C-taal voor beginners - hoofdstuk 1

identifiers, declaraties, types,scherminput/output

1.1 identifiers

Vooraleer ook maar iets in C te programmeren moet u een identifier kunnen benoemen of declareren. Een identifier wordt gebruikt voor variabelen, functies, data definities, enz. Het declareren gebeurt voordat de variabele gebruikt wordt, zodat het programma weet wat ermee aan te vangen. In C is een identifier een combinatie van alfanumerieke karakters, met als eerste letter een alfabetisch teken of een underline.

Belangrijk:

- identifiers zijn hoofdletterafhankelijk: produkt is niet hetzelfde als proDukT...
- de meeste compilers zullen maximum 31 tekens als significant beschouwen voor een variabele

Wat spaties, lege lijnen, enz. betreft kan u bijna doen wat u wilt. Tijdens het compileren worden de lege ruimtes als het ware weggegooid. Het is echter wel aan te raden na elke nieuwe opdracht een nieuwe regel te beginnen en inspringingen en dergelijke te gebruiken. Dit vergemakkelijkt het lezen voor uzelf én voor degenen die het misschien ooit zullen moeten aanpassen. Hoofdstuk 10 spitst zich toe op stijlen die u het best kan gebruiken bij het programmeren in C.

1.2 standaardwoorden in C

C op zichzelf herkent slechts 32 woorden. Deze zijn voorgeprogrammeerd en kunnen niet voor andere zaken gebruikt worden. Ze moeten in kleine letters toegepast worden. De lijst:

auto	**double**	**int**	**struct**
break	**else**	**long**	**switch**
case	**enum**	**register**	**typedef**
char	**extern**	**return**	**union**
const	**float**	**short**	**unsigned**
continue	**for**	**signed**	**void**
default	**goto**	**sizeof**	**volatitle**
do	**if**	**static**	**while**

1.3 een eerste programma

```
/* 1.3.1.c */
/* Dit is mijn eerste programma */
#include<stdio.h>
int main(void)
{
    printf("Hallo wereld, dit is mijn eerste programma!\n");
    return 0;
}
```

Dit simpele programma zal de zin afdrukken die tussen de dubbele quotes **" "** staat.

Belangrijke componenten:

- alles wat tussen **/* */** staat is commentaar. De compiler zal er niet op letten. U zal dit voornamelijk gebruiken om duidelijkheid in het programma op te nemen. In feite kunnen commentaarlijnen overal staan, zelfs middenin identifiers... .
- **#include<stdio.h>** voegt de code toe om de standaard I/O bibliotheek te kunnen gebruiken. Dit is hier nodig omdat die het commando **printf()** bevat, dat toelaat om uitvoer op het scherm te drukken. We zeggen dat **stdio.h** een header-file is.
- **main()** omvat het eigenlijke programma. Alles wat tussen de **{ }** staat behoort tot de functie **main()**. U kan vele verschillende functies en functienamen gebruiken, maar één functie moet de naam **main** hebben.
- **printf()** zorgt ervoor dat hetgeen tussen **" "** staat op het scherm afgedrukt wordt, (voor meer zie 1.7).
- Let op de puntkomma **;** die gebruikt wordt om een commando af te sluiten.
- De **int** voor de **main** en de **void** erachter en de **return 0;** zal later worden uitgelegd, plaats het er nu gewoon bij.

1.4 variabelen definiëren

```
/* 1.4.1.c */
/* Dit programma berekent de som van 2 getallen */
#include<stdio.h>
int main(void)
{
int som;
int getal1;
int getal2;

    printf("Geef het eerste getal: ");
    scanf("%d",&getal1);

    printf("Geef het tweede getal: ");
    scanf("%d",&getal2);

    som=getal1+getal2;

    printf("De som van %d en %d is %d\n",getal1,getal2,som);
    return 0;
}
```

Het programma zal twee getallen vragen, daarvan de som berekenen en afdrukken op het scherm.

U merkt een paar nieuwe dingen op:

```
int som;
int getal1;
int getal2;
```

op deze manier wordt een variabele gedefinieerd. M.a.w. we zeggen tegen het programma wat voor type variabelen het zijn. In dit geval is dat van het type integer. **Som**, **getal1** en **getal2** zijn de desbetreffende variabelen. De waarde van de getallen wordt door het programma opgenomen van de gebruiker met het commando **scanf("%d",&getal1)**. Dit leest een waarde in en wijst die toe aan de variabele **getal1**. Met **%d** duiden we aan dat het om een integer gaat. Let ook op het **&** teken (ampersand). Dit wil letterlijk zeggen "adres van". (zie 1.6)
Merk op dat variabelen van hetzelfde type ook in één lijn mogen gedeclareerd worden.

```
int som;
int getal1;
int getal2;
```

is hetzelfde als:

```
int som, getal1, getal2;
```

Ook is het mogelijk om een integer in de declaratie al een waarde te geven, zoals in de volgende voorbeelden getoont wordt:

```
int som, getal1 = 10, getal2;
int som, getal1 = 10, getal2 = 300;
float getal = 1.34542;
```

Volgende types variabelen zijn mogelijk:

1.4.1 constanten: const

Constanten leveren een manier om een variabele te definiëren die nergens anders in het programma kan aangepast worden:

decimaal: van 0 tot 9, voorgesteld met **%d**
octaal: van 0 tot 7, voorgesteld met **%o** en aangeduid met 0, bv: 0357 om het octale getal 357 aan te geven
hexadecimaal: van 0 tot F, voorgesteld met **%x** en aangeduid met 0X of 0x, bv: 0X1A om het hexadecimale getal 1A aan te geven en 0x1A om het hexadecimale getal 1a aan te geven

1.4.2 integer: int

16 bits: van -32768 tot 32767
32 bits: heeft een dubbel bereik

1.4.3 floating point (met enkelvoudige precisie): float

[ref. C FAQ - Sectie 14]

dit type laat toe om (grote) kommagetallen te gebruiken, bv: 95846.9478573532
let op: een komma wordt voorgesteld door een punt **"."**

1.4.4 floating point met dubbele precisie: double

voor grotere **floats** met grotere precisie
een **double** gebruikt dubbel zoveel geheugen dan een gewone **float**

1.4.5 karakters: char

één teken tussen enkele quotes **' '**. Toegelaten karakters zijn degenen uit de ASCII code (decimaal tussen 0 en 127). Sommige karakters kunnen niet voorgesteld worden zonder de '\', maar dit wordt dan nog steeds als één karakter bekeken. Een uitzondering is '\n' dat

op het Dos systeem 2 tekens beslaat, maar op UNIX maar 1. In feite kan een karakter variabele ook een getal bevatten.

1.5 qualificators

Daarnet zag u dat u de precisie van een **float** variabele kunt verhogen door aan de variabele het type **double** toe te wijzen. Het is ook mogelijk om het waardenbereik van een integer variabele te regelen. Volgende qualificators zijn mogelijk. De aangeduide geheugenruimtes kan u niet strikt nemen (zie 1.6):

1.5.1 long

voor de typespecificator **int**, geeft de gedeclareerde integer een uitgebreid waardenbereik.

```
bv:    long int groot;
       long groot;
```

Een **long** gebruikt 4 bytes geheugenruimte.

1.5.2 short

voor de typesecificator **int** geeft de genoemde variabele een beperkt waardenbereik. Dit wordt vooral gebruikt om geheugenruimte te sparen.

```
bv: short int teller;
    short teller;
```

Een **short** gebruikt 2 bytes geheugenruimte.

1.5.3 unsigned

Geeft aan dat de integer variabelen tot positieve waarden beperkt zijn.
Aangezien de ruimte voor negatieve getallen niet nodig is, heeft u de mogelijkheid tot dubbel zoveel positieve getallen. Als een signed type variabele bijvoorbeeld een maximum bereik heeft tot 32767, dan heeft hetzelfde unsigned type variabele een bereik tot 65535.

```
bv: unsigned int geheel;
    unsigned geheel;
```

Een **unsigned** gebruikt 4 bytes geheugenruimte.

De unsigned specificator beperkt zich niet alleen tot integers. Dingen als **unsigned char**, **unsigned short** en **unsigned long** zijn even goed mogelijk.

1.6 geheugengebruik: bits en bytes

Bij de beschrijving van de verschillende types hierboven mag u de aanduidingen voor het geheugengebruik niet strikt nemen. Deze zullen (meestal) wel gelden voor het Windows besturingssysteem, draaiende op een intel PC. Je kunt er niet vanuit gaat dat een integer 16 of 32 bits is. Onder DOS is een integer meestal 16-bits en een long 32-bits. Onder Windows is een integer meestal 32-bits en een long 32-bits. Er zijn echter ook systemen waar een long 64-bits is. Het komt er dus op neer dat

als u het exacte geheugengebruik op uw specifieke machine wilt weten, dan zal u uw computerdocumentatie moeten raadplegen (zie ook de macro-methode hieronder):

De representaties en waardes van de verschillende types van integers:

type	16-bit bytes	16-bit minimum	16-bit maximum	32-bit bytes	32-bit minimum	32-bit maximum
char	1	-128	127	1	-128	127
signed char	1	-128	127	1	-128	127
unsigned char	1	0	255	1	0	255
signed short	2	-32768	32767	2	-32768	32767
unsigned short	2	0	65535	2	0	65535
signed int	2	-32768	32767	4	-2147483648	2147483647
unsigned int	2	0	65535	4	0	4294967295
signed long	4	-2147483648	2147483647	4	-2147483648	2147483647
unsigned long	4	0	4294967295	4	0	4294967295
float	4	3.4 E-38	3.4 E+38	4	3.4 E-38	3.4 E+38
double	8	1.7 E-308	1.7 E+308	8	1.7 E-308	1.7 E+308
long double	10	1.2 E-4923	1.2 E+4923	10	1.2 E-4923	1.2 E+4923

De standaard ANSI-header **limits.h** bevat een aantal macro's (voor een uiteenzetting over macro's, zie later hoofdstuk), die erg nuttig kunnen zijn bij het bepalen van de minimum en maximumwaardes. Een overzicht:

CHAR_BIT	Aantal bits in een byte
SCHAR_MIN	Minimum waarde voor een signed char
SCHAR_MAX	Maximum waarde voor een signed char
UCHAR_MAX	Maximum waarde voor een unsigned char
CHAR_MIN	Minimum waarde voor een char
CHAR_MAX	Maximum waarde voor een char
SHRT_MIN	Minimum waarde voor een short int
SHRT_MAX	Maximum waarde voor een short int
USHRT_MAX	Maximum waarde voor een unsigned short int
INT_MIN	Minimum waarde voor een int
INT_MAX	Maximum waarde voor een int
UINT_MAX	Maximum waarde voor een unsigned int
LONG_MIN	Minimum waarde voor een long int
LONG_MAX	Maximum waarde voor een long int
ULONG_MAX	Maximum waarde voor een unsigned long int

Dit kan u dan bijvoorbeeld als volgt toepassen:

```
printf("Aantal bits in een byte is %d en de maximum waarde voor
short int is %d\n",
 CHAR_BIT, SHRT_MIN);
```

Om het aantal bytes van een expressie of type te achterhalen gebruiken we het **sizeof** sleutelwoord:

```
printf("Het aantal bytes in een integer is %d.\n", sizeof(int));
```

1.7 standaard invoer- en uitvoer

1.7.1 printf

U zag reeds eerder het gebruik van **printf()** in het voorbeeldprogramma. Deze functie bevindt zich in het **stdio.h** header-file en zorgt voor een output op het scherm. Alles tussen **" "** noemt men de string en wordt op het scherm getoond; **\n** zorgt ervoor dat er een nieuwe regel begonnen wordt. Voorbeeld:

```
printf("Dit is de eerste regel\n");
printf("Dit is de tweede regel\n");
```

geeft als output:

```
Dit is de eerste regel
Dit is de tweede regel
```

maar:

```
printf("Dit is de eerste regel");
printf("Dit is de tweede regel");
```

geeft als output:

```
Dit is de eerste regelDit is de tweede regel
```

Om tekens weer te geven die in de gewone code moeilijk of niet weer te geven zijn, kan u gebruik maken van escape sequenties. Een escape sequentie begint met een backslash **** gevolgd door één karakter:

\a	waarschuwingsbel
\r	carriage return
\n	regelovergang
\t	horizontale tab
\v	verticale tab
\f	form feed
\b	backspace
\'	enkel aanhalingsteken
\''	dubbel aanhalingsteken
\\	backslash
\?	vraagteken

U zag ook reeds eerder op welke manier een variabele opgenomen wordt in het argument, namelijk met het **%** teken gevolgd door een karakter, afhankelijk van het type variabele. Na de string - dat is na de dubbele quotes **" "** -staat er telkens een komma en daarna de naam van de variabele waarop het procentteken en karakter betrekking hebben.
Voorbeeld:

```
int prijs = 157;
int aantal = 23;
printf("U kocht %d producten tegen de prijs van %d
Bef.\n",aantal,prijs);
```

dit levert u het volgende op:

```
U kocht 23 producten tegen de prijs van 157 Bef.
```

Enkele andere mogelijke voorbeelden:

```
int getal1 = 10, getal2 = 300;
product = getal1 * getal2;
printf("Het produkt van %d en %d is %d\n",getal1,getal2,product);
```

```
float getal, opl;
getal = 20.43234;
opl = getal / 5.0;
printf("Het kommagetal %f gedeeld door 5 geeft %f\n",getal,opl);
```

```
int aantal;
float prijs, bedrag;
aantal = 234;
bedrag = 12.5;
prijs = aantal * bedrag;
printf("De totale kostprijs: %f",prijs);
```

Om het procentteken weer te geven in een string moet u het tweemaal naast elkaar plaatsen: **%%**:

```
printf("De BTW bedraagt op het moment 19%%. Dat is minder dan
vorige maand.\n");
```

1.7.2 scanf

Scanf is de theoretische tegenhanger van **printf()**, om output op het scherm te tonen. De syntax verschilt nauwelijks:

```
scanf("%d", &getal);
```

Nieuw is het adresteken **&**. Dit wil eenvoudig uitgedrukt zeggen: "adres van". De waarde van de variabele wordt niet meegegeven, wel het adres ervan zodat **scanf()** de waarde kan vervangen. Dit is nu niet belangrijk. Wanneer we later met pointers gaan werken wordt de betekenis van geheugenplaatsen duidelijker.

De functie **scanf()** leest een invoerlijn in, totdat er een eerste dataveld gevonden wordt. Leidende blanco's worden genegeerd en de waarde wordt ingelezen tot aan de volgende blanco of illegaal karakter, dan stopt het lezen en wordt de waarde teruggestuurd. Onthoudt dat **scanf()** niets doet, vooraleer er een return gegenereerd wordt, zelfs niet als er verschillende spaties ingegeven worden [ref. C FAQ - Sectie 12, Punt 18]. Om input op te nemen en te gebruiken kunnen we als volgt tewerk gaan:

```
/* 1.7.2.1.c */
#include<stdio.h>
int main(void)
{
int prijs;
float te_betalen, BTW, totaal;
    printf("Geef de prijs van het artikel in -> ");
    scanf("%d", &prijs);
    printf("Wat is het BTW tarief? ");
    scanf("%f", &BTW);
    te_betalen = prijs * BTW / 100;
    printf("De te betalen BTW is %f \n", te_betalen);
    totaal = te_betalen + prijs;
    printf("Het totaal te betalen bedrag is %f\n", totaal);
    return 0;
}
```

1.7.3 getchar

De functie **getchar()** leest één enkel karakter in van het standaard invoerapparaat, het toetsenbord:

```
/* 1.7.3.1.c */
#include<stdio.h>
int main(void)
{
int karakter;
    printf("Geef een karakter in -> ");
    karakter = getchar();
    printf("U gaf het volgende karakter in: %c", karakter);
    return 0;
}
```

Let op: een variabele die gebruikt wordt met een **getchar()**, is van het type integer. Om het karakter te tonen gebruiken we de **%c** specificatie, niettegenstaande de integer declaratie. Het gebruik van **%d** zou natuurlijk een decimale waarde tonen.

1.7.4 putchar

De tegenhanger **putchar()**, gebruikt het standaard uitvoerapparaat om één enkel karakter te tonen. Merk op dat na de uitvoer van het karakter de cursor reeds een plaats opschuift, klaar voor het volgende karakter. Op deze manier bespaart de compiler ons een paar, anders noodzakelijke, ingrepen. Voorbeeld:

```
/* 1.7.4.1.c */
#include<stdio.h>
int main(void)
{
int karakter;
    printf("Geef een karakter in -> ");
    karakter = getchar(); getchar();
    printf("U gaf het volgende karakter in: ");
    putchar(karakter);
    return 0;
}
```

Let op de tweede **getchar()**. Dit is een standaardmethode om de return, die volgt na het ingeven van het karakter, op te vangen en als het ware tegen te houden. Op die manier kan deze bij een volgende **getchar()** geen problemen geven.

1.7.5 lengtespecificatie

Alhoewel er een reeks opties zijn om uitvoer en invoer te formatteren zal ik op dit moment enkel de lengtespecificatie bespreken. Met de lengtespecificatie kan de manier waarop iets uitgevoerd wordt aangepast worden:

printf("De oplossing is %5d\n", opl);	voorziet een veldlengte van 5 bytes
printf("De som is %.4f\n, som);	toont de inhoud van de float variabele met 4 bytes na de komma
printf("De BTW bedraagt %5.2f\n, btw);	een totaallengte van 5, met 2 plaatsen na de komma
printf("Het getal is %.0f\n", getal);	toont de inhoud van de float variabele zonder cijfers na de komma

Ook bij de invoer kan men de lengtespecificatie toepassen. Een voorbeeld:

```
printf("Geef een cijfer in -> ");
scanf("%1d", &cijfer);
```

In dit geval zal enkel het eerste teken van het antwoord in de variabele cijfer gestoken worden. Het is echter duidelijk dat hier even goed -of zelfs beter- een **getchar()** kan gebruikt worden.

Opmerking: mintekens, decimale punten, enz... nemen ook een byte in beslag. Wanneer de inhoud van een variabele groter is dan de breedte die voorzien werd, zal deze toch volledig getoond worden.

1.8 typecasting

Typecasting laat toe om een variabele zich te laten gedragen als een variabele van een ander type. De methode van typecasting wordt gedaan door de variabele vooraf te gaan door het gewenste type tussen ronde haken. De variabele op zich wordt niet aangepast [ref. C FAQ - Sectie 3, Punt 14] [ref. C FAQ - Sectie 4, Punt 5].

Voorbeeld:

```
float getal1 = 30;
int getal2 = (int)getal1;
```

Dit voorbeeld typecast **getal1** om zich te gedragen als een integer getal.

---

opdrachten

1.1 Schrijf een programma dat twee getallen vraagt, deze dan van plaats verwisseld en op het scherm toont.

1.2 Schrijf een programma dat een integer, een karakter en een float vraagt en deze dan in één string op het scherm toont.

1.3 Schrijf een programma dat de volgende 3 zinnen drukt:

```
Een tab is \t
Een backspace is \b
Een waarschuwingsbel is \a
Hij zei: "C is 10% verstand en 90% oefenen".
```

1.4 Schrijf een programma dat de prijs excl BTW vraagt en daarna de prijs incl BTW berekent en weergeeft. Het BTW-percentage is 20.5% en is een constante.

1.5 Schrijf een programma om de verschillende mogelijkheden van de lengtespecificatie aan te tonen. Experimenteer voldoende met deze vorm van geformatteerde uitvoer en invoer.

1.6 Experimenteer voldoende met alle onderwerpen uit dit hoofdstuk, om een goede basis te hebben voor de volgende delen van de cursus.

C-taal voor beginners - hoofdstuk 2

programmacontrole

2.1 operatoren

In C zijn o.a. de volgende operatoren gedefinieerd:

relationele operatoren	
<	kleiner dan
>	groter dan
==	gelijk aan (twee gelijk aan tekens naast elkaar)
<=	kleiner dan of gelijk aan
>=	groter dan of gelijk aan
!=	niet gelijk aan
logische operatoren	
&&	EN
||	OF
!	NIET
bit operatoren	
&	AND
|	OR
^	XOR
~	INVERT
rekenkundige operatoren	
+	plusteken/optellen
-	minteken/aftrekken
*	vermenigvuldigen
/	delen
%	rest

Onthoudt dat in C de rekenregels i.v.m. **( ), +, -, *, /** dezelfde zijn als degene die u vroeger leerde in de wiskundeles. Eerst wordt hetgeen tussen haakjes uitgevoerd, dan de vermenigvuldiging en de deling, nadien de optelling en de aftrekking. Wat nieuw is is de rest-operator %. Deze geeft de rest van een deling:

22 % 5	geeft 2
20 % 4	geeft 0
-18 % 4	geeft -2
18 % -4	geeft 2

U merkt dat de rest hetzelfde teken heeft als het deeltal.

2.1.1 de toekenningsoperator

We gebruiken het gelijkheidsteken om een waarde toe te kennen aan een variabele:

```
getal1 = 200;
getal2 = getal1 + 500000;
som = getal1 + getal2;
snelheid = afstand / tijd;
intrest = som * rente * looptijd / 100;
```

De waarden worden toegekend van rechts naar links. Bij wijze van initialisatie kan u de volgende methode gebruiken om een boel variabelen in één keer op dezelfde waarde te zetten:

```
int g1, g2, g3, g4, g5;
g1 = g2 = g3 = g4 = g5 = 150;
```

2.1.2 de samengestelde toekenningsoperator

In C komt het veel voor dat aan de waarde van een variabele een waarde wordt toegekend die het resultaat is van een bewerking waar diezelfde variabele in voorkomt. Omdat dit nogal ingewikkeld klinkt een paar voorbeelden:

```
teller = teller + 2;
teller = teller - 1;
getal = getal * getal2;
rest = rest % 5;
deler = deler / 2.4;
```

Dit is niet echt een functionele manier van werken. Daarom werd de samengestelde toekenningsoperator voorzien, welke het best bij wijze van voorbeeld uitgelegd wordt:

oude manier	**nieuwe manier**
teller = teller + 2;	teller+=2;
teller = teller - 1;	teller-=1;
getal = getal * getal2;	getal*=getal2;
rest = rest % 5;	rest%=2.4;
deler = deler / 2.4;	deler/= 2.4;

Eerst wordt de expressie uitgevoerd, daarna de operatie:

```
getal*=3+9- 2
```

is evenredig met

```
getal = getal * (3 + 9 - 2)
```

maar niet met

```
getal = getal * 3 + 9 - 2
```

2.1.3 incrementie en decrementie

Er is nog een manier om tijd te besparen in C. Een variabele met 1 verhogen deden we tot hier toe op de volgende manier:

```
getal = getal + 1
```

Ook mogelijk is het volgende:

```
getal += 1
```

Vanaf nu maken we gebruik van de incrementoperator:

pre-incrementie	**post-incrementie**
++getal	getal++

In beide gevallen wordt bij de variabele 1 bijgeteld. Het verschil tussen de twee wordt duidelijk als er een andere expressie bijkomt:

```
getal + = ++getal2;
getal + = getal2++;
```

In het eerste geval wordt de variabele **getal2** verhoogd en daarna bij de variabele **getal** opgeteld. In het tweede geval wordt eerst **getal2** bij **getal** opgeteld, daarna wordt **getal2** verhoogd.

Op dezelfde manier is er de decrementoperator, om van de variabele de waarde 1 af te trekken.

pre-decrementie	**post-decrementie**
--getal	getal--

2.2 conditionele statements

Om bepaalde voorwaarden en dergelijke te testen maken we gebruik van de volgende conditionele statements. Deze zijn de werkelijke kracht van de C-taal. In elk degelijk programma komen er verschillende voor. Vaak zorgen ze er voor dat een stuk code korter en duidelijker wordt.

2.2.1 if

Het **if()** statement begint met het sleutelwoord **if** en wordt gevolgd door een expressie tussen ronde haken **( )**. Deze expressie (voorwaarde) wordt geëvalueerd en indien waar, wordt de uitdrukking op de volgende lijn uitgevoerd. Indien de voorwaarde niet waar is wordt de uitdrukking overgeslagen:

```
if(getal == 1000)
    printf("Het getal is duizend.\n");
```

Hier wordt dus gecontroleerd of de waarde van de variabele **getal** gelijk is aan duizend. Merk op dat direkt na de voorwaarde geen puntkomma **;** mag staan. In dit voorbeeld hebben we slechts één uitdrukking die uitgevoerd moet worden als de voorwaarde waar is. Meestal zal u echter meerdere opdrachten willen

gebruiken. In dat geval plaatsen we de uitdrukkingen tussen accolades **{ }**. Dit is trouwens zo bij alle conditionele statements:

```
if(getal < 1000)
{
    printf("Het getal is kleiner dan duizend.\n");
    printf("Het getal is mogelijk ook kleiner dan honderd.\n");
    printf("Maar het staat vast dat het niet groter is dan
1000.\n");
}
```

Bij de **if()** structuur is nog een tweede sleutelwoord beschikbaar: **else**. Dit kunnen we letterlijk vertalen door "anders". Dit betekent: "als de voorwaarde tussen de ronde haken waar is, voer dan de volgende expressie(s) uit; als dit niet het geval is, voer dan hetgeen na de **else** komt uit". Twee voorbeelden:

```
if(letter != 'A')
    printf("De letter is niet A.\n");
else printf("De letter is A.\n");
```

```
if(getal < 1000 && getal > 500)
{
    printf("Het getal is kleiner dan duizend, ");
    printf("maar het is groter dan vijfhonderd.\n);
}
else
{
    printf("Het getal is ofwel kleiner dan vijfhonderd, ");
    printf("ofwel is het groter dan duizend.\n");
}
```

Ook kan u verschillende **if()** structuren in andere **if()** structuren gebruiken. Zo kan u eigenlijk alle soorten statements tezamen en in elkaar gebruiken, zoals u later in dit hoofdstuk zal zien. Als we een **if()** statement in een ander **if()** statement gebruiken spreken we van een geneste if-structuur:

```
if(getal < 1000)
    printf("Het getal is kleiner dan duizend.\n");
else if(getal < 2000)
    printf("Het getal is kleiner dan tweeduizend, maar groter dan
duizend.\n");
else if(getal < 3000)
    printf("Het getal is kleiner dan drieduizend, maar groter dan
tweeduizend.\n");
else if(getal <= 4000)
{
    printf("Het getal is waarschijnlijk kleiner dan
vierduizend.\n");
    printf("Het getal kan echter ook exact vierduizend zijn.\n");
}
```

2.2.2 while

Bij **while()** wordt -net als bij **if()**- eerst de voorwaarde tussen de ronde haken bekeken. Indien deze waar is worden de volgende opdrachten uitgevoerd. Bij **if()** wordt de voorwaarde echter maar éénmaal

geëvalueerd. Bij de while-structuur worden de opdrachten uitgevoerd, zolang de voorwaarde waar is. Drie voorbeelden:

```
int teller = 0;
while(teller != 10)
{
    teller++;
    printf("Aantal keren dat de loop uitgevoerd werd: %d\n",
teller);
}
```

```
int getal = 5;
while(getal == 5)
    printf("Het getal is vijf\n");
```

```
int getal = 0, teller = 0;
while(teller < 4)
{
    getal + = 5;
    if(getal != 20)
        printf("Het getal is nog niet 20.\n");
    else printf("Het getal is nu 20.\n");
    teller++;
}
```

Telkens als de loop éénmaal uitgevoerd is wordt er teruggekeerd naar het begin van het **while()** statement en wordt de voorwaarde opnieuw gecontroleerd. Is deze nog altijd waar, wordt de loop weer uitgevoerd. Let op: het tweede voorbeeld is een oneindige lus. Aangezien er met de variabele **getal** nooit iets gebeurt en er aan de waarde dus niets verandert, zal de voorwaarde altijd waar zijn en blijven. De loop zal simpelweg altijd uitgevoerd worden. Het derde voorbeeld illustreert het gebruik van verschillende statements tegelijk. U kan natuurlijk ook meerdere **while()** statements in elkaar gebruiken.

2.2.3 do while

Een variatie op de gewone **while()** loop is de **do while()** loop. Deze doet exact hetzelfde, behalve dat de loop éénmaal doorlopen wordt, alvorens de voorwaarde te testen:

```
int teller = 0;
do
{
    printf("De teller staat op %d\n", teller);
    teller++;
}while(teller < 11);
```

Ook hier kan u weer gebruik maken van geneste statements. Onthoudt dus dat de opdracht(en) altijd één keer uitgevoerd worden, alvorens de voorwaarde te controleren. Deze vorm van **while()** zal dan ook handiger blijken.

2.2.4 for

De **for()** loop bestaat uit het sleutelwoord **for** gevolgd door een nogal uitgebreide expressie tussen ronde haken. Het eerste veld bevat de initialisatie van de variabele. Meestal wordt deze op de waarde 0 of 1 geïnitialiseerd. De expressie in dit veld wordt uitgevoerd, voordat de loop binnengegaan wordt. In feite staat er geen grens op wat er in het eerste veld gebeurt. U kan verschillende initialisaties doen, gescheiden door een komma. Meestal wordt het echter simpel gehouden. Vb: **teller = 0;**
Het tweede veld bevat de test die gedaan zal worden, telkens als de loop uitgevoerd wordt. Dit kan eender welke voorwaarde zijn, zolang ze maar een waar (TRUE) of onwaar (FALSE) tot resultaat heeft. Vb: **teller < 10;**
De expressie in het derde en laatste veld wordt uitgevoerd, telkens nadat de loop is uitgevoerd. Ook hier kan men meerdere operaties gebruiken, gescheiden door komma's. Meestal gaat het om incrementie of decrementie. Vb:

```
int index;
for(index=0;index<6;index++)
    printf("De waarde van de index is %d.\n", index);
```

Dit geeft het volgende:

```
De waarde van de index is 1.
De waarde van de index is 2.
De waarde van de index is 3.
De waarde van de index is 4.
De waarde van de index is 5.
```

Ter verduidelijking: in het eerste veld wordt de variabele **index** geïnitialiseerd op 0. In het tweede veld wordt de voorwaarde bepaalt. De loop wordt hier uitgevoerd zolang de waarde van de variabele **index** kleiner is dan 6. In het derde veld past men hier de incrementoperator toe. Elke keer, na het uitvoeren van de lus wordt bij de waarde van de variabele **index** 1 bijgeteld. Hier kan u natuurlijk ook weer meerdere statements en opdrachten hebben tussen accolades **{ }**.

2.2.5 switch

De **switch()** functie is de meest uitgebreide methode van een voorwaardelijk statement. Hiermee kunnen zoveel voorwaarden als gewenst tegelijk getest worden. Elke voorwaarde noemt men een case. Bij elke case hoort een constante expressie, deze moet uniek zijn en mag geen vergelijking zijn:

```
int x = 5;
switch(x)
{
    case 1 : printf("x is 1.\n");
    break;
    case 2 : printf("x is 2.\n");
    break;
    case 3 : printf("x is 3.\n");
    break;
    case 4 : printf("x is 4.\n");
    break;
    case 5 : printf("x is 5.\n");
    break;
    case 6 : printf("x is 6.\n");
    break;
    case 7 : printf("x is 7.\n");
    break;
    default : printf("x is 0 of groter dan 7.\n");
    break;
```

```
}
```

Tussen de ronde haken na het woord **switch** kunnen eventueel ook vergelijkingen staan zoals **x * (y + z)**. Dan wordt dit eerst uitgevoerd en daarna vergeleken met de case. In het voorbeeld komt ook het woord break voor. Dit dient om uit de lus te springen, zodra de juiste voorwaarde gevonden en uitgevoerd is. Break is niet verplicht, maar als u het niet gebruikt zullen alle opdrachten die bij de cases staan, die na de juistecase komen ook uitgevoerd worden. Het woord **default** wil gewoon zeggen:"in het geval dat geen enkele van de cases waar is". Ook dit is niet verplicht.

