: `typedef`.

10 In- en uitvoer

10.1 Inleiding

C bevat geen speciale taalconstructies voor in- en uitvoer. Men heeft hiervoor een verzameling van functies voorzien die samengebracht zijn in de standaard C-bibliotheek. Voorbeelden zijn `printf` en `scanf`. In dit hoofdstuk zullen andere I/O functies besproken worden die gebruikt worden bij

- andere in- en uitvoer dan die van en naar de terminal (toetsenbord en beeldscherm);
- binair in plaats van geformatteerd;
- directe toegankelijkheid van bestanden.

Naast deze in- en uitvoer functies zijn in deze bibliotheek nog andere functies opgenomen, bijv. `malloc`.
Om de in- en uitvoer functies te kunnen gebruiken (d.w.z. oproepen) moet een include gedaan worden van de header file `stdio.h`.

10.2 Geformatteerde output

Een oproep van `printf` heeft de vorm:

```
printf(formaat, arg1, arg2, ... );
```

De *formaat* string bevat twee soorten objecten: gewone af te drukken karakters en conversiespecificaties. Voor elk van de argumenten `arg1`, `arg2`, ... wordt de rvalue afgedrukt overeenkomstig de conversiespecificatie in de string. Zo'n specificatie begint met een `%` en eindigt op een conversieteken.

conversieteken	het corresponderende argument wordt afgedrukt als
`%d`	een geheel getal in decimale notatie
`%o`	een geheel getal in octale notatie
`%x`	een geheel getal in hexidecimale notatie
`%u`	een geheel getal in *unsigned* decimale notatie
`%f`	een reëel getal in vaste komma notatie
`%e`	een reëel getal in drijvende komma notatie
`%g`	`e` of `f` conversie, de kortste vorm
`%c`	een karakter
`%s`	een string

Tussen `%` en het conversieteken kan staan:

`-`	links in de beschikbare ruimte
getal	de veldbreedte
`.`	scheiding tussen veldbreedte en precisie
getal	de precisie - getal: het aantal cijfers rechts van het punt - string: het aantal afgedrukte karakters
`l`	argument is van type `long int` (`double`) in plaats van `int` (`float`)
`h`	argument is van type `short int` in plaats van `int`

Wanneer na de `%` een karakter volgt dat niet in bovenstaande lijsten opgenomen is, wordt dat karakter gewoon afgedrukt. Dit is handig om het procent-teken zelf af te drukken, door middel van `%%`. Om het `"` teken af te drukken, moet dit teken in de formaatstring voorafgegaan worden door een backslash: `\"`.
Voorbeeld:

3456		314.159265358979323846	
%10d	3456	%20.10f	314.1592653590
%10o	6600	%20.10e	3.1415926536e+02
%10x	d80	%20.10g	314.1592654
%10u	3456	%-20.10f	314.1592653590
%-10d	3456	%-20.10e	3.1415926536e+02
%-10u	3456	%-20.10g	314.1592654
	1234567890		12345678901234567890

Afhankelijk van de implementatie geeft de functie `printf` ook een waarde terug. Deze is dan gelijk aan het aantal overgedragen karakters of negatief in het geval van een fout.

10.3 Geformatteerde input

Een oproep van `scanf` heeft de vorm:

```
scanf(formaat, arg1, arg2, ...);
```

De argumenten `arg1`, `arg2`, ... zijn adressen (pointers), die aangeven waar de ingelezen gegevens moeten opgeborgen worden.
De conversietekens die in *formaat* kunnen voorkomen, zijn `d`, `o`, `x`, `f`, `c`, `s`. De tekens `d`, `o` en `x` moeten voorafgegaan worden door een `l` wanneer het corresponderende argument een pointer is naar een `long` in plaats van een `int`; analoog wordt `h` gebruikt om aan te geven dat het over een `short` gaat. Wanneer het corresponderende argument een pointer naar een `double` is, wordt `f` voorafgegaan door een `l`.
Tussen `%` en het conversieteken kan een `*` staan. Hiermee wordt aangegeven dat wel iets moet gelezen worden, maar dat geen toekenning aan een variabele moet plaatsvinden. Daarnaast kan met behulp van een getal ook de maximale veldbreedte aangegeven worden.
Buiten de conversiespecificaties kunnen in het eerste argument nog voorkomen:

- spaties, tabs en newlines: deze worden genegeerd;
- gewone karakters (behalve `%`): deze moeten dan ook in de invoer op de corresponderende plaats voorkomen.

De functie `scanf` geeft een waarde terug. Deze is gelijk aan het aantal gelezen en aan variabelen toegekende waarden. Als bij het lezen iets fout loopt, is de return waarde gelijk aan `0` of `EOF` (in `stdio.h` gedefinieerd als `-1`). Het resultaat `0` geeft aan dat geen enkele variabele een waarde gekregen heeft, omdat de gespecificeerde conversie in de formaatstring niet mogelijk is op basis van de ingelezen tekens. Het resultaat `-1` geeft het einde van de invoer aan, bijvoorbeeld bij het intikken van CTRL-Z.

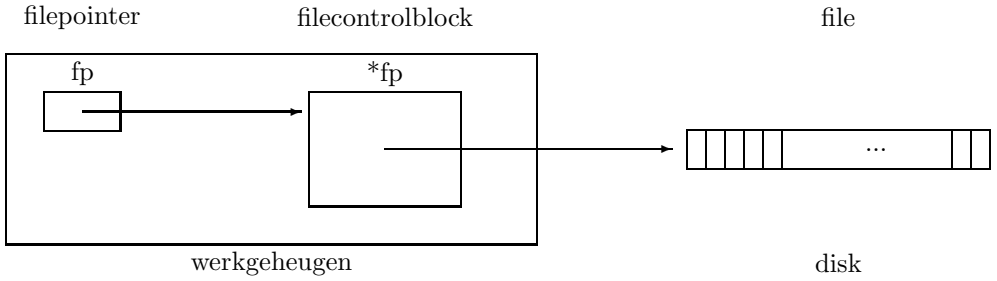
10.4 Via bestanden

In plaats van input van het toetsenbord en output naar het beeldscherm, is het nodig gegevens van bestanden te kunnen lezen en resultaten naar bestanden te kunnen schrijven. Hiervoor worden twee analoge functies gebruikt, wanneer het over geformatteerde in- en uitvoer gaat:

```
fprintf(filepointer, formaat, arg1, arg2, ...);
fscanf(filepointer, formaat, arg1, arg2, ...);
```

Deze twee functies hebben elk één bijkomend argument, namelijk een *filepointer*. Een filepointer is een variabele die een pointer bevat naar een structure waarin allerlei administratieve gegevens zitten omtrent het bestand dat gebruikt wordt. De inhoud van het bestand zelf is ergens op een extern medium (bijv. disk) gelokaliseerd maar om in- of uitvoer naar het bestand te doen, moeten deze administratieve gegevens snel toegankelijk zijn en moeten deze dus in het werkgeheugen aanwezig zijn.

Deze administratieve gegevens zijn verzameld in het *file-control-block*. Hierin zit o.a. waar ergens op de disk de data van het bestand kan gevonden worden. De precieze inhoud is gedefinieerd door middel van het type `FILE`. De definitie hiervan kan teruggevonden worden in `stdio.h`. De filepointer zelf is dus van type `FILE *`.



Om in het file-control-block de verschillende velden te initialiseren met de administratieve gegevens van het bestand waarop in- of uitvoer gedaan gaat worden, moet het bestand *geopend* worden. Dit gebeurt met de functie `fopen`. Wanneer het lezen van of het schrijven naar het bestand afgelopen is, kan het file-control-block terug opgeruimd worden door het bestand te *sluiten* mbv de functie `fclose`.

```
FILE *fopen(char *naam, char *mode);
fclose(FILE *fp);
```

De functie `fopen` heeft twee argumenten. Het eerste argument, `naam`, specificeert welk bestand van disk moet geopend worden. Dit gebeurt met behulp van de naam van het bestand zoals die op de disk gekend is. Wanneer het om een nieuw te creëren bestand gaat, duidt dit argument de naam aan, die vanaf dan kan gebruikt worden om het bestand te benoemen. Het tweede argument, `mode`, geeft aan hoe het bestand moet geopend worden. Er zijn verschillende mogelijkheden:

`"r"`	lezen van bestaand bestand
`"w"`	schrijven op een nieuw bestand (creatie van bestand)
	eventueel reeds bestaande inhoud is verloren
`"a"`	achteraan toevoegen aan een bestaand bestand
`"r+"`	lezen van en schrijven op bestaand bestand
`"w+"`	schrijven op een nieuw bestand (creatie van bestand)
	ook lezen is mogelijk
`"a+"`	achteraan toevoegen aan een bestaand bestand
	ook lezen is mogelijk

De return-waarde van `fopen` is een pointer naar een nieuw gecreëerd filecontrolblock. Indien er toch iets misloopt bij het openen van het bestand, wordt de waarde `NULL` teruggegeven.
Eens een bestand geopend, kan ervan gelezen worden en kan erop geschreven worden met bijvoorbeeld de `fscanf` en `fprintf` functies.

```
/*
 * leesgetal.c : leest een aantal getallen van een bestand
 */
#include <stdio.h>
void main(void)
{
    FILE   *fin;
    FILE   *fuit;
    int    tel = 0;
    double getal;
```

```
12      fin = fopen("inweg.txt", "r");
        fuit = fopen("uitweg.txt", "w");
14      while ( fscanf(fin, "%lf%*c", &getal) != EOF )
        {
16            printf(      " %4d : %10.4f\n", tel, getal);
             fprintf(fuit, " %4d : %10.4f\n", tel++, getal);
18      }
        fclose(fin);
20      fclose(fuit);
        printf(" %4d getallen gelezen\n", tel);
22  }
```

Naast deze twee functies zijn er nog andere functies, bijvoorbeeld om karakter per karakter of lijn per lijn te werken:

```
int fgetc(FILE *fp);
int fputc(int ch, FILE *fp);
char *fgets(char *str, int maxlen, FILE *fp);
int fputs(char *str, FILE *fp);
```

Met `fgets` worden maximaal `maxlen-1` tekens ingelezen, zodat in de laatste positie plaats is voor een `'\0'`.
De volgende twee programma's kunnen gebruikt worden om een copie van een bestand te maken.

```
 /*
2  * teken.c : copieert een bestand teken per teken
   */
4 #include <stdio.h>
  void main(void)
6 {
      FILE *fin;
8     FILE *fuit;
      int  teken;
10
      fin = fopen("a.c", "r");
12    fuit = fopen("weg.txt", "w");
      while ( (teken=fgetc(fin)) != EOF )
14        fputc(teken, fuit);
      fclose(fin);
16    fclose(fuit);
  }
```

Wanneer het einde van een bestand bereikt is, geeft de functie `fgetc` de waarde `EOF` terug.

```
1  /*
    * lijn.c : copieert een bestand lijn per lijn
3   */
  #include <stdio.h>
5 #define LLEN    80
  void main(void)
7 {
      FILE *fin;
9     FILE *fuit;
      char lijn[LLEN];
11
      fin = fopen("a.c", "r");
13    fuit = fopen("weg.txt", "w");
```

```
    while ( fgets(lijn, LLEN, fin) != NULL )
        fputs(lijn, fuit);
    fclose(fin);
    fclose(fuit);
}
```

Wanneer het einde van een bestand bereikt is, geeft de functie `fgets` de waarde `NULL` terug. Soms kan het nuttig zijn om een rij tekens te lezen tot aan een bepaald teken, maar zonder dat teken zelf. Dit wordt opgelost door het teken eerst te lezen, maar dan wordt dit lezen ongedaan gemaakt. Het is alsof het teken weer in de invoer gezet wordt.

```
/*
 * twoord.c : tel het aantal woorden in een bestand
 */
#include <stdio.h>
#include <ctype.h>
#define WLEN 80
void leesspaties( FILE *fp, int *pi );

void main(void)
{
    FILE  *fin;
    int    spaties = 0;
    int    tel = 0;
    char   woord[WLEN];

    fin = fopen("inweg.txt", "r");
    leesspaties(fin, &spaties);
    while ( fscanf(fin, "%s", woord) != EOF )
    {
        tel++;
        printf("%4d : %s\n", tel, woord);
        leesspaties(fin, &spaties);
    }
    fclose(fin);
    printf("%4d woorden gelezen\n", tel);
    printf("%4d spaties weggelaten\n", spaties);
}

void leesspaties( FILE *fp, int *pi )
{
    int  ch;

    do
    {
        ch = fgetc(fp);
        (*pi)++;
    }
    while ( isspace(ch) );
    (*pi)--;
    ungetc(ch,fp);
}
```

De functie `isspace` test of het argument een spatie, tab-teken, new-line, carriage return of form feed is.

Voor invoer van toetsenbord kan men de filepointer `stdin` gebruiken en voor uitvoer naar beeldscherm `stdout`. De bijhorende filecontrolblocks worden door het run-time systeem geïnitialiseerd wanneer het programma gestart wordt.

```
Vereenvoudigde vorm:
    int scanf("...", ...);
    int printf("...", ...);
    int getchar();
    int putchar(int ch);
    char *gets(char *str);
    int puts(char *str);
```

```
Algemene vorm:
   int fscanf(stdin, "...", ...);
   int fprintf(stdout, "...", ...);
   fgetc(stdin);
   fputc(ch, stdout);
   fgets(str, maxlen, stdin);
   fputs(str, stdout);
```

Daarnaast is er nog een derde filepointer, `stderr`, die ook voor uitvoer naar het beeldscherm kan gebruikt worden.