2.3 de conditionele operator

De conditionele operator heeft drie expressis als operanden. De algemene syntax is de volgende:

```
expressie1 ? expressie2 : expressie3
```

De eerste operand wordt gebruikt om na te gaan welke van de twee andere operanden moet worden uitgewerkt. Als de eerste expressie de logische waarde "waar" (niet nul) heeft, is het resultaat van de conditionele expressie gelijk aan dat van epressie 2. Als de eerste expressie "onwaar" (nul) oplevert, is expressie 3 het resultaat van de conditionele expressie. Bij wijze van voorbeeld kunnen we de conditionele operator gebruiken om het kleinste van twee waarden te bepalen:

```
/* 2.3.1.c */
#include<stdio.h>
int main(void)
{
int min, a, b;
    a = 5;
    b = 3;
    min = a < b ? a : b;
    printf("Het minimum van %d en %d is %d\n", a, b, min);
    return 0;
}
```

2.4 bit operatoren

Deze operatoren en de operaties die zij uitvoeren, worden in volgende tabel weergegeven:

&	als beide bits 1 zijn, is het resultaat 1
|	als één van de twee 1 is, is het resultaat 1
^	als slechts één bit 1 is, is het resultaat 1
~	als de bit 1 is, is het resultaat 0 en omgekeerd

Een voorbeeldprogramma:

```
/* 2.4.1.c */
#include<stdio.h>

int main(void)
{
int een = 1, niets = 0;

    printf("1 & 0 geeft %d\n", een & niets);
    printf("1 | 0 geeft %d\n", een | niets);
```

```
    printf("1 ^ 0 geeft %d\n", een ^ niets);
    return 0;
}
```

Wat het volgende zou moeten opleveren:

```
1 & 0 geeft 0
1 | 0 geeft 1
1 ^ 0 geeft 1
```

Het gebruik van 0 en 1 in dit verband kan een beetje verwarrend zijn. Volgend voorbeeld is bijna identiek:

```
/* 2.4.2.c */
#include<stdio.h>

int main(void)
{
int een = 5, niets = 7;

    printf("%d & %d geeft %d\n", een, niets, (een & niets));
    printf("%d | %d geeft %d\n", een, niets, (een | niets));
    printf("%d ^ %d geeft %d\n", een, niets, (een ^ niets));

    return 0;
}
```

Met als resultaat:

```
5 & 7 geeft 5
5 | 7 geeft 7
5 ^ 7 geeft 2
```

---

opdrachten

1. Schrijf een programma dat uw naam tien maal op het scherm toont. Schrijf dit programma drie maal, elke keer met een ander soort statement.

2. Schrijf een programma dat van 1 tot 10 telt en deze waarden elk op een nieuwe lijn toont op het scherm. Zorg er voor dat er een boodschap wordt weergegeven als de tel 3 is, en een andere boodschap als de tel 10 is.

C-taal voor beginners - hoofdstuk 3

strings en arrays

3.1 strings

[ref. C FAQ - Sectie 8]

Een string is een groep karakters, meestal letters van het alfabet. Om een goed ogende output te verkrijgen, die betekenisvolle namen en titels heeft en esthetische voldoening geeft, heeft u de de mogelijkheid nodig om tekstuele data te kunnen verwerken. Een exacte definitie van een string is "een reeks data van het type **char**, afgesloten door een nulkarakter". Wanneer C een string gaat gebruiken, kopiëren, of wat dan ook, worden de functies zo opgesteld dat ze doen wat er gevraagd wordt dat ze doen, totdat er een nulkarakter gededecteerd wordt. Zulk een string noemt men in vaktermen een ASCII-Z string.

Even een overzicht zodat u in dit hoofdstuk niet verward geraakt door de verschillende definities van **0**:

NULL	De macro **NULL**, veelal gebruikt bij pointers.
nulkarakter	Het (afsluitende) **nulkarakter**.
null-statement	De puntkomma **;** na een conditie. Bijvoorbeeld: **while((c = getchar()) == ' ');**
nul	Het getal **0**.
null	Kan zowel **'\0'** zijn als getal **0** (Engelstalige benaming).

De macro NULL is niet hetzelfde als het nulkarakter '\0'. Zoals u in een later hoofdstuk zal leren is de NULL macro handig om bepaalde dingen "naar niets te laten wijzen". Het afsluitende karakter van een string is echter het nulkarakter, dit is '\0' [ref. C FAQ - Sectie 5].

3.2 arrays

Een array is een reeks van homogene stukken data, allemaal identiek in type. Dit type kan wel zeer complex zijn. Een string is simpelweg een speciaal array, een reeks data van het type **char**.

De beste manier om deze principes te bekijken is bij wijze van voorbeeld:

```
/* 3.2.1.c */
#include<stdio.h>
int main(void)
{
char naam[5]; /* definitie van een string van karakters */
    naam[0] = 'J' ;
    naam[1] = 'o' ;
    naam[2] = 'h' ;
    naam[3] = 'n' ;
    naam[4] = '\0' ; /* nulkarakter = einde van de tekst */
    printf("De naam is %s\n", naam);
    printf("Eén letter is %c\n", naam[2]);
    printf("Een deel van de naam is %s\n", &naam[1]);
    return 0;
}
```

Het eerste nieuw stukje code is **char naam[5];** Dit definieert een string van het **char** type. De vierkante haken definiëren een array subscript in C en de **5** tussen de vierkante haken definieert vijf data velden van het type **char**, allemaal deel van de string variabele **naam**. In de C-taal beginnen alle subscripts met 0. We hebben hier vijf **char** type variabelen: **naam[0], naam[1], naam[2], naam[3], naam[4]**. U mag niet vergeten dat in C de subcripts gaan van 0 tot 1 minder dan het nummer in de definitie (tussen de vierkante haken). Dit werd zo origineel bepaald in de C-taal en kan niet door de programmeur veranderd worden.

3.3 het gebruik van een string

De variabele **naam** is dus een string met ruimte voor vijf karakters, maar aangezien we ruimte nodig hebben voor het afsluitende nulkarakter dat meetelt als één van de vijf karakters, hebben we eigenlijk maar vier bruikbare karakters. Om iets in de string te laden hebben we vijf toekenningsstatements gebruikt, welke elk één karakter toekent aan één van de stringkarakters. Uiteindelijk wordt de laatste plaats van de string gevuld met '\0' als de eindindicator en onze string is compleet. Nu dat we hem volledig gedefinieerd hebben drukken we deze af d.m.v. **printf("De naam is %s\n", naam);** De **%s** is de outputdefinitie die gebruikt wordt voor een string. Het programma zal karakters printen, beginnend met het eerste in de string **naam**, totdat het nulkarakter gevonden wordt. Dit wordt zelf niet afgedrukt. Merk op dat in het **printf()** statement enkel de naam van de variabele gegeven moet worden, zonder subscript, aangezien we de ganse string willen afdrukken, beginnend bij het begin. Het is belangrijk te weten dat de variabele **naam** op zichzelf naar de ganse **string** verwijst, maar dat **naam[ ]** met een waarde tussen de vierkante haken naar één enkel karakter in de string verwijst.

3.4 een deel van een string outputten

printf("Eén letter is %c\n", naam[2]); maakt duidelijk dat we elk afzonderlijk karakter van de string kunnen afdrukken door **%c** te gebruiken en het individuele karakter van de variabele **naam** te benoemen, samen met het subscript. In hoofdstuk 1 leerden we reeds dat **%c** een karakter aanduid en dat een karakter slechts één teken is. **printf("Een deel van de naam is %s\n", &naam[1]);** illustreert hoe we een een deel van de string kunnen afdrukken, door het begin aan te duiden met een subscript. Het **&** teken specifieert hier het adres van **naam[1]**. Dit wordt uitvoerig bestudeerd in het volgende hoofdstuk, dus maak er u nu niet al te veel zorgen over.

Het voorbeeldprogramma dat we tot nu toe gebruikten zal u waarschijnlijk de indruk geven dat strings nogal omslachtig zijn, aangezien elk karakter één voor één geïnitialiseerd moet worden. Het zou moeilijk werken zijn als we een string op die manier moesten gebruiken, maar ik deed het tot nu toe om duidelijk te maken hoe het in elkaar zit. Het volgende voorbeeld laat zien dat strings in feite zeer gemakkelijk in gebruik zijn.

3.5 stringfuncties

```
/* 3.5.1.c */
#include <stdio.h>
#include <string.h>

int main(void)
{
char naam1[12], naam2[12], mixed[25];
char titel[20];

    strcpy(naam1, "Rosalinda");
    strcpy(naam2, "Tom");
    strcpy(titel, "Dit is de titel.");

    printf("          %s\n\n", titel);
```

```
    printf("Naam 1 is %s\n", naam1);
    printf("Naam 2 is %s\n", naam2);

    if(strcmp(naam1, naam2) > 0) /* geeft 1 als naam1 > naam2 */
        strcpy(mixed, naam1);
    else
        strcpy(mixed, naam2);

    printf("De grootste alfabetische naam is %s\n", mixed);

    strcpy(mixed, naam1);
    strcat(mixed, " ");
    strcat(mixed, naam2);
    printf("Beide namen zijn %s\n", mixed);
    return 0;
}
```

Dit geeft als output:

```
        Dit is de titel.

Naam 1 is Rosalinda
Naam 2 is Tom
De grootste alfabetische naam is Tom
Beide namen zijn Rosalinda Tom
```

Om de verschillende stringfuncties te kunnen gebruiken includeren we de header **string.h** in ons programma.

3.5.1 strcpy

Dit programma is een goed voorbeeld van het gebruik van een paar stringfuncties. Eerst declareren we 4 strings. Vervolgens komen we aan een nieuwe functie die u zeer nuttig zal vinden, de **strcpy()** functie, of string-kopieerfunctie. Deze kopieert van één string naar een andere totdat het null karakter gevonden wordt in de bronstring. Herinner u dat de null eigenlijk een 0 is die door het systeem aan de string wordt toegevoegd. Het is duidelijk welke string gekopieerd wordt als u het bekijkt als een toekennings-statement. Net als bij, bijvoorbeeld, **x = 23;** wordt de data van rechts naar links gekopieerd, zodat in ons voorbeeld na de uitvoering van het eerste statement de variabele **naam1** de **string "Rosalinda"** bevat, maar zonder de dubbele quotes **" "**, deze zijn gewoon een manier voor de compiler om te weten dat hij met strings te maken heeft. Het zou duidelijk moeten zijn dat **strcpy(naam1, "Rosalinda");** een stringconstante naar een string variabele kopieert.

Op dezelfde manier wordt de string **"Tom"** naar **naam2** gekopieerd, daarna wordt de titelstring gekopieerd naar de string genaamd **titel**. De titel en allebei de namen worden dan afgedrukt. Merk op dat het niet noodzakelijk is voor de ontvangende string om exact dezelfde grootte te zijn als de string die het moet herbergen, zolang hij maar tenminste zolang is als de bronstring.

3.5.2 strcmp

De volgende functie die we zullen bekijken is **strcmp()**, ofwel de string-vergelijkingsfunctie. **strcmp(naam1, naam2)** geeft een 1 indien de eerste string groter is dan de tweede, een 0 als de twee gelijk zijn van lengte en dezelfde karakters bevatten en een -1 als de eerste string kleiner is dan de tweede. In ons voorbeeld wordt één van de twee strings, afhankelijk van de **strcmp()**, gekopieerd in de string variabele **mixed** en de grootste naam wordt afgedrukt met een **printf()** statement. Het is duidelijk

dat **"Tom"** alfabetisch groter is. De lengte doet er niet toe, enkel de relatieve positie in het alfabet. Het resultaat is ook afhankelijk van het feit dat het hoofdletters of kleine letters zijn.

3.5.3 strcat

U merkt ook nog een derde functie in het voorbeeld: **strcat()**, ofwel de stringconcatenatie. Deze voegt simpelweg de karakters van een string aan het einde van een andere string toe, automatisch de null aanpassend, zodat alles in orde is. In dit geval wordt **naam1** naar **mixed** gekopieerd, daarna worden er twee blanco's aan toegevoegd, om dan de inhoud van **naam2** eraan toe te voegen. Het resultaat wordt afgedrukt met een **printf()** functie.

Strings zijn niet moeilijk te gebruiken en zijn zeer nuttig, maar ze vereisen een zekere aandacht. Het is verkeerd een string naar een string te kopiëren, die korter werd gedefinieerd dan de bronstring, maar de compiler zal de kopieeropdracht toch uitvoeren, waardoor andere data waarschijnlijk overschreven wordt. De compiler kent geen manier om u hiervoor te waarschuwen, dus een beetje voorzichtigheid is geboden.

Een snelle controle van de documentatie van één compiler toonde zo'n achttien stringfuncties beschikbaar voor gebruik. Sommige worden gebruikt om strings te kopiëren met bovengrenzen op het aantal te kopiëren karakters. Er zijn stringfuncties om naar bepaalde karakters in een string te zoeken, andere tellen karakters bij op bepaalde plaatsen. Natuurlijk kan u ook karakters verwijderen, op de plaats waar u dat wilt. Het zou u veel opleveren als u de referenties en documentatie van uw compiler raadpleegde, om te kijken welke functies er beschikbaar zijn. Anders zou u misschien ingewikkelde procedures moeten bedenken om bepaalde zaken te verwezenlijken, die eigenlijk op een gemakkelijke manier tot een goed einde gebracht kunnen worden.

3.5.4 strlen

Deze functie kan u gebruiken om de lengte van een string te berekenen:

```
/* 3.5.4.1.c */
#include <stdio.h>
#include <string.h>

int main(void)
{
char string[50] = "Een voorbeeldstring";

    printf("De string bevat is %d tekens\n", strlen(string));
    return 0;
}
```

Het null karakter wordt niet meegeteld.

U wist reeds dat de index (lengte of grootte) groter mag zijn dan werkelijke lengte, maar niet kleiner. Als u geen index definieert, zal de compiler zelf de karakters moeten tellen en geheugenruimte voorzien [ref. C FAQ - Sectie 1, Punt 32] :

```
char mijn_string[ ] = "Dit is een voorbeeldstring";
```

Alhoewel dit perfect ANSI-C is, kan deze methode wel problemen geven bij concatenatie en het kopiëren van strings en dergelijke. Op dezelfde manier is ook het volgende mogelijk:

```
int mijn_array[ ] = {1,23,17,4,-5,100};
```

3.5.5 strchr

De functie **strchr()** scant een string totdat een bepaald karakter gevonden is. Er wordt in voorwaartse richting gezocht. De functie vindt het eerste voorkomen van het karakter in de string. Het null karakter wordt als een deel van de string beschouwd. Een voorbeeld:

```
/* 3.5.5.1.c */
#include<stdio.h>
#include<string.h>
int main(void)
{
char string[17];
char *ptr, c = 'r';

    strcpy(string, "Dit is een string");
    ptr = strchr(string, c);
    if (ptr)
        printf("Het karakter %c staat op positie: %d\n", c, ptr-
string);
    else
        printf("Het karakter werd niet gevonden\n");
    return 0;
}
```

3.5.6 gets en puts

De twee meest gebruikte stringfuncties zijn waarschijnlijk **gets()** en **puts()**. Deze kunnen een volledige string opnemen en outputten, inclusief spaties, tabs, enz... .

De **gets()** functie verzamelt een string van karakters, totdat het newline teken '\n' gevonden wordt. Dit wil meestal zeggen totdat de return toets ingedrukt wordt. Het newline (of nieuwe-lijn) karakter wordt door **gets()** vervangen door het null karakter. Alles tot aan de newline wordt in de variabele gestopt, tussen de ronde haken. De **gets()** functie is niet lengte-bepaald. Als de inputstring voldoende groot is kan er dus data overschreven worden.

De **puts()** functie toont een volledige string op het scherm. Deze functies zijn handig omdat **scanf()** en dergelijke problemen ondervinden als er lege ruimtes in een reeks karakters voorkomen. Ze bevinden zich in de standaardheader, dus **string.h** is niet vereist. Ter illustratie twee voorbeelden:

```
/* 3.5.6.1.c */
#include <stdio.h>

int main(void)
{
char mijn_string[80];

    printf("Geef een string in: ");
    gets(mijn_string);
    printf("De string input was: %s\n", mijn_string);
    return 0;
}
```

/ 3.5.6.2.c */*

```
#include <stdio.h>

int main(void)
{
    char mijn_string[ ] = "Dit is een voorbeeld output string\n";

    puts(mijn_string);
    return 0;
}
```

3.6 een array van integers

Een voorbeeld van een programma dat gebruik maakt van een array van integers:

```
/* 3.6.1.c */
#include<stdio.h>
int main(void)
{
int waarden[12];
int index;
    for(index=0;index<12;index++)
        waarden[index] = 2 * (index+4);
    for(index=0;index<12;index++)
        printf("De waarde bij index=%2d is %3d\n",
                    index, waarden[index]);
    return 0;
}
```

Dit geeft als output:

```
De waarde bij index= 0 is   8
De waarde bij index= 1 is  10
De waarde bij index= 2 is  12
De waarde bij index= 3 is  14
De waarde bij index= 4 is  16
De waarde bij index= 5 is  18
De waarde bij index= 6 is  20
De waarde bij index= 7 is  22
De waarde bij index= 8 is  24
De waarde bij index= 9 is  26
De waarde bij index=10 is  28
De waarde bij index=11 is  30
```

Merk op dat het array op dezelfde manier als bij een string gedefinieerd wordt, nu is deze echter van het type integer. We hebben dus twaalf integer variabelen om mee te werken, plus één genaamd **index**. De namen van de variabelen zijn: **waarden[0]**, **waarden[1]**, ..., en **waarden[11]**. Vervolgens hebben we een **for()** loop die een boel nonsens toekent, het maakt wel duidelijk dat aan elk van de twaalf variabelen data wordt toegekend, daarna worden ze alle twaalf afgedrukt in de volgende loop. Elk element van het array is simpelweg een integer variabele, geschikt om integer waarden te herbergen. Het enige verschil tussen de variabelen **index** en, bijvoorbeeld, **waarden[2]** is de manier waarop ze geadresseerd worden. U zou geen problemen mogen ondervinden bij het begrijpen van dit programma. Compileer en run het en experimenteer wat met de waarden.

Zoals u bij hierboven reeds zag is er nog een mogelijkheid om een string te initialiseren. De groep karakters kan onmiddelijk meegegeven worden aan de variabele:

```
char mijn_string[30] = "Dit is een voorbeeldstring";
```

Om een array met integers te initialiseren kan u het volgende doen:

```
int mijn_array[30] = {1,23,17,4,-5,100};
```

Deze getallen kunnen ook negatief zijn. Het gebruik van de index (of subscript) is op dezelfde manier als bij een string: **1** staat hier op index **0**, **23** op index **1**, **17** op index **2**, ...

3.7 strings als input

Tot nu toe hebben we voornamelijk gezien hoe we een string definiëren en als uitvoer verwerken. Om een string in te scannen kan u gewoon het bekende **scanf()** gebruiken. Het nadeel is echter dat we slechts één woord kunnen opnemen, dus geen string met spaties, zoals dat bij de **gets()** functie wel het geval is. We gebruiken **%s** om aan te duiden dat het om een string gaat:

```
/* 3.7.1.c */
#include<stdio.h>
int main(void)
{
char v_naam[30];
    printf("Geef uw voornaam in -> ");
    scanf("%s", &v_naam);
    printf("Uw voornaam is dus %s", v_naam);
    return 0;
}
```

3.8 multi-dimensionele arrays

Voor een vlugge kijk op multi-dimensionele arrays, bekijken we de volgende initialisatie:

```
char multi[5][10];
```

Wat betekent dit? Laten we het in het volgende licht beschouwen:

```
char multi[5][10];
```

We bespreken het onderlijnde deel van de "naam" van het array. Met de **char** er voor en **[10]** er achter hebben we een array van tien karakters. Maar de naam **multi[5]** is op zichzelf een array dat aangeeft dat er vijf elementen zijn, die elk op zichzelf een array van tien karakters zijn. We hebben nu dus een array van vijf arrays van tien karakters elk.

Stel dat we dit 2-dimensioneel array met wat willekeurige data gevuld hebben. In het computergeheugen ziet dit er mogelijk uit alsof het gevormd werd door het initialiseren van vijf aparte arrays:

```
multi[0] = {'0', '1', '2', '3', '4', '5', '6', '7', '8', '9'}
multi[1] = {'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j'}
multi[2] = {'A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'I', 'J'}
multi[3] = {'9', '8', '7', '6', '5', '4', '3', '2', '1', '0'}
multi[4] = {'J', 'I', 'H', 'G', 'F', 'E', 'D', 'C', 'B', 'A'}
```

Individuele elementen zijn adreseerbaar op de volgende methode:

```
multi[0][3] = '3'
multi[1][7] = 'h'
multi[4][0] = 'J'
```

Een voorbeeldprogramma:

```
/* 3.8.1.c */
#include <stdio.h>

int main(void)
{
int i, j;
int groot[8][8], breed[25][12];

    for (i = 0 ; i < 8 ; i++)
        for (j = 0 ; j < 8 ; j++)
            groot[i][j] = i * j;     /* een
vermenigvuldigingstafel */

    for (i = 0 ; i < 25 ; i++)
        for (j = 0 ; j < 12 ; j++)
        breed[i][j] = i + j;     /* een optellingstafel */

    groot[2][6] = breed[24][10] * 22;     /* lijn 1 (zie tekst) */
    groot[2][2] = 5;
    groot[groot[2][2]][groot[2][2]] = 177;     /* lijn 2 (zie
tekst) */

    for (i = 0 ; i < 8 ; i++)
    {
        for (j = 0 ; j < 8 ; j++)
        printf("%5d ", groot[i][j]);
        printf("\n");     /* nieuwe lijn voor elke verhoging van i
*/
    }
    return 0;
}
```

Dit programma werkt met dubbel-gedimensioneerde arrays. De variabele **groot** is een 8 bij 8 array dat in totaal 8 x 8 of 64 elementen bevat. Het eerste element is **groot[0][0]**, het laatste is **groot[7][7]**. Een ander array genaamd **breed** is ook gedefinieerd, maar is niet vierkant, om aan te tonen dat dit niet verplicht is. Beide arrays worden gevuld met data, waarvan er één een vermenigvuldigingstafel voorstelt en waarvan de andere tot een optellingstafel gevormd wordt.

Om te illustreren dat individuele elementen naar behoefte aangepast kunnen worden, wordt één van de elementen van **groot** toegewezen aan de waarde van één van de elementen van **breed**, na vermenigvuldigd te zijn met **22** (lijn 1). Vervolgens wordt **groot[2][2]** toegewezen aan de willekeurige waarde **5** en deze waarde wordt gebruikt voor de subscripts van het toekenningsstatement op lijn 2. Lijn 2 is eigenlijk hetzelfde als **groot[5][5] = 177;** omdat elk van de subscripts de waarde **5** bevat. Dit wordt hier enkel gedaan om te illustreren dat elke legale expressie mag gebruikt worden in het subscript. Het moet aan twee zaken voldoen: het moet een integer zijn (alhoewel **char** ook toegelaten wordt) en de waarde moeten binnen het bereik van het subscript, waarvoor het gebruikt wordt, liggen. De volledige matrix **groot** wordt gedrukt in een vierkant, zodat u de waarden kan controleren.

opdrachten

1. Schrijf een programma met drie korte strings, van ongeveer 6 karakters elk en gebruik **strcpy()** om de string literals **"een"**, **"twee"** en **"drie"** in die strings te kopiëren. Gebruik concatenatie om de drie strings in één grotere string van ongeveer 30 karakters te kopiëren en druk het resultaat tien maal op het scherm af. Vergeet de header **string.h** niet te includeren.

2. Definieer twee arrays van het type integer, **array1** en **array2** genaamd. Gebruik een loop om een boel waarden in elk van hen te stoppen en tel ze telkens op bij een ander array van tien elementen breed, genaamd **array1_en_array2**. Uiteindelijk druk je alle resultaten af in een tabel, via een index.

3. Vraag aan de gebruiker een voornaam, daarna een achternaam. Stop deze waarden in twee verschillende arrays, van het juiste type. Gebruik één of meerdere loops naar keuze om eerst de naam gewoon af te drukken en daarna de naam in omgekeerde volgorde af te drukken:

```
Wat is uw naam: Janssen
Wat is uw voornaam: Jan
U bent dus Jan Janssen.
In omgekeerde volgorde is dat: nessnaJ naJ
```

C-taal voor beginners - hoofdstuk 4

functies, prototypes en geheugenklassen

4.1 functies

Een C-programma bestaat uit één of meerdere functies. Met functies kunnen we grote programma's opdelen in kleinere eenheden, wat het programmeren vergemakkelijkt. Verder kunnen functies die in het ene programma zijn ontwikkeld, in andere programma's worden gebruikt. Een functie beschrijft het rekenproces dat op de data van de functie moet worden uitgevoerd. De functie moet daarom zowel de data-objecten (declaraties) als de werking (statement) beschrijven.

Een programma kan uit meerdere functies bestaan, maar één van die functies moet de naam **main** hebben. Een functie heeft steeds één ingang en één uitgang. De statements worden van begin tot einde uitgevoerd. Als een functie een tweede functie aanroept, is het effect dat de uitvoering van de eerste functie tijdelijk wordt opgeschort op het punt dat de tweede functie wordt aangeroepen. Deze tweede functie wordt dan binnengegaan (bij de ingang), uitgevoerd en verlaten. Daarna gaat de aanroepende functie verder bij het statement, dat onmiddelijk volgt op het punt van de aanroep.

In het volgende voorbeeldprogramma moet vier keer de tijd in uren, minuten en seconden ingelezen worden en telkens de tijd in seconden berekend en afgedrukt worden. We kunnen dat oplossen door een functie **bereken()** viermaal aan te roepen:

```
/* 4.1.1.c */
#include<stdio.h>

void bereken(void); /* declaratie */
int main(void)
{

    bereken();    /* aanroep */
    bereken();    /* aanroep */
    bereken();    /* aanroep */
    bereken();    /* aanroep */
    return 0;
}

void bereken(void) /* definitie */
{
int uren, minuten, seconden, tijd;

    printf("\nGeef uren, minuten en seconden: ");
    scanf("%d%d%d", &uren, &minuten, &seconden);

    tijd=(60 * uren + minuten) * 60 + seconden;

    printf("\nDe tijd %d:%d:%d is omgerekend %d seconden\n",
                                        uren, minuten, seconden,
tijd);

    return;
}
```

Bovenstaand programma bevat 2 functies: **main()** en **bereken()**. Via de functie-aanroep **bereken();** wordt de functie **bereken()** aangeroepen. De statements van de functie worden dan uitgevoerd. Als laatste instructie vinden we **return;** Deze instructie keert terug naar de aanroepende functie. De volgorde bij het gebruik van een functie is altijd: declareren, aanroepen en definiëren, zoals in het programma aangetoond wordt.

4.1.1 de waarde van een functie

In ons voorbeeld werd een ondergeschikte functie **bereken()** gebruikt, die verantwoordelijk was voor het berekenen van één tijd. Tussen de functies **main()** en **bereken()** werden geen waarden uitgewisseld.

In de C-taal bestaat een mechanisme om het resultaat van een door een aangeroepen functie uitgevoerde berekening terug te zenden naar de aanroepende functie. Dit gebeurt met een **return** statement. Het statement geeft aan dat er vanuit de aangeroepen functie wordt teruggekeerd en dat de waarde aan de aanroepende functie moet worden doorgegeven. Deze waarde kan door de aanroepende functie met behulp van een passend statement worden vastgehouden. Voorbeeld:

```
/* 4.1.1.1.c */
#include<stdio.h>

int bereken(void);
int main(void)
{
int sec;

    sec = bereken(); /* vasthouden van de waarde */

    printf("Omgerekend zijn dit %d seconden.", sec);
    return 0;
}

int bereken(void)
{
int uren, minuten, seconden, tijd;

    printf("\nGeef uren, minuten en seconden: ");
    scanf("%d%d%d", &uren, &minuten, &seconden);

    tijd=(60 * uren + minuten) * 60 + seconden;

    return tijd;
}
```

In de functie **bereken()** wordt het aantal seconden niet meer afgedrukt, maar wel doorgegeven aan het hoofdprogramma met behulp van het **return** statement. Bij de declaratie vertellen we wat voor soort waarde wordt teruggestuurd; hier een integer. Als een functie een waarde aflevert, moet het type van de af te leveren waarde in de functienaam vóór de naam van de functie worden geplaatst. Deze typespecificator is één van de types die C kent (zie hoofdstuk 1). Uiteraard moet dit type overeenkomen met (of kunnen geconverteerd worden naar) het type van de expressie die via de **return** wordt teruggegeven. Een **return** kan ook een expressie terugleveren zoals **x + y - z**. Dit wordt dan geëvalueerd en het resultaat wordt teruggelevert.

Een **void** functie is een functie die geen waarde aflevert. Dus als u met een **return** een waarde teruggeeft, mag de afleverende functie nooit van het type **void** zijn. Een functie die zonder typespecificator wordt gedefinieerd heeft impliciet het type **int**:

```
bereken(void)
{
    ....
}
```

De computer verwacht in dit geval dat een waarde van het type **int** zal worden teruggestuurd. Als u een programma schrijft met een functie die niet als het type **void** gedeclareerd is, maar die ook geen waarde terugleverd, kan u best als laatste lijn **return 0;** plaatsen, dit om waarschuwingen van de compiler te vermijden:

```
#include<stdio.h>
int main(void)
{
    ...
    return 0;
}
```

Zoals ik reeds eerder zei moeten we bij het gebruik van functies altijd drie zaken respecteren, in de juiste volgorde: declaratie, aanroep en definitie. De declaratie is noodzakelijk opdat de C-compiler gegevens over de attributen van het object kan verzamelen en deze informatie kan gebruiken als er naar het object verwezen wordt. Naar een functie mag echter wel verwezen (aangeroepen) worden, voordat deze is gedefinieerd. Als een functie eerst wordt aangeroepen en dan pas gedeclareerd, moet in de aanroepende functie een functieprototype staan zodat het type van de af te leveren waarde duidelijk is. Als u bijvoorbeeld in een programma twee integer functies heeft: **functie1** en **functie2**, dan declareren we deze als volgt:

```
int functie1();
int functie2();
```

Ook mogelijk, maar moeilijker te onderhouden is het volgende:

```
int functie1(), functie2();
```

4.1.1.1 het return type van main

In de originele definitie van C, gaven alle functies standaard een integer type variabele terug, behalve als de programmeur dit anders definieerde. Omdat het expliciet returnen van een waarde bij het verlaten van een functie optioneel was, werd het grootste deel van C als volgt geschreven:

```
main()
{
    ....
}
```

Omdat **main()** zonder argumenten (zie volgende paragraaf) volgens velen niets terrugleverde, plaatste men veelal het **void** type, waardoor het volgende toegepast werd:

```
void main()
{
    ....
}
```

Toen de ANSI-C standaard werd vervolledigd, was het enige legale teruglevertype het type **int**, op deze manier:

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

De meeste compilers zullen de **void** methode ondersteunen, maar enkel de **int** return methode word toegelaten door de ANSI-C standaard. In deze cursus zal overal de laatste (juiste ANSI-C) methode gebruikt worden.

4.1.2 argumenten van functies

In voorgaande programma's werd tussen de afzonderlijke functies geen data uitgewisseld, of werd slechts één waarde afgeleverd via het **return** statement. Er bestaan echter veel programmeerproblemen waarvoor functies nodig zijn die meer dan één object aan de aanroepende functie afleveren. Bovendien is het ook mogelijk dat de aanroepende functie waarden aan de aangeroepen functie wil doorgeven. In deze paragraaf bespreken we hoe de aanroepende functie argumenten (parameters) kan doorgeven.

We hebben reeds gebruik gemaakt van het doorgeven van argumenten van de aanroepende naar de aangeroepen functies. In de meeste programma's die we reeds ontwikkeld hebben, werd de functie **printf()** opgeroepen. Bij de aanroep van deze functie worden argumenten opgegeven, die als een lijst tussen haakjes en gescheiden komma's van de functie staan. In de aanroep:

```
printf("De som van %d en %d is %d", getal1, getal2, som);
```

worden vier argumenten doorgegeven: het stringargument en de drie expressies die de af te drukken waarden weergeven. We noemen deze argumenten de actuele argumenten. De manier waarop een functie wordt aangeroepen wordt de "calling sequence" genoemd. De functiedefinitie met argumenten heeft de vorm:

```
typespecificator functie(argumentenlijst);
```

De typespecificator is natuurlijk één van de types die C kent. Nieuw is de argumentenlijst tussen de ronde haken. Deze noemen we de formele argumenten. De actuele en formele argumenten moeten in aantal en type overeenkomen. Voor elk formeel argument komt precies één corresponderend acteel argument voor bij de functieaanroep. Voorbeeld:

```
/* 4.1.2.1.c */
#include<stdio.h>

int main(void)
{
float som(float, float);
float x, y, resultaat;

    printf("Geef twee getallen: ");
    scanf("%f%f", &x, &y);

    resultaat = som(x, y);

    printf("De som van %f en %f = %f", x, y, resultaat);
    return 0;
}
```

```
float som(float a, float b)
{
float hulp;

    hulp = a + b;

    return hulp;
}
```

Een functie **som()** heeft twee floating point waarden als argumenten en geeft de som van deze twee via het **return** statement terug aan de oproepende functie.