10.5 Binaire informatie

Geformatteerde data is alleen nodig om de informatie toonbaar te maken voor de gebruiker zelf. Een voorbeeld hiervan is het opdelen van de informatie in lijnen, zodat er iets netjes op paper kan afgedrukt worden. Vele bestanden in een computersysteem bevatten informatie die alleen door het systeem zelf gelezen of aangepast wordt. Deze informatie hoeft dan niet geformatteerd te zijn. De externe representatie van de informatie (op een bestand) is dan dezelfde als de interne voorstelling (in de variabelen in het werkgeheugen). Er wordt geen conversie uitgevoerd.
Bestanden die zo'n informatie bevatten worden soms *binaire bestanden* genoemd in tegenstelling tot *tekst-bestanden*. In sommige C-omgevingen moet bij de functie `fopen` aangegeven worden dat het over een binair bestand gaat. Dit gebeurt door in het `mode` argument de letter **b** bij te voegen. Om bewerkingen op binaire bestanden uit te voeren, zijn volgende functies beschikbaar:

```
size_t fread(void *ptr, size_t len, size_t nobj, FILE *fp);
size_t fwrite(void *ptr, size_t len, size_t nobj, FILE *fp);
int fseek(FILE *fp, long offset, int code);
long ftell(FILE *fp);
```

De functies `fread` en `fwrite` geven als resultaat het werkelijk aantal getransporteerde elementen. Wanneer een fout opgetreden is of het einde van het bestand bereikt is, wordt de waarde 0 teruggegeven. De eerste parameter is van type `void *`: geeft aan dat het een adres in het werkgeheugen is naar een plaats waar zowel iets van type `int`, `float`, ... of een zelf gedefinieerde structure zit. Het type `size_t` is in het headerbestand `stdio.h` met behulp van `typedef` gedefinieerd als een `unsigned int`.
Naast lezen en schrijven is ook de functie `fseek` voorzien om de positie in het bestand waar de operatie zal uitgevoerd worden, te wijzigen. Deze functie heeft normaal 0 als resultaat, tenzij er iets misgegaan is. In dat geval is de terugkeerwaarde gelijk aan -1. De functie `ftell` geeft de actuele offset in een bestand, relatief ten opzichte van het begin en uitgedrukt in aantal bytes.

ptr	het adres van een gebied in het werkgeheugen	fread, fwrite
len	de lengte van een element in dit gebied	fread, fwrite
nobj	het aantal elementen in de operatie betrokken	fread, fwrite
fp	de filepointer	fread, fwrite, fseek, ftell
offset	relatieve positie (in bytes) tov een startpunt	fseek
code	indicatie voor het startpunt	fseek
	`SEEK_SET` : tov het begin van het bestand	
	`SEEK_CUR` : tov de actuele positie in het bestand	
	`SEEK_END` : tov het einde van het bestand	

Omdat met behulp van de `fseek` functie naar een willekeurige record in het bestand kan gegaan worden, zonder alle voorgaande records te lezen, wordt zo'n binair bestand ook een bestand met

directe toegankelijkheid genoemd. Op een binair bestand worden elementen of objecten weggeschreven. Meestal spreekt men van *records*. Dus een binair bestand is een opeenvolging van records, beginnend met record 0. De lengte van de verschillende records in een binair bestand kan constant zijn. Men spreekt dan van een *bestand met vaste record lengte*. In andere gevallen kan een bestand records van verschillende lengte bevatten (*bestand met veranderlijke record lengte*).

Opgave. Lees een geformatteerd tekst bestand bestaande uit een eerste lijn met twee gehele getallen (aantal rijen en aantal kolommen van een matrix) en een aantal volgende lijnen met per lijn de elementen van een rij van de matrix. Schrijf deze matrix op een binair bestand: eerste record is het aantal rijen, de tweede record is het aantal kolommen en de volgende records zijn de verschillende rijen van de matrix.

```
/*
 * f2b.c : omzetting geformatteerd naar binair
 */
#include <stdio.h>

#define TNAAM "mat.txt"
#define BNAAM "mat.dat"
#define PMAX 25

void main(void)
{
    FILE    *finp;          /* input : tekst bestand */
    FILE    *fdirp;         /* output : binair bestand */
    float   p[PMAX];
    float   x;
    int     m;              /* aantal rijen */
    int     n;              /* aantal kolommen */
    int     i, j;

    finp = fopen(TNAAM, "r");
    fscanf(finp, "%d%d%*c", &m, &n);
    fdirp = fopen(BNAAM, "wb");
    fwrite(&m, sizeof(int), 1, fdirp);
    fwrite(&n, sizeof(int), 1, fdirp);
    for (i=1; i<=m ; i++)
    {
        for (j=1; j<=n ; j++)
        {
            fscanf(finp, "%f", &x);
            p[j] = x;
        }
        fscanf(finp, "%*c");
        fwrite(&p[1], sizeof(float), n, fdirp);
    }
    fclose(fdirp);
    fclose(finp);
}
```

Opgave. Lees het binair bestand gecreëerd in vorige opgave. Schrijf de matrix in geformatteerde vorm uit.

```
/*
```

```
 * b2f.c : omzetting binair naar geformatteerd
 */
#include <stdio.h>

#define BNAAM "mat.dat"
#define PMAX 25

void main(void)
{
    FILE    *fdirp;         /* input : binair bestand */
    float   p[PMAX][PMAX];
    int   m;              /* aantal rijen */
    int   n;              /* aantal kolommen */
    int   i, j;

    fdirp = fopen(BNAAM, "rb");
    fread(&m, sizeof(int), 1, fdirp);
    fread(&n, sizeof(int), 1, fdirp);
    for (i=1; i<=m; i++)
    {
        fread(&p[i][1], sizeof(float), n, fdirp);
    }
    fclose(fdirp);
    for (i=1; i<=m ; i++)
    {
        for (j=1; j<=n ; j++)
            printf("%10.4f", p[i][j]);
        printf("\n");
    }
}
```

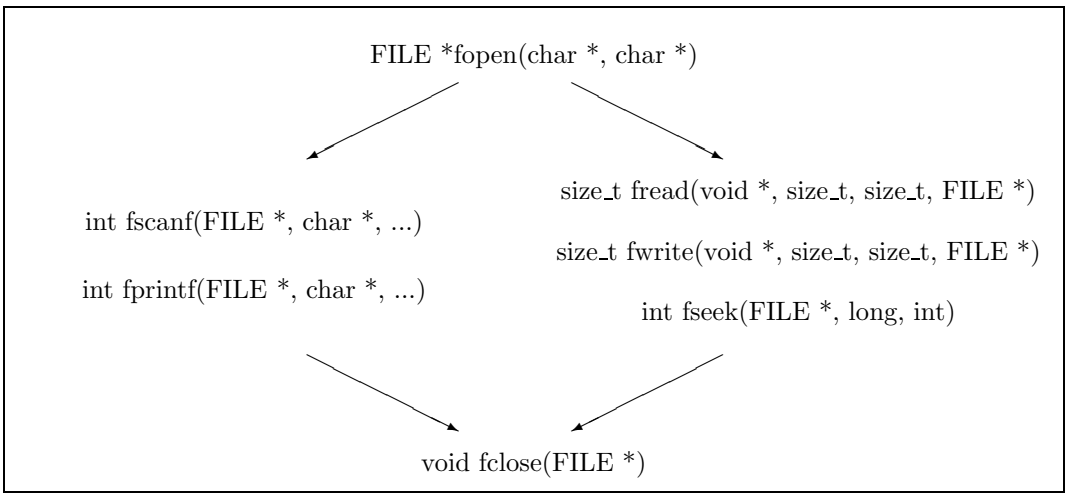
Opgave. Gegeven een binair bestand met vaste record lengte. Elke record bestaat uit drie floats. Gevraagd een binair bestand met daarin de records in omgekeerde volgorde.

```
/*
 * binv.c : records in omgekeerde volgorde
 */
#include <stdio.h>

#define INAAM "koppel.dat"
#define UNAAM "kopinv.dat"

typedef struct
        {
            float     x;
            float     y;
            float     z;
        } Punt;

void main(void)
{
    FILE    *finp;
    FILE    *fuitp;
    Punt    pnt;
```

```
21      int     i;

23      finp = fopen(INAAM, "rb");
        fuitp = fopen(UNAAM, "w+b");
25      fseek(finp, -(long)sizeof(Punt), SEEK_END);
        for (i=0;    ; i++)
27      {
            if ( fread(&pnt, sizeof(Punt), 1, finp) <= 0 )
29              break;
            printf("Record %d wordt geschreven %f %f %f\n",
31                                          i, pnt.x, pnt.y, pnt.z);
            fwrite(&pnt, sizeof(Punt), 1, fuitp);
33          if ( fseek(finp, -2L*sizeof(Punt), SEEK_CUR) < 0 )
                break;
35      }
        i /= 2;
37      fseek(fuitp, (long)i*sizeof(Punt), SEEK_SET);
        fread(&pnt, sizeof(Punt), 1, fuitp);
39      printf("Record %d : %f %f %f\n", i, pnt.x, pnt.y, pnt.z);
        fclose(finp);
41      fclose(fuitp);
    }
```

10.6 Begrippen

- Een bestand, een filepointer en een filecontrolblock.
- Geformatteerde in- en uitvoer, tekst-bestanden.
- Directe toegankelijkheid, binaire bestanden.
- Overzicht:



10.7 Een adressenbestand

Opgave. Schrijf een programma dat een aantal bewerkingen kan uitvoeren op binair bestand met records van lengte 128 bytes. Een record bevat de naam, het adres en de telefoonnummer van een persoon. De bewerkingen zijn: het maken van een lijst, het zoeken van een record, het toevoegen van een record en het verwijderen van een record.

Het header bestand:

```
/*
 * adres.h :  definities en declaraties voor adressenbestand
 */
#define TNAAM "adres.txt"
#define BNAAM "adres.dat"
#define SCHEIDING ';'
#define NLEN     32
#define PLEN      4
#define TLEN     10
#define MAXLEN 128

typedef struct
        {
            char naam[NLEN];
            char adres[NLEN];
            char postco[PLEN];
            char woonpla[NLEN];
            char tel[TLEN];
            char rest[18];
        } Adres;

void   drukaf(Adres *ap, int i);
```

Het bestand met hoofdprogramma en functies:

```
/*
 * adres.c : bewerkingen op binair bestand
 */
#include <stdio.h>
#include <stdlib.h>
#include <memory.h>
#include "adres.h"

char *menutekst[] =
        { "",
          " lijst",
          " zoeken",
          " toevoegen",
          " verwijderen",
          " stoppen",
          0
        };
void lijst(void);
int  zoeken(void);
void toevoegen(void);
```

```
void verwijderen(void);

int menu(void)
{
      int i;

      i = 1;
      while ( menutekst[i] )
      {
            printf(" %2d : %s\n", i, menutekst[i]);
            i++;
      }
      printf("Keuze : ");
      scanf("%d%*c", &i);
      return i;
}

void main(void)
{
      int  keuze;
      int  n;

      while ( 1 )
      {
           do
           {
                keuze = menu();
           }
           while ( keuze < 1  && keuze > 5 );
           switch ( keuze )
           {
               case 1 :
                  lijst();
                  break;
               case 2 :
                  n = zoeken();
                  break;
               case 3 :
                  toevoegen();
                  break;
               case 4 :
                  verwijderen();
                  break;
               default:
                  exit(0);
           }
      }
}

void lijst(void)                              /* sequentieel doorlopen */
{
     FILE   *fdirp;          /* binair bestand */
     int    n = 0;           /* aantal records */
     Adres  a;               /* in te lezen record */
```

```
76          fdirp = fopen(BNAAM, "rb");
            while ( fread(&a, sizeof(Adres), 1, fdirp) != 0 )
78          {
                if ( a.naam[0] != '\0' )
80                  drukaf(&a, n);
                n++;
82          }
            fclose(fdirp);
84      }

86      int zoeken(void)                                    /* lineair zoeken */
        {
88          FILE   *fdirp;          /* binair bestand */
            int    n = 0;           /* aantal records */
90          Adres  a;               /* in te lezen record */
            char   naam[NLEN];
92
            fdirp = fopen(BNAAM, "rb");
94          printf("naam : ");      gets(naam);
            while ( fread(&a, sizeof(Adres), 1, fdirp) != 0 )
96          {
                if ( strncmp(a.naam, naam, NLEN) == 0 )
98              {
                    drukaf(&a, n);
100                 break;
                }
102             n++;
            }
104         fclose(fdirp);
            return n;
106     }

108     void toevoegen(void )                           /* achteraan toevoegen */
        {
110         FILE   *fdirp;          /* binair bestand */
            int    n = 0;           /* recordnr */
112         Adres  a;               /* nieuw record */
            int    ok;
114
            fdirp = fopen(BNAAM, "ab");
116         n = ftell(fdirp)/sizeof(Adres);
            memset(&a, '\0', MAXLEN);
118         printf("naam : ");      gets(a.naam);
            printf("adres : ");     gets(a.adres);
120         printf("postco : ");    gets(a.postco);
            printf("woonpla : ");   gets(a.woonpla);
122         printf("tel : ");       gets(a.tel);
            drukaf(&a, n);
124         printf("Zijn de gegevens juist ? [j/n] : ");
            ok = getchar();
126         if ( ok == 'j' )
                fwrite(&a, sizeof(Adres), 1, fdirp);
128         fclose(fdirp);
```

```
}

void verwijderen(void )     /* door structure-inhoud op nul te zetten */
{
     FILE   *fdirp;         /* binair bestand */
     int    n = 0;          /* aantal records */
     Adres  a;              /* in te lezen record */
     int    ok;

     n = zoeken();
     printf("Deze gegevens verwijderen ? [j/n] : ");
     ok = getchar();
     if ( ok != 'j' )
          return;
     memset(&a, '\0', MAXLEN);
     fdirp = fopen(BNAAM, "r+b");
     fseek(fdirp, (long)n*sizeof(Adres), SEEK_SET);
     fwrite(&a, sizeof(Adres), 1, fdirp);
     fclose(fdirp);
}

void drukaf(Adres *ap, int i)
{
     printf("%3d %-19.19s %-19.19s %4.4s %-16.16s %-10.10s\n",
         i, ap->naam, ap->adres, ap->postco, ap->woonpla, ap->tel);
}
```