Merk op: de namen van de actuele en formele argumenten mogen verschillend zijn. Bij de aanroep wordt immers de waarde van het actuele argument doorgegeven naar het formele argument. De variabelen **resultaat** en **hulp** zijn eigenlijk overbodig in het vorige programma. Als volgt kan het immers ook:

```
/* 4.1.2.2.c */
#include<stdio.h>

int main(void)
{
float som(float, float);
float x, y;

    printf("Geef twee getallen: ");
    scanf("%f%f", &x, &y);

    printf("De som van %f en %f = %f", x, y, som(x, y));
    return 0;
}

float som(float a, float b)
{
    return a + b;
}
```

U merkt dat we in een **printf()** statement evengoed een functie kunnen oproepen.

Indien een functie geen parameters aanneemt, kan u als u wil het **void** type in de hoofding gebruiken om dit aan te duiden:

```
int mijn_functie(void)
{
    ....
}
```

In deze cursus worden functies zonder argumenten trouwens op deze manier gebruikt.

Als een actueel argument een expressie is, wordt eerst de expressie geëvalueerd en wordt de waarde als argument doorgegeven. Dit noemen we ook wel call by value.

4.1.2.1 de argumenten van main

Een programma begint door het oproepen van de functie **main()**. Deze vereist geen prototype en kan dus zonder parameters gedefinieerd worden:

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

Er bestaat echter ook een mogelijkheid om parameters te gebruiken:

```
int main( int argc, char *argv[ ] )
{
    ....
    return 0;
}
```

De namen **argc** en **argv** zijn niet vereist, maar gewoon een standaardmanier om de argumenten te benoemen. U kan eender welke benaming gebruiken als die geldig is voor functieargumenten. Wanneer **main()** aangeroepen wordt (dit is onmiddelijk bij uitvoering van het programma) zal **argc** een teller zijn voor het aantal argumenten en **argv** zal een array zijn van die argumenten. Aangezien elk woord een string is die voorgesteld wordt als een pointer-naar-char, is **argv** een array-van-pointers-naar-char.

U vraagt zich waarschijnlijk af hoe we waarden aan de **main()** kunnen doorgeven. In feite gaat het hier om de command line, zoals we die kennen in Dos of Unix. Indien u bijvoorbeeld een programma 'mijn_prog.exe' heeft dat een bestandsnaam (argument of parameter) aanneemt en daar dan iets mee doet, zou u dat in Dos als volgt kunnen oproepen:

```
mijn_prog c:\map\ergens\mijn_bestand.txt
```

Sommige besturingssystemen laten zelfs toe dat u het bestand er op 'laat vallen'. U moet weten dat **argv[0]** de bestandsnaam van het programma zelf voorstelt en dus voor niets anders gebruikt kan worden. Bekijk het volgende voorbeeld dat simpelweg de inhoud van een opgegeven bestand toont.

```
/* 4.1.2.1.1.c */
#include <stdio.h>

int main(int argc, char *argv[ ])
{
int i;
FILE *fp;
int c;

    printf("De naam van dit programma is %s.\n", argv[0]);
    getchar(); /* wachten op een druk op <enter> alvorens verder
te gaan */
    for(i = 1; i < argc; i++)
    {
        fp = fopen(argv[i], "r");
        if(fp == NULL)
            printf("kan %s niet openen\n", argv[i]);

        while((c = getc(fp)) != EOF)
            putchar(c);
```

```
            fclose(fp);
        }

        return 0;
    }
```

4.2 pointers

De hierboven beschreven manier kan gebruikt worden om parameters door te geven van de aanroepende functie naar de aangeroepen functie. Er bestaat echter geen gelijkaardig mechanisme voor de tegengestelde richting. Veranderingen in de waarde van een formeel mechanisme binnen een functie brengt geen verandering in de waarde van het oorspronkelijke actuele argument. Dit is het gevolg van het feit dat het doorgeven van het argument via "call by value" is, zoals eerder besproken. Om een aangeroepen functie de toegang te verlenen tot een waarde in de aanroepende functie, moet de aanroepende functie het adres van haar lokale variabelen doorgeven. Als de aangeroepen functie dan het adres van een variabele in de aanroepende functie heeft, kan op dat adres een nieuwe waarde worden toegekend. We hebben dit reeds toegepast bij de **scanf()** functie, waarbij we het adres doorgeven van de variabele waarop de in te lezen waarde moet worden geplaatst (met het **&** teken). Om functies te kunnen schrijven die met adressen werken hebben we het begrip pointer nodig. (Zie hoofdstuk 5)

Een pointer is een variabele die het adres van een ander object bevat. Aangezien een pointer een variabele is, moet deze net als alle andere variabelen een gebied in het geheugen beslaan, dat zelf een adres heeft. De inhoud van dat gebied in het geheugen is het adres van een andere variabele. Als, bijvoorbeeld, **ptrx** een variabele is die kan gebruikt worden om de integer variabele **x** te adresseren, kunnen we dit adres als volgt aan de pointer toekennen:

```
ptrx = &x;
```

ptrx verwijst nu naar de variabele **x** en bevat het adres waarop variabele **x** in het geheugen te vinden is. De indirectieoperator ***** neemt nu de inhoud van de geheugenplaats waarnaar de pointer verwijst. Dus

```
*ptrx
```

geeft de waarde van de variabele **x**. Bekijk **&** als "adres van" en ***** als "inhoud van". We kunnen een pointer op dezelfde manier declareren als variabelen, bijvoorbeeld:

```
int *ptrx;
```

Hiermee wordt een pointer gedeclareerd die verwijst naar een variabele van het type integer. Een ander voorbeeld:

```
int a = 14, b;
int *q;
q = &a;  /* q verwijst naar het adres van variabele a */
b = *q;  /* b neemt de inhoud van de variabele waarnaar pointer q
verwijst */
```

Hieruit kunnen we dan afleiden dat **b = 14**.

4.3 call by reference

Voorbeeld:

```c
/* 4.3.1.c */
#include<stdio.h>

int main(void)
{
int uren, minuten, seconden, tijd;
void bereken(int, int, int, int*); /* declaratie */

    printf("\nGeef uren, minuten en seconden: ");
    scanf("%d%d%d", &uren, &minuten, &seconden);

    bereken(uren, minuten, seconden, &tijd); /* aanroep */

    printf("Berekende tijd: %d", tijd);
    return 0;
}

void bereken(int u, int m, int s, int *t) /* definitie */
{
    *t = (60 * u + m) * 60 + s;
}
```

In een functie **main()** wordt een tijd ingelezen in uren, minuten en seconden. De bedoeling is dit om te rekenen naar het totaal aantal seconden door gebruik te maken van een ondergeschikte functie **bereken()**. Het resultaat van de berekening moet teruggestuurd worden naar **main()** vermits dit daar wordt afgedrukt. De functie **bereken()** heeft dus vier argumenten: de eerste drie stellen de oorspronkelijke tijd voor, terwijl de laatste de door de functie berekende tijd voorstelt. Dit laatste argument moet een adres (of pointer) zijn, zodat de functie **bereken()** de berekende waarde in de desbetreffende geheugenplaats kan plaatsen.

4.4 geheugenklassen

Tot nu toe hebben we altijd met hetzelfde soort variabelen gewerkt. In de C-taal kan men echter meerdere types gebruiken. Er zijn 4 geheugenklassen voor variabelen in C:

4.4.1 automatisch

Het bereik van een automatische variabele is beperkt tot de functie waarin hij gedeclareerd is. Automatische variabelen hebben waarden die lokaal zijn ten opzichte van de functies waarin ze gedeclareerd zijn. Dat wil zeggen dat een variabele **x** die gedeclareerd is in een functie kan veranderd worden, zonder dat dit een invloed heeft op enige **x** elders in het programma.

Automatische variabelen bestaan slechts totdat de functie afgelopen is. De geheugenruimte die door de compiler gereserveerd werd is dan niet langer voorzien en de waarden van deze variabelen is dan ook verloren. Als de functie nadien opnieuw wordt opgeroepen zijn de waarden van de vorige keer niet meer beschikbaar.

Om aan te duiden dat we een automatische variabele willen plaatsen we het woordje **auto** voor de declaratie:

```c
auto char letter;
auto int i, j, k;
auto float btw = 20.5;
```

De automatische geheugenklasse is de default geheugenklasse voor variabelen in C in een functie. Daarom kunnen we **auto** ook weglaten, wat we normaal ook doen.

```
auto char letter;
auto int i, j, k;
auto float btw = 20.5;
```

is equivalent met

```
char letter;
int i, j, k;
float btw = 20.5;
```

Dus waar het op neerkomt is dat u het woordje **auto** helemaal niet moet kennen of bestuderen, en zeker niet moet toepassen. De geheugenklasse op zich is echter wel belangrijk. [ref. C FAQ - Sectie 1, Punt 12]

4.4.2 extern

Elke variabele die buiten functies gedeclareerd wordt is extern. Deze variabele kan dan gebruikt worden doorheen het ganse programma. Wanneer hun namen niet gebruikt worden in een functie, dan worden zij "verborgen" maar hun waarde kan niet vernietigd worden:

```
/* 4.4.2.1.c */
#include<stdio.h>

int x;

int main(void)
{
void functie1(void);
void functie2(void);

    x = 1;
    functie1();
    printf("x = %d\n", x);
    functie2();
    printf("x = %d\n", x);
    return 0;
}

void functie1(void) { x *= 10; }

void functie2(void) { x += 2; }
```

Met als uitvoer:

```
x = 10
x = 12
```

Hier is de variabele **x** dus **extern**. In de drie functies is deze beschikbaar en kan hij niet anders gedeclareerd worden. Variabele **x** is "globaal" beschikbaar. Het bereik van **x** gaat van het begin van het programma tot het einde en moet daarom dus niet als parameter doorgegeven worden. De verleiding is groot om zoveel mogelijk variabelen als **extern** te declareren. Dit brengt echter een heleboel problemen

met zich mee bij grotere programma's. Gebruik dus niet te pas en te onpas externe variabelen. [ref. C FAQ - Sectie 1, Punt 7 en 11]

De default geheugenklasse voor functies in C is de geheugenklasse **extern**.

Met dit in gedachte kan je een variabele of een functie op verschillende manieren definieren:

1. Lokaal in een functie: komt (meestal) op de stack (zie 4.5).
2. Lokaal static in een functie. Wordt 1 keer ongesteld op de geven waarde en veranderd dan niet tussen calls naar die routine. Deze waarde is dus alleen binnen de betreffende functie beschikbaar.
3. Globaal in een source file. Dit kan door alle functies in het ganse programma gebruikt worden.
4. Globaal static in een source file. Dit kan door alle functies in de betreffende source file gebruikt worden. Andere source files kunnen hem niet gebruiken.
5. Extern. Het object is gedefinieerd in een andere source file en is van type 3.

4.4.3 statisch

We weten nu dat we variabelen binnen (lokaal) of buiten (globaal) een functie kunnen declareren. Statische variabelen vormen een nieuwe geheugenklasse: zij kunnen zowel lokaal als globaal zijn. We duiden een statische variabele aan met het woordje **static**.

Lokale statische variabelen zijn intern ten opzichte van de functie waarin ze worden gedeclareerd, maar in tegensteling tot automatische variabelen blijven statische variabelen bestaan. Dit wil zeggen: zij behouden hun waarde ook nadat de functie is afgelopen. Daardoor is de waarde die de variabele had toen de functie de vorige keer verlaten werd nog dezelfde wanneer die functie opnieuw aangeroepen wordt.

Een lokale statische variabele kan met een constante expressie worden geïnitialiseerd, éénmaal aan het begin van het programma. Daarna is de waarde van een statische variabele bij het binnengaan van een functie gelijk aan de waarde die de variabele had toen de functie de vorige keer werd verlaten:

```
/* 4.4.3.1.c */
#include<stdio.h>
int main(void)
{
int i;
void functie(void);
    for(i=0;i<5;i++)
        functie();
    return 0;
}
void functie(void)
{
static int st_var = 0;
int au_var = 0;
    printf("auto=%d, static=%d\n", au_var, st_var);
    au_var++;
    st_var++;
}
```

De functie **functie()** bevat declaraties voor twee lokale variabelen. **au_var** is een automatische variabele van het type integer en **st_var** is een statische variabele van het type integer. De automatische variabele

wordt bij elke aanroep op nul gezet, maar de initialisatie van de statische variabele wordt slechts één keer uitgevoerd bij het aanroepen van de functie. De uitvoer van het programma zal dan ook zijn:

```
auto=0, static=0
auto=0, static=1
auto=0, static=2
auto=0, static=3
auto=0, static=4
```

Als u het voorbeeld compileert kan het zijn dat u een waarschuwing krijgt in de vorm van "au_var is assigned a value that is never used". Als u een beetje nadenkt begrijpt u waarom. Het programma zou toch het gewenste resultaat moeten geven. De term **static** betekent niet alleen dat een variabele permanent is, maar ook dat deze in zekere mate privé-eigendom is. Interne statische variabelen zijn alleen bekend binnen de functie waarin ze gedeclareerd zijn.

4.4.4 register

Register variabelen worden in C gebruikt als de verwerkingssnelheid van belang is. De idee achter de register geheugenklasse is om de compiler één van de processor-registers te laten reserveren. Uiteraard is de register geheugenklasse bedoeld voor variabelen die intensief gebruikt worden, zoals loop-tellers. De opslagklasse register is alleen toepasbaar op automatische variabelen en functie-argumenten. Ze wordt aangegeven door het gereserveerde woord **register** voor een normale declaratie. Voor registervariabelen gelden echter enkele beperkingen i.v.m. het aantal beschikbare registers en de variabeltypes.

De compiler is op geen enkele manier verplicht om uw suggestie te volgen. Als deze ervoor kiest om uw raad links te laten liggen, heeft de variabele dezelfde opslagklasse als wanneer hij gewoon gedeclareerd zou zijn. Van een variabele een register variabele proberen te maken verhindert u ook om het adres ervan te gebruiken en er een pointer naar te richten, aangezien er geen mogelijkheid is om een geheugenpointer naar een CPU-register te richten.

4.5 enkele begrippen i.v.m. het geheugen

Er zijn drie gebieden in het geheugen die speciale mogelijkheden hebben. Deze worden aangewezen door de compiler en de linker: (Deze begrippen en het gebruik ervan is wel nogal systeemspecifiek)

4.5.1 stack

De stack wordt door het systeem toegekend met een vaste grootte en wordt van boven naar onder gevuld met één element per keer. Elementen worden van ook boven naar onder verwijderd, één element per keer. Dus, het laatste element dat toegevoegd werd aan de stack is het eerste element dat verwijdert wordt wanneer het niet langer nodig is. Elk element op de stack is potentieel beschikbaar voor verwijzing of aanpassing door de uitvoerend code, maar de elementen worden enkel vanaf de top toegevoegd.

4.5.2 heap

Dit is het gebied dat naast de stack ligt. De heap wordt door het systeem voorbehouden met een vaste grootte en gebruikt in willekeurige volgorde. Dit wil niet zeggen dat er geen volgorde inzit, het betekent dat het geheugen niet op een vooraf vastgelegde manier wordt toegekend. In feite kan het gebruikt worden in blokken als het nodig is vanuit de heap. Geheugen in de heap dat niet gebruikt wordt staat in de free list, die aangesproken kan worden, indien nodig.

4.5.3 globaal geheugen

Dit is een verzameling van al het geheugen van de machine, dat niet aan de stack of heap toegewezen wordt.

4.6 optimalisatie

Het declareren van een functie buiten de scope van de **main()** is netter en eenvoudiger te onderhouden. Merk ook op dat de **main()** een integer returned. Voorbeelden in deze cursus zullen dit voorbeeld misschien niet altijd volgen, omdat het ook niet altijd verplicht is. Het zou een goede oefening zijn om de voorbeelden proberen te optimaliseren. Om de uitvoeringssnelheid van een programma te verbeteren kan u altijd het volgende proberen. Pas wel op dat dit de leesbaarheid van de broncode niet al te veel ten nadele komt.

1. Gebruik arrays in plaats van switch of if/else statements.
2. Reduceer. Verwerk niets als het ook niet nodig is.
3. Gebruik bibliotheekfuncties in plaats van een zelf ontworpen for loop, indien mogelijk.
4. Verander indien nodig de manier waarop een programma is georganiseerd.
5. Kijk hoe de compiler bepaalde zaken vertaalt. Herschrijf uw eigen functies om dit te optimaliseren.

---

opdrachten

In dit hoofdstuk werden een boel programma's getoond. Compileer en run ze en schrijf dan soortgelijke programma's, waaraan u telkens een paar dingen verandert en het resultaat daarvan bestudeert.

1. Schrijf een programma dat uw naam tien maal op het scherm drukt, door een functie op te roepen om het schrijven te doen. Verplaats de oproepfunctie ook eens voor de **main** om te kijken of uw compiler het toelaat.

2. Schrijf een programma voor de berekening van oppervlakten van vierkanten, rechthoeken en trapezia.

- Het oppervlak van een vierkant met zijde a is gelijk aan a^2.
- Het oppervlak van een rechthoek met zijden a en b is ab.
- Van een trapezium met evenwijdige zijden a en b en hoogte h is het oppervlak ½(a+b)h.

Gebruik zoveel verschillende functies als u nodig acht.

3. Schrijf een functie die een datum uitschrijft op grond van drie getallen: één voor de dag, één voor de maand en één voor hetjaar. Als de eeuwaanduiding ontbreekt, dan moet die er bij gezet worden. Je mag ervan uitgaan dat alle data tussen 1900 en 2000 liggen. Bijvoorbeeld:

Functie aanroepen met	*Uitvoer van de functie*
31, 1 en 50	31 januari 1950
16, 10 en 1960	16 oktober 1960

4. Schrijf een programma waarin je de functie **ggd()** opneemt die op een recursieve manier de grootste gemene deler van twee niet-negatieve integers bepaalt. Het inlezen van de twee getallen en het

afdrukken van hun grootste gemene deler gebeurt in het hoofdprogramma. Het berekenen gebeurt in de functie m.b.v. het volgend algoritme:

ggd(a,b)=a als a=b

ggd(a,b)=ggd(b,a%b) als a>b en a%b is niet 0

C-taal voor beginners - hoofdstuk 5

pointers en geheugenmanipulatie

5.1 wat is een pointer?

Eén van de dingen die beginners in de C-taal als moeilijk ervaren is het concept van pointers. Ik heb ondervonden dat de voornaamste reden een zwak of minimaal gevoel voor variabelen is. In het eerste hoofdstuk werd het begrip variabele reeds uit de doeken gedaan. Omdat het begrijpen ervan zeer belangrijk is, even een andere benadering.

Een variabele in een programma is iets met een naam, waarvan de waarde kan variëren. De compiler kent een specifiek blok geheugen toe aan een variabele, om de waarde van die variabele in de computer bij te kunnen houden. De grootte van dat blok hangt af van het bereik waarover de variabele mag variëren. De grootte van het type hangt feitelijk af van de compiler.

Wanneer we een variabele declareren informeren we de compiler over twee dingen: de naam van de variabele en het type van de variabele. We declareren, bijvoorbeeld, een variabele van het type integer met de naam **k** op de volgende manier:

```
int k;
```

Wanneer de compiler de "int" tegenkomt zet deze bijvoorbeeld 2 bytes geheugen opzij om de waarde van die integer vast te houden. Er wordt ook een tabel met symbolen opgesteld. In die tabel wordt het symbool **k** toegevoegd en het relatieve adres in het geheugen, waar die 2 bytes opzij gezet werden. Dus als we later schrijven:

```
k = 2;
```

verwachten we dat, wanneer dit statement uitgevoerd wordt, de waarde **2** in die geheugenlocatie wordt geplaatst die reeds opzij gezet werd voor de opslag van de waarde van **k**. In C verwijzen we naar een variabele zoals de integer **k** als een "object". Eigenlijk zijn er twee "waarden" die geassocieerd worden met het object **k**. Eén ervan is de waarde van de integer, **2** in ons voorbeeld, en de andere is de "waarde" van de geheugenlocatie, het adres van **k**. Men verwijst naar deze twee waarden met de naamgeving "rvalue" (rechtse waarde) en "lvalue" (linkse waarde). We kunnen lvalue beschouwen als de waarde die toegelaten wordt aan de linkse zijde van de toekenningsoperator **=**. De rvalue is degene aan de rechtse kant van de toekenningsoperator, in ons voorbeeld is dat de **2**. Dus **2 = k;** is illegaal.

Beschouw nu het volgende:

```
int j, k;
k = 2;
j = 7; /* lijn 1 (zie tekst) */
k = j; /* lijn2  (zie tekst) */
```

In dit voorbeeld interpreteert de compiler de **j** in lijn 1 als het adres van de variabele **j** (lvalue) en creëert de code om de waarde **7** naar dat adres te kopiëren. In lijn 2, anderzijds, wordt de **j** geïnterpreteerd als zijn rvalue, aangezien deze aan de rechtse kant van de **=** staat. Dat wil zeggen: de **j** verwijst naar de waarde die in de geheugenplaats zit, die opzij gehouden werd voor **j** (**7** in dit geval). Dus de **7** wordt gekopieerd naar het adres dat gereserveerdd werd door de lvalue van **k**.

Stel nu dat we een variabele willen om een lvalue vast te houden (een adres). De grootte die daarvoor vereist is hangt af van het systeem. Op oudere desktop computers kan het geheugen in 2 bytes bijgehouden worden. Nieuwere systemen hebben meer geheugen nodig om een adres te bewaren. De eigenlijke grootte is niet zo belangrijk, zolang we de compiler maar laten weten dat we een adres willen bewaren. Zulk een variabele noemt men een pointer variabele. In C definiëren we een pointer variabele door er een asterisk ***** voor te plaatsen. We geven ook het type weer dat, in dit geval, verwijst naar het type van data opgeslagen in het adres waar we onze pointer gaan bewaren:

```
int *ptr;
```

ptr is de naam van onze variabele. De ***** informeert de compiler dat we een pointer variabele willen. De **int** laat weten dat we onze variabele willen gebruiken om het adres van een integer vast te houden. Zulk een pointer "wijst" of wordt als het ware gericht naar een integer. In het Engels zegt men "point to", waarvan de benaming pointer afgeleid werd. Merk op dat toen we **int k;** schreven we **k** geen waarde gaven. Indien zo'n definitie buiten een functie gedaan wordt, zal de compiler de waarde op 0 zetten. Dit is ook het geval met een pointer variabele, zodat deze zeker niet naar een object in C verwijst. Een pointer die op deze manier gedeclareerd wordt is een "null" pointer. In feite wijst zulk een pointer naar "het niets". Het eigenlijke patroon dat gebruikt wordt voor een null pointer is niet altijd gelijk aan 0, aangezien dit afhangt van het systeem. Om de broncode toch compatibel te maken tussen verschillende compilers, is er een macro voorzien om een null pointer te representeren, onder de naam **NULL**. Dus, een pointer initialiseren door hem naar **NULL** te wijzen, zoals **ptr = NULL**, verzekert ons dat deze echt een null pointer geworden is. Op dezelfde manier waarop men een integer test op de waarde 0, zoals in **if(k == 0**), kan men een null pointer testen: **if(ptr == NULL)**.

Terug naar onze nieuwe variabele **ptr**. Veronderstel nu dat we er het adres van onze integer variabele **k** in willen stoppen. U weet reeds dat we hiervoor de adresoperator **&** gebruiken:

```
ptr = &k;
```

Wat min of meer wil zeggen: "kopieer het adres van k naar ptr". De **&** operator neemt de lvalue (adres) van **k**, niettemin dat **k** aan de rechterkant van de toekenningsoperator **=** staat, en kopieert dit naar de inhoud van onze pointer **ptr**. Nu wijst **ptr** dus naar **k**. Naast de adresoperator **&** hebben we ook nog de indirectieoperator * (asterisk). Deze wordt als volgt gebruikt:

```
*ptr = 7;
```

De waarde **7** wordt gekopieerd naar het adres waarnaar **ptr** wijst. Dus als **ptr** wijst naar (bevat het adres van) **k**, zal ***ptr = 7;** de waarde van **k** op **7** zetten. Dat is zo als we de ***** op deze manier gebruiken, namelijk om naar de waarde te verwijzen van datgene waarnaar **ptr** wijst, niet naar de waarde van de pointer zelf. Op dezelfde manier zouden we kunnen schrijven:

```
printf("%d\n", *ptr);
```

om de integer waarde af te drukken die bewaard wordt op het adres waar **ptr** naar wijst. Om aan te tonen hoe dit allemaal bij elkaar past, een voorbeeldprogramma. Compileer en run dit en bekijk de code en output aandachtig:

```
/* 5.1.1.c */
#include<stdio.h>

int j, k;
int *ptr;

int main(void)
```

```
{
    j = 1;
    k = 2;

    ptr = &k;

    printf("\n");
    printf("j heeft de waarde %d en wordt bewaard op adres %p\n",
j, (void *)&j);
    printf("k heeft de waarde %d en wordt bewaard op adres %p\n",
k, (void *)&k);
    printf("ptr heeft de waarde %p en wordt bewaard op adres
%p\n", ptr, (void *)&ptr);
    printf("De waarde van de integer waarnaar ptr wijs is %d\n",
*ptr);
    return 0;
}
```

Nieuw is **%p**, dat gebruikt wordt om het adres van een pointer af te drukken. We moeten ook de aspecten van C nog bespreken, die het gebruik van de **(void *)** expressie vereisen. Includeer het op dit moment gewoon in uw code. We verklaren de reden achter deze expressie later in dit hoofdstuk. Omdat het begrijpen van pointers belangrijk is, een overzicht van deze paragraaf:

- een variabele wordt gedeclareerd door hem een type en een naam te geven (vb: **int k;**)
- een pointer variabele wordt gedeclareerd door hem een type en een naam te geven (vb: **int *ptr**), waar de asterisk de compiler vertelt dat de variabele **ptr** een pointer variabele is en het type vertelt de compiler naar welk type de variabele moet wijzen, integer in dit geval
- eens een variabele is gedeclareerd, kunnen we zijn adres verkrijgen door de naam voor te gaan met **&**, zoals in **&k**
- om een pointer te "indirecteren", anders gezegd om de waarde ervan te krijgen, gebruiken we de indirectieoperator *****, zoals in ***ptr**
- **&** en ***** zijn unaire operators
- een lvalue van een variabele is niets anders dan de waarde van zijn adres

5.2 pointertypes en arrays

[ref. C FAQ - Sectie 6]

Opmerking van een lezer bij deze paragraaf: "Het probleem is dat jij er vanuit gaat dat het adres van **mijn_array[0]** lager is dan het adres van **mijn_array[1]**. Op je PC (en vele andere machines) zal dit best kloppen maar daar kan je niet 100% zeker van zijn. Er zijn namelijk machines die het precies andersom doen (het zijn er niet veel, maar het is toch een reëel probleem)."

Het is dus mogelijk dat hetgeen hier uitgelegd wordt niet helemaal overeenkomt met de manier waarop uw systeem werkt. Meestal zal het echter wel kloppen en kan u het veilig toepassen.

Beschouw het volgende:

```
int mijn_array[ ] = {1,23,17,4,-5,100};
```

Hier hebben we een array met als inhoud zes integers. We verwijzen naar elk van deze integers met een subscript, bijvoorbeeld **mijn_array[0]** t.e.m. **mijn_array[5]**. Als alternatief kunnen we ook een pointer gebruiken:

```
int *ptr;
ptr = &mijn_array[0]; /* wijst de pointer naar de eerste integer
in ons array */
```

Dan kunnen we ons array afdrukken, gebruik makende van de arraynotatie of van de indirectieoperator. De volgende code illustreert dit duidelijk:

```
/* 5.2.1.c */
#include<stdio.h>

int mijn_array[ ] = {1,23,17,4,-5,100};
int *ptr;

int main(void)
{
    int i;

    ptr = &mijn_array[0];

    printf("\n\n");

    for(i=0;i<6;i++)
    {
        printf("mijn_array[%d] = %3d     ", i, mijn_array[i]); /*
lijn A (zie tekst) */
        printf("ptr + %d = %d\n", i, *(ptr + i)); /* lijn B (zie
tekst) */
    }

    return 0;
}
```

Met als uitvoer:

```
mijn_array[0] = 1       ptr + 0 = 1
mijn_array[1] = 23      ptr + 1 = 23
mijn_array[2] = 17      ptr + 2 = 17
mijn_array[3] = 4       ptr + 3 = 4
mijn_array[4] = -5      ptr + 4 = -5
mijn_array[5] = 100     ptr + 5 = 100
```

Bestudeer lijn A en B en merk op dat de compiler in beide gevallen dezelfde waarden afdrukt. Bekijk ook hoe we de indirectie gebruikten in lijn B. Verander nu lijn B als volgt:

```
printf("ptr + %d = %d\n", i, *ptr++);
```

en compileer en run het programma opnieuw. Verander lijn B daarna als volgt:

```
printf("ptr + %d = %d\n", i, *(ptr++));
```

en probeer opnieuw. Voorspel elke keer de uitkomst en bestudeer de output.

In C is het de standaard om waar we **&var_naam[0]** kunnen gebruiken, dat veranderen in **var_naam**, dus in onze code kunnen we **ptr = &mijn_array[0];** veranderen in **ptr = mijn_array;** om hetzelfde resultaat te verkrijgen. Dit heeft als gevolg dat velen zeggen dat de naam van een array een pointer is. Ik verkies het volgende te denken: "de naam van het array is het adres van het eerste element in het array". Vele beginners worden verward als ze het bekijken als een pointer. Bijvoorbeeld, terwijl we wel kunnen schrijven:

```
ptr = mijn_array;
```

kunnen we niet schrijven:

```
mijn_array = ptr;
```

De reden is dat **ptr** een variabele en **mijn_array** een constante is. De locatie waar het eerste element van **mijn_array** zal opgeslagen worden kan niet veranderd worden vanaf het moment dat **mijn_array[ ]** gedeclareerd is. Laten we nu wat dieper ingaan op het verschil tussen de namen **ptr** en **mijn_array**, zoals we ze hierboven gebruikten. Sommige schrijvers verwijzen naar een naam van een array als een constante pointer. Om de term "constant" in deze context te begrijpen gaan we even terug naar onze definitie van de term "variabele". Wanneer we een variabele declareren zetten we een stukje geheugen opzij om de waarde van het juiste type vast te houden. Eénmaal dat gebeurd is kan de naam van de variabele op een paar manieren geïnterpreteerd worden. Indien de variabele gebruikt wordt aan de linkse kant van de toekenningsoperator, zal de compiler dit bekijken als de geheugenplaats waar de waarde, die het resultaat is van de rechtse statement(s), naar verplaatst moet worden. Maar, wanneer aan de rechtse kant gebruikt, wordt de de variabele bekeken als de waarde om in die geheugenlocatie te plaatsen. Dat klinkt allemaal nogal complex, maar is het in feite niet.

Met het vorige in gedachten, laten we de simpelste manier van constanten beschouwen:

```
int i, k;
i = 2;
```

Terwijl **i** een variabele is en dus ruimte inneemt in het data gedeelte van het geheugen, is **2** een constante die direct in het code gedeelte van het geheugen terecht komt. Wanneer we iets schrijven als **k = i;** vertellen we de compiler om code te creëeren die tijdens het runnen naar de geheugenlocatie **&i** zal kijken om de waarde te bepalen om naar **k** te verplaatsen. Maar **i = 2;** zal simpelweg **2** in de code plaatsen en naar niets anders verwijzen. Dus **k** en **i** zijn objecten, **2** niet.
Op dezelfde manier, aangezien **mijn_array** een constante is zal de compiler het adres kennen van **mijn_array[0]**, éénmaal hij beslist waar het array op zichzelf opgeslagen zal worden. Dus bij het zien van

```
ptr = mijn_array;
```

gebruikt de compiler dit adres als een constante en is er verder geen andere verwijzing.

Dit is een goed moment om het gebruik van de **(void *)** expressie verder te verklaren, die we eerder in een programma gebruikten. Zoals u reeds weet kunnen we pointers hebben van verschillende types. Tot dusver hebben we pointers naar integers en pointers naar karakters besproken. We hebben ook geleerd dat op verschillende systemen de grootte van een array kan variëren. Het is echter ook mogelijk dat de grootte van een pointer kan variëren, afhankelijk van het data type van het object waarnaar de pointer wijst. Dus net zoals met integers, waarbij je problemen kunt ervaren bij het toekennen van een **long int** aan een variabele van het type **short int**, kunnen er problemen opdagen bij het toekennen van de waarde van een pointer aan pointer variabelen van andere types. Om dit probleem te minimaliseren, voorziet C het type **void** voor een pointer. We kunnen zo'n pointer als volgt declareren:

```
void *ptr;
```

Een void pointer is eigenlijk een soort van een generieke pointer. Bijvoorbeeld: alhoewel C een vergelijking tussen een pointer naar het type **int** en een pointer naar het type **char** niet toelaat, kunnen deze wel vergeleken worden met een **void** pointer. U kan natuurlijk wel ingewikkelde manieren bedenken om een pointer van het ene type naar het andere te converteren.