Opgave. Gegeven een tekstbestand met per lijn volgende gegevens: naam, adres, postcode, woonplaats en telefoonnummer. Deze velden zijn van elkaar gescheiden met een ";". Gevraagd een programma dat dit tekstbestand lijn per lijn leest en de gegevens gebruikt om een binair bestand met records zoals in het vorige voorbeeld te initialiseren.

```
/*
 * txt2bin.c : omzetting tekstbestand nar binair bestand
 */
#include <stdio.h>
#include "adres.h"
Adres  *ontleed(char *lijn);

void main(void)
{
     FILE   *ftxtp;          /* input : tekst bestand */
     FILE   *fdirp;          /* output : binair bestand */
     char   lijn[MAXLEN];    /* de invoerlijn */
     int    n = 0;           /* aantal records */
     Adres  *ap;             /* pointer naar opgebouwde record */

     ftxtp = fopen(TNAAM, "r");
     fdirp = fopen(BNAAM, "wb");
     while ( fgets(lijn, MAXLEN, ftxtp) != NULL )
     {
          ap = ontleed(lijn);
          drukaf(ap, n);
          fwrite(ap, sizeof(Adres), 1, fdirp);
```

```
            n++;
24      }
        fclose(fdirp);
26      fclose(ftxtp);
        printf("Totaal aantal records %d\n", n);
28  }

30  void drukaf(Adres *ap, int    i)
    {
32      printf("%3d %-19.19s %-19.19s %4.4s %-16.16s %-10.10s\n",
            i, ap->naam, ap->adres, ap->postco, ap->woonpla, ap->tel);
34  }

36  Adres *ontleed(char *lijn)
    {
38      char   *zoek(char *cp);
        static Adres a;
40      char   *cpv;
        char   *cpt;
42
        cpv = lijn;
44      cpt = zoek(cpv);
        strncpy(a.naam, cpv, NLEN);
46      cpv = cpt+1;
        cpt = zoek(cpv);
48      strncpy(a.adres, cpv, NLEN);
        cpv = cpt+1;
50      cpt = zoek(cpv);
        strncpy(a.postco, cpv, PLEN);
52      cpv = cpt+1;
        cpt = zoek(cpv);
54      strncpy(a.woonpla, cpv, NLEN);
        cpv = cpt+1;
56      cpt = zoek(cpv);
        strncpy(a.tel, cpv, TLEN);
58      return &a;
    }
60
    char *zoek(char *v)
62  {
        char *t = v;
64
        while ( *t != SCHEIDING && *t != '\n')
66          t++;
        *t = '\0';
68      return t;
    }
```

Eeen voorbeeld van een gegeven tekstbestand:

```
Drieskens Marc;Vendelstraat 15;3012;Wilsele;016/563227
Leupe Dirk;Bergstraat 56;2500;Lier;03/4577001
Ooms Willy;Voort 27;2310;Rijkevorsel;03/3124519
```

11 Diversen

11.1 Een overzicht van de operatoren

De operatoren gerangschikt van hoge naar lage prioriteit, met bijhorende associativiteit:

`()`	`[]`	`->`	`.`								links
`!`	`~`	`++`	`--`	`-`	(type)	`*`	`&`	sizeof			rechts
`*`	`/`	`%`									links
`+`	`-`										links
`<<`	`>>`										links
`<`	`<=`	`>`	`>=`								links
`==`	`!=`										links
`&`											links
`^`											links
`|`											links
`&&`											links
`||`											links
`?:`											rechts
`=`	`+=`	`-=`	`*=`	`/=`	`%=`	`<<=`	`>>=`	`&=`	`^=`	`|=`	rechts
`,`											links

Merk op dat elk van de tekens `-`, `*` en `&` tweemaal voorkomt. De context van het desbetreffende teken in het programma bepaalt om welke operator het gaat.

11.1.1 Een ternaire operator

Aan een variabele een waarde toekennen die afhankelijk is van een conditie kan met behulp van het `if` statement:

```
if ( a < b )
    z = a + 1;
else
    z = b - 1;
```

Dit kan op een kortere manier met behulp van de `?:` operator:

```
z =  a<b ? a+1 : b-1;
```

Indien a kleiner is dan b, dan is de waarde van de expressie $a+1$, anders $b-1$. Deze waarde wordt toegekend aan de variabele z. De operator heeft drie operands : `a<b`, `a+1` en `b-1`.
In plaats van de waarde van de expressie toe te kennen, kan ze natuurlijk ook in een andere expressie gebruikt worden.

```
/*
 * tabel.c : 1-dim rij afdrukken in tabel vorm
 */
void rij2tab(int a[], int n, int m)
{
    int i;

    for (i=0; i<n; i++)
        printf("%4d%c", a[i], ((i+1)%m ? ' ' : '\n') );
    if ( n%m )
        printf("\n");
}
```

11.1.2 De komma operator

Expressies kunnen onderling gescheiden worden met een komma. Deze komma wordt als een operator opgevat: de expressies worden van links naar rechts uitgewerkt en de laatste, dus de meest rechtse bepaalt de waarde van het geheel.

```
/*
 * invers.c : omdraaien van de elementen in een array
 */
void draai(int a[], int b[], int n)
{
    int i, j;

    for (i=0, j=n-1; i<n; i++, j--)
        b[j] = a[i];
}
```

Met deze operator kan men er ook voor zorgen dat bij een iteratie-lus de beëindigingstest niet in het begin en ook niet op het einde, maar ergens in het midden van de lus gebeurt.

```
/*
 * lustest.c : eindtest in het midden
 */
#include <stdio.h>
void main(void)
{
    int n;
    int som;

    som = 0;
    while ( scanf("%d%*c", &n), n>0 )
        som += n;
    printf("De som is %d\n", som);
}
```

De lus bestaat uit drie acties:

1. het lezen van een getal;
2. de test op het einde van de lus;
3. het verhogen van `som` met `n`.

Hetzelfde zou kunnen gerealiseerd worden met een `break`. Maar volgens sommigen is dit geen stijlvolle constructie.

Merk op dat de komma-operator een zeer lage prioriteit heeft:

```
int a = 5, b=8 ;
int r;

r = a++, b++;
```

11.2 De keywords van de taal

Sommige woorden zijn gereserveerd als "keyword"; zij hebben een vaste betekenis en mogen daarom niet als naam voor iets anders (bijvoorbeeld variabele) gebruikt worden.

Variabele declaratie :	`auto`	`char`	`const`	`double`	`extern`
	`float`	`int`	`long`	`register`	`short`
	`signed`	`static`	`unsigned`	`void`	`volatile`
Data type declaratie :	`enum`	`struct`	`typedef`	`union`	
Controle statement :	`break`	`case`	`continue`	`default`	
	`do`	`else`	`for`	`goto`	
	`if`	`return`	`switch`	`while`	
Operator :	`sizeof`				

11.2.1 Type qualifiers

Register. De geheugenklasse `register` meldt aan de computer dat de bijbehorende lokale variabelen opgeslagen moeten worden in high-speed geheugenregisters, mits zulks fysisch en semantisch mogelijk is. Dit heeft alleen zin als snelheid een essentiële factor is en voor die variabelen waarop het meest frequent operaties uitgevoerd worden.
Voorbeeld:

```
/*
 *   vbreg.c : programma met een register-variable
 */
#include <stdio.h>
#define LIMIET 1000

void main(void)
{
    register int i;      /* register variabele  */

    for (i=0; i<LIMIET; i++)
    {
        printf("%8d\n", i);
    }
}
```

De variabele `i` is een lokale variabele die omwille van efficiëntie in een register gestopt wordt. Voor de rest is het een *lokale* variabele. De register toekenning gebeurt wanneer de functie start en wordt ongedaan gemaakt bij het einde van de functie.