5.3 pointers en strings

Het bestuderen van strings is nuttig voor het begrijpen van de relatie tussen pointers en arrays. Het maakt het ook makkelijk te illustreren hoe sommige van de standaard C stringfuncties toegepast kunnen worden. Uiteindelijk maakt het duidelijk hoe en wanneer pointers kunnen en zouden moeten gepasseerd worden naar functies.

U wist reeds dat in C strings eigenlijk karakterarrays zijn. Dit is niet altijd zo bij andere programmeertalen. In BASIC, Pascal, Fortran en een reeks andere talen, heeft een string zijn eigen datatype. In C is dit niet zo. Daar is een string een array van het type **char**, afgesloten met een binair null karakter (geschreven als '\0' of soms gewoon 0). Om onze discussie te beginnen zullen we een stukje code schrijven dat, aangezien het enkel als illustratie dient, u waarschijnlijk nooit zou gebruiken in uw eigen programma. Bijvoorbeeld:

```
char mijn_string[40];
mijn_string[0] = 'T';
mijn_string[1] = 'e';
mijn_string[2] = 'd';
mijn_string[3] = '\0';
```

Het eindresultaat is een string die eigenlijk een array van karakters is, afgesloten met het null karakter. Let op: "null" is niet hetzelfde als "NULL". De null verwijst naar een 0, zoals gedefinieerd met de escape sequentie '\0'. Dit neemt 1 byte geheugen in beslag. **NULL** is de naam van de macro om null pointers te initialiseren en is gedefinieerd in een header van uw C compiler; null is mogelijk niet eens gedefinieerd. Aangezien de code hierboven zeer tijdrovend is, voorziet C twee alternatieve manieren om hetzelfde resultaat te verkrijgen. Ten eerste kan u schrijven:

```
char mijn_string[40] = {'T', 'e', 'd', '\0'};
```

Maar dit is ook niet echt efficiënt. Daarom laat C het volgende toe:

```
char mijn_string[40] = "Ted";
```

Wanneer de dubbele quotes **" "** gebruikt worden, in plaats van de enkele quotes **' '**, wordt het null karakter '\0' automatisch achteraan de string geplaatst. In elk van de bovenvernoemde gevallen gebeurt hetzelfde. De compiler zet een aaneengrenzend stuk geheugen van 40 bytes opzij om karakters vast te houden en initialiseert de eerste 4 karakters als "Ted\0".
Tot hier de korte herhaling van strings en arrays. Beschouw nu het volgende programma.

```
/* 5.3.1.c */
#include<stdio.h>

char strA[80] = "Een string die gebruikt wordt als demonstratie";
char strB[80];

int main(void)
{
```

```
    char *pA;    /* een pointer naar het char type */
    char *pB;    /* nog een pointer naar het char type */

    puts(strA);  /* toon string A */
    pA = strA;   /* wijs pA naar string A */
    puts(pA);    /* toon waar pA naar wijst */
    pB = strB;   /* wijs pB naar string B */
    putchar('\n');      /* een lege lijn */

    while(*pA != '\0')  /* lijn A (zie tekst) */
    {
        *pB++ = *pA++;  /* lijn B (zie tekst) */
    }

    *pB = '\0';  /* lijn C (zie tekst) */
    puts(strB);  /* toon strB */

    return 0;
}
```

In ons voorbeeldprogramma beginnen we met het definiëren van twee karakter arrays van elk 80 karakters. Omdat deze globaal zijn, worden ze door de compiler eerst geïnitialiseerd op '\0', zoals eerder uitgelegd. Daarna worden de gegeven karakters van **strA** geïnitialiseerd tussen enkele quotes. We declareren twee karakter pointers en tonen de string op het scherm. Daarna "wijzen" of richten we de pointer **pA** naar **strA**. In feite kopiëren we het adres van **strA[0]** naar onze variabele **pA**. Daarna gebruiken we **puts()** om hetgeen te tonen waar **pA** naar verwijst. In hoofdstuk 3 hadden we het reeds even over **gets()** en **puts()**. Deze functies dienen respectievelijk om een string om te nemen en om op het scherm te tonen. De variabele (de string) wordt tussen de ronde haken geplaatst. Het functieprototype (als gedefinieerd door de compiler) voor **puts()** is als volgt:

```
int puts(const char *s);
```

Negeer op het moment de "const". De parameter die naar **puts()** gepasseerd wordt is een pointer, meerbepaald de waarde van een pointer (aangezien alle parameters in C via "passed by value" werken). De waarde van een pointer is het adres van hetgeen waarnaar hij wijst. Dus wanneer we **puts(strA);** schrijven geven we het adres van **strA[0]** door. Als we **puts(pA);** schrijven geven we hetzelfde adres door, aangezien we **pA = strA;** schreven. Met dat gegeven in gedachte, bekijken we de code van de **while()** structuur op lijn A. Lijn A zegt in feite: zolang het karakter waar **pA** naar wijst niet het null karakter is, doe het volgende: kopieer het karakter waar **pA** naar wijst naar de ruimte waar **pB** naar wijst. Incrementeer **pA** daarna zodat deze naar het volgende karakter wijst en **pB** zodat deze naar de volgende ruimte wijst (lijn B). Wanneer we het laatste karakter gekopieerd hebben wijst **pA** naar het afsluitende null karakter en de loop eindigt. We hebben het null karakter zelf niet gekopieerd, maar in C moet een string wel met een nul karakter afgesloten worden. Daarom voegen we dit toe (lijn C).

Bekijk opnieuw het prototype voor **puts()**. De "const", die gebruikt wordt als parameterwijziger, informeert de gebruiker dat de functie de string waar **s** naar wijst niet zal aanpassen of veranderen, dus wordt deze string behandelt als een constante.
Wat het bovenstaande voorbeeldprogramma illustreert is een simpele manier om een string te kopiëren. Nadat u het voorbeeld voldoende bestudeerde en volledig begrijpt, kunnen we onze eigen vervanging voor **strcpy()**, dat in hoofdstuk 3 besproken werd, verzinnen. Dit kan er als volgt uitzien:

```
char *mijn_strcpy(char *bestemming, char *bron)
{
    char *p = bestemming;
    while (*bron != '\0')
```

```
    {
        *p++ = *bron++;
    }
    *p = '\0'
    return bestemming;
}
```

Ik gebruik hier de Nederlandse benamingen "bestemming" en "bron", maar meestal zal u respectievelijk de Engelse benamingen "destination" en "source" tegenkomen in referenties en compilerdocumentatie. In dit geval heb ik de standaardroutine voor het terugkeren (**return**) van een pointer naar de bestemming gebruikt. Opnieuw is de functie ontworpen om de waarden van twee karakter pointers aan te nemen. Dus in ons programma zouden we kunnen schrijven:

```
int main(void)
{
    mijn_strcpy(strB, strA);
    puts(strB);
    return 0;
}
```

Ik wijk hier misschien lichtjes af van de vorm die in standaard C gebruikt wordt, welke er als volgt zou uitzien:

```
char *mijn_strcpy(char *bestemming, const char *bron);
```

Hier wordt de "const" bepaling gebruikt om te verzekeren dat de functie de inhoud, waarnaar de bronpointer naar wijst, niet zal veranderen. Dit kan bewezen worden door bovenstaande functie aan te passen, alsook het prototype, om de "const" bepaling te includeren, zoals getoond. In de functie kan u dan een statement toevoegen dat probeert om de inhoud te veranderen van datgene waar de bron naar wijst, zoals:

```
*bron = 'X';
```

wat normaal gezien het eerste karakter van de string in een 'X' zou veranderen. Door de **const** zou uw compiler dit als een fout moeten zien. Probeer dit uit.

Laten we nu enkele zaken beschouwen die de voorbeelden ons getoond hebben. Ten eerste, beschouw het feit dat ***ptr++** geïnterpreteerd moet worden als het teruggeven (**return**) van de waarde waar **ptr** naar wijst, waarna de pointerwaarde geïncrementeerd wordt. Dit heeft te maken met de prioriteit van de operators. Indien we **(*ptr)++** schreven zouden we niet de pointer incrementeren, maar wel datgene waar hij naar wijst, aangezien ronde haken voorgaan op de incrementie. Bijvoorbeeld, indien toegepast op het eerste karakter van bovenstaande voorbeeldstring, zou de 'T' geïncrementeerd worden tot 'U'. U kan zelf een simpel stukje code schrijven om dit te illustreren.

Merk op: wanneer we een integer doorgeven, maken we een kopie van deze integer. Als de doorgegeven waarde dan wordt gemanipuleerd heeft dit totaal geen effect op de originele integer. Maar bij arrays en pointers kunnen we het adres van de variabele doorgeven en vanaf dan de waarde manipuleren van de originele variabelen.

We hebben in een korte tijd reeds heel wat voortuitgang geboekt. Laten we even terugkeren naar hetgeen we deden bij het kopiëren van strings, maar nu in een ander licht. Beschouw de volgende functie:

```
char *mijn_strcpy(char dest[ ], char source[ ])
{
```

```
    int i = 0;
    while(source[i] != '\0')
    {
        dest[i] = source[i];
        i++;
    }
    dest[i] = '\0';
    return dest;
}
```

Ik gebruik hier voor de verandering de Engelse benamingen voor "bestemming" en "bron".
Herinner u dat strings arrays zijn van het type **char**. Hier hebben we gekozen voor een arraynotatie in plaats van een pointernotatie om het eigenlijke kopiëren te doen. Het resultaat is hetzelfde: de string wordt gekopieerd. Dit brengt een paar interessante zaken aan het licht.

Bij zowel het doorgeven van een karakter pointer als het doorgeven van de naam van het array, zoals hierboven, wordt het adres van het eerste element van elk array doorgegeven. Bijgevolg is de numerieke waarde van de doorgegeven parameter dezelfde, of we nu een karakter pointer of een array-naam als parameter gebruiken. Dit zou suggereren dat op de één of andere manier **source[i]** hetzelfde is als ***(source+i)**. In feite is dit waar. Wanneer iemand bijvoorbeeld **a[i]** schrijft, dan kan dat vervangen worden door ***(a+i)**, zonder enige problemen. De compiler creëert in beide gevallen dezelfde code. We kunnen dus zien dat pointer-rekenkunde hetzelfde is als array-indexering. Beide syntaxen geven hetzelfde resultaat. Dit wil niet zeggen dat pointers en arrays hetzelfde zijn; dat is niet zo. Het betekent gewoon dat om een gegeven element van een array te identificeren, we de keuze hebben tussen twee mogelijkheden, met hetzelfde resultaat als gevolg.

Bekijk nu de uitdrukking **(a+i)**. Dit is een simpele optelling, gebruik makend van de **+** operator en de regels van C dat zulk een berekening commutatief is. Dus **(a+i)** is hetzelfde als **(i+a)**. Bijgevolg kunnen we ***(i+a)** evengoed als ***(a+i)** schrijven. Maar ***(i+a)** kon ook afkomstig zijn van **i[a]**. Dit alles heeft het merkwaardige gevolg dat indien:

```
char a[20];
int i;
```

en we schrijven:

```
a[3] = 'x';
```

dit hetzelfde is als:

```
3[a] = 'x';
```

Let op: dit laatste kan u beter niet toepassen, aangezien velen het niet zullen aanvaarden als legaal programmeren. Ik toon het hier enkel om een merkwaardigheid duidelijk te maken.

Laten we nu onze bovenstaande functie opnieuw bekijken, waar we schrijven:

```
dest[i] = source[i];
```

U weet ondertussen dat we dit ook kunnen schrijven als:

```
*(dest + i) = *(source + i);
```

Dit vereist echter twee optellingen voor elke waarde die **i** aanneemt. Dus, de pointerversie kan een beetje sneller zijn dan de arrayversie. Dit is ook weer erg compiler en platform specifiek. Soms kan het ook gewoon omgekeerd zijn. Een andere manier om de methode met de pointers te versnellen zou zijn door

```
while(*source != '\0')
```

te vervangen door

```
while (*source)
```

aangezien de waarde tussen de haken op hetzelfde moment naar 0 zal gaan, in beide gevallen.

5.4 pointers naar arrays

Pointers kunnen natuurlijk naar elk type data object "gericht" worden, inclusief arrays. Dat wist u ondertussen al wel, maar het is belangrijk om dit uit te breiden naar hoe we dit doen als het om multi-dimensionele arrays gaat.

Om even te herhalen, in paragraaf 5.2 leerden we dat, indien we een array van integers hebben, we een integer pointer naar dat array kunnen wijzen, als volgt:

```
int *ptr;
ptr = &mijn_array[0]; /* wijst de pointer naar de eerste integer
in ons array */
```

Het type van de pointer variabele moet dus hetzelfde zijn als het type van het eerste element van het array. Bovendien kunnen we een pointer als een formele parameter, van een functie die ontworpen is om een array te manipuleren, gebruiken. Bijvoorbeeld: gegeven is het volgende:

```
int array[3] = {'1', '5', '7'};
void een_functie(int *p);
```

Sommige programmeurs zouden het functieprototype als volgt schrijven:

```
void een_functie(int p[ ]);
```

wat andere gebruikers van de functie ervan zou moeten inlichten dat ze ontworpen is om elementen van een array de manipuleren. Hetgeen in beide gevallen wordt doorgegeven is de waarde van een pointer naar het eerste element van het array, onafhankelijk van de notatie die gebruikt werd in het functieprototype of in de definitie. Merk op dat, indien de arraynotatie gebruikt wordt, het niet nodig is om de eigenlijke dimensie van het array door te geven, aangezien we enkel het adres van het eerste element doorgeven en niet het ganse array.

Laten we nu het probleem van het 2-dimensionele array beschouwen. C interpreteert een 2-dimensioneel array als een array van 1-dimensionele arrays. Bijgevolg is het eerste element van een 2-dimensioneel array van integers, een 1-dimensioneel array van integers. Een pointer naar een 2-dimensioneel array van integers moet een pointer zijn van datzelfde datatype. Eén manier om dit te verwezenlijken is door gebruik te maken het sleutelwoord **typedef**. Dit kent een nieuwe naam toe aan een gespecifieerd data type, bijvoorbeeld:

```
typedef unsigned char byte;
```

dit zal de naam **byte** als het type unsigned char definiëren. Van dan af zal bijvoorbeeld

```
byte b[10];
```

een array zijn van unsigned karakters. Merk op dat in de **typedef** declaratie, het woord **byte** hetgeen heeft vervangen dat normaal de naam zou zijn van onze **unsigned char**. Want de regel bij het gebruik van **typedef** is dat de nieuwe naam van het data type de naam is die in de definitie van het data type gebruikt wordt. Dus bij

```
typedef int Array[10];
```

wordt **Array** een data type voor een array van tien integers. Zo zal, bijvoorbeeld, **Array mijn_arr;** de variabele **mijn_arr** declareren als een array van tien integers en **Array arr2d[5];** maakt van **arr2d** een array van vijf arrays van tien integers elk.
Merk ook op dat **Array *p1d;** er voor zorgt dat **p1d** een pointer is naar een array van tien integers. Omdat ***p1d** naar hetzelfde type wijst als **arr2d**, het adres van het 2-dimensinele array **arr2d** aan **p1d** toekennend, is de pointer naar een 1-dimensioneel array van tien integers acceptabel, zoals **p1d = &arr2d[0];** of **p1d = arr2d;** Deze zijn beide correct.

Niettegenstaande het gebruik van **typedef** de dingen duidelijker en makkelijker maakt voor de gebruiker en programmeur, is het niet echt noodzakelijk. Wat we nodig hebben is een manier om een pointer te declareren zoals **p1d** zonder gebruik van het **typedef** sleutelwoord. Dit kan op de volgende manier gedaan worden:

```
int (*p1d) [10];
```

waarbij **p1d** dus een pointer is naar een array van tien integers, net zoals het was onder de declaratie bij het gebruik van het arraytype. Dit is niet hetzelfde als

```
int *p1d[10];
```

wat **p1d** de naam van een array van tien pointers naar het type integer zou maken.

Tenslotte: dit hoofdstuk is veruit het meest ingewikkelde van de cursus. Toch is het ook zowat het belangrijkste. Geen enkel degelijk C-programma werkt zonder pointers en, alhoewel u dit nu waarschijnlijk moeilijk kunt geloven, pointers maken het leven van de programmeur een stuk eenvoudiger. Leer dit hoofdstuk grondig en bestudeer alle voorbeelden. Neem rustig de tijd, want als u dit onder de knie heeft zal de rest meevallen.

---

opdrachten

1. Probeer uw eigen versies the schrijven van stringfuncties als **strlen()**, **strcat()**, **strchr()**, enz.. .

2. Definieer een karakter array en gebruik **strcpy()** om er een string naar te kopiëren. Druk de string af door een loop te gebruiken met een pointer om één karakter per keer af te drukken. Initialiseer de pointer op het eerste element en gebruik incrementatie. Neem een aparte integer variabele om de af te drukken karakters te tellen.

3. Schrijf een programma dat aan de gebruiker vraagt om een string in te geven. Met een loop neem je telkens één karakter op van die string. In een aparte functie druk je de string terug af. In een andere functie druk je de string in omgekeerde volgorde af. De **main()** roept **functie1()** op en **functie1()** roept **functie2()** op. Gebruik waar mogelijk pointers. Tip: om nadien de string in omgekeerde volgorde te

kunnen afdrukken, sla je elk ingescand karakter op in een string, gebruik makend van een incrementerende index.

C-taal voor beginners - hoofdstuk 6

de preprocessor

6.1 wat is een preprocessor?

[ref. C FAQ - Sectie 10]

De preprocessor is een programma dat wordt uitgevoerd, vlak voor de de uitvoering van de compiler. De werking is onopvallend voor de programmeur, maar toch zeer belangrijk. Het verwijdert alle commentaar uit de broncode en voert een reeks textuele vervangingen uit, waarna het resultaat naar de compiler gestuurd wordt.

Beschouw het volgende voorbeeld voor het gebruik van een paar defines en macro's:

```
/* 6.1.1.c */
#include <stdio.h>

#define START 0  /* start van de loop */
#define EINDE 9          /* einde van de loop */
#define MAX(A,B) ((A)>(B)?(A):(B))      /* Max macro definitie */
#define MIN(A,B)  ((A)>(B)?(B):(A))     /* Min macro definitie */

int main(void)
{
int index, mn, mx;
int teller = 5;

    for (index = START ; index <= EINDE ; index++)
    {
        mx = MAX(index, teller);/* lijn 1 (zie tekst) */
        mn = MIN(index, teller);/* lijn 2 (zie tekst) */
        printf("Max is %d en min is %d\n", mx, mn);
    }

    return 0;
}
```

Met als uitvoer:

```
Max is 5 en min is 0
Max is 5 en min is 1
Max is 5 en min is 2
Max is 5 en min is 3
Max is 5 en min is 4
Max is 5 en min is 5
Max is 6 en min is 5
Max is 7 en min is 5
Max is 8 en min is 5
Max is 9 en min is 5
```

Let op de lijnen die beginnen met **#define**. Dit is de manier waarop alle defines en macro's worden gedeclareerd. Voordat de eigenlijke compilatie begint, gaat de compiler doorheen een preprocessor-fase om alle defines op te lossen. In ons huidige voorbeeld zal de preprocessor elk voorkomen van het woord

START vervangen door een **0**. De compiler zelf zal het woord **START** nooit zien, dus voor deze waren de nullen altijd al aanwezig. Merk op dat indien het woord in een string constante of in een commentaarlijn voorkomt, het niet veranderd zal worden. Het zou duidelijk moeten zijn dat het woord **START** in het programma, in plaats van de numeral **0**, enkel voor het gemak is en in feite als toelichting dient, aangezien het woord duidelijk maakt waarvoor de nul gebruikt wordt.

In geval van een zeer klein programma, zoals ons voorbeeld, maakt het niet echt uit wat u gebruikt. Als u echter een programma met een paar duizend lijnen heeft, met 27 referenties voor **START**, zou het een heel andere zaak zijn. Indien u alle voorkomens van **START** in het programma wilt veranderen in een nieuw getal, is het simpel om dat ene **#define** statement te veranderen naar de nieuwe waarde. Wordt deze techniek niet gebruikt, zou het moeilijk zijn om alle referenties in het programma met de hand te gaan veranderen. Bij het missen van één of twee voorkomens kunnen de gevolgen al rampzalig zijn.

Op dezelfde manier gaat de preprocessor het woord **EINDE** vinden en vervangen door **9**, daarna zal de compiler het programma uitvoeren zonder te weten dat **EINDE** ooit bestond.

Het is tamelijk algemeen in het C-programmeren dat symbolische constanten, zoals **START** en **EINDE**, met volledige hoofdletters geschreven worden en dat namen van variabelen in kleine letters geschreven worden. U kan echter eender welke methode gebruiken, aangezien het een kwestie van persoonlijke smaak is.

6.2 macro's

Een macro is niets meer dan een andere define, maar aangezien hij in staat is om logische beslissingen te nemen en rekenkundige bewerkingen uit te voeren, heeft hij een unieke naam.

Bekijk de derde **#define** van ons programma voor een voorbeeld van een macro. In dit geval verwacht de preprocessor twee termen tussen ronde haken, elke keer als hij het woord **MAX** tegenkomt, en zal hij een vervanging doen van de termen in het tweede deel van de definitie. Dus, de eerste term zal elke **A** in het tweede deel van de definitie vervangen en de tweede term zal elke **B** in het tweede deel van de definitie vervangen. In de **for()** loop zal **index** vervangen worden voor elke **A** en zal **teller** vervangen worden voor elke **B**. Dus als lijn 1 aan de compiler gegeven wordt, zal deze er als volgt uitzien:

```
mx=((index)>(teller)?(index):(teller))
```

Ik benadruk nogmaals dat string constanten en commentaarlijnen niet aangepast zullen worden. **mx** zal de maximumwaarde van **index** of **teller** ontvangen. Op dezelfde manier zal de **MIN** macro er voor zorgen dat **mn** de minimumwaarde van **index** of **teller** ontvangt (lijn 2). Deze twee bijzondere macro's komen veel voor.

Let op. Bij het definiëren van een macro is het van belang dat er geen witruimte voorkomt tussen de macronaam en het eerste (openende) haakje. Anders kan de compiler niet bepalen of het een macro is en zal het bijgevolg als een simpel vervangend **#define** statement behandelen.

De resultaten van het macrogebruik in ons voorbeeld worden dan in de loop afgedrukt. Er lijken een boel overbodige haakjes aanwezig te zijn in de macro definitie, maar deze zijn essentieel. We zullen dit in ons volgende voorbeeld bespreken. Compileer en run het voorbeeld dat we tot nu toe besproken hebben en kijk of het doet wat we verwachtten, alvorens verder te gaan naar het volgende programma.

Laten we nu eens kijken naar een voorbeeld van een slechte macro:

```
/* 6.2.1.c */
#include <stdio.h>
```

```
#define FOUT(A) A*A*A    /* foute macro voor cubus */
#define CUBUS(A) (A)*(A)*(A) /* bijna juiste macro voor cubus */
#define VRKW(A) (A)*(A) /* juiste macro voor vierkantswortel */
#define TOEV_FOUT(A) (A)+(A) /* foute macro voor optelling */
#define TOEV_JUIST(A) ((A)+(A)) /* juiste macro voor optelling */
#define START 1
#define STOP   7

int main(void)
{
int i, tegengewicht;

    tegengewicht = 5;
    for (i = START ; i <= STOP ; i++)
    {
        printf("De vierkantswortel van %3d is %4d, en de cubus is
%6d\n",
         VRKW(i+tegengewicht),i+tegengewicht,
                CUBUS(i+tegengewicht)); /* lijn 1 (zie tekst) */
        printf("Het foute van %3d is %6d\n",
        i+tegengewicht, FOUT(i+tegengewicht));
    }

    printf("\nProbeer nu de optellings macro's\n");
    for (i = START ; i <= STOP ; i++)
    {
        printf("Foute optellings macro = %6d, en juist = %6d\n",
/* lijn 2 (zie tekst) */
        5*TOEV_FOUT(i), 5*TOEV_JUIST(i));      /* lijn 3 (zie
tekst) */
    }

    return 0;
}
```

dit programma geeft als uitvoer het volgende:

```
De vierkantswortel van  36 is    6, en de cubus is    216
Het foute van   6 is     16
De vierkantswortel van  49 is    7, en de cubus is    343
Het foute van   7 is     27
De vierkantswortel van  64 is    8, en de cubus is    512
Het foute van   8 is     38
De vierkantswortel van  81 is    9, en de cubus is    729
Het foute van   9 is     49
De vierkantswortel van 100 is   10, en de cubus is   1000
Het foute van  10 is     60
De vierkantswortel van 121 is   11, en de cubus is   1331
Het foute van  11 is     71
De vierkantswortel van 144 is   12, en de cubus is   1728
Het foute van  12 is     82

Probeer nu de optellings macro's
Foute optellings macro =      6, en juist =     10
Foute optellings macro =     12, en juist =     20
Foute optellings macro =     18, en juist =     30
```

```
Foute optellings macro =     24, en juist =     40
Foute optellings macro =     30, en juist =     50
Foute optellings macro =     36, en juist =     60
Foute optellings macro =     42, en juist =     70
```

Als eerste definiëren we een macro genaamd **FOUT**, die de cubus van **A** lijkt te evalueren. In sommige gevallen is dit zo, maar soms gaat dit mis. De tweede macro **CUBUS** doet het bijna altijd goed. We zullen weldra zien waarom deze niet altijd juist functioneren.

Bekijk het programma waar de **CUBUS** van **i+tegengewicht** berekend wordt (lijn 1). Als **i** 1 is, hetgeen zo de eerste keer is, dan zijn we aan het zoeken naar de cubus van 1 + 5 = 6, wat zal resulteren in 216. Wanneer we **CUBUS** gebruiken groeperen we de waarden als volgt: (1+5)*(1+5)*(1+5) = 6*6*6 = 216. Als we echter **FOUT** gebruiken dan groeperen we ze als 1+5*1+5*1+5 = 1+5+5+5 = 16, wat een verkeerd antwoord is. De ronde haken zijn daarom vereist om de variabelen op een correcte manier in te delen. Het is duidelijk dat zowel **CUBUS** als **FOUT** het juiste antwoord zouden geven als het om een enkele-term-vervanging ging, zoals in het vorige programma. Nog in de loop worden de correcte waarden van de cubus en de vierkantswortel van de cijfers afgedrukt, alsook de verkeerde waarden van de inspectie.

We definiëren ook een macro **TOEV_FOUT** volgens bovenstaande regels, maar we hebben nog steeds een probleem indien we de macro proberen te gebruiken in lijnen 2 en 3. In lijn 2, wanneer we zeggen dat we het programma **5*TOEV_FOUT(i)** willen laten bereken met i = 1, krijgen we het resultaat 5*1 + 1 welk gelijk is aan 5 + 1 of 6. Dit is zeker niet wat we in gedachten hadden. Eigenlijk wilden we het resultaat 5*(1 + 1) = 5*2 = 10 wat het antwoord is dat we krijgen als we de macro **TOEV_JUIST** gebruiken, omwille van de extra haken rond de ganse expressie in de definitie van de macro. Een kleine inspanning om het programma en zijn uitkomst te bestuderen zal zeker in uw voordeel spelen bij het begrijpen en toepassen van macro's.

Om bovenstaande problemen te vermijden includeren de meeste programmeurs ronde haken rond elke variabele in een macro en een extra paar rond de ganse expressie. Dit zorgt er voor dat elke macro juist functioneert en is tevens de reden dat onze macro **CUBUS** nog steeds fout is. Deze vereist haken rond de volledige expressie. De rest van het programma is simpel en ik laat het dan ook aan u over om het verder te bekijken.

Algemene opmerking over macro's: gebruik ze zo min mogelijk en het liefst niet voor functies. Een wijze raad die sommigen toch niet opvolgen door zeer ver te gaan met het declareren van een macro.

6.3 voorwaardelijke compilatie

```
/* 6.3.1.c */
#include <stdio.h>

#define OPTIE_1  /* Dit definieert de preprocessor controle */

#ifdef OPTIE_1   /* lijn 1 (zie tekst ) */
    int teller_1 = 17;  /* Dit bestaat alleen indien OPTIE_1
gedefinieerd is */
#endif   /* lijn 2 (zie tekst ) */

int main(void)
{
int index;

    for (index = 0 ; index < 6 ; index++)
    {
        printf("In de loop, index = %d", index);
```

```
#ifdef OPTIE_1   /* lijn 3 (zie tekst) */
        printf(" teller_1 = %d", teller_1);     /* Dit wordt
mogelijk afgedrukt */
#endif   /* lijn 4 (zie tekst) */
        printf("\n");
    }

    return 0;
}
#undef OPTION_1  /* lijn 5 (zie tekst) */
```

Dit is ons eerste voorbeeld van voorwaardelijke compilatie of "conditional preprocessing". **OPTIE_1** wordt gedefinieerd en als gedefinieerd beschouwd voor het ganse programma. Daarom, wanneer de preprocessor aan lijn 1 komt, houdt deze de tekst tussen lijn 1 en 2 in het programma en geeft deze door aan de compiler. Als **OPTIE_1** niet gedefinieerd zou zijn, zou de preprocessor alles tussen lijn 1 en 2 weggooien en zou de compiler het nooit zien. Insgelijks wordt hetgeen tussen lijn 3 en 4 staat voorwaardelijk gecompileerd, gebaseerd op het feit dat **OPTIE_1** werd gedefinieerd of niet. Dit is een zeer nuttige constructie, maar niet op de manier die we hier gebruiken. In het algemeen wordt het gebruikt om een onderwerp te includeren als we een specifieke processor gebruiken met een specifiek besturingssysteem, of zelfs een speciaal stuk hardware.

Compileer het voorbeeld en run het zoals het is. Verwijder daarna de definitie van **OPTIE_1** en compileer en run het opnieuw. U zal merken dat de extra lijn niet afgedrukt wordt omdat het weggegooit zal worden door de preprocessor. Onthoud dat de preprocessor enkel tekstuele substitutie of tekstverwijdering toepast en u het effectief zal kunnen gebruiken. Lijn 5 illustreert een undefine commando. Dit verwijdert het feit dat **OPTIE_1** was gedefinieerd en vanaf dit punt zal het programma werken alsof het **OPTIE_1** nooit gedefinieerd werd. Hier heeft dat natuurlijk geen effect omdat na onze undefine geen code meer komt. Plaats de undefine definitie voor de loop en compileer en run uw programma opnieuw. U zal zien dat het net is alsof **OPTIE_1** nooit gedefinieerd werd. Het gebruik van undefine is uiterst handig als u de waarde van een **#define** tijdens het programma wilt veranderen. Bijvoorbeeld:

```
/* 6.3.2.c */
#include <stdio.h>

#define WAARDE 1

main(void)
{

    printf("De waarde is nu %d\n", WAARDE);

#undef WAARDE

#define WAARDE 2

    printf("De waarde is nu %d\n", WAARDE);

}
```

Dit geeft het volgende:

```
De waarde is nu 1
De waarde is nu 2
```

In de meeste gevallen is de **#undef**, zoals we ze in de illustratie hierboven gebruikten, niet echt nodig. Het getuigt echter van goede stijl als u het wel zo doet.

Beschouw nu ons tweede voorbeeld:

```
/* 6.3.3.c */
#include <stdio.h>

#define OPTIE_1  /* Dit definieert de preprocessor controle */
#define PRINT_DATA      /* Als dit gedefinieerd is, zullen we
afdrukken */

#ifndef OPTIE_1
    int teller_1 = 17;   /* Dit bestaat als OPTIE_1 niet
gedefinieerd is */
#endif

int main(void)
{
int index;

#ifndef PRINT_DATA
    printf("Geen resultaten zullen gedrukt worden met deze versie
"
                          " van het programma\n");
#endif

    for (index = 0 ; index < 6 ; index++)
    {
#ifdef PRINT_DATA
        printf("In de loop, index = %d", index);
#ifndef OPTIE_1
        printf(" teller_1 = %d", teller_1);    /* Dit wordt
mogelijk afgedrukt */
#endif
        printf("\n");
#endif
    }

    return 0;
}
```

Dit illustreert de preprocessorinstructie die code includeert indien een symbool niet gedefinieerd is. De **#ifndef** instructie will letterlijk zeggen "if not defined", ofwel "indien niet gedefinieerd" en deze wijst zichzelf dan ook uit. Dit programma is een echte oefening in logica voor de ijverige student, maar zou verstaanbaar moeten zijn met een kleine inspanning. Het symbool **OPTIE_1** is omgekeerd ten opzichte van het vorige programma en **PRINT_DATA** wordt gebruikt om het afdrukken in gang te steken als het niet gedefinieerd is. Is dit het geval dan zal er een output zijn. Ook dit programma is weer een beetje dwaas maar het illustreert op een duidelijke manier de preprocessorinstructies. Ons volgende programma is al iets praktischer:

```
/* 6.3.4.c */
#include <stdio.h>

#define MIJN_DEBUG
```

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

    for (index = 0 ; index < 6 ; index++)
    {
        printf("Index is nu %d", index);
        printf(" en we kunnen de data verwerken");
        printf("\n");
#ifdef MIJN_DEBUG
        printf("The processor is nog niet gedebugged!
************\n");
#else
        for (count = 1 ; count < index * 5 ; counter++)
        {
            waarde = (zie pagina 16 van uw boek)
            grens = (vraag Bill naar een char definitie)
            Linda gebruikt for loops waar ze maar wil
            printf("teller = %d, waarde = %d, grens = %d\n,
teller, waarde, grens);
        }
#endif
    }

    return 0;
}
```

We definiëren een symbool genaamd **MIJN_DEBUG** bij het begin van het programma. Wanneer we de code in de **main()** functie tegenkomen zien we waarom het is gedefinieerd. Blijkbaar hebben we niet genoeg informatie om deze code te vervolledigen, dus hebben we ze zomaar bijeen gegooid, totdat we een kans hebben om Bill en Linda om verdere informatie te vragen. Ondertussen wensen we verder te werken aan andere delen van het programma, dus gebruiken we de preprocessor om deze tot nu toe oncompileerbare code tijdelijk "weg te gooien". In dit geval gaat het slechts om een paar lijnen code, maar het zou evengoed een groot blok kunnen zijn. We kunnen deze techniek ook gebruiken om verschillende grote blokken code te behandelen, waarvan sommige in andere modules, totdat Bill terugkomt en de analyse uitlegt waarmee we het kunnen afmaken. Compileer en run het programma en bekijk de output. Verwijder nadien de definitie van MIJN_DEBUG en u zal ongetwijfeld een boel foutmeldingen krijgen bij het compileren. U ziet dus dat indien het wel gedefinieerd is de code simpelweg niet aan de compiler getoond wordt.

Er is ook de mogelijkheid om een gewone **if** te gebruiken als voorwaarde:

```
/* 6.3.5.c */
#include<stdio.h>
#define TEST 1
int main(void)
{
int tijd, seconden, minuten, uren;
    printf("Geef het aantal uren, minuten, seconden: ");
    scanf("%d %d %d", &uren, &minuten, &seconden);
#if TEST
    printf("\nIngelezen tijd %d:%d:%d", uren, minuten, seconden);
#else
    printf("\nTEST staat op 0");
#endif
    tijd = (60 * uren + minuten) * 60 + seconden;
```

```
    printf("\nBerekende tijd : %d", tijd);
    return 0;
}
```

In plaats van **TEST** op **0** of **1** te definiëren kunnen we deze ook op **TRUE** (waar) of **FALSE** (niet-waar) zetten. De compiler aanvaardt dit op dezelfde manier. De **#else** is niet verplicht en na de **#if** en **#else** mag natuurlijk meer dan één opdracht komen. De methode met true en false kan handig zijn voor het inbouwen van testfuncties in uw programma. Als het dan éénmaal af is zet u de **#define** constante gewoon op **FALSE** en dan worden deze genegeerd.

Als praktiche variant op de debug macro kan u bijvoorbeeld ook het volgende toepassen:

```
#ifdef DEBUG
#define DEB_MSG(a) printf(a);
#else
#define DEB_MSG(a)
#endif
```

Dan kan je je programma vol zetten met debug informatie regels als:

```
DEB_MSG("We zitten nu in functie XYZ()\n")
```

Indien je **DEBUG** gedefinieerd hebt wordt het afgedrukt, anders niet.

6.4 performance consequenties van macro's

Als u een groot programma schrijft, met veel macro's kan u best rekening houden met de snelheid. Over het algemeen zijn functies sneller. Zeker als het gaat om herhaling van opdrachten. Marcos schrijven is eigenlijk text verwerken. Stel dat u de volgende macro definieert:

```
#define CUBE(A) ((A)*(A)*(A))
```

En u schrijft in uw code het volgende:

```
CUBE(CUBE(func()));
```

Dan maakt de preprocessor daar dit van :

```
((((func())*(func())*(func()))))*(((func())*(func())*(func()))))*(((
func())*(func())*(func()))))) ;
```

Als een aanroep naar **func()** nu 1 seconde duurt en een vermenigvuldiging 0.1 seconde dan kost de macro versie dus 9*1+8*0.1 = 9.8 seconden. Is **CUBE** een functie dan kost het 1+4*0.1 = 1.4 seconden. Het verschil is duidelijk.

6.5 enumeratie

Beschouw het volgende programma:

```
/* 6.5.1.c */
#include <stdio.h>

int main(void)
```

```
{
enum {GEWONNEN, GELIJKSPEL, DAG, VERLOREN, NIET_AANWEZIG} result;
/* lijn 1 (zie tekst) */
enum {ZON, MAA, DIN, WOE, DON, VRI, ZAT} dagen;

    result = GEWONNEN;
    printf("GEWONNEN = %d\n", result);
    result = VERLOREN;
    printf("VERLOREN = %d\n", result);
    result = GELIJKSPEL;
    printf("GELIJKSPEL = %d\n", result);
    result = DAG;
    printf("DAG = %d\n", result);
    result = NIET_AANWEZIG;
    printf("NIET_AANWEZIG = %d\n\n", result);

    for(dagen = MAA ; dagen < VRI ; dagen++)
        printf("The dag code is %d\n", dagen);

    return 0;
}
```

met als resultaat:

```
GEWONNEN = 0
VERLOREN = 3
GELIJKSPEL = 1
DAG = 2
NIET_AANWEZIG = 4

The dag code is 1
The dag code is 2
The dag code is 3
The dag code is 4
```

voor een illustratie van het gebruik van de **enum** type variabele.

Lijn 1 definieert de eerste enum type variabele genaamd **result**, welke een variabele is die elk van de waardes tussen de haken kan aannemen. In feite is de variabele **result** een integer type variabele en kan deze elke waarde, gedefinieerd voor een integer variabele, toegewezen krijgen. De namen tussen de haken zijn constanten van het type integer. Deze kunnen overal gebruikt worden waar het legaal is om een integer constante te gebruiken. De constante **GEWONNEN** krijgt de waarde **0** toegewezen, **GELIJKSPEL** de waarde **1**, **DAG** de waarde **2**, enz... .

De variabele **result** wordt gebruikt als een gewone integer variabele, zoals het programma aantoont. Het **enum** type van variabele is bedoeld om gebruikt te worden door u, de programmeur, als een hulpmiddel. U kan een constante genaamd **MAA** namelijk gemakkelijker in controle structuren gebruiken dan de waarde van **1**. Merk op dat **dagen** de waarden van de weekdagen toegekend krijgt in de rest van het programma. Indien u een **switch()** statement zou gebruiken, zou het nuttiger zijn om de labels **ZON**, **MAA**, enz... te gebruiken i.p.v. het meer vreemde **0**, **1**, **2**, enz... .

Ik gebruikte hoofdletters voor de enumeratiewaarden in dit programma. Dit enkel uit persoonlijke voorkeur omdat het allemaal constanten zijn. Het is echter niet vereist, maar wel de standaard geworden in de professionele C-taal.

6.6 pragma

Een pragma is een instructie aan de compiler om bepaalde eigenaardige acties uit te voeren tijdens het compileren. Alhoewel pragma's kunnen variëren van compiler tot compiler en ze niet gestandariseerd zijn, voeren ze een paar nuttige functies uit. Uw compiler ondersteund waarschijnlijk wel één of andere manier om u de optimalistie-methode te laten kiezen, door het invoegen van een pragma in de broncode. Controleer de documentatie bij uw compiler voor de pragma's die u kan gebruiken.

---

opdrachten

1. Schrijf een programma om van 7 tot -5 te tellen door af te tellen. Gebruik **#define** statements om de grensen te definiëren. Tip: u zal een decrementerende variabele nodig hebben in het derde deel van de **for()** loop controle.

C-taal voor beginners - hoofdstuk 7

bestandsinput/output

7.1 een bestand openen

Tot nu toe hebben we alleen met het scherm gewerkt. De grote kracht achter C is het kunnen gebruiken en manipuleren van bestanden. Bekijk het volgende voorbeeld voor het schrijven van data naar een bestand:

```
/* 7.1.1.c */
#include<stdio.h>
#include<string.h>
int main(void)
{
FILE *fp;
char een_string[25];
int index;

    fp = fopen("TIENLIJNEN.TXT", "w");  /* openen voor schrijven,
lijn 1 (zie tekst) */
    strcpy(een_string, "Dit is een voorbeeldlijn.");

    for (index = 1 ; index <= 10 ; index++)
        fprintf(fp, "%s Lijn nummer %d\n", een_string, index); /*
lijn 2 (zie tekst) */

    fclose(fp);  /* het bestand sluiten */

    return 0;
}
```

We beginnen zoals altijd met het **#include** statement voor **stdio.h** en we includeren ook de header voor de string functies. Daarna definiëren we een paar variabelen, inclusief een nogal vreemd uitziend nieuw type. Het type **FILE** noemt men een structure en dit is gedefinieerd in het **stdio.h** bestand. Het wordt gebruikt voor het definiëren van een bestandspointer voor het gebruik in bestandsoperaties. C vereist een pointer naar een **FILE** type voor het verstrekken van toegang tot een bestand en zoals gewoonlijk kan de naam elke geldige variabelnaam zijn. Veel schrijvers gebruiken **fp** voor de naam van het eerste voorbeeld van een pointer naar een bestand, dus zullen wij ook zo beginnen.

Voordat we naar een bestand kunnen schrijven moeten we het openen. Wat dit eigenlijk betekent is dat we aan het systeem moeten vertellen dat we een bestand willen schrijven en wat de bestandsnaam is. Dit doen we met de **fopen()** functie, zoals in het voorbeeld aangegeven. De pointer naar het bestand, **fp** in ons geval, zal naar de structuur voor het bestand en twee argumenten, die vereist zijn voor deze functie, wijzen. Het eerste argument is de bestandsnaam, het tweede het bestandsattribuut. De bestandsnaam mag eender welke naam zijn, zolang hij maar in regel is met de bepalingen voor naamgeving van uw besturingssysteem (zoals Windows). Hij mag dus kleine letters en hoofdletters bevatten. Allebei de argumenten staan afzonderlijk tussen dubbele qoutes **" "**, net als bij een string constante (zie lijn 1). Voor dit voorbeeld hebben we de naam **TIENLIJNEN.TXT** gekozen. Breng het voorbeeld in uw compiler en sla het ergens op. Zorg ervoor dat er in die map geen bestand met de naam **TIENLIJNEN.TXT** bestaat, anders wordt dit overschreven. Compileer nu het programma en run het. Ga nu naar de directory waar u het bestand opsloeg en het bestand **TIENLIJNEN.TXT** zou aangemaakt moeten zijn, een tiental bijna identieke regels bevattend.

Merk op dat we niet verplicht zijn om een string constante voor de bestandsnaam te gebruiken, zoals we dat hier deden. Dit werd enkel zo gedaan voor het gemak. We kunnen een string variabele gebruiken die reeds de bestandsnaam bevat en deze dan als het eerste argument van de **fopen()** functie nemen. Dit zal later in dit hoofdstuk duidelijk worden.

De tweede parameter is het bestandsattribuut en dit kan één van de volgende zijn (de kleine-letter-notatie is verplicht):

modus	**open voor...**
r	lezen in tekst modus
w	schrijven in tekst modus
a	toevoegen in tekst modus
rb	lezen in binaire modus
wb	schrijven in binaire modus
ab	toevoegen in binaire modus
r+	lezen en schrijven in tekst modus
w+	lezen en schrijven in tekst modus
a+	lezen en toevoegen in tekst modus
r+b of rb+	lezen en schrijven in binaire modus
w+b of wb+	lezen en schrijven in binaire modus
a+b of ab+	lezen en schrijven in binaire modus

Sommige lijken dus exact hetzelfde te doen. Het verschil zit hem in het feit dat een bestand reeds moet bestaan of niet:

7.1.1 lezen ("r")

In feite zijn er nog extra attributen beschikbaar in C om meer flexibele input en output toe te laten. Wanneer u dit hoofdstuk volledig bestudeert heeft, kan u de documentatie van uw compiler daarover raadplegen. Als er een "r" gebruikt wordt, wordt het bestand geopend voor lezen (reading), een "w" opent een bestand om ernaar te schrijven (writing) en een "a" duid aan dat we data willen toevoegen (appending), aan de data die reeds aanwezig is in het bestand. Een bestand voor lezen openen vereist dat het bestand reeds bestaat. Indien dit niet zo is zal de bestandspointer op **NULL** gezet worden en dit kan gecontroleerd worden door het programma. In ons huidige voorbeeld is dit niet zo, maar dat kunnen we gemakkelijk doen als volgt:

```
if(fp == NULL)
{
    printf("Het bestand kon niet geopend worden\n");
    return EXIT_FAILURE;
}
```

Een degelijke programmeur zal altijd zijn bestandspointers controleren om geen problemen te ondervinden bij het werken met bestanden. In vorige voorbeelden liet ik het even opzij liggen om het niet te ingewikkeld te maken. De **EXIT_FAILURE** zal zodadelijk besproken worden.

7.1.2 schrijven ("w")

Wanneer een bestand geopend is om te schrijven, wordt het aangemaakt als het nog niet bestaat en zal het gereset worden als het al wel bestaat. Dit heeft als resultaat dat data die reeds aanwezig is gewist zal worden. Als het bestand toch niet geopend kan worden, hetgeen het geval kan zijn als u een alleen-lezen

bestand wilt overschrijven, wordt er een **NULL** teruggestuurd naar de pointer. Dit zou dus best weer getest worden zoals hierboven.

7.1.3 toevoegen ("a")

Een bestand dat geopend is voor toevoegen, zal aangemaakt worden als het nog niet bestaat en het zal dan initieel leeg zijn. Bestaat het wel, dan wordt het data invoer punt op het einde van de data die reeds bestaat geplaatst, zodat nieuwe data er onmiddelijk achter komt te staan. Controleer ook hier uw **NULL** pointer.

7.2 een bestand sluiten

Om een bestand te sluiten gebruiken we de functie **fclose()** met de bestandspointer tussen de ronde haken. Dit is zeer simpel en misschien niet altijd nodig, aangezien het systeem probeert alle geopende bestanden te sluiten alvorens naar het besturingssysteem terug te keren. Ik raad u echter aan om elk bestand dat u opent ook weer te sluiten. Dit is een goede gewoonte en helpt u ook herinneren welke bestanden er op het einde van programma allemaal geopend zijn.

U kan een bestand openen voor schrijven, het sluiten, het openen voor lezen, het sluiten, het openen voor toevoegen, enz... . Elke keer zou u dezelfde pointer kunnen gebruiken, maar dit is niet verplicht. De bestandspointer is simpelweg een handig middel om naar een bestand te wijzen en u beslist naar welk bestand hij wijst.

Verwijder het bestand **TIENLIJNEN.TXT** nog niet. We zullen het verder in dit hoofdstuk nog gebruiken.

7.3 uitvoer naar een bestand

7.3.1 fprintf

Dit functioneert op bijna dezelfde manier als we dat tot nu toe gewoon waren. Het verschil is de nieuwe functienaam en de toevoeging van de bestandspointer als een van de functieargumenten. Op lijn 2 van ons programma vervangt **fprintf()** het bekende **printf()** en onze pointer naar het bestand is het eerste argument. De rest van het statement is identiek aan het **printf()** statement. U kan dus net doen alsof u naar het scherm wilt schrijven, maar nu voegt u de bestandspointer toe, zodat de compiler weet naar welk bestand hij moet schrijven.

7.3.2 één karakter per keer

```
/* 7.3.2.1.c */
#include <stdio.h>
#include <string.h>
#include <stdlib.h>     /* voor de EXIT_SUCCESS */

int main(void)
{
FILE *point;
char anderen[35];
int indexer, teller;

    strcpy(anderen, "Extra lijnen.");
    point = fopen("tienlijnen.txt", "a");
    if (point == NULL)
    {
        printf("Het bestand kon niet geopend worden\n");
```

```
        exit (EXIT_FAILURE);
    }

    for (teller = 1 ; teller <= 10 ; teller++)
    {
        for (indexer = 0 ; anderen[indexer] ; indexer++)
            putc(anderen[indexer], point);     /* één karakter
outputten */
        putc('\n', point);      /* een linefeed outputten */
    }
    fclose(point);

    return EXIT_SUCCESS;
}
```

Het programma begint met de **#include** statements. Daarna worden enkele variabelen gedefininieerd, inclusief een bestandspointer. Deze pointer werd **point** genoemd, maar we zouden eender welke naam kunnen gebruiken, zolang deze voldoet aan de regels van variabelnamen. Daarna definiëren we een string van karakters om in de output functie te gebruiken met een **strcpy()**. We zijn dan klaar om het bestand te openen voor toevoegen en dit doen we dan ook met de **fopen()** functie, behalve dat we dit keer kleine letters voor de bestandsnaam gebruiken. Dit doe ik hier enkel om te illustreren dat sommige systemen, zoals Windows en Dos, geen onderscheid maken tussen kleine letters of hoofdletters. Merk op dat het bestand voor toevoeging zal geopend worden en de uitvoer zal na de uitvoer van ons vorige programma komen staan, tenminste als u het programma in dezelfde directory compileert en runt. Als het bestand niet geopend kan worden, zal er een **NULL** waarde teruggegeven worden door de **fopen()** functie.

De constante **EXIT_FAILURE** is gedefinieerd in het **stdlib.h** bestand en is meestal gedefinieerd om de waarde **1** te hebben. De constante **EXIT_SUCCESS** is ook gedefinieerd in het **stdlib.h** bestand en heeft meestal de waarde **0**. Het besturingssysteem op uw computer kan deze teruggegeven waarden gebruiken om te bepalen of het programma normaal tot zijn einde kwam en kan dan de nodige acties ondernemen, indien nodig. Bijvoorbeeld: als een tweedelig programma wordt uitgevoerd en het eerste deel geeft een error (fout) terug, dan is er geen behoefte om het tweede deel van het programma uit te voeren. Uw compiler werkt waarschijnlijk in verschillende stappen, waarbij elke uitvoering afhangt van het succes van de vorige stap.

Ons programma is eigenlijk niet meer dan twee geneste **for()** loops. De buitenste loop is een teller tot tien zodat we de binnenste loop tien keer doorlopen. Deze roept de functie **putc()** herhaaldelijk op totdat een karakter string genaamd **anderen** nul geworden is. Dit is de afsluitende null voor de string.

Het deel van het programma waarin we geïnteresseerd zijn is the **putc()** functie. Dit voert één karakter per keer uit, waarbij het karakter het eerste argument tussen de haken is en de bestandspointer het tweede en laatste argument. Waarom de ontwikkelaar van C de pointer eerst plaatste in de **fprintf()** functie en als laatste in de **putc()** functie is een goede vraag. Dit is een punt tot discussie, waar we niet op in zullen gaan.

Wanneer de tekstlijn **anderen** uitgeput is, is er een nieuwe lijn (newline) nodig omdat deze niet in bovenstaande definitie opgenomen was. Een enkele **putc()** wordt dan uitgevoerd om het "\n" karakter, wat hetzelfde is als een **return** op het toetsenbord, terug te geven en een linefeed te doen.

Vanaf het moment dat de buitenste loop tien keer doorlopen is wordt het bestand gesloten en het programma eindigt. Compileer en run dit programma en zorg dat u het goed begrijpt. Normaal zou na het runnen **TIENLIJNEN.TXT** een paar extra lijnen moeten bevatten. Telkens u het runt komen er bij.

7.4 een bestand lezen

7.4.1 één karakter per keer

```
/* 7.4.1.1.c */
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
FILE *grappig;
int c;

    grappig = fopen("TIENLIJNEN.TXT", "r");

    if (grappig == NULL)
    {
        printf("Het bestand bestaat niet\n");
        exit (EXIT_FAILURE);
        getchar();
    }
    else
    {
        do
        {
            c = getc(grappig);  /* één karakter van het bestand
opnemen */
            putchar(c);  /* het karakter op het scherm tonen */
        } while (c != EOF);     /* herhalen tot EOF (end of file)
*/
    }
    fclose(grappig);

    return EXIT_SUCCESS;
}
```

Dit is ons eerste programma dat van een bestand kan lezen. Het begint met de ondertussen bekende **#include** statements, een paar data definities en het statement om een bestand te openen. Dit vereist geen extra uitleg, behalve voor het feit dat we hier "r" gebruiken omdat we van het bestand willen kunnen lezen. We controleren in dit voorbeeld of het bestand bestaat en indien dit zo is executeren we het algemene gedeelte. Als het niet bestaat drukken we een boodschap af en stoppen we. De pointer zal in dit geval weer op NULL gezet worden. U ziet dat ik hier voor de verandering ook een **getchar()** toegevoegd heb, na de foutmelding. Dit doe ik omdat anders de boodschap gedrukt wordt en er onmiddelijk afgesloten wordt, zonder dat u iets ziet. Met de **getchar()** wacht het programma op een return alvorens te stoppen. Dit is een zeer geliefde methode om ervoor te zorgen dat een programma tijdens het compileren te bekijken is alvorens alles gebeurd is. In de vorige voorbeelden kan u dit zelf toepassen.

Het hoofddeel van ons programma is een **do while()** loop waarin één enkel karakter van het invoerbestand wordt gelezen en getoond wordt op het scherm, totdat een **EOF** ('end of file' of 'einde van het bestand') wordt gevonden. Het bestand wordt dan gesloten en het programma stopt.

Op dit moment moeten we opletten voor één van meest voorkomende en verbijsterende problemen bij het programmeren in C. De variabele die teruggegeven wordt bij de **getc()** functie is een karakter, dus kunnen we een **char** variabele gebruiken voor dit doel. Er is een probleem dat zich hier zou kunnen ontwikkelen indien we een **unsigned char** zouden gebruiken, omdat C een -1 teruggeeft voor een **EOF**. Een **unsigned char** type variabele kan geen negatieve waarden bevatten, maar enkel de waarden van 0 tot 255, dus wordt er een 255 teruggeven voor een -1, wat nooit op de juiste manier kan vergeleken worden met de **EOF**. Dit is een zeer frustrerend probleem als men het zelf zou moeten vinden. Het programma zal

nooit de **EOF** vinden en daarom zal de loop nooit eindigen. Dit kan echter voorkomen worden. Gebruik altijd een **int** type variabele als de return een **EOF** kan zijn, omdat een **int** altijd **signed** (positief en negatief) is. Volgens de ANSI-C standaard kan een **char** ofwel als **signed**, ofwel als **unsigned** toegepast worden, door welk type van compiler dan ook.

Sommige compilers gebruiken een **char** type dat niet 8 bits lang is. Als uw compiler een ander aantal bits dan 8 gebruikt voor het **char** type, dan gelden dezelfde mogelijke problemen. Gebruik geen **unsigned** type als u op een **EOF** moet controleren, teruggeven door de functie, omdat **EOF** meestal als -1 gedefinieerd wordt en dit dus niet zal passen. Er schuilt echter nog een ander probleem in ons programma, maar daar zullen we ons zorgen om maken als we bij het volgende programma komen en we zullen het oplossen met het daaropvolgende programma.

Nadat u het voorbeeld compileerde en runde zou het een goede oefening zijn om de naam van **TIENLIJNEN.TXT** te veranderen om te controleren of de **NULL** test ook werkt zoals we het bedoelden. In een echt produktieprogramma zou u het programma eigenlijk niet beëindigen. U zou de gebruiker de mogelijkheid kunnen geven om een andere bestandsnaam op te geven. We hebben interesse in het illustreren van de basis van de bestands-handelingstechnieken, dus gebruiken we een zeer eenvoudige error-manipulatie methode.

7.4.2 één woord per keer

```
/* 7.4.2.1.c */
#include <stdio.h>

int main(void)
{
FILE *fp1;
char eenwoord[100];
int c;

    fp1 = fopen("TIENLIJNEN.TXT", "r");

    do
    {
        c = fscanf(fp1, "%s", eenwoord);       /* één woord uit
het bestand opnemen */
        printf("%s\n", eenwoord);       /* het woord op het scherm
*/
    } while (c != EOF);  /* herhalen tot EOF */

    fclose(fp1);

    return 0;
}
```

Dit programma is bijna identiek aan het vorige, behalve dat we nu één woord per keer opnemen uit het bestand, met de **fscanf()** functie. Deze functie zal, net zoals onze bekende **scanf()** functie, stoppen met lezen zodra er een een witruimte of een newline karakter gevonden wordt. Dit drukken we dan telkens af met een gewone **printf()**. U kan dit zien als u het compileert en runt, in dezelfde map als het bestand **TIENLIJNEN.TXT**, maar eerst bestuderen we nog een programmeerprobleem.

Het is aan u om op de juiste manier een controle in te voegen voor het openen van het bestand. Zorg ook voor een juiste respons indien het niet geopend kan worden. Een betekenisvolle respons is om simpelweg een foutboodschap te tonen en te "exiten" naar het systeem, zoals we in ons vorige voorbeeld al deden.

Een gedetailleerde kijk op ons programma zal onthullen dat wanneer we data inlezen en de **EOF** vinden, we iets afdrukken vooraleer we op de **EOF** controleren en daardoor een extra lijn afdrukken. Wat we meestal afdrukken is hetzelfde dat afgedrukt werd de vorige keer dat we door de loop gingen, aangezien dit nog in de buffer **eenwoord** zit. Daarom moeten we op de **EOF** controleren, voordat we de **printf()** functie uitvoeren. In ons volgende programma wordt dit toegepast.

Compileer en run het programma dat we aan het bespreken waren en observeer de output. Als u **TIENLIJNEN.TXT** niet veranderde zullen de woorden uit dit bestand één voor één op het scherm getoond worden, met als laatste twee identieke woorden. Dit komt door het probleem dat we zonet besproken. Merk op dat sommig compilers de buffer ledigen na het printen, dus het kan zijn dat u een extra lege lijn als laatste krijgt, in plaats van twee dezelfde woorden.

```
/* 7.4.2.2.c */
#include<stdio.h>
int main(void)
{
FILE *fp1;
char eenwoord[100];
int c;

    fp1 = fopen("TIENLIJNEN.TXT", "r");

    do          /* lijn 1 (zie tekst) */
    {
        c = fscanf(fp1, "%s", eenwoord);
        if (c != EOF)
            printf("%s\n", eenwoord);
    } while (c != EOF);         /* lijn 2 (zie tekst) */

    fclose(fp1);

    return 0;
}
```

Compileer en run dit programma en u zal zien dat het extra woord niet verschijnt, omdat we een extra controle op de **EOF** inbouwden in het midden van de loop. Dit was drie programma's terug al een probleem, maar ik koos ervoor om er daar nog niet op in te gaan omdat de fout niet zo voor de hand lag. Ook in dit programma is er weer geen openingscontrole, maar u weet ondertussen hoe u dat zelf kan doen.

Een ervaren C-programmeur zou niet de code gebruiken die wij hier gebruikten, omdat **c** tweemaal vergeleken wordt met **EOF**, elke keer als de loop uitgevoerd wordt. We gebruikten deze methode omdat ze makkelijk te begrijpen is, maar hoe meer ervaring u opdoet, hoe meer u gebruik zal willen maken van meer efficiënte methodes, zelfs als deze moeilijker te lezen en begrijpen zijn. Een ervaren C-programmeur zal de volledige structuur van lijn 1 t.e.m. 2 vervangen door:

```
while((c = fscanf(fp1, "%s", eenwoord)) != EOF)
{
    printf("%s\n", oneword);
}
```

Er is geen twijfel mogelijk dat deze code moeilijker te lezen is, maar als u het nauwkeurig bekijkt ziet u dat ze identiek is aan de code in ons programma en dat de extra controle niet meer nodig is. Alhoewel meer efficiënt, is het niet duidelijk of de kleine tijdswinst de slechte leesbaarheid waard is. Stel dat uw programma tien milliseconden bespaart bij het lezen van een bestand dat één keer per dag gebruikt

wordt, en het kost de programmeur een uur langer om uw code aan te passen, een jaar nadat ze geschreven werd. Dan zal u weinig voordeel halen uit de korte methode. U zal veel soortgelijke beslissingen moeten maken tijdens het schrijven van een programma.

7.4.3 een volledige lijn

```
/* 7.4.3.1.c */
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
FILE *fp1;
char eenwoord[100];
char *c;

    fp1 = fopen("TIENLIJNEN.TXT", "r");
    if (fp1 == NULL)
    {
        printf("Het bestand kon niet geopend worden\n");
        exit (EXIT_FAILURE);
    }

    do
    {
        c = fgets(eenwoord, 100, fp1); /* neemt een volledige lijn
uit het bestand */
        if (c != NULL)
            printf("%s", eenwoord);
    } while (c != NULL);

    fclose(fp1);

    return EXIT_SUCCESS;
}
```

Dit programma lijkt hard op onze vorige voorbeelden, behalve dat we nu een volledige lijn inlezen van het bestand. We gebruiken **fgets()** wat een volledige lijn opneemt inclusief het newline karakter "\n", in een geheugenbuffer. De buffer is het eerste argument in de functieaanroep. Het maximum aantal karakters dat per lijn gelezen mag worden is het tweede argument. Deze functie zal karakters in de buffer inlezen tot er een newline karakter gevonden wordt, of tot het maximum aantal karakters - 1 bereikt wordt. Het laatste wordt gebruikt voor het null karakter. Wanneer de **EOF** gevonden wordt, wordt er een **NULL** teruggegeven. In ons voorbeeld zal pointer **c** dan de waarde **NULL** krijgen. Onthoud dat **NULL** als **0** gedefinieerd is in uw **stdio.h** bestand.

Wanneer we merken dat de pointer **c** de waarde **NULL** gekregen heeft kunnen we stoppen met het verwerken van data, maar we moeten dit controleren voordat we printen, net zoals in ons vorige programma. Als laatste, natuurlijk, sluiten we het bestand.

7.5 een bestandsnaam als variabele

Bekijk het volgende programma als een voorbeeld voor het lezen van een willekeurig bestand:

```
/* 7.5.1.c */
#include <stdio.h>
```

```
#include <stdlib.h>

int main(void)
{
FILE *fp1;
char eenwoord[100], bestandsnaam[25];
char *c;

    printf("Geef de bestandsnaam -> ");
    scanf("%s", bestandsnaam); /* de gekozen bestandsnaam opnemen
*/
    fp1 = fopen(bestandsnaam, "r");
    if (fp1 == NULL)
    {
        printf("Het bestand kon niet geopend worden\n");
        exit (EXIT_FAILURE);
    }

    do
    {
        c = fgets(eenwoord, 100, fp1);
        if (c != NULL)
            printf("%s", eenwoord);
    } while (c != NULL);

    fclose(fp1);

    return EXIT_SUCCESS;
}
```

Dit programma vraagt de gebruiker om een bestandsnaam en leest deze in door hem in een string op te slaan. Daarna wordt het bestand geopend om te lezen. Het volledige bestand wordt dan ingelezen en getoond op het scherm. U zou de rest moeten begrijpen, dus worden er geen extra opmerkingen gemaakt.

Compileer en run dit programma. Wanneer het een bestandsnaam vraagt geeft u een naam en extensie in van een bestand dat tekst bevat. Dit kan zelfs een C-bronbestand zijn. Bij het ingeven van de bestandsnaam gelden dezelfde regels als bij de Dos-prompt. D.w.z. dat u een pad kan opgeven zoals "C:\windows\desktop\mijn_tekst_bestand.txt". In onze vorige programma's gebruikten we altijd slechts de bestandsnaam, waardoor deze in dezelfde directory als het programma moest staan. Ook hier kunt u een volledig pad opgeven. Let op: in een string wordt de backslash **** weergegeven door de escape sequentie toe te passen:

```
fp1 = fopen("c:\\windows\\desktop\\tienlijnen.txt", "r");
```

Indien u Windows gebruikt zal het volgende waarschijnlijk ook toegelaten worden:

```
fp1 = fopen("c:/windows/desktop/tienlijnen.txt", "r");
```

7.6 afdrukken naar de printer

```
/* 7.6.1.c */
#include <stdio.h>
#include <stdlib.h>
```

```
int main(void)
{
FILE *grappig, *printer;
int c;

    grappig = fopen("TIENLIJNEN.TXT", "r");  /* inputbestand
openen */
    if (grappig == NULL)
    {
        printf("Het bestand kon niet geopend worden\n");
        getchar();
        exit (EXIT_FAILURE);
    }

    printer = fopen("PRN", "w");      /* printerbestand openen */
    if (printer == NULL)
    {
        printf("De printer is niet beschikbaar \n");
        getchar();
        exit (EXIT_FAILURE);
    }

    do
    {
        c = getc(grappig);      /* één karakter van het bestand
opnemen */
        if (c != EOF)
        {
            putchar(c);  /* op het scherm tonen */
            putc(c, printer);   /* het karakter afdrukken op de
printer */
        }
    } while (c != EOF);         /* herhalen tot EOF */

    fclose(grappig);
    fclose(printer);

    return 0;
}
```

Dit programma is een voorbeeld van hoe we naar de printer kunnen afdrukken. Het zou geen verassingen moeten opleveren dus gaan we er snel overheen.

Opnieuw openen we **TIENLIJNEN.TXT** om te lezen en we openen **PRN** om te schrijven. Printen is identiek aan het schrijven van data naar een schijf behalve dat we een standaard naam gebruiken als bestandsnaam. Veel C compilers gebruiken het gereserveerde woord **PRN** dat de compiler instructeert om de uitvoer naar de printer te sturen. Er zijn andere namen als **LPT**, **LPT1** of **LPT2**. Raadpleeg de documentatie bij uw compiler. Heeft u een nogal nieuwe compiler kan de vooraf gedefinieerde pointer **stdprn** ook mogelijk zijn. Indien u het programma runt en u krijgt de foutboodschap voor de printer, probeer dan **LPT1**. Lukt dit nog niet dan kan het nuttig zijn om de softwarematige instellingen van uw printer na te kijken voor de juiste printerpoort. Vergeet ook niet om uw printer aan te zetten.

Het programma is simpelweg een loop waarin een karakter ingelezen wordt en als het niet **EOF** is, wordt het getoond en afgedrukt. Merk op dat een goede programmeur beide bestandspointers controleert om te verzekeren dat de bestanden geopend kunnen worden. U kan nu **TIENLIJNEN.TXT** verwijderen, we zullen het verder niet meer gebruiken.

7.7 exit

Om een programma te beëindigen gebruikten we in dit hoofdstuk **return** met één van afsluitmacro's uit **stdlib.h**. Er is nog een andere functie die ongeveer hetzelfde verwezenlijkt. Indien we exit gebruiken worden eerst alle geopende bestanden en dergelijke gesloten en daarna wordt het programma afgesloten. Deze functie heeft een integer waarde als parameter. Meestal wordt 0 gebruikt om een goede programma-beëindiging aan te geven en een niet-nul waarde indien er een fout voorkwam (ofwel **EXIT_SUCCESS** of **EXIT_FAILURE**). Zo kan u bijvoorbeeld als laatste regel van uw programma het volgende plaatsen:

```
exit(0);
```

Deze functie bevindt zich eveneens in de **stdlib.h** header.

---

opdrachten

1. Leesopdracht: besteed wat tijd aan het bestuderen van de documentatie bij uw compiler en lees over de volgende functies. U zal er niet alles over begrijpen, maar u zal een goed idee krijgen van hoe bibliotheekfuncties gedocumenteerd zijn. Bekijk ook de declaratie voor het woord **FILE**.