Auto. Omdat het aantal registers beperkt is, worden de meeste lokale variabelen ergens in het werkgeheugen bewaard. Wannneer de functie start, wordt een plaats toegewezen aan de variabele. Dit gebeurt automatisch, vandaar dat men soms ook van *automatische* variabelen spreekt. Men kan dit expliciet in de declaratie aangeven:

```
auto int    i;
auto double f;
```

Dit wordt echter bijna nooit gedaan, meestal wordt de `auto` qualifier gewoon weggelaten.

Daarnaast biedt ANSI-C de mogelijkheid een extra *type-kwalificatie* te geven:

const : de variabele kan na initialisering nooit gewijzigd worden (een alternatief voor een naamconstante); wordt ook vaak gebruikt bij het specificeren van formele parameters; een voorbeeld:

```
const int weeklengte = 7;
```

volatile : (te beschouwen als het omgekeerde van `const`) deze variabelen kunnen in principe ook gewijzigd worden door bijvoorbeeld de hardware (het gebruik is systeemafhankelijk).

Extern. Dit keyword wordt gebruikt bij de declaratie van globale variabelen en functies wanneer het programma uit verschillende bronbestanden bestaat.

11.2.2 Het `goto` statement

In *oude* programma's kan men soms nog het `goto` statement terugvinden:

```
goto identifier;
```

waarbij de `identifier` *label* genoemd wordt. Er moet namelijk in dezelfde functie een statement voorkomen dat voorafgegaan wordt door deze *label*.

```
void main(void)
{
        register int i = 0;
        const int n = 10;

    opnieuw:
        i++;
        ...
        if ( i > n )
            goto gedaan;
        goto opnieuw;
    gedaan:
        printf("Het is gedaan\n");
        exit(0);
}
```

Sinds de tweede helft van de jaren zeventig (opkomst van *gestructureerd programmeren*) wordt het ten stelligste afgeraden dit statement te gebruiken. Er bestaan steeds andere, meer *leesbare* manieren om het hetzelfde effect te bekomen.
Een `goto` statement is ook vrij beperkt: er kan alleen een sprong binnen een functie uitgevoerd worden. Men kan bijvoorbeeld niet uit een een hulpfunctie springen naar het einde van de functie `main`. Om de uitvoering van een programma (voortijdig) te beëindigen, kan men gebruik maken van de functie `exit()`. Tijdens de uitvoering van `exit` worden alle geopende bestanden netjes gesloten.

11.3 Pointers naar functies

Omdat functies, net als gegevens, worden opgeslagen in het werkgeheugen, hebben zij een beginadres. Het is in C mogelijk pointervariabelen te declareren waaraan het beginadres van een functie kan toegekend worden. Na deze toekenning kan de functie opgeroepen worden via de oorspronkelijke naam maar ook via de pointervariabele.

```
1   /*
     * funpoin.c : pointers naar functies
3    */
    #include <stdio.h>
5   #define PI 3.14159

7   float omtrek(float straal)
    {
9       return 2.0*PI*straal;
    }
11
    float opp(float straal)
13  {
        return PI*straal*straal;
15  }

17  void main(void)
    {
19      float (*pf)(float x);
        float r;
```

```
21          int    k;

23          printf("Geef de straal : ");
            scanf("%f%*c", &r);
25          printf("Omtrek=1  Oppervlakte=2 : ");
            scanf("%d%*c", &k);
27          pf = (k==1 ? omtrek : opp);
            printf("Het resultaat is %f\n", (*pf)(r) );
29  }
```

In dit voorbeeld is de pointer naar de functie niet echt nodig:

```
if ( k==1 )
    printf("Het resultaat is %f\n", omtrek(r) );
else
    printf("Het resultaat is %f\n", opp(r) );
```

Pointers naar functies hebben hun nut in grote programma's waar in het begin van het programma een keuze moet gemaakt worden uit een aantal functies en waar de gekozen functie op verschillende plaatsen in het programma moet opgeroepen worden. Door gebruik te maken van een pointer die naar de gekozen functie wijst, kan de functie via deze pointer telkens opgeroepen worden.
Daarnaast maakt een pointer naar een functie het ook mogelijk een functie als actuele parameter door te geven naar een andere functie.

Opgave. Schrijf een functie die de trapeziumregel voor de berekening van een benaderende waarde voor een bepaalde integraal implementeert. De functie heeft drie argumenten: de functie, de ondergrens en de bovengrens.

```
/*
 * funtrap.c : functie trapeziumregel
 */
#include <math.h>
#define  EPS  1.0e-4

double trapezium(double (*pf)(double), double a, double b)
{
    int     i, n;
    double stap, x;
    double integraal, vorig;

    n = 1;
    vorig = 0.0;
    integraal = ( (*pf)(a) + (*pf)(b) )/2.0;
    while ( fabs((vorig-integraal)/integraal) > EPS )
    {
        vorig = integraal;
        n *= 2;
        stap = (b-a)/n;
        x = a;
        integraal = ( (*pf)(a) + (*pf)(b) )/2.0;
        for (i=1; i<n; i++)
        {
            x += stap;
            integraal += (*pf)(x) ;
        }
        integraal *= stap;
```

```
        }
        return integraal;
    }
```

De waarde van de expressie `trapezium(sin, 0.0, 1.0)` wordt berekend door de functie `trapezium` op te roepen met drie argumenten: het beginadres van de functie `sin` en twee getallen. Het resultaat is gelijk aan 0.4597.

11.4 De `main` functie

Een programma bestaat uit een aantal functies en een hoofdfunctie `main`. Deze hoofdfunctie heeft een terugkeerwaarde van type `int` en kan ook parameters hebben.

```
1  /*
    * mainarg.c : programmaparameters
3   */
   #include <stdio.h>
5
   int main(int argc, char *argv[])
7  {
       int  i;
9
       printf("argc    = %d\n", argc);
11     for (i=0; i<argc; i++)
           printf("argv[%d] = %s\n", i, argv[i]);
13     return 0;
   }
```

Veronderstel na compilatie de uitvoerbare versie in het bestand `vbmp` zit. Wanneer het programma `vbmp` gestart wordt, door de naam ervan in te tikken, kunnen een willekeurig aantal argumenten bijgevoegd worden:

```
vbmp appel peer genoeg
```

De uitvoer is dan

```
argc    = 4
argv[0] = vbmp
argv[1] = appel
argv[2] = peer
argv[3] = genoeg
```

Tijdens het starten van het programma, heeft het systeem ervoor gezorgd dat er actuele argumenten gemaakt worden die doorgegeven worden aan de functie `main`. Er zijn twee argumenten:

argc : het aantal ingetikte woorden, inclusief de naam van het programma;

argv : een array van `argc` elementen, waarbij elk element een pointer is naar een string, en een bijkomend element met waarde `(char *)NULL`. Element i in deze array bevat het beginadres van het i-de ingetikte woord.