```
fopen(), fclose(), putc(), putchar(), printf(), fprintf(),
scanf(), fscanf(), fgets()
```

2. Schrijf een programma dat een bestandsnaam vraagt voor een invoerbestand en een bestand om naar te schrijven en ze beide opent, tezamen met een bestand naar de printer. Gebruik een loop die telkens één karakter leest en dit uitvoert naar het bestand, de printer en het scherm. Stop bij **EOF**.

3. Schrijf een programma dat een bestandsnaam vraagt. Lees het bestand lijn per lijn in en toon het op het scherm, met lijnnummers.

4. Tip: plaats de regel "FILES=40" in uw config.sys bestand, dat onder de C: root staat. Dan kan u zeker genoeg bestanden tegelijk openen en bewerken met een programma. Indien u deze regel er niet staat kan u normaal gezien zo'n acht bestanden tegelijk openen.

C-taal voor beginners - hoofdstuk 8

de 15 ANSI-C headers

8.1 inleiding

Dit hoofdstuk is een vertaling van "The C Library Reference Guide", door Eric Huss. De auteur gaf mij de wettelijke toestemming tot het vertalen en publiceren van dit materiaal.

Deze cursus behandelt de C-taal globaal gezien. C biedt echter ontelbare mogelijkheden die het programmeren vergemakkelijken en een programma krachtiger maken. Een goede compiler heeft ontelbare functies en ik zal hier degene bespreken die voldoen aan de ANSI-C standaard. Alles wat in dit hoofdstuk voorkomt zou u dus moeten kunnen toepassen als uw compiler de standaard C-taal ondersteund. Een korte terminologie is nodig omdat vele functies en macro's met woorden werken die niet echt kunnen vertaald worden naar het Nederlands:

abort	afbreken, onderbreken, stoppen
error	fout
nonzero	true, niet 0, 1 (meestal)
carriage return	alsof de enter toets ingedrukt wordt
new line	\n
stream	stroom, richting
default	standaard, vanzelfsprekend
offset	tegengewicht, compensatie
allocate, alloceren	toewijzen (aan)

Zoekt u een specifieke functie of macro, gebruik dan de zoekfunctie van uw browser. Als u bij een bepaalde header geen macro's of functies ziet staan, dan zijn die er niet.

Belangrijke tip: indien een voorbeeldprogramma in dit hoofdstuk de functie **exit()** gebruikt en uw compiler geeft u hier een waarschuwing over, dan zal u waarschijnlijk **stdlib.h** moeten invoegen. Als er gesproken wordt over "teruggeven", dan gaat het meestal om de **return**.

Er zijn vijftien ANSI-C headers:

8.2 assert.h

De **assert** header wordt gebruikt voor debugging doeleinden.

Macro's:

```
assert();
```

Externe referenties:

```
NDEBUG
```

8.2.1 macro's

8.2.1.1 assert

Declaratie:

```
void assert(int expressie);
```

De **assert()** macro laat toe om diagnostische informatie naar het standaard foutbestand te schrijven.

Als de expressie op 0 (false) geëvalueerd wordt, worden de expressie, de bestandsnaam van het bronbestand en het lijnnummer naar de standaard error geschreven, waarna de abort functie aangeroepen wordt. Indien de identifier **NDEBUG** gedefinieerd is doet de assert macro niets. De normale wijze van foutuitvoer is in deze vorm:

```
Assertion failed: expressie, file bestandsnaam, line lijnnummer
```

Voorbeeld:

```
#include<assert.h>
void open_record(char *record_naam)
{
  assert(record_naam != NULL);
  /* rest van de code ... */
}
int main(void)
{
  open_record(NULL);
}
```

8.3 ctype.h

De **ctype** header wordt gebruikt voor het testen en converteren van karakters.

Functies:

```
isalnum();
isalpha();
iscntrl();
isdigit();
isgraph();
islower();
isprint();
ispunct();
isspace();
isupper();
isxdigit();
tolower();
toupper();
```

8.3.1 functies

8.3.1.1 is... functies

Deze functies testen het gegeven karakter en geven een nonzero (true) terug als er aan de voorwaarde voldaan wordt. Andes wordt er een 0 (false) teruggegeven.

Declaraties:

```
int isalnum(int karakter);
```

een letter (A tot Z of a tot z) of een getal (0 tot 9)

```
int isalpha(int karakter);
```

een letter (A tot Z of a tot z)

```
int iscntrl(int karakter);
```

eender welk controle karakter (0x00 tot 0x1F of 0x7F)

```
int isdigit(int karakter);
```

een getal (0 tot 9)

```
int isgraph(int karakter);
```

eender welk afdrukkarakter behalve het spatiekarakter (0x21 tot 0x7E)

```
int islower(int karakter);
```

een kleine letter (a tot z)

```
int isprint(int karakter);
```

eender welk afdrukkarakter (0x20 tot 0x7E)

```
int ispunct(int karakter);
```

eender welk interpunctieteken (eendere welk afdrukkarakter behalve spatie of isalnum)

```
int isspace(int karakter);
```

een witruimtekarakter (spatie, tab, carriage return, new line, vertikale tab of formfeed)

```
int isupper(int karakter);
```

een hoofdletter (A tot Z)

```
int isxdigit(int karakter);
```

een hexadecimaal getal (0 tot 9, A tot F of a tot f)

8.3.1.2 to... functies

Deze functies leveren een manier om een enkel karakter om te vormen. Voldoet het karakter aan de voorwaarde, wordt het verandert. Anders blijft het ongewijzigd.

Declaraties:

```
int tolower(int karakter);
```

Als het karakter een hoofdletter is (A tot Z) wordt het naar een kleine letter geconverteerd (a tot z)

```
int toupper(int karakter);
```

Als het karakter een kleine letter is (tot z) wordt het naar een hoofdletter geconverteerd (A tot Z)

Voorbeeld:

```
#include<ctype.h>
#include<stdio.h>
#include<string.h>
int main(void)
{
  int loop;
  char string[] = "DIT IS EEN TEST";
  for(loop = 0; loop < strlen(string); loop++)
          string[loop] = tolower(string[loop]);
  printf("%s\n", string);
  return 0;
}
```

8.4 errno.h

De **errno** header wordt gebruikt voor debugging doeleinden.

Macro's:

```
EDOM
ERANGE
```

Variabelen:

```
errno
```

8.4.1 macro's

8.4.1.1 EDOM

Declaratie:

```
#define EDOM een_waarde
```

EDOM is een identiefier macro die gedeclareerd wordt via **#define**. Zijn waarde vertegenwoordigt een domeinfout die teruggelevert wordt door bepaalde wiskundige functies wanneer een domeinfout voorkomt.

8.4.1.2 ERANGE

Declaratie:

```
#define ERANGE een_waarde
```

ERANGE is een identiefier macro die gedeclareerd wordt via **#define**. Zijn waarde vertegenwoordigt een fout in een gebied die teruggelevert wordt door bepaalde wiskundige functies wanneer een fout in een gebied voorkomt.

8.4.2 variabelen

8.4.2.1 errno

Declaratie:

```
int errno;
```

De **errno** variabele heeft een waarde van nul bij het begin van het programma. Indien er een error voorkomt, dan krijgt de variabele een waarde afhankelijk van het foutnummer.

8.5 float.h

De **float** header definieert de minimum en maximum grenzen voor waarden van floating point getallen.

8.5.1 gedefinieerde waarden

Een floating point getal wordt op deze manier gedefinieerd:

teken waarde E *exponent*

Waarbij *teken* plus of min, *waarde* de waarde van het getal en *exponent* de waarde van het exponent is.

Volgende waarden worden gedefinieerd met het **#define** statement. Deze waarden zijn gebruiksspecifiek, maar mogen niet lager zijn dan wat hier getoond wordt. Merk op dat in alle gevallen **FLT** op het type **float**, **DBL** op het type **double** en **LDBL** op het type **long double** slaat.

FLT_ROUNDS	Definieert de manier waarop floating point getallen worden afgerond. **-1** onbepaalbaar **0** naar nul **1** naar dichtsbijzijnde **2** naar positieve oneindigheid **3** naar negatieve oneindigheid
FLT_RADIX 2	Definieert de basis (radix) weergave van het exponent (vb basis-2 is binair, basis-10 is decimal en basis-16 is hexadecimaal.
FLT_MANT_DIG **DBL_MANT_DIG** **LDBL_MANT_DIG**	Definieert het aantal getallen (digits) in het cijfer (in de **FLT_RADIX** basis)
FLT_DIG 6 **DBL_DIG 10** **LDBL_DIG 10**	Het maximum aantam decimale cijfers (basis-10) dat kan weergegeven worden zonder verandering na afronden.
FLT_MIN_EXP **DBL_MIN_EXP** **LDBL_MIN_EXP**	De minimum negatieve integer waarde voor een exponent in basis **FLT_RADIX.**
FLT_MIN_10_EXP -37 **DBL_MIN_10_EXP -37** **LDBL_MIN_10_EXP -37**	De minimum negatieve integer waarde voor een exponent in basis-10.
FLT_MAX_EXP **DBL_MAX_EXP** **LDBL_MAX_EXP**	De maximum integer waarde voor een exponent in basis **FLT_RADIX**.
FLT_MAX_10_EXP +37 **DBL_MAX_10_EXP +37** **LDBL_MAX_10_EXP +37**	De maximum integer waarde voor een exponent in basis-10.
FLT_MAX 1E+37 **DBL_MAX 1E+37** **LDBL_MAX 1E+37**	Maximum eindige floating point waarde.
FLT_EPSILON 1E-5 **DBL_EPSILON 1E-9** **LDBL_EPSILON 1E-9**	Minst beduidend getal dat kan weergegeven worden.
FLT_MIN 1E-37 **DBL_MIN 1E-37** **LDBL_MIN 1E-37**	Minimum floating point waarden.

8.6 limits.h

De **limits** header definieert de krakteristieken van variabeltypes.

8.6.1 gedefinieerde waarden

Volgende waarden worden gedefinieerd met het **#define** statement. Ze zijn gebruikersspecifiek, maar mogen niet lager zijn dan wat hier getoond wordt.

CHAR_BIT 8	Aantal bits in een **byte**.
SCHAR_MIN -127	Minimum waarde voor een **signed char**.
SCHAR_MAX +127	Maximum waarde voor een **signed char**.
UCHAR_MAX 255	Maximum waarde voor een **unsigned char**.
CHAR_MIN	Minimum waarde voor een **char**.
CHAR_MAX 1	Maximum waarde voor een **char**.
SHRT_MIN -32767	Minimum waarde voor een **short int**.
SHRT_MAX +32767	Maximum waarde voor een **short int**
USHRT_MAX 65535	Maximum waarde voor een **unsigned short int**.
INT_MIN 32767	Minimum waarde voor een **int**.
INT_MAX +32767	Maximum waarde voor een **int**.
UINT_MAX 65535	Maximum waarde voor een **unsigned int**.
LONG_MIN 2147483647	Minimum waarde voor een **long int**.
LONG_MAX +2147483647	Maximum waarde voor een **long int**.
ULONG_MAX 4294967295	Maximum waarde voor een **unsigned long int**.

8.7 locale.h

De **locale** header is handig voor het bepalen van locatiespecifieke informatie.

Variabelen:

```
struct lconv
```

Macro's:

```
NULL
LC_ALL
LC_COLLATE
LC_CTYPE
LC_MONETARY
LC_NUMERIC
LC_TIME
```

Functies:

```
localeconv();
setlocale();
```

8.7.1 functies

8.7.1.1 setlocale

Declaratie:

```
char *setlocale(int categorie, const char *locale);
```

Bepaalt of leest locatieafhankelijke informatie.

category kan één van volgende macro's zijn:

LC_ALL	Bepaal alles.
LC_COLLATE	Beïnvloedt de **strcoll** en **strxfrm** functies.
LC_CTYPE	Beïnvloedt alle karakter functies.
LC_MONETARY	Beïnvloedt de monetaire informatie geleverd door de **localeconv** functie.
LC_NUMERIC	Beïnvloedt de komma formattering en de informatie geleverd door de **localeconv** functie.
LC_TIME	Beïnvloedt de **strftime** functie.

Een waarde van "C" voor **locale** zet de locale op de normale C vertalingsomgevingsinstellingen (default). Een null waarde ("") zet de natuurlijke omgevingsinstellingen. Een null pointer (**NULL**) zorgt ervoor dat **setlocale** een pointer teruggeeft aan de string geassocieërd met deze categorie voor de huidige instellingen (geen veranderingen). Alle andere waarden zijn gebruiksspecifiek. Na een succesvolle uitvoering geeft **setlocale** een pointer terug aan een string die de vorige locatie instellingen vertegenwoordigt. Bij mislukking wordt **NULL** teruggegeven.

Voorbeeld:

```
#include<locale.h>
#include<stdio.h>
int main(void)
{
  char *oude_locale;
  oude_locale = setlocale(LC_ALL, "C");
  printf("De vorige instellingen waren %s.\n", oude_locale);
}
```

8.7.1.2 localeconv

Declaratie:

```
struct lconv *localeconv(void);
```

Bepaalt de structure **lconv** om de huidige locatie instellingen te vertegenwoordigen.

De string pointers in de structure kunnen naar een null string ("") wijzen wat aangeeft dat de waarde niet beschikbaar is. De **char** types zijn niet-negatieve cijfers. Als de waarde gelijk is aan **CHAR_MAX**, dan is de waarde niet beschikbaar.

lconv variabelen:

char *decimal_point	Komma karakter gebruikt voor niet-monetaire waarden.
char *thousands_sep	Karakter om duizendtallen te onderscheiden voor niet-monetaire waarden.
char *grouping	Een string die de grootte van elke groep getallen weergeeft in niet-monetaire hoeveelheden. Elk karakter vertegenwoordigt een integer waarde die het aantal cijfers in de huidige groep aangeeft. Een O waarde betekent dat de vorige waarde gebruikt moet worden voor de rest

	van de groepen.
char *int_curr_symbol	Een string van de gebruikte internationale valuta symbolen. De eerste drie karakters zijn diegene, gespecifieerd volgens ISO 4217:1987 en het vierde is het karakter dat het valuta symbool onderscheidt van de monetaire hoeveelheid.
char *currency_symbol	Het locale symbool dat gebruikt wordt voor de valuta.
char *mon_decimal_point	Komma karakter gebruikt voor monetaire waarden.
char *mon_thousands_sep	Karakter om duizendtallen te onderscheiden voor monetaire waarden.
char *mon_grouping	Een string waarvan de elementen de grootte van de groepering van getallen in monetaire waarden bepalen. Elk karakter vertegenwoordigt een integer waarde die het aantal cijfers in de huidige groep aangeeft. Een O waarde betekent dat de vorige waarde gebruikt moet worden voor de rest van de groepen.
char *positive_sign	Karakter gebruikt voor positieve monetaire waarden.
char *negative_sign	Karakter gebruikt voor negatieve monetaire waarden.
char int_frac_digits	Aantal getallen na de komma in internationale monetaire waarden.
char frac_digits	Aantal getallen na de komma in monetaire waarden.
char p_cs_precedes	Indien gelijk aan 1, dan verschijnt **currency_symbol** voor een positieve monetaire waarde. Indien gelijk aan 0, fan verschijnt **currency_symbol** na de positieve monetaire waarde.
char p_sep_by_space	Indien gelijk aan 1, dan wordt **currency_symbol** gescheiden van een postieve monetaire waarde door een spatie. Indien gelijk aan 0, dan is er geen spatie tussen **currency_symbol** en een postitieve monetaire waarde.
char n_cs_precedes	Indien gelijk aan 1, de gaat **currency_symbol** een negatieve monetaire waarde vooraf. Indien gelijk aan 0, dan komt **currency_sybol** na een negatieve monetaire waarde.
char n_sep_by_space	Indien gelijk aan 1, dan verschijnt **currency_symbol** voor een negatieve monetaire waarde. Indien gelijk aan 0, fan verschijnt **currency_symbol** na de negatieve monetaire waarde.
char p_sign_posn	Vertegenwoordigt de positie van **positive_sign** in een positieve monetaire waarde.
char n_sign_posn	Vertegenwoordigt de positie van **positive_sign** in een negatieve monetaire waarde.

Volgende waarde worden gebruikt voor **p_sign_posn** en **n_sign_posn**:

0	Ronde haken omringen de waarde en **currency_symbol**.
1	Het teken gaat de waarde en **currency_symbol** vooraf.
2	Het teken komt na de waarde en **currency_symbol**.
3	Het teken gaat de waarde en **currency_symbol** onmiddelijk vooraf.
4	Het teken komt onmiddelijk na de waarde en **currency_symbol**.

Voorbeeld:

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

int main(void)
{
  struct lconv locale_structure;
  struct lconv *locale_ptr = &locale_structure;

  locale_ptr = localeconv();
  printf("Komma: %s\n", locale_ptr->decimal_point);

  return 0;
}
```

8.8 math.h

De **math** header definieert verscheidene wiskundige functies.

Macro's:

```
HUGE_VAL
```

Functies:

```
acos();
asin();
atan();
atan2();
ceil();
cos();
cosh();
exp();
fabs();
floor();
fmod();
frexp();
ldexp();
log();
log10();
modf();
pow();
sin();
sinh();
sqrt();
tan();
tanh();
```

8.8.1 macro's

8.8.1.1 HUGE_VAL

Alle **math.h** functies behandelen fouten (errors) gelijkaardig.

In het geval dat het argument dat aan de functie doorgegeven wordt het bereik van die functie overtreft, dan wordt de variabele **errno** op **EDOM** gezet. De waarde die de functie teruggeeft is gebruikersspecifiek.

In het geval dat de teruggegeven waarde te groot is om weergegeven te worden in een **double**, dan geeft de functie de macro **HUGE_VAL** terug en de variabele **errno** wordt op **ERANGE** gezet om een overflow te vertegenwoordigen. Is de waarde te klein om in een **double** weergegeven te worden, dan stuurt de functie nul terug.

errno, EDOM and **ERANGE** zijn gedefinieerd in de **errno.h** header.

8.8.2 functies

8.8.2.1 trigonometrie

8.8.2.1.1 acos

Declaratie:

```
double acos(double x);
```

Return:

De cirkelcosinus van *x* in straal.

Bereik:

De waarde *x* moet binnen de grens van -1 tot +1 liggen (inclusief). De teruggegeven waarde ligt in het bereik van 0 tot pi (inclusief).

8.8.2.1.2 asin

Declaratie:

```
double asin(double x);
```

Return:

De cirkelsinus van *x* in straal.

Bereik:

De waarde van *x* moet binnen de grens van -1 tot +1 liggen (inclusief). De teruggegeven waarde ligt in het bereik van -p/2 tot +p/2 (inclusief).

8.8.2.1.3 atan

Declaratie:

```
double atan(double x);
```

Return:

De cirkeltangens van *x* in straal.

Bereik:

De waarde van *x* heeft geen bereik. De teruggegeven waarde ligt in het bereik van -p/2 tot +p/2 (inclusief).

8.8.2.1.4 atan2

Declaratie:

```
double atan2(double y, double x);
```

Return:

De cirkeltangens van *y/x* in straal, gebaseerd op de tekens van beide waarden om het juiste kwadrant te bepalen.

Bereik:

De waarde van *x* heeft geen bereik. De teruggegeven waarde ligt in het bereik van -p/2 tot +p/2 (inclusief).

8.8.2.1.5 cos

Declaratie:

```
double cos(double x);
```

Return:

De cosinus van een straalhoek *x*.

Bereik:

De waarde van *x* heeft geen bereik. De teruggegeven waarde ligt in het bereik van -1 tot +1 (inclusief).

8.8.2.1.6 cosh

Declaratie:

```
double cosh(double x);
```

Return:

De hyperbolische cosinus van *x*.

Bereik:

Er is geen grens wat betreft het bereik van het argument of de teruggegeven waarde.

8.8.2.1.7 sin

Declaratie:

```
double sin(double x);
```

Return:

De sinus van een straalhoek *x*.

Bereik:

De waarde van *x* heeft geen bereik. De teruggegeven waarde ligt in het bereik van -1 tot +1 (inclusief).

8.8.2.1.8 sinh

Declaratie:

```
double sinh(double x);
```

Return:

De hyperboliche sinus van *x*.

Bereik:

Er is geen grens wat betreft het bereik van het argument of de teruggegeven waarde.

8.8.2.1.9 tan

Declaratie:

```
double tan(double x);
```

Return:

De tangens van een straalhoek *x*.

Bereik:

Er is geen grens wat betreft het bereik van het argument of de teruggegeven waarde.

8.8.2.1.10 tanh

Declaratie:

```
double tanh(double x);
```

Return:

De hyperbolische tangens van *x*.

Bereik:

De waarde van *x* heeft geen bereik. De teruggegeven waarde ligt in het bereik van -1 tot +1 (inclusief).

8.8.2.2 expentioneel, logarithmisch, machtsfuncties

8.8.2.2.1 exp

Declaratie:

```
double exp(double x);
```

Return:

De waarde van e tot de macht *x*.

Bereik:

Er is geen grens wat betreft het bereik van het argument of de teruggegeven waarde.

8.8.2.2.2 frexp

Declaratie:

```
double frexp(double x, int *exponent);
```

Return:

Het floating point cijfer *x* wordt opgebroken in een mantissa en een exponent. De teruggegeven waarde is de mantissa en de integer waar *exponent* naar wijst is het exponent. De resulterende waarde is **x = mantissa * 2 ^exponent**.

Bereik:

De mantissa ligt binnen het bereik van .5 (inclusief) en 1 (exclusief).

8.8.2.2.3 ldexp

Declaratie:

```
double ldexp(double x, int exponent);
```

Return:

x vermenigvuldigd met 2 verheven tot de macht van *exponent*.

Bereik:

Er is geen grens wat betreft het bereik van het argument of de teruggegeven waarde.

8.8.2.2.4 log

Declaratie:

```
double log(double x);
```

Return:

Het natuurlijke logaritme (basis-e logaritme) van *x*.

Bereik:

Er is geen grens wat betreft het bereik van het argument of de teruggegeven waarde.

8.8.2.2.5 log10

Declaratie:

```
double exp(double x);
```

Return:

Het gewone logaritme (basis-10 logaritme) van *x*.

Bereik:

Er is geen grens wat betreft het bereik van het argument of de teruggegeven waarde.

8.8.2.2.6 modf

Declaratie:

```
double modf(double x, double *interger);
```

Return:

Floating-point cijfer *x* wordt opgebroken in een integer en fractie componenten. De teruggegeven waarde is het fractie component (het deel achter de decimaal) en zet *integer* naar het integer component.

Bereik:

Er is geen grens wat betreft het bereik van het argument of de teruggegeven waarde.

8.8.2.2.7 pow

Declaratie:

```
double pow(double x, double y);
```

Return:

x tot de macht van *y*.

Bereik:

x mag niet negatief zijn als *y* een fractionele waarde is en *x* mag niet nul zijn als *y* kleiner dan of gelijk aan nul is.

8.8.2.2.8 sqrt

Declaratie:

```
double sqrt(double x);
```

Return:

De vierkantswortel van *x*.

Bereik:

Het argument mag niet negatief zijn. De teruggegeven waarde is altijd positief.

8.8.2.3 andere wiskundige functies

8.8.2.3.1 ceil

Declaratie:

```
double ceil(double x);
```

Return:

De kleinste integer waarde groter dan of gelijk aan *x*.

Bereik:

Er is geen grens wat betreft het bereik van het argument of de teruggegeven waarde.

8.8.2.3.2 fabs

Declaratie:

```
double fabs(double x);
```

Return:

De absolute waarde van *x* (een negatieve waarde wordt positief, een positieve waarde blijft positief).

Bereik:

Er is geen grens wat betreft het bereik van het argument. De teruggegeven waarde is altijd positief.

8.8.2.3.3 floor

Declaratie:

```
double floor(double x);
```

Return:

De grootste integer waarde kleiner dan of gelijk aan *x*.

Bereik:

Er is geen grens wat betreft het bereik van het argument of de teruggegeven waarde.

8.8.2.3.4 fmod

Declaratie:

```
double fmod(double x, double y);
```

Return:

De rest van *x* gedeeld door *y*.

Bereik:

Er is geen grens wat betreft het bereik van de teruggegeven waarde. Als *y* nul is, dan zal ofwel een bereiksfout voorkomen ofwel zal de functie nul teruggeven (gebruikers-gedefinieerd).

8.9 setjmp.h

De **setjmp** header wordt gebruikt voor het controleren van low-level aanroepen en returns naar en van functies.

Macro's:

```
setjmp();
```

Functies:

```
longjmp();
```

Variabelen:

```
typedef jmp_buf
```

8.9.1 variabelen en definities

Het variabel type **jmp_buf** is een array type gebruikt voor het vasthouden van informatie voor **setjmp** en **longjmp**.

8.9.2 macro's

8.9.2.1 setjmp

Declaratie:

```
int setjmp(jmp_buf omgeving);
```

Bewaart de omgeving naar de variabele *omgeving*. Indien er een niet-nul waarde wordt teruggegeven, dan indiceert dit dat het punt in de broncode werd bereikt met een **longjmp**. In het andere geval wordt nul teruggegeven, aangevend dat de omgeving werd bewaard.

8.9.3 functies

8.9.3.2 longjmp

Declaratie:

```
void longjmp(jmp_buf omgeving, int waarde);
```

De omgeving wordt hersteld van een **setjmp** aanroep waar de omgevingsvariabele werd opgeslagen. De uitvoering van het programma gaat naar de **setjmp** locatie alsof **setjmp** de waarde van de variabele *waarde* had teruggegeven. De variabele *waarde* kan niet nul zijn. Alhoewel, als nul doorgegeven wordt, dan wordt 1 vervangen. Als de functie waar **setjmp** werd aangeroepen beëindigd is, dan zijn de resultaten ongedefinieerd.

Voorbeeld:

```
#include<setjmp.h>
#include<stdio.h>
void een_functie(jmp_buf);
int main(void)
{
  int waarde;
  jmp_buf omgeving_buffer;
  waarde=setjmp(omgeving_buffer);
  if(waarde!=0)
  {
          printf("Bereikte dit punt met een longjmp met waarde=-
d.\n",waarde);
          exit(0);
  }
  printf("Aanroepende functie.\n");
  een_functie(omgeving_buffer);
  return 0;
}
void een_functie(jmp_buf omg_buf)
{
  longjmp(omg_buf,5);
}
```

De uitvoer van dit programma zou moeten zijn:

```
Aanroepende functie.
Bereikte dit punt met een longjmp met waarde=5.
```

8.10 signal.h

De **signal** header voorziet mogelijkheden om signalen te verwerken, die tijdens de uitvoering van een programma gemeld worden.

Macro's:

```
SIG_DFL
SIG_ERR
SIG_IGN
SIGABRT
SIGFPE
SIGILL
SIGINT
SIGSEGV
SIGTERM
```

Functies:

```
signal();
raise();
```

Variabelen:

```
typedef sig_atomic_t
```

8.10.1 variabelen en definities

Het **sig_atomic_t** type is van het type **int** en wordt gebruikt als variabele in een signaalverwerker. De **SIG_** macro's worden gebruikt met de signaalfunctie om signaalfuncties te definiëren.

SIG_DFL	Default verwerker.
SIG_ERR	Vertegenwoordigt een signaalfout.
SIG_IGN	Signaalfout.

8.10.2 macro's

De **SIG** macro's worden gebruikt om een signaalnummer te vertegenwoordigen in de volgende situaties:

SIGABRT	Abnormale beëindiging (voortgebracht door de abort functie).
SIGFPE	Floating-point fout (fout veroorzaakt door deling door nul, ongeldige operatie, enz...).
SIGILL	Illegale operatie (instructie).
SIGINT	Interactief attentie signaal (zoals ctrl+C).
SIGSEGV	Ongeldige toegang tot opslag (schending van segment en geheugen).
SIGTERM	Verzoek tot beëindigen.

8.10.3 functies

8.10.3.1 signal

Declaratie:

```
void (*signal (int sig, void (*func) (int))) (int);
```

Controleert hoe een signaal verwerkt wordt. *sig* vertegenwoordigt het signaalnummer compatibel met de **SIG** macro's. *func* is de functie die geroepen moet worden wanneer het signaal voorkomt. Als *func*

SIG_DFL is, dan wordt de default verwerker aangeroepen. Als *func* **SIG_IGN** is, dan wordt het signaal genegeerd. Als *func* naar een functie wijst dan, wanneer een signaal gevonden wordt, wordt de default functie aangeroepen (**SIG_DFL**), daarna wordt de functie aangeroepen. De functie moet één argument aannemen van het type **int** welk het signaalnummer vertegenwoordigt. De functie mag afsluiten met **return**, **abort**, **exit**, of **longjmp**. Wanneer de functie stopt wordt de uitvoering hervat waar deze werd onderbroken (behalve als het een **SIGFPE** signaal was, in dat geval is het resultaat ongedefinieerd).

Als de aanroep tot **signal** succesvol is, dan wordt er een pointer naar de vorige signaalverwerker voor het bepaalde signaaltype teruggegeven.

8.10.3.2 raise

Declaratie:

```
int raise(int sig);
```

Zorgt ervoor dat er een signaal *sig* wordt gegenereerd. Het *sig* argument is compatibel met de **SIG** macro's. Als de aanroep succesvol is, wordt nul teruggegeven. Anders wordt er een niet-nul waarde teruggegeven.

Voorbeeld:

```
#include<signal.h>
#include<stdio.h>
void vang_functie(int);
int main(void)
{
  if(signal(SIGINT, vang_functie)==SIG_ERR) {
          printf("Er was een fout bij het bepalen van de
signaalverwerker.\n");
          exit(0);
  }
  printf("Het interactieve attentie signaal wordt verhoogd.\n");
  if(raise(SIGINT)!=0) {
          printf("Fout bij het verhogen van het signaal.\n");
          exit(0);
  }
  printf("Einde.\n");
  return 0;
}
void vang_functie(int signal)
{
  printf("Het interactieve attentie signaal werd aangenomen.\n");
}
```

Indien er geen fouten voorkwamen zou de output als volgt moeten zijn:

```
Het interactieve attentie signaal wordt verhoogd.
Het interactieve attentie signaal werd aangenomen.
Einde.
```

8.11 stdarg.h

De **stdarg** header definieert verschillende macro's die gebruikt worden om de argumenten in een functie te nemen, als het aantal argumenten onbekend is.

Macro's:

```
va_start();
va_arg();
va_end();
```

Variabelen:

```
typedef va_list
```

8.11.1 variabelen en definities

Het **va_list** type is een type dat gebruikt kan worden voor toegang tot de argumenten in een functie met de **stdarg** macro's.

Een functie van variabele argumenten wordt gefinieerd met de ellips **(, ...)** op het einde van de parameterlijst.

8.11.2 macro's

8.11.2.1 va_start

Declaratie:

```
void va_start(va_list ap, laatste_arg);
```

Initialiseert *ap* voor gebruik met de **va_arg** en **va_end** macro's. *laatste_arg* is het laatste gekende vaste argument dat doorgegeven wordt naar de functie (het argument voor de ellips).

Merk op dat **va_start** geroepen moet worden alvorens **va_arg** en **va_end** te gebruiken.

8.11.2.2 va_arg

Declaratie:

```
type va_arg(va_list ap, type);
```

Expandeert naar het volgende argument in de parameterlijst van de functie met het type *type*. Merk op dat *ap* geinitialiseerd moet zijn met **va_start**. Als er geen volgende argument is, dan is het resultaat onbepaald.

8.11.2.3 va_end

Declaratie:

```
void va_end(va_list ap);
```

Zorgt ervoor dat een functie met variabele argumenten die gebruik maakte van de **va_start** macro kan teruggeven. Als **va_end** niet geroepen wordt alvorens terug te keren naar de functie, dan is het resultaat

onbepaald. De variabele argumentenlijst *ap* mag niet langer worden gebruikt na een aanroep tot **va_end** zonder aan aanroep to **va_start**.

Voorbeeld:

```
#include<stdarg.h>
#include<stdio.h>
void som(char *, int, ...);
int main(void)
{
  som("De som van 10+15+13 is %d.\n",3,10,15,13);
  return 0;
}
void som(char *string, int aant_arg, ...)
{
  int som=0;
  va_list ap;
  int loop;
  va_start(ap, aant_arg);
  for(loop=0;loop<aant_arg;loop++)
         som+=va_arg(ap,int);
  printf(string,som);
  va_end(ap);
}
```

8.12 stddef.h

De **stddef** header definieert verscheidene standaard definities. Veel van deze definities komen ook voor in andere headers.

Macro's:

```
NULL
offsetof();
```

Variabelen:

```
typedef ptrdiff_t
typedef size_t
typedef wchar_t
```

8.12. variabelen en definties

ptrdiff_t is het verschil van de aftrekking van twee pointers. **size_t** is het unsigned integer resultaat van het **sizeof** sleutelwoord. **wchar_t** is een integer type van de grootte van een brede karakter constante. **NULL** is de waarde van een null pointer constante.

Declaratie:

```
offsetof(type, lid-aanduider)
```

Dit resulteert in een constante integer van het type **size_t** wat de offset (tegengewicht) in bytes is van een structure lid van het begin van de structure. Het lid wordt gegeven door *lid-aanduider* en de naam van de structure wordt gegeven in *type*.

Voorbeeld:

```
#include<stddef.h>
#include<stdio.h>
int main(void)
{
  struct gebruiker {
         char naam[50];
         char alias[50];
         int level;
  };
  printf("level is byte %d in de structure.\n",
                          offsetof(struct gebruiker, level));
}
```

De output zou als volg moeten zijn:

```
level is byte 100 in de structure.
```

8.13 stdio.h

De assert header wordt gebruikt voor debugging doeleinden.

8.2.1 macro's

De **stdio** header voorziet functies voor het uitvoeren en verwerken van input en output.

Macro's:

```
NULL
_IOFBF
_IOLBF
_IONBF
BUFSIZ
EOF
FOPEN_MAX
FILENAME_MAX
L_tmpnam
SEEK_CUR
SEEK_END
SEEK_SET
TMP_MAX
stderr
stdin
stdout
```

Functies:

```
clearerr();
fclose();
feof();
ferror();
fflush();
fgetpos();
```

```
fopen();
fread();
freopen();
fseek();
fsetpos();
ftell();
fwrite();
remove();
rename();
rewind();
setbuf();
setvbuf();
tmpfile();
tmpnam();
fprintf();
fscanf();
printf();
scanf();
sprintf();
sscanf();
vfprintf();
vprintf();
vsprintf();
fgetc();
fgets();
fputc();
fputs();
getc();
getchar();
gets();
putc();
putchar();
puts();
ungetc();
perror();
```

Variabelen:

```
typedef size_t
typedef FILE
typedef fpos_t
```

8.13.1 variabelen en definities

size_t	Het unsigned integer resultaat van het **sizeof** sleutelwoord.
FILE	Een type geschikt voor het opslaan van gegevens voor een bestandstream.
fpos_t	Een type geschikt voor het opslaan van een positie in een bestand.
NULL	De waarde van een null pointer constante.
_IOFBF, **_IOLBF** en **_IONBF**	Worden gebruikt in de **setvbuf** functie.

BUFSIZ	Een integer die de waarde van een buffer, gebruikt bij **setbuf**, vertegenwoordigt.
EOF	Een negatieve integer die aangeeft dat het einde-van-bestand werd bereikt.
FOPEN_MAX	Een integer die het maximum aantal bestanden weergeeft, dat het systeem gegaranteerd kan openen.
FILENAME_MAX	Een integer die de langste lengte van een karakter array weergeeft, geschikt voor het bevatten van de langst mogelijke bestandsnaam. Als het gebruik geen limiet aanduid, dan zou deze waarde de aanbevolen maximum waarde moeten zijn.
L_tmpnam	Een integer die de langste lengte van een karakter array weergeeft, geschikt voor het bevatten van de langst mogelijke tijdelijke bestandsnaam, gecreëerd door de **tmpnam** functie.
TMP_MAX	Het maximum aantal unieke bestandsnamen dat de functie **tmpnam** kan aanmaken.
stderr, **stdin** en **stdout**	Pointers naar **FILE** types die overeenstemmen met de standaard fout, standaard input en standaard output streams.

8.13.2 streams en bestanden

Streams vergemakkelijken een manier om een niveau van abstractie te creëeren tussen het programma en een invoer/uitvoer apparaat. Dit maakt een gekende methode mogelijk voor het sturen en ontvangen van data onder de verscheidene beschikbare types van apparaten. Er zijn twee types van streams: tekst en binair.

Tekst streams zijn samengesteld uit lijnen (regels). Elke lijn heeft nul of meer karakters en wordt afgesloten door een nieuwe-lijn karakter ('\n'), welk het laatste karakter van een lijn is. Conversies kunnen voorkomen bij tekst streams tijdens invoer en uitvoer. Tekst streams bestaan enkel uit (af)drukbare karakters, het tab karakter en het nieuwe-lijn karakter. Spaties mogen niet verschijnen voor het nieuwe-lijn karakter, alhoewel het toepassings-gedefinieerd of het lezen van een tekst stream deze lege ruimtes verwijdert. Een toepassing moet lijnen met tot minimum 254 karakters ondersteunen, inclusief het nieuwe-lijn arakter.

Binaire streams voeren data in en uit bij een verhouding van precies 1:1. Er bestaat geen conversie en alle karakters kunnen overgezet worden.

Wanneer een programma begint zijn er reeds drie beschikbare streams: standaard invoer, standaard uitvoer en standaard fout (error).

Bestanden zijn geassocieerd met streams en moeten geopend worden om ze te kunnen gebruiken. De positie van I/O in een bestand wordt bepaald door de bestandspositie. Wanneer een bestand wordt geopend, wijst de bestandspositie naar het begin van het bestand behalve wanneer het geoepend is voor toevoegen (appending). In dat geval wijst de positie naar het einde van het bestand. De bestandspositie volget de lees en schrijf operaties om aan te duiden waar de volgende operatie zal plaatsvinden.

Als een bestand wordt gesloten, kunnen er geen acties meer op uitgevoerd worden totdat het opnieuw geopend wordt. Bij het verlaten van de **main** functie worden alle geopende bestanden gesloten.

8.13.3 bestandsfuncties

8.13.3.1 clearerr

Declaratie:

```
void clearerr(FILE *stream);
```

Ledigt de einde-van-bestand indicator en foutindicator voor de gegeven stream. Zolang de foutindicator is bepaald, geven alle stream operaties een fout terug totdat **clearerr** of **rewind** wordt aangeroepen.

8.13.3.2 fclose

Declaratie:

```
int clearerr(FILE *stream);
```

Sluit de stream. Alle buffers worden geledigt.

Indien succesvol, wordt er nul teruggegeven. Bij een fout wordt er **EOF** teruggegeven..

8.13.3.3 feof

Declaratie:

```
int feof(FILE *stream);
```

Test de einde-van-bestand indicator voor de gegeven stream. Als de stream het einde-van-bestand bereikt heeft, dan wordt er een niet-nul waarde teruggegeven. In het andere geval is de return nul.

8.13.3.4 ferror

Declaratie:

```
int ferror(FILE *stream);
```

Test de foutindicator voor de gegeven stream. Als de foutindicator is bepaald, dan geeft het een niet-nul waarde terug. Is deze niet bepaald, dan is de return nul.

8.13.3.5 fflush

Declaratie:

```
int fflush(FILE *stream);
```

Ledigt de uitvoerbuffer van een stream. Als *stream* een null pointer is, dan worden alle uitvoerbuffers geledigd.

8.13.3.6 fgetpos

Declaratie:

```
int fgetpos(FILE *stream, fpos_t *pos);
```

Neemt de huidige bestandspositie van de stream en schrijft deze naar *pos*.

Indien succesvol, geeft het nul terug. Bij een fout geeft het een niet-nul waarde terug en wordt het foutnummer in de variabele **errno** opgeslagen.

8.13.3.7 fopen

Declaratie:

```
FILE *fopen(const char * bestandsnaam, const char *modus);
```

Opent de bestandsnaam aangeduid door *bestandsnaam*. Het modus argument mag één van de volgende constante strings zijn:

r	lezen in tekst modus
w	schrijven in tekst modus
a	toevoegen in tekst modus
rb	lezen in binaire modus
wb	schrijven in binaire modus
ab	toevoegen in binaire modus
r+	lezen en schrijven in tekst modus
w+	lezen en schrijven in tekst modus
a+	lezen en toevoegen in tekst modus
r+b of rb+	lezen en schrijven in binaire modus
w+b pf wb+	lezen en schrijven in binaire modus
a+b pf ab+	lezen en schrijven in binaire modus

Als het bestand niet bestaat en het word geopend in lees modus, dan mislukt de openactie.

Als het bestand met de **a** modus geopend wordt, dan worden alle schrijfacties uitgevoerd op het einde van het bestand, ongeacht de huideige bestandspositie.

Als het bestand in de update modus (**+**) geopend owrdt, dan kan kan uitvoer niet direkt gevolgd worden door invoer en omgekeerd, zonder tussenkomst van **fseek**, **fsetpos**, **rewind** of **fflush**.

Bij succes wordt een pointer naar de bestandsstream teruggegeven. Bij mislukking wordt er een null pointer teruggegeven.

8.13.3.8 fread

Declaratie:

```
size_t fread(void *ptr, size_t grootte, size_t aantalelementen,
FILE *stream);
```

Leest data van de gegeven stream in in een array aangewezen door *ptr*.Het leest *aantalelementen* elementen van grootte *grootte*. Het totaal aantal gelezen bytes is ***(grootte * aantalelementen)***.

Bij succes wordt het aantal gelezen elementen teruggegeven. Bij een fout of einde-van-bestand is de return het totaal aantal succesvol gelezen elementen (wat nul kan zijn).

8.13.3.9 freopen

Declaratie:

```
FILE *freopen(const char *bestandsnaam, const char *modus, FILE
*stream);
```

Associeert een nieuwe bestandsnaam met de gegeven open stream. Het oude bestand is *stream* wordt gesloten. Als er een fout voorkomt bij het sluiten van het bestand, dan wordt de fout genegeerd. Het modus argument is hetzelfde als beschreven bij de **fopen** opdracht. Meestal gebruikt voor het reassociëren van **stdin**, **stdout** of **stderr**.

8.13.3.10 fseek

Declaratie:

```
int fseek(FILE *stream, long int offset, int waarvandaan);
```

Zet de bestandspositie van de stream tot de gegeven offset. Het argument *offset* duid het aantal bytes aan om te zoeken vanaf de gegeven waar vandaan positie. Het argument *waarvandaan* kan zijn:

SEEK_SET	Zoekt vanaf het begin van het bestand.
SEEK_CUR	Zoekt vanaf de huidige positie.
SEEK_END	Zoekt vanef het einde van het bestand.

Bij een tekst stream zou *waarvandaan* **SEEK_SET** moeten zijn en *offset* zou ofwel nul moeten zijn of een waarde teruggegeven van **ftell**.

De einde-van-bestand indicator wordt gereset, de fout indicator niet.

Bij succes wordt nul teruggegeven. Bij een fout wordt er een niet-nul waarde teruggegeven.

8.13.3.11 fsetpos

Declaratie:

```
int fsetpos(FILE *stream, fpos_t *pos);
```

Neemt de huidige bestandspositie van de stream en schrijft deze naar *pos*.

Zet de bestandspositie van de gegeven stream op de gegeven positie. Het argument *pos* is een positie gegeven door de functie **fgetpos**. De einde-van-bestand indicator wordt geledigd.

Bij succes wordt nul teruggegeven. Bij een fout wordt er een niet-nul waarde teruggegeven en wordt de variabele **errno** ingesteld.

8.13.3.12 ftell

Declaratie:

```
long int ftell(FILE *stream);
```

Geeft de huidige bestandspositie van de huidige stream terug. Als het een binaire stream is, dan is de waarde het aantal bytes vanaf het begin van het bestand. Als het een tekst stream is, dan is de waarde een waarde bruikbaar voor de **fseek** functie om de bestandspositie tot de huidige positie terug te geven.

Bij succes wordt de huide bestandspositie teruggegeven. Bij een fout wordt een waarde van **-1L** teruggegeven en wordt **errno** ingesteld.

8.13.3.13 fwrite

Declaratie:

```
size_t fwrite(const void *ptr, size_t grootte, size_t
aantalelementen, FILE *stream);
```

Schrijft data van het array aangewezen door *ptr* naar de gegeven stream. Het schrijft *aantalelementen* elementen van grootte *grootte*. Het totaal aantal geschreven bytes is **(*grootte * aantalelementen*)**.

Bij succes wordt het aantal geschreven elementen teruggegeven. Bij een fout of einde-van-bestand is de return het totaal aantal succesvol geschreven elementen (wat nul kan zijn).

8.13.3.14 remove

Declaratie:

```
int remove(const char *bestandsnaam);
```

Verwijdert de gegeven bestandsnaam zodat deze niet langer toegankelijk is. Als het bestand geopend is, dan is het resultaat gebruikers-gedefinieerd.

Bij succes wordt nul teruggegeven. Bij mislukking wordt er een niet-nul waarde teruggegeven.

8.13.3.15 remove

Declaratie:

```
int rename(const char *oude_bestandsnaam, const char
*nieuwe_bestandsnaam);
```

Zorgt er voor dat *oude_bestandsnaam* wordt verandert in *nieuwe_bestandsnaam*. Als de nieuwe bestandsnaam reeds bestaat, dan is het resultaaat gebruikers-gedefinieerd.

8.13.3.16 rewind

Declaratie:

```
int rewind(FILE *stream);
```

Zet de bestandspostie naar het begin van het bestand van de gegeven stream. De fout indicator en einde-van-bestand indicator worden gereset.

8.13.3.17 setbuf

Declaratie:

```
void setbuf(FILE *stream, char *buffer);
```

Definieert hoe een stream gebufferd zou moeten worden. Dit zou aangeroepen moeten worden nadat de stream werd geopend, maar voordat er een operatie op werd gedaan op de stream. Invoer en uitvoer wordt volledig gebufferd. Het default **BUFSIZ** is de grootte van de buffer. Het argument *buffer* wijst naar een array om gebruikt te worden als de buffer. Als *buffer* een null pointer is, dan wordt de stream ge-onbufferd.

8.13.3.18 setvbuf

Declaratie:

```
int setvbuf(FILE *stream, char *buffer, int modus, size_t
grootte);
```

Definieert hoe een stream gebufferd zou moeten worden. Dit zou aangeroepen moeten worden nadat de stream werd geopend, maar voordat er een operatie op werd gedaan op de stream. Het argument *modus* definieert hoe de stream als volgt zou moeten gebufferd worden.

_IOFBF	Invoer en uitvoer wordt volledig gebufferd. Als de buffer leeg is probeert een invoeroperatie deze te vullen. Bij uitvoer wordt de buffer volledig gevuld alvorens er informatie naar het bestand wordt geschreven (of de stream gesloten wordt).
_IOLBF	Invoer en uitvoer wordt gelijnbufferd. Als de buffer leeg is probeert een invoeroperatie deze te vullen. Bij uitvoer wordt de buffer geledigd wanneer er een nieuwe-lijn karakter wordt geschreven.
_IONBF	Invoer en uitvoer wordt niet gebufferd. Er wordt geen bufferoperatie vericht.

Het argument *buffer* wijst naar een array dat gebruikt wordt als de buffer. Als *buffer* een null pointer is, dan gebruikt **setvbuf malloc** om zijn eigen buffer te creëeren.

Het argument *grootte* determineert de grootte van het array.

Bij succes wordt nul teruggegeven. Bij mislukking wordt er een niet-nul waarde teruggegeven.

8.13.3.19 tmpfile

Declaratie:

```
FILE *tmpfile(void);
```

Creëert een tijdelijk bestand in binaire update modus (**wb+**). Het tijdelijke bestand wordt verwijdert wanneer het programma eindigt of wanneer de stream gesloten wordt.

8.13.3.20 tmpnam

Declaratie:

```
char *tmpnam(char *str);
```

Genereert en geeft een geldige tijdelijke bestandsnaam terug die niet bestaat. Tot **TMP_MAX** verschillende bestandsnamen kunnen aangemaakt worden.

Als het argument *str* een null pointer is, dan geeft de functie een pointer naar een geldige bestandsnaam terug. Als *ptr* een geldige pointer naar een array is, dan wordt de bestandsnaam geschreven naar het array en een pointer naar datzelfde array wordt teruggegeven. De bestandsnaam mag tot **L_tmpnam** karakters lang zijn.

8.13.4 geformatteerde I/O functies

8.13.4.1 ...printf functies

Declaraties:

```
int fprintf(FILE *stream, const char *formaat, ...);
int printf(const char *formaat, ...);
int sprintf(char *str, const char *formaat, ...);
int vfprintf(FILE *stream, const char *formaat, va_list arg);
int vprintf(const char *formaat, va_list arg);
int vsprintf(char *str, const char *formaat, va_list arg);
```

De **...printf** functies voorzien een manier om geformatteerde informatie naar een stream uit te voeren.

fprintf	Stuurt geformatteerde uitvoer naar een stream.
printf	Stuurt geformatteerde uitvoer naar **stdout**.
sprintf	Stuurt geformatteerde uitvoer naar een string.
vfprintf	Stuurt geformatteerde uitvoer naar een stream, gebruik makend van een argumentenlijst.
vprintf	Stuurt geformatteerde uitvoer naar **stdout**, gebruik makend van een argumentenlijst.
vsprintf	Stuurt geformatteerde uitvoer naar een string, gebruik makend van een argumentenlijst.

Deze functies nemen de formaatstring gespecifieerd door het *formaat* argument en passen elk volgend argument tot de formaat specifieerders in de string toe, van links naar rechts. Elk karakter in de formaatstring wordt naar de string gekopieerd behalve de conversie karakters die een formaat specifieerder specifiëren.

De string opdrachten (**sprintf** en **vsprintf**) voegen een null karakter toe aan het einde van de string. Dit null karakter wordt niet meegerekend bij de karaktertelling.

De argumentenlijst opdrachten (**vfprintf**, **vprintf** en **vsprintf**) gebruiken een argumentenlijst die wordt klaargemaakt door **va_start**. Deze opdrachten roepen **va_end** zelf niet aan (de gebruiker moet dit aanroepen).

Een conversie specifieerder begint met het **%** karakter. Na dit karakter komt het volgende in deze volgorde:

[**vlaggen**]	Controleert de conversie (optioneel).
[**breedte**]	Definieert het aantal te drukken karakters (optioneel).
[**.precisie**]	Definieert de hoeveelheid van precisie om af te drukken voor een cijfertype (optioneel).
[**aanpasser**]	Overschrijft de grootte (type) van het argument (optionaal).
[**type**]	Het toe te passen conversietype (verplicht).

Vlaggen:

- De waarde wordt links uitgelijnd (default is rechts). Overschrijft de **0** vlag.

+ Forceert het teken (+ of -) om altijd getoond te worden. Default is enkel het - teken. Overschrijft de spatie vlag.

spatie Dwingt een positieve waarde een spatie voor het teken te tonen. Negatieven tonen nog steeds het - teken.

Alternatieve vorm:

Conversie karakter	Resulaat
o	De precisie wordt verhoogd om het eerste teken een nul te maken.
X or x	Niet-nul zal 0x of 0X als prefix hebben.
E, e, f, g, or G	Het resultaat zal altijd een komma (decimaal punt) hebben.
G or g	Achterloopnullen zullen niet verwijderd worden.

0 Voor **d**, **i**, **o**, **x**, **X**, **e**, **E**, **f**, **g** en **G** worden voorloopnullen gebruikt om het veld vol te maken in plaats van spaties. Dit is enkel nuttig met een breedte specifieerder. Precisie overschrijft deze vlag.

Breedte:

De breedte van het veld is hier gespecifieerd met een decimale waarde. Als de waarde niet groot genoeg is om de breedte te vullen, dan wordt de rest van het veld gevuld met spaties (behalve als de 0 vlag ingesteld is). Als de waarde de breedte van het veld overloopt (overflow), dan specifieert het volgende argument (dat van het type integer moet zijn) de breedte van het veld. Merk op: bij het gebruiken van de ***** met de breedte en/of precisie specifieerder, komt het breedte argument eerst, dan het precisie argument en dan de te converteren waarde.

Precisie:

De precisie begint met een punt (**.**) om zichzelf te onderscheiden van het breedte argument. De precisie kan gegeven worden als decimale waarde of als een asterisk (*****). Als een ***** gebruikt wordt, dan specifieert het volgende argument (dat van het type integer is) de precisie. Merk op: bij het gebruiken van de ***** met de breedte en/of precisie specifieerder, komt het breedte argument eerst, dan het precisie argument en dan de te converteren waarde. Precisie beïnvloedt het c type niet.

[.precisie]	**Resultaat**
(*geen*)	Default precisie waarden: 1 voor **d**, **i**, **o**, **u**, **x**, **X** types. Het minimum aantal te verschijnen cijfers. 6 voor **f**, **e**, **E** types. Specifieert het aantal cijfers na de komma. Voor **g** or **G** types worden alle significante cijfers getoond. Voor het **s** type worden alle karakters in de string gedrukt tot aan, maar niet inclusief, het null karakter.
. of **.0**	Voor **d**, **i**, **o**, **u**, **x**, **X** types wordt de default precisie waarde gebruikt, behalve als de waarde nul is. In dat geval er geen karakters worden gedrukt. Voor **f**, **e**, **E** types worden er geen deciamaal punt karakters of cijfers gedrukt. Voor **g** or **G** types wordt aangenomen dat de precisie **1** is.
.n	Voor **d**, **i**, **o**, **u**, **x**, **X** types worden er tenminste **n** cijfers gedrukt (opvullend met nullen indien nodig). Voor **f**, **e**, **E** types specifieert het aantal cijfers na de komma (decimaal punt). Voor **g** or **G** types het aantal significante cijfers om te drukken. Voor **het s** type specifieert het maximum aantal te drukken karakters.

Aanpasser:

Een aanpasser verandert de manier waarop een conversie specifieerder type wordt geïnterpreteerd.

[aanpasser]	[type]	Effect
h	**d**, **i**, **o**, **u**, **x**, **X**	De waarde wordt eerst geconverteerd naar een **short int** of een **unsigned short int**.
h	**n**	Specifieert dat de pointer naar een **short int** wijst.
l	**d**, **i**, **o**, **u**, **x**, **X**	De waarde wordt eerst geconverteerd naar een **long int** of een **unsigned long int**.
l	**n**	Specifieert dat de pointer naar een **long int** wijst.
L	**e**, **E**, **f**, **g**, **G**	De waarde wordt eerst naar een **long double** geconverteerd.

Conversie specifieerder type:

De conversie specifieerder specifieert als welk type het argument moet behandelt worden.

[type]	Uitvoer
d, **i**	Type **signed int**.
o	Type **unsigned int** gedrukt in octaal.
u	Type **unsigned int** gedrukt in decimaal.
x	Type **unsigned int** gedrukt in hexadecimaal als dddd gebruik makend van a, b, c, d, e, f.
X	Type **unsigned int** gedrukt in hexadecimaal als dddd gebruik makend van A, B, C, D, E, F.
f	Type **double** gedrukt als [-]ddd.ddd.
e, **E**	Type **double** gedrukt als [-]d.dddeñdd waar er één cijfer voor de decimaal gedrukt wordt (nul als de waarde nul is). Het exponent bevat tenminste twee cijfers. Als het type E is dan wordt het exponent met hoofdletter E gedrukt.
g, **G**	Type **double** gedrukt als type e of E als het exponent minder is dan -4 of gelijk is aan de precisie. Anders gedrukt als type f. Achterloopnullen worden verwijdert. Decimaal punt karakters verschijnen alleen als er een niet-nul decimaal cijfer is.
c	Type **char**. Een enkel karakter wordt gedrukt.
s	Type pointer naar array. De string wordt afgedrukt volgens de precisie (geen precisie drukt de volledige string).
p	Drukt de waarde van een pointer (de geheugenlocatie die hij vasthoudt).
n	Het argument moet een pointer naar een **int** zijn. Slaat het aantal gedrukte karakters tot zover op in de integer. Er worden geen karakters gedrukt.
%	Een **%** teken wordt gedrukt.

Het aantal gedrukte karakters wordt teruggegeven. Als er een fout voorkwam, wordt **-1** teruggegeven.

8.13.4.2 ...scanf functies

Declaraties:

```
int fscanf(FILE *stream, const char *formaat, ...);
int scanf(const char *formaat, ...);
int sscanf(const char *str, const char *formaat, ...);
```

De **...scanf** functies voorzien een manier om geformatteerde informatie uit stream op te nemen.

fscanf	Leest geformatteerde invoer uit een stream.
scanf	Leest geformateerde invoer uit **stdin**.
sscanf	Leest geformatteerde invoer uit een string.

Deze functies nemen invoer op op een manier die gespecifieerd is door het *formaat* argument en slaan elk invoerveld op in de volgende argumenten, van links naar rechts.

Elk uitvoerveld wordt gespecifieerd in de formaat string met een conversie specifieerder die specifieert hoe de invoer moet worden opgeslagen in de juiste variabele. Andere karakters in de formaat string specifiëren karakters die moeten overeenkomen vanuit de invoer, maar die in geen van de volgende argumenten worden opgeslagen. Als de invoer niet overeenstemt dan stopt de functie met scannen en keert deze terug. Een witkarakter kan overeenkomen met elk ander witkarakter (spatie, tab, carriage return, nieuwe-lijn, verticale tab of formfeed) of het volgende niet-compatibele karakter.

Een invoerveld wordt gespecifieerd met een conversiespecifieerder die begint met het **%** karakter. Na het **%** karakter komt het volgende in deze volgorde:

[*] Toewijzingsonderdrukker (optioneel).
[breedte] Definieert het maximum aantal te lezen karakters (optioneel).
[aanpasser] Overschrijft de grootte (type) van het argument (optioneel).
[type] De toe te passen type conversie (verplicht).

Toewijzingsonderdrukker:

Zorgt er voor dat het invoerveld wordt gescand maar niet wordt opgeslagen in een variabele.

Breedte:

De maximum breedte wordt hiet gespecifieerd met een decimale waarde. Als de invoer kleiner is dan de aangegeven breedte, dan wordt hetgeen dat tot dusver gelezen was geconverteerd en in de variabele gestoken.

Aanpasser:

A aanpasser verandert de wijze waarop een conversie specifieerder wordt geïnterpreteerd.

[aanpasser]	[type]	Effect
h	**d**, **i**, **o**, **u**, **x**	Het argument is een **short int** of **unsigned short int**.
h	**n**	Specifieert dat de pointer wijst naar een **short int**.
l	**d**, **i**, **o**, **u**, **x**	Het argument is een **long int** of **unsigned long int** .
l	**n**	Specifieert dat de pointer wijst naar een **long int**.
l	**e**, **f**, **g**	Het argument is een **double**.
L	**e**, **f**, **g**	Het argument is een **long double**.

Conversie specifieerder type:

Deze specifieert welk type het argument is. Het controleert ook wat een geldig converteerbaar karakter is (welk soort van karakters het kan lezen zodat het naar iets compatibel kan geconverteerd worden).

[type]	Input
d	Type **signed int** vertegenwoordit in basis 10. Cijfers 0 tot en met 9 en het teken (+ of -).
i	Type **signed int**. De basis (radix) is afhankelijk van de eerste twee karakters. Als het eerste karakter een cijfer is van 1 tot 9, dan is de basis 10. Als het eerste cijfer een nul is en het tweede

cijfer is een cijfer van 1 tot 7, dan is de basis 8 (octaal). Als het eerste cijfer een nul is en het tweede karakter is een x of X, dan is de basis 16. (hexadecimaal).

o Type **unsigned int**. De invoer moet in basis 8 zijn (octaal). Alleen cijfers van 0 tot en met 7.

u Type **unsigned int**. De invoer moet in basis 10 zijn (decimaal). Alleen cijfers van 0 tot en met 9.

x, X Type **unsigned int**. De invoer moet in basis 16 zijn (hexadecimaal). Alleen cijfers van 0 tot en met 9 of A tot en met Z of a tot en met z.

e, **E**, **f**, **g**, **G** Type **float**. Begint met een optioneel teken. Dan één of meet cijfers, gevolg door een optioneel decimaal punt en een decimale waarde. Uiteindelijk beëindigd met een optionele van teken voorziene exponent waarde aangeduid met een e of een E.

s Type karakter array. Voert een sequentie van niet-witkarakters in. Het array moet groot genoeg zijn om de sequentie plus het toegevoegde null karakter vast te houden.

[...] Type karakter array. Voorziet een zoekset van karakters. Voorziet invoer van alleen die karakters tussen de vierkante haken (de scanset). Als het eerste karakter een carrot (**^**) is, dan wordt de scanset geïnverteerd en wordt elk ASCII teken toegelaten, behalve die tussen de vierkante haken staan. Op sommige systemen kan een bereik gespecifieerd worden met het dash karakter (**-**). Door het specifiëren van het beginnende karakter, een dash en een beëindigend karakter kan een bereik van karakters geïncludeerd worden in de scanset. Een null karakter wordt toegevoegd aan het einde van het aray.

c Type karakter array. Voert het aantal karakters in gespecifieerd in het breedte veld. Als er geen breedte veld gespecifieerd is, dan wordt 1 verondersteld. Er wordt geen null karakter toegevoegd aan het einde van het array.

p Pointer naar een pointer. Voert een geheugenadres in op dezelfde manier van het **%p** type geproduceert door de **printf** functie.

n Het argument moet een pointer naar een **int** zijn. Slaat het aantal tot dusver toe gelezen karakters op in de integer. Er worden geen karakter ingelezen uit de invoer stream.

% Vereist een overeenkomend **%** karakter uit de invoer.

Het lezen van een invoerveld (aangeduid met een conversie specifieerder) eindigt wanneer een niet-compatibel karakter wordt tegengekomen of als het breedte veld is voldaan.

Bij succes wordt het aantal geconverteerde en opgeslagen invoervelden teruggegeven. Als er een invoerfout voorkwam, dan wordt EOF teruggegeven.

8.13.5 karakter I/O functies

8.13.5.1 fgetc

Declaratie:

```
int fgetc(FILE *stream);
```

Neemt het volgende karakter (een **unsigned char**) van de gespecifieerde stream en schuift de positie indicator voor de stream vooruit.

8.13.5.2 fgets

Declaratie:

```
char fgets(char *str, int n, FILE *stream);
```

Leest een lijn van de gespecifieerde stream en slaat deze op in de string waar *str* naar wijst. Dit stopt wanneer ofwel (n-1) karakters zijn gelezen, het nieuwe-lijn karakter wordt gelezen of het einde-van-bestand is bereikt, wat dan ook eerst komt. Het nieuwe-lijn karakter wordt gekopieerd naar de string. Een null karakter wordt toegevoegd aan het einde van de string.