Dus `argv[0]` wijst naar een string met waarde de naam van het programma; `argv[1]` naar het eerste argument, ..., `argv[argc-1]` naar het laatste argument. Het laatste (NULL) element is nodig om het einde van de array aan te geven. Deze lengte kan niet op voorhand vastgelegd worden, omdat een willekeurig aantal argumenten ingetikt kunnen worden.

Opgave. Schrijf een programma dat een willekeurig aantal invoerbestanden copieert naar één uitvoerbestand.

```
/*
 * mcopie.c : resultaatfile infile1 infile2 infile3 ...
 */
#include <stdio.h>
#include <stdlib.h>
FILE *openen(char *nm, char *mod);

int main(int argc, char *argv[])
{
    FILE *fin;
    FILE *fuit;
    int  teken;
    int  i;

    fuit = openen(argv[1], "w");
    for (i=2; i<argc; i++)
    {
        fin = openen(argv[i], "r");
        while ( (teken=fgetc(fin)) != EOF )
            fputc(teken, fuit);
        fclose(fin);
    }
    fclose(fuit);
    return 0;
}

FILE *openen(char *naam, char *mode)
{
    FILE *res;
    res = fopen(naam, mode);
    if ( res == NULL )
    {
        printf("Probleem met openen van %s\n", naam);
        exit(1);
    }
    return res;
}
```

11.5 De preprocessor

11.5.1 Directieven

Voordat de compiler het echte werk doet, wordt het bronbestand behandeld door de preprocessor. De C-preprocessor behandelt die lijnen die in de eerste kolom beginnen met het `#` symbool. Volgende *compiler control lines* zijn onder meer beschikbaar.

define : hiermee kan aan een symbolische naam een constante toegekend worden; daarnaast kan hiermee ook een *macro* gedefinieerd worden.

undef : aangeven dat de compiler vanaf dan de definitie van de bijhorende symbolische naam moet vergeten.

include : geeft aan dat het bestand moet tussengevoegd worden in het bronbestand. Wanneer de bestandsnaam tussen dubbel quotes (") staat, wordt het bestand eerst gezocht in de huidige directory en daarna in de *algemeen toegankelijke* directories (systeemafhankelijk). Een naam tussen <> geeft aan dat het bestand alleen in de algemeen toegankelijke directories moet gezocht worden. Deze bestanden worden dikwijls *header* bestanden genoemd, vandaar dat ze meestal een extensie `.h` hebben.

ifdef : om het compileren van bepaalde gedeelten van de programmatekst te laten afhangen van een voorwaarde (*conditionele compilatie*).

line : met twee argumenten: het eerste argument is een getal vanaf waar de compiler de volgende programmaregels zal nummeren; het tweede argument is een naam die vanaf dan de compiler als naam van het bestand zal gebruiken.

11.5.2 Macro's

Een macro is te vergelijken met een functie. Er is een macro-definitie

```
#define max(a,b) ((a) > (b) ? (a) : (b))
```

en een macro-oproep

```
y = 2 * max(i+1, j-1) + 3;
```

Door de preprocessor zal de macro-oproep (*in line*) vervangen worden door de macro-definitie (*macro-expansie*):

```
y = 2 * ((i+1) > (j-1) ? (i+1) : (j-1)) + 3;
```

De C-compiler krijgt dus een expressie te verwerken. Er wordt geen functie-oproep met doorgeven van argumenten (naar parameters) uitgevoerd.
Merk op dat er in de macro-definitie nogal wat haakjes gebruikt worden. Dit heeft te maken met de prioriteit van de operatoren. Met de definitie

```
#define kwadraat(x) x * x
```

en de oproep

```
x = kwadraat(a+b);
```

wordt de macro-expansie

```
x = a+b * a+b;
```

Dit is echter niet de bedoeling. Er zijn dus haakjes nodig

```
#define kwadraat(x) ((x) * (x))
```

De macro-expansie wordt dan

```
x = ((a+b) * (a+b));
```

Het open-haakje dat het begin van de parameters aangeeft, moet direct volgen op de macro-naam (zonder spatie). Anders zou de parameterlijst gebruikt worden als het eerste deel van de definitie.

Wanneer een macro-definitie te lang wordt voor één lijn, dan kan de definitie gesplitst worden over meerdere lijnen. Dit gebeurt door de eerste lijn te eindigen met een backslash (\).

In het headerbestand `<ctype.h>` zijn een aantal macro's gedefinieerd om het type karakter te testen. Omdat op karaktervariabelen bewerkingen kunnen uitgevoerd worden zijn deze macro's vrij eenvoudig. Bijvoorbeeld:

```
#define isdigit(c) ((c)>='0'&&(c)<='9')
#define tolower(c) ((unsigned char)(c)-'A'+'a')
```

11.5.3 Conditionele compilatie

```
#define VROEGER
void main(void)
{
    ...
#ifdef VROEGER
    /* programmadeel A  */
#else
    /* programmadeel B  */
#endif
    ...
}
```

Wanneer `VROEGER` gedefinieerd is (zoals in het voorbeeld), zal de compiler niet programmadeel B compileren; alleen programmadeel A wordt opgenomen in de uitvoerbare versie. Wanneer `VROEGER` niet gedefinieerd zou zijn, zou alleen programmadeel B gecompileerd worden.

11.6 Begrippen

- Operatoren : conditionele en komma.
- Keywords: type qualifiers (`register`, `auto`, `const`, `volatile`) en `goto`.
- Pointers naar functies.
- De `main` functie: terugkeerwaarde en parameters.
- Preprocessor: `define`, `include`, `ifdef`, `line`.

12 Ontwerpen van programma's

12.1 Ontwerp

Voor het ontwerpen en schrijven van computerprogramma's zijn twee zaken nodig:

- een goed begrip van de elementen van een programmeertaal;
- het begrijpen van de definitie en structuur van het probleem dat moet opgelost worden of de taak die moet uitgevoerd worden.

In de vorige hoofdstukken werden de basiselementen van de programmeertaal C uitgelegd. Daarnaast zijn dus technieken nodig om een probleem te analyseren en daaruit een oplossing af te leiden welke kan omgevormd worden tot een programma.
Volgens G. Polya (eind jaren veertig) zijn er 4 stappen nodig om een algoritme te ontwerpen:

1. Begrijp het probleem.
2. Tracht een idee te vormen over hoe een algoritmische procedure het probleem zou kunnen oplossen.
3. Formuleer het algoritme en schrijf het neer als een programma.
4. Evalueer het programma op nauwkeurigheid en op de mogelijkheden om het als middel te gebruiken bij het oplossen van andere problemen.

Voor de meeste problemen moet de ontwerper een iteratieve oplossingsbenadering toepassen. Beginnen met een vereenvoudigd probleem en hiervoor een betrouwbare oplossing construeren. Daardoor krijgt men meer vat op het probleem. Dit beter begrijpen kan dan gebruikt worden om meer details in te vullen en een verfijnde oplossingsprocedure uit te denken, totdat men komt tot een bruikbare oplossing voor het realistische originele probleem.
De eerste twee stappen kunnen in praktijk zeer moeilijk zijn en in het geval van enkele wiskundige problemen, zal de oplossing equivalent zijn met het ontdekken en bewijzen van nieuwe stellingen voordat een computerprogramma kan geschreven worden. Het is bijvoorbeeld tot op dit moment niet geweten of volgende procedure altijd zal stoppen voor een willekeurige initiële waarde.

```
1  /*
   *  stop.c :  stopt dit programma ?
3  */
   #include <stdio.h>
5  int main(int argc, char *argv[])
   {
7       int getal;
        int stap = 0;
9
        getal = atoi(argv[1]);
11      while ( getal != 1 )
        {
13          if ( getal%2 == 0 )
                getal = getal/2;
15          else
                getal = 3*getal+1;
17          stap++;
        }
19      printf("programma stopt na %d stappen\n", stap);
        return 0;
21 }
```

Niet alle problemen kunnen opgelost worden door middel van computerprogramma's. En zelfs wanneer een algoritme bestaat, is het niet altijd mogelijk dit algoritme te implementeren in een programma omwille van beperkingen van de grootte van het werkgeheugen en de beschikbare rekentijd.

12.2 Structuur van een programma

Men kan weinig algemene richtlijnen geven om een programma te structureren, omdat de structuur afhankelijk is van het origineel probleem en het algoritme dat gebruikt wordt. Toch is het nuttig om een programma op te delen in drie fazen:

Initialisatie : declaratie en initialisatie van de variabelen. Het is nuttig om alle beschikbare informatie in te voeren *voordat* de berekeningen starten, zodat controles op juistheid van gegevens kan gebeuren. Een complexe simulatie uitvoeren kan uren duren en het zou spijtig zijn dat het programma halverwege vastloopt omdat de gebruiker een foutieve invoer doet.

Data verwerking : het grootste gedeelte van het programma. De implementatie van de oplossingsprocedure met behulp van functies, iteratie-lussen en keuze-statements.

Stockeren van de resultaten : op het einde van het programma. Het uitschrijven van de resultaten naar het beeldscherm en/of naar een bestand.

Opgave. Schrijf een programma dat de elementen van een vierkante matrix invult zodat de som van elke rij gelijk is aan de som van elke kolom (een *magisch* vierkant).

Algoritme. In het geval dat het aantal rijen (kolommen) oneven is, bestaat er een eenvoudige methode. De getallen $1, 2, \ldots, n^2$ worden ingevuld waarbij de plaats op de volgende manier berekend wordt:

1. begin in het midden van de laatste rij;
2. het volgend element wordt ingevuld linksboven het huidig element;
3. tenzij buiten de grenzen van het vierkant gegaan wordt, in dat geval is het de laatste rij en/of laatste kolom;
4. indien de plaats reeds ingevuld is, dan wordt het volgende element ingevuld net onder het huidig element;
5. tenzij buiten de grens van het vierkant gegaan wordt, in dat geval is het de eerste rij.

```
1   /*
     * vier.c  : magisch vierkant
3    */
    #include <stdio.h>
5   #include <stdlib.h>

7   #define NMAX 16

9   int hoegroot(void);
    void invullen(int a[][NMAX], int n);
11  void afdrukken(int x[][NMAX], int m);

13  int main(int argc, char *argv[])
    {
15      int     a[NMAX][NMAX];          /* het vierkant */
        int     n;                      /* grootte vierkant */
17
        n = hoegroot();
```

```
19          invullen(a,n);
            afdrukken(a,n);
21      }

23      int hoegroot(void)
        {
25          int     z;

27          printf("grootte van het vierkant : ");
            scanf("%d%*c", &z);
29          if ( !( (z%2) && z<NMAX ) )     /* of if ( !(z%2) || z>=NMAX ) */
            {
31              fprintf(stderr, "grootte %d is niet oneven of te groot\n", z);
                exit(1);
33          }
            return z ;
35      }

37      void invullen(int a[][NMAX], int n)
        {
39          int     n2;                 /* kwadraat van n           */
            int     k;                  /* in te vullen waarde      */
41          int     vi, i;              /* rij index (en vorige)    */
            int     vj, j;              /* kolom index (en vorige)  */
43
            for (i=1; i<=n; i++)
45              for (j=1; j<=n; j++)
                    a[i][j] = 0;
47          n2 = n*n;
            k = 1;
49          j = n/2 + 1;                    /* begin in het midden     */
            i = n;                          /*      van de laatste rij */
51          while ( k <= n2 )
            {
53              a[i][j] = k++;
                vj = j--;                   /* ga voort naar links     */
55              vi = i--;                   /*           en naar boven */
                if ( i == 0 )               /* indien buiten vierkant  */
57                  i = n;                  /*        naar laatste rij */
                if ( j == 0 )               /* indien buiten vierkant  */
59                  j = n;                  /*      naar laatste kolom */
                if ( a[i][j] )              /* indien reeds ingevuld   */
61              {
                    i = vi + 1;             /* naar volgende rij      */
63                  if ( i > n )               /* indien buiten vierkant */
                        i = 1;                 /*    naar eerste rij     */
65                  j = vj;                    /*    in dezelfde kolom   */
                }
67          }
        }
69
        void afdrukken(int x[][NMAX], int m)
71      {
            int     i, j;
```

```
73          int       rijsom;
            int       kolomsom[NMAX];
75
            printf("\nOplossing:\n");
77          for (j=1; j<=m; j++)
                kolomsom[j] = 0;
79          for (i=1; i<=m; i++)
            {
81              rijsom = 0;
                for (j=1; j<=m; j++)
83              {
                    printf("%4d", x[i][j]);
85                  rijsom += x[i][j];
                    kolomsom[j] += x[i][j];
87              }
                printf(" : %6d\n", rijsom);
89          }
            rijsom = 0;
91          for (j=1; j<=m; j++)
            {
93              printf("%4d", kolomsom[j]);
                rijsom = rijsom + kolomsom[j];
95          }
            printf(" : %6d\n", rijsom);
97      }
```

Het resultaat voor $n = 3$:

7	5	3
2	9	4
6	1	8

Deze matrix wordt in het werkgeheugen als één lange rij gestockeerd:

?	7	5	3	$\cdots$	?	2	9	4	$\cdots$	?	6	1	8	$\cdots$

Het resultaat voor $n = 5$ (de som is telkens 65):

16	14	7	5	23
22	20	13	6	4
3	21	19	12	10
9	2	25	18	11
15	8	1	24	17

Het resultaat voor $n = 7$ (de som is telkens 175):

29	27	18	9	7	47	38
37	35	26	17	8	6	46
45	36	34	25	16	14	5
4	44	42	33	24	15	13
12	3	43	41	32	23	21
20	11	2	49	40	31	22
28	19	10	1	48	39	30