Bij succes wordt er een pointer naar de string teruggegeven. Bij een fout wordt een null pointer teruggegeven. Als het einde-van-bestand voorkomt voordat er een karakter gelezen werd, dan blijft de string onveranderd.

8.13.5.3 fputc

Declaratie:

```
int fputc(int kar, FILE *stream);
```

Schrijft een karakter (een **unsigned char**) gespecifieerd door het argument *kar* naar de aangegeven stream en schuift de positie indicator voor de stream vooruit.

Bij succes wordt het karakter teruggegeven. Als er een fout voorkomt, dan wordt de fout indicator voor de stream bepaald en EOF wordt teruggegeven.

8.13.5.4 fputs

Declaratie:

```
int fputs(const char *str, FILE *stream);
```

Schrijft een string naar de gespecifieerde stream tot, maar niet inclusief, het null karakter.

Bij succes wordt een niet-negatieve waarde teruggegeven. Bij een fout wordt EOF teruggegeven.

8.13.5.5 getc

Declaratie:

```
int getc(FILE *stream);
```

Neemt het volgende karakter (een **unsigned char**) van de gespecifieerde stream en schuift de positie indicator voor de stream vooruit.

Dit is mogelijk een macro versie van **fgetc**.

Bij succes wordt het karakter teruggegeven. Als het einde-van-bestand wordt ontmoet, dan wordt EOF teruggegeven en de einde-van-bestand indicator wordt ingesteld. Als er een fout voorkomt, dan wordt de fout indicator voor de stream ingesteld en EOF wordt teruggegeven.

8.13.5.6 getchar

Declaratie:

```
int getchar(void);
```

Neemt een karakter (een **unsigned char**) van **stdin**.

Bij succes wordt het karakter teruggegeven. Als het einde-van-bestand wordt ontmoet, dan wordt EOF teruggegeven en de einde-van-bestand indicator wordt ingesteld. Als er een fout voorkomt, dan wordt de fout indicator voor de stream ingesteld en EOF wordt teruggegeven.

8.13.5.7 gets

Declaratie:

```
char *gets(char *str);
```

Leest een lijn van **stdin** en slaat deze op in de string waar *str* naar wijst. Het stopt wanneer het nieuwe-lijn karakter wordt tegengekomen, of wanneer het einde-van-bestand werd ontmoet. Het nieuwe-lijn karakter wordt niet naar de string gekopieerd. Een null karakter wordt toegevoegd aan het eonge van de string.

8.13.5.8 putc

Declaratie:

```
int putc(int kar, FILE *stream);
```

Schrijft een karakter (een **unsigned char**) gespecifieerd door het argument *kar* naar de aangegeven stream en schuift de positie indicator voor de stream vooruit.

Dit is mogelijk een macro versie van **fputc**.

Bij succes wordt het karakter teruggegeven. Als er een fout voorkomt, dan wordt de fout indicator voor de stream bepaald en EOF wordt teruggegeven.

8.13.5.9 putchar

Declaratie:

```
int putchar(int kar);
```

Schrijft een karakter (een **unsigned char**) gespecifieerd door het argument *kar* naar **stdout**.

Dit is mogelijk een macro versie van **fputc**.

Bij succes wordt het karakter teruggegeven. Als er een fout voorkomt, dan wordt de fout indicator voor de stream bepaald en EOF wordt teruggegeven.

8.13.5.10 puts

Declaratie:

```
int puts(const char *str);
```

Schrijft een string naar **stdout** tot, maar niet inclusief, het null karakter. Een nieuwe-lijn karakter wordt toegevoegd aan de output.

Bij succes wordt er een niet-negatieven waarde teruggegeven. Bij een fout wordt EOF teruggegeven.

8.13.5.11 ungetc

Declaratie:

```
int ungetc(int kar, FILE *stream);
```

Duwt het karakter *kar* (een **unsigned char**) op de gespecifieerde stream zodat dit het volgende karakter is dat wordt gelezen. De funcites **fseek**, **fsetpos** en **rewind** gooien elk karakter dat op de stream geduwt wordt weg.

Veelvoudige karakters die op de stream geduwd worden worden op een FIFO manier gelezen.

Bij succes wordt het geduwde karakter teruggegeven. Bij een fout wordt EOF teruggegeven.

8.13.6 fout functies

8.13.6.1 perror

Declaratie:

```
void perror(const char *str);
```

Drukt een omschrijvende foutboodschap naar **stderr**. Eerst wordt de string gedrukt gevolgd door een dubbelpunt en een spatie. Dan wordt er een foutboodschap gebaseerd op de huidige instelling van de variabele **errno** gedrukt.

8.14 stdlib.h

De **stdlib** header definieert verscheidene algemene operatie functies en macro's.

Macro's:

```
NULL
EXIT_FAILURE
EXIT_SUCCESS
RAND_MAX
MB_CUR_MAX
```

Variabelen:

```
typedef size_t
typedef wchar_t
struct div_t
struct ldiv_t
```

Functies:

```
abort();
abs();
atexit();
```

```
atof();
atoi();
atol();
bsearch();
calloc();
div();
exit();
free();
getenv();
labs();
ldiv();
malloc();
mblen();
mbstowcs();
mbtowc();
qsort();
rand();
realloc();
srand();
strtod();
strtol();
strtoul();
system();
wcstombs();
wctomb();
```

8.14.1 variabelen en definities

size_t	Het unsigned integer resultaat van het **sizeof** sleutelwoord.
wchar_t	Een integer type van de grootte van een brede karakter constante.
div_t	De structuur teruggegeven door de **div** functie.
ldiv_t	De structuur teruggegeven door de **ldiv** functie.

NULL	De waarde van een null pointer constante.
EXIT_FAILURE en **EXIT_SUCCESS**	Waarden voor de **exit** functie om de afsluitstatus terug te geven.
RAND_MAX	De maximum waarde teruggegeven door de **rand** functie.
MB_CUR_MAX	Het maximum aantal bytes in een multibyte karakterset die niet groter kan zijn dan **MB_LEN_MAX**.

8.14.2 string functies

8.14.2.1 atof

Declaratie:

```
double atof(const char *str);
```

De string waar het argument *str* naar wijst wordt geconverteerd naar een floating-point getal (type **double**). Initiële witkarakters (spatie, tab, carriage return, nieuwe-lijn, vertikale tab of formfeed) worden

overgeslagen. Het getal kan bestaan uit een optioneel teken, een string van cijfers met een optioneel decimaal karakter en een optionele **e** of **E** gevolgd door een optioneel exponent met teken. Conversie stopt bij het eerste onherkenbare teken.

Als de waarde te klein is om als return voor het type **double** te dienen, dan wordt nul teruggegeven en **ERANGE** wordt opgeslagen in de variabele **errno**.

8.14.2.2 atoi

Declaratie:

```
int atoi(const char *str);
```

De string waar het argument *str* naar wijst wordt geconverteerd naar een integer (type **int**). Initiële witkarakters (spatie, tab, carriage return, nieuwe-lijn, vertikale tab of formfeed) worden overgeslagen. Het getal kan bestaan uit een optioneel teken en een string van cijfers. Conversie stopt bij het eerste onherkenbare teken.

Bij succes wordt het geconverteerde getal teruggegeven. Als het getal niet kan geconverteerd worden, dan wordt **0** teruggegeven.

8.14.2.3 atol

Declaratie:

```
long int atol(const char *str);
```

De string waar het argument *str* naar wijst wordt geconverteerd naar een long integer (type **long int**). Initiële witkarakters (spatie, tab, carriage return, nieuwe-lijn, vertikale tab of formfeed) worden overgeslagen. Het getal kan bestaan uit een optioneel teken en een string van cijfers. Conversie stopt bij het eerste onherkenbare teken.

Bij succes wordt het geconverteerde getal teruggegeven. Als het getal niet kan geconverteerd worden, dan wordt **0** teruggegeven.

8.14.2.4 strtod

Declaratie:

```
double strod(const char *str, char **eindptr);
```

De string waar het argument *str* naar wijst wordt geconverteerd naar een floating-point getal (type **double**). Initiële witkarakters (spatie, tab, carriage return, nieuwe-lijn, vertikale tab of formfeed) worden overgeslagen. Het getal kan bestaan uit een optioneel teken, een string van cijfers met een optioneel decimaal karakter en een optionele **e** of **E** gevolgd door een optioneel exponent met teken. Conversie stopt bij het eerste onherkenbare teken.

Het argument *eindptr* is een pointer naar een pointer. Het adres van het karakter dat de scan stopte wordt opgeslagen in de pointer waar *eindptr* naar wijst.

Bij succes wordt het geconverteerde getal teruggegeven. Als er geen conversie gedaan kon worden, dan wordt nul teruggegeven. Als de waarde buiten het bereik valt van het type **double**, dan wordt **HUGE_VAL**

teruggegeven, met het juiste teken, en **ERANGE** wordt opgeslagen in de variabele **errno**. Als de waarde te klein is om als return te dienen vor het type **double**, dan wordt nul teruggegeven en **ERANGE** wordt opgeslagen in de variabele **errno**.

8.14.2.5 strtol

Declaratie:

```
long int strtol(const char *str, char **eindptr, int basis);
```

De string waar het argument *str* naar wijst wordt geconverteerd naar een long integer (type **long int**). Initiële witkarakters (spatie, tab, carriage return, nieuwe-lijn, vertikale tab of formfeed) worden overgeslagen. Het getal kan bestaan uit een optioneel teken en een string van cijfers. Conversie stopt bij het eerste onherkenbare teken.

Als het basis (radix) argument nul is, dan hang de conversie af van de eerste twee karakters. Als het eerste karakter een cijfer is van 1 tot 9, dan is het basis 10. Als het eerste cijfer een nul is en het tweede cijfer is een cijfer van 1 tot 7, dan is het basis 8 (octaal). Als het eerste cijfer een nul is en het tweede karakter is een x of X, dan is het basis 16 (hexadecimaal).

Als het *basis* argument van 2 tot 36 is, dan wordt die basis gebruikt en karakters die buiten die basisdefinitie vallen worden als onconverteerbaar beschouwd. Voor basis 11 tot 36, worden de karakters A tot Z (of a tot z) gebruikt. Als de basis 16 is, dan mogen de karakters 0x of 0X het getal voorafgaan.

Het argument *eindptr* is een pointer naar een pointer. Het adres van het karakter dat de scan stopte wordt opgeslagen in de pointer waar *eindptr* naar wijst.

Bij succes wordt het geconverteerde getal teruggegeven. Als er geen conversie gedaan kon worden, dan wordt nul teruggegeven. Als de waarde buiten het bereik valt van het type **long int**, dan wordt **LONG_MAX** of **LONG_MIN** teruggegeven, met teken van de juiste waarde, en **ERANGE** wordt opgeslagen in de variabele **errno**. Als de waarde te klein is om als return te dienen vor het type **double**, dan wordt nul teruggegeven en **ERANGE** wordt opgeslagen in de variabele **errno**.

8.14.2.6 strtoul

Declaratie:

```
unsigned long int strtol(const char *str, char **eindptr, int
basis);
```

De string waar het argument *str* naar wijst wordt geconverteerd naar een unsigned long integer (type **unsigned long int**). Initiële witkarakters (spatie, tab, carriage return, nieuwe-lijn, vertikale tab of formfeed) worden overgeslagen. Het getal kan bestaan uit een optioneel teken en een string van cijfers. Conversie stopt bij het eerste onherkenbare teken.

Als het basis (radix) argument nul is, dan hang de conversie af van de eerste twee karakters. Als het eerste karakter een cijfer is van 1 tot 9, dan is het basis 10. Als het eerste cijfer een nul is en het tweede cijfer is een cijfer van 1 tot 7, dan is het basis 8 (octaal). Als het eerste cijfer een nul is en het tweede karakter is een x of X, dan is het basis 16 (hexadecimaal).

Als het *basis* argument van 2 tot 36 is, dan wordt die basis gebruikt en karakters die buiten die basisdefinitie vallen worden als onconverteerbaar beschouwd. Voor basis 11 tot 36, worden de karakters A tot Z (of a tot z) gebruikt. Als de basis 16 is, dan mogen de karakters 0x of 0X het getal voorafgaan.

Het argument *eindptr* is een pointer naar een pointer. Het adres van het karakter dat de scan stopte wordt opgeslagen in de pointer waar *eindptr* naar wijst.

Bij succes wordt het geconverteerde getal teruggegeven. Als er geen conversie gedaan kon worden, dan wordt nul teruggegeven. Als de waarde buiten het bereik valt van het type **unsigned long int**, dan wordt **ULONG_MAX** en **ERANGE** wordt opgeslagen in de variabele **errno**.

8.14.3 geheugenfuncties

8.14.3.1 calloc

Declaratie:

```
void *calloc(size_t nvoorwerpen, size_t grootte);
```

Alloceert het gevraagde geheugen en geeft een pointer naar dat geheugen terug. De gevraagde grootte is *nvoorwerpen*, elk *grootte* bytes lang (totaal gevraagd geheugen is *nvoorwerpen * grootte*). De ruimte wordt geïnitialiseerd tot allemaal nul bits.

Bij succes wordt een pointer tot de gevraagde ruimte teruggegeven. Bij mislukking wordt er een null pointer teruggegeven.

8.14.3.2 free

Declaratie:

```
void free(void ptr);
```

Dealloceert het geheugen vooraf gealloceerd door een aanroep tot **calloc**, **malloc**, of **realloc**. Het argument *ptr* wijst naar de ruimte die voordien werd gealloceerd. Als *ptr* wijst naar een geheugenblok dat niet werd gealloceerd met **calloc**, **malloc**, or **realloc**, of dat een ruimte is die gedealloceerd werd, dan is het resultaat ongedefinieerd.

Er is geen return.

8.14.3.3 malloc

Declaratie:

```
void *malloc(size_t grootte);
```

Alloceert het gevraagde geheugen en geeft een pointer naar dat geheugen terug. De gevraagde grootte is in *grootte* bytes. De waarde van de ruimte is onbeslist.

Bij succes wordt een pointer tot de gevraagde ruimte teruggegeven. Bij mislukking wordt er een null pointer teruggegeven.

8.14.3.4 realloc

Declaratie:

```
void *realloc(void *ptr, size_t grootte);
```

Pobeert het geheugenblok van grootte te veranderen waar *ptr* naar wijst en dat voordien werd gealloceerd met een aanroep tot **malloc** of **calloc**. De inhoud waar *ptr* naar wijst blijft onverandert. Als de waarde van *grootte* groter is dan de vorige grootte van het blok, dan hebben de extra bytes een onbesliste waarde. Als de waarde *grootte* kleiner is dan de vorige grootte van het blok, dan worden de overige bytes aan het einde van het blok geledigd. Als *ptr* null is, dan gedraagt deze zich als **malloc**. Als *ptr* naar een geheugenblok wijst dat niet gealloceerd werd met **calloc** of **malloc**, of het is een ruimte die werd gedealloceerd, dan is het resultaat niet gedefinieerd. Als de nieuwe ruimte niet gealloceerd kan worden, dan wordt de inhoud waar *ptr* naar wijst niet veranderd. Als *grootte* nul is, dan word het geheugenblok volledig geledigd.

Bij succes wordt een pointer tot het geheugenblok teruggegeven (dit kan in een andere locatie staan als voordien). Bij mislukking of als *grootte* nul is, dan wordt een null pointer teruggegeven.

8.14.4 omgevingsfuncties

8.14.4.1 abort

Declaratie:

```
void abort(void);
```

Veroorzaakt een abnormale beëindiging van het programma. Verhoogt het **SIGABRT** signaal en een onsuccesvolle beëindigingsstatus wordt naar de omgeving teruggekeerd. Of geopende streams wel of niet gesloten worden is gebruikers-afhankelijk.

Er is geen return mogelijk.

8.14.4.2 atexit

Declaratie:

```
int atexit(void (*func) (void));
```

Veroorzaakt een aanroep tot de gespecifieerde functie wanneer het programma normaal eindigt. Tenminste 32 functies kunnen gerigistreerd worden om aangeroepen worden wanneer het programma eindigt. Ze worden aangeroepen op een laatste-in, eerste-out basis (de laatst geregistreerde functie wordt eerst aangeroepen).

Bij succes wordt nul teruggegeven. Bij mislukking wordt een niet-nul waarde teruggegeven.

8.14.4.3 exit

Declaratie:

```
void exit(int status);
```

Veroorzaakt een normale beëindiging van het programma. Eerst worden de functies geregistreerd door **atexit** aangeroepen, daarna worden alle open streams geledigd en gesloten en alle tijdelijke bestanden geopend met **tmpfile** worden verwijderd. De waarde van *status* wordt teruggegeven aan de omgeving. Als *status* **EXIT_SUCCESS** is, dan geeft dit een succesvolle beëindiging aan. Als *status* **EXIT_FAILURE** is, dan geeft dit een onsuccesvolle beëindiging aan. Alle andere waarden zijn gebruikers-gedefinieerd.

Er is geen return mogelijk.

8.14.4.4 getenv

Declaratie:

```
char *getenv(const char naam);
```

Zoekt naar de omgevingsstring waar *naam* naar wijst en geeft de geassocieerde waarde aan de string terug. Naar de teruggegeven waarde zou niet mogen geschreven worden.

Als de string gevonden wordt, dan wordt een pointer naar de string zijn geassocieerde waarde teruggegeven. Als de string niet gevonden wordt, dan wordt een null pointer teruggegeven.

8.14.4.5 system

Declaratie:

```
int system(const char *string);
```

De opdracht gespecifieerd door *string* wordt doorgegeven aan de gastheeromgeving om uitgevoerd te worden door de command processor (bijvoorbeel Dos). Een null pointer kan gebruikt worden om na te gaan of de command processor al dan niet bestaat.

Als *string* een null pointer is en de processor bestaat, dan wordt nul teruggegeven. Alle andere return waarden zijn gebruikers-gedefinieerd.

8.14.5 zoek- en sorteerfuncties

8.14.5.1 bsearch

Declaratie:

```
void *bsearch(const void *sleutel, const void *basis, size_t
nvoorwerpen,
                size_t grootte, int (*vgl) (const void *, const
void *));
```

Doet een binaire zoekactie. De gespecifieerde *basis* wijst naar het begin van het array. Er word gezocht naar een voorwerp gelijk aan datgene waar *sleutel* naar wijst. Het array is *nvoorwerpen* lang met elk element in het array *grootte* bytes lang.

De vergelijkingsmethode wordt gespecifieerd door de *vgl* functie. Deze functie neemt twee argumenten, het eerste is de sleutel pointer en het tweede is het huidige element in het array dat vergeleken wordt. Deze functie moet minder dan nul teruggeven als de vergeleken waarde kleiner is dan de gespecifieerde sleutel. Ze moet nul teruggeven als de vergeleken waarde gelijk is aan de gespecifieerde sleutel. Ze moet groter dan nul teruggeven als de vergeleken waarde groter is dan de gespecifieerde sleutel.

Het array moet zo opgesteld zijn dat elementen die minder dan de sleutel vergelijken eerst komen, gelijke elementen daarop volgen en als laatste grotere elementen geplaatst worden.

Als er een overeenkomst gevonden wordt, dan wordt een pointer naar die overeenkomst teruggestuurd. Anders wordt er een null pointer teruggekeerd. Indien er meerdere overeenkomstige sleutels gevonden worden, dan is het ongespecifieerd welke sleutel wordt teruggegeven.

8.14.5.2 qsort

Declaratie:

```
void *qsort(void *basis, size_t nvoorwerpen, size_t grootte,
                         int (*vgl) (const void *, const void *));
```

Sorteert een array. De gespecifieerde *basis* wijst naar het begin van het array. Het array is *nvoorwerpen* lang met elk element in het array *grootte* bytes lang.

De elementen worden gesorteerd in oplopende volgorde volgens de *vlg* functie. Deze functie neemt twee argumenten. Deze argumenten zijn twee elementen die vergeleken worden. Deze functie moet minder dan nul teruggeven als het eerste argument kleiner is dan het tweede. Ze moet nul teruggeven als het eerste argument gelijk is aan het tweede. Ze moet groter dan nul teruggeven als het eerste argument groter is dan het tweede.

Indien meerdere elementen gelijk zijn, dan is de manier waarop zij gesorteerd worden ongespecifieerd.

Er wordt geen waarde teruggegeven.

Voorbeeld:

```
#include<stdlib.h>
#include<stdio.h>
#include<string.h>
int main(void)
{
char string_array[10][50]={"John","Jane","Mary","Roger","Dave",

  "Paul","Beavis","Astro","George","Elroy"};
  /* De lijst sorteren */
  qsort(string_array,10,50,strcmp);
  /* Naar "Elroy" zoeken en dit afdrukken */
  printf("%s",bsearch("Elroy",string_array,10,50,strcmp));
  return 0;
}
```

8.14.6 wiskundige functies

8.14.6.1 abs

Declaratie:

```
int abs(int x);
```

Geeft de absolute waarde van *x* terug. Merk op dat bij het twee-complement het meeste maximum getal niet kan weergegeven worden als positief getal. Het resultaat in dit geval is ongedefinieerd.

De absolute waarde wordt teruggegeven.

8.14.6.2 div

Declaratie:

```
div_t div(int numer, int denom);
```

Deelt *numer* (numerator) door *denom* (denominator). Het resultaat wordt opgeslagen in **div_t** dat twee leden heeft:

```
int quot;
int rem;
```

waar *quot* het quotiënt is en *rem* de rest (remainder). In het geval van niet-exacte deling, wordt *quot* naar de dichtbijzijnde integer afgerond. De waarde *numer* is gelijk aan (**quot * denom + rem**).

De waarde van de deling wordt teruggegeven in de strucuture.

8.14.6.3 abs

Declaratie:

```
long int labs(long int x);
```

Geeft de absolute waarde van *x* terug. Merk op dat bij het twee-complement het meeste maximum getal niet kan weergegeven worden als positief getal. Het resultaat in dit geval is ongedefinieerd.

De absolute waarde wordt teruggegeven.

8.14.6.4 ldiv

Declaratie:

```
ldiv_t ldiv(int numer, int denom);
```

Deelt *numer* (numerator) door *denom* (denominator). Het resultaat wordt opgeslagen in **ldiv_t** dat twee leden heeft:

```
long int quot;
long int rem;
```

waar *quot* het quotiënt is en *rem* de rest (remainder). In het geval van niet-exacte deling, wordt *quot* naar de dichtbijzijnde integer afgerond. De waarde *numer* is gelijk aan (**quot * denom + rem**).

De waarde van de deling wordt teruggegeven in de strucuture.

8.14.6.5 rand

Declaratie:

```
int rand(void);
```

Geeft een pseudo-willekeurig getal terug in het bereik van **0** tot **RAND_MAX**.

8.14.6.6 srand

Declaratie:

```
void rand(unsigned int seed);
```

Deze functie plaatst de willekeurig-getal generator gebruikt door de functie **rand**. Het plaatsen van **srand** met dezelfde plaatsing zal er voor zorgen dat **rand** dezelfde sequentie van pseudo-willekeurige getallen teruggeeft. Als **srand** niet aangeroepen wordt, dan gedraagt **rand** zich alsof **srand(1)** werd aangeroepen.

8.14.7 multibyte functies

8.14.7.1 mblen

Declaratie:

```
int mblen(const char *str, size_t n);
```

Geeft de lengte terug van de lengte van een multibyte karakter waar het argument *str* naar wijst. Ten meeste *n* bytes zullen bekeken worden.

Als *str* een null pointer is, dan wordt nul teruggegeven als multibyte karakters niet staat-afhankelijk zijn (shift state). Anders wordt een niet-nul waarde teruggegeven als multibyte karakters staat-afhankelijk zijn.

Als *str* niet null is, dan wordt het aantal bytes teruggegeven dat wordt bijgehouden in het multibyte karakter waar *str* naar wijst. Nul wordt teruggegeven als *str* naar een null karakter wijst. Een waarde van -1 wordt teruggegeven als *str* niet naar een geldig multibyte karakter wijst.

8.14.7.2 mbstowcs

Declaratie:

```
size_t mbstowcs(schar_t *pwcs, const char *str, size_t n);
```

Converteert de string van multibyte karakters waar het argument *str* naar wijst naar de string waar *pwcs* naar wijst. Er worden niet meer dan *n* waarden in het array opgeslagen. Conversie stopt wanneer het null karakter ontmoet wordt, of wanneer *n* waarden werden opgeslagen. Het null karakter wordt als nul opgeslagen in het array maar wordt niet geteld in de teruggegeven waarde.

Indien een ongeldig multibyte karakter wordt bereikt wordt de waarde -1 teruggegeven. Anders wordt het aantal waardes opgeslagen in het array teruggegeven niet inclusief het afsluitende nul karakter.

8.14.7.3 mbtowc

Declaratie:

```
int mbtowc(wchar_t *pwc, const char *str, size_t n);
```

Examineert het multibyte karakter waar het argument *str* naar wijst. De waarde wordt geconverteerd en opgeslagen in het argument *pwc* als *pwc* niet null is. Er worden ten meeste *n* bytes gelezen.

Als *str* een null pointer is, dan wordt nul teruggegeven als multibyte karakters niet staat-afhankelijk zijn (shift state). Anders wordt een niet-nul waarde teruggegeven als multibyte karakters staat-afhankelijk zijn.

Als *str* niet null is, dan wordt het aantal bytes teruggegeven dat wordt bijgehouden in het multibyte karakter waar *str* naar wijst. Nul wordt teruggegeven als *str* naar een null karakter wijst. Een waarde van -1 wordt teruggegeven als *str* niet naar een geldig multibyte karakter wijst.

8.14.7.4 wcstombs

Declaratie:

```
size_t wcstombs(char *str, const wchar_t *pwcs, size_t n);
```

Converteert de code opgeslagen in het array *pwcs* naar multibyte karakter en slaat deze op in de string *str*. Ten meeste *n* bytes worden naar de string gekopieerd. Als een multibyte karakter de *n* grens te buiten gaat, dan worden geen van de bytes van dat multibyte karakter gekopieerd. Conversie stopt wanneer het null karakter ontmoet wordt, of wanneer *n* bytes naar de string werden gekopieerd. Het null karakter wordt in de string opgeslagen maar wordt niet geteld in de teruggegeven waarde.

Indien een ongeldige code wordt bereikt wordt de waarde -1 teruggegeven. Anders wordt het aantal bytes opgeslagen in de string teruggegeven niet inclusief het afsluitende nul karakter.

8.14.7.5 wctomb

Declaratie:

```
int wctomb(char *str, wchar_t wchar);
```

Examineert de code die correspondeert met een multibyte karakter gegeven door het argument *wchar*. De code wordt geconverteerd naar een multibyte karakter en opgeslagen in de string waar het argument *str* naar wijst als *str* niet null is.

Als *str* een null pointer is, dan wordt nul teruggegeven als multibyte karakters niet staat-afhankelijk zijn (shift state). Anders wordt een niet-nul waarde teruggegeven als multibyte karakters staat-afhankelijk zijn.

Als *str* niet null is, dan wordt het aantal bytes teruggegeven dat wordt bijgehouden in het multibyte karakter *wchar*. Een waarde van -1 wordt teruggegeven als *wchar* geen geldig multibyte karakter is.

8.15 string.h

De **string** header voorziet vele functies die nuttig zijn voor het manipuleren van strings (karakter arrays).

Macro's:

```
NULL
```

Variabelen:

```
typedef size_t
```

Functies:

```
memchr();
memcmp();
memcpy();
memmove();
memset();
strcat();
strncat();
strchr();
strcmp();
strncmp();
strcoll();
strcpy();
strncpy();
strcspn();
strerror();
strlen();
strpbrk();
strrchr();
strspn();
strstr();
strtok();
strxfrm();
```

8.15.1 variabelen en definities

size_t	Het unsigned integer resultaat van het **sizeof** sleutelwoord.
NULL	De waarde van een null pointer constante.

8.15.2 functies

8.15.2.1 memchr

Declaratie:

```
void *memchr(const void *str, int k, size_t n);
```

Zoekt naar het eerste voorkomen van karakter *k* (een **unsigned char**) in de eerste *n* bytes van de string waar het argument *str* naar wijst.

Geeft een pointer terug die wijst naar het eerste overeenkomende karakter, of null indien er geen overeenkomst gevonden werd.

8.15.2.2 memcmp

Declaratie:

```
int memcmp(const void *str1, const void *str2, size_t n);
```

Vergelijkt de eerste *n* bytes van *str1* en *str2*. Stopt niet met vergelijken, zelfs niet als het null karakter ontmoet wordt (er worden altijd *n* karakters gecontroleerd).

Geeft nul terug als de eerste *n* bytes van *str1* en *str2* gelijk zijn. Geeft minder dan nul of meer dan nul terug als *str1* respectievelijk kleiner of groter dan *str2* is.

8.15.2.3 memcpy

Declaratie:

```
void *memcpy(void *str1, const void *str2, size_t n);
```

Kopieert *n* karakters van *str2* naar *str1*. Als *str1* en *str2* elkaar overlappen dan is het gedrag ongedefinieerd.

Geeft het argument *str1* terug.

8.15.2.4 memmove

Declaratie:

```
void *memmove(void *str1, const void *str2, size_t n);
```

Kopieert *n* karakters van *str2* naar *str1*. Als *str1* en *str2* elkaar overlappen wordt de informatie eerst volledig gelezen van *str1* en daarna naar *str2* gelezen zodat de karakters op de juiste manier gekopieerd worden.

Geeft het argument *str1* terug.

8.15.2.5 memset

Declaratie:

```
void *memset(void *str, int k, size_t n);
```

Kopieert het karakter *k* (een **unsigned char**) naar de eerste *n* karakters van de string waar het argument *str* naar wijst.

Geeft het argument *str* terug.

8.15.2.6 strcat

Declaratie:

```
char *strcat(char *str1, const char *str2);
```

Voegt de string waar *str2* naar wijst toe aan het einde van de string waar *str1* naar wijst. Het afsluitende null karakter van *str1* wordt overschreven. Het kopiëren stopt wanneer het afsluitende null karakter van *str2* gekopieerd werd. Als er overlapping voorkomt, dan is het resultaat ongedefinieerd.

Geeft het argument *str1* terug.

8.15.2.7 strncat

Declaratie:

```
char *strncat(char *str1, const char *str2, size_t n);
```

Voegt de string waar *str2* naar wijst toe aan het einde van de string waar *str1* naar wijst tot *n* karakters lang. Het afsluitende null karakter van *str1* wordt overschreven. Het kopiëren stopt wanneer *n* karakters gekopieerd werden of wanneer het afsluitende null karakter van *str2* gekopieerd werd. Een afsluitend null karakter wordt altijd toegevoegd aan *str1*. Als er overlapping voorkomt, dan is het resultaat ongedefinieerd.

Geeft het argument *str1* terug.

8.15.2.8 strchr

Declaratie:

```
char *strchr(const char *str, int k);
```

Zoekt naar het eerste voorkomen van het karakter *k* (een **unsigned char**) in de string waar *str* naar wijst. Het afsluitende null karakter wordt beschouwd als een deel van de string.

De return is een pointer die wijst naar het eerste overeenkomstige karakter, of null als er geen overeenkomst voorkwam.

8.15.2.9 strcmp

Declaratie:

```
int strcmp(const char *str1, const char *str2);
```

Vergelijkt de string waar *str1* naar wijst met de string waar *str2* naar wijst.

Geeft nul terug als *str1* en *str2* gelijk zijn. Geeft minder dan nul of meer dan nul terug als *str1* respectievelijk kleiner of groter dan *str2* is.

8.15.2.10 strncmp

Declaratie:

```
int strncmp(const char *str1, const char *str2, size_t n);
```

Vergelijkt ten meeste de eerste *n* bytes van *str1* en *str2*. Stopt met vergelijken als het null karakter ontmoet wordt.

Geeft nul terug als de eerste *n* bytes (of null afgesloten lengte) van *str1* en *str2* gelijk zijn. Geeft minder dan nul of meer dan nul terug als *str1* respectievelijk kleiner of groter dan *str2* is.

8.15.2.11 strcoll

Declaratie:

```
int strcoll(const char *str1, const char *str2);
```

Vergelijkt string *str1* met *str2*. Het resultaat is afhankelijk van de **LC_COLLATE** instelling van de locatie.

Geeft nul terug als *str1* en *str2* gelijk zijn. Geeft minder dan nul of meer dan nul terug als *str1* respectievelijk kleiner of groter dan *str2* is.

8.15.2.12 strcpy

Declaratie:

```
char *strcpy(char *str1, const char *str2);
```

Kopieert de string waar *str2* naar wijst naar de string waar *str1* naar wijst. Kopieert tot en inclusief het null karakter van *str2*. Als *str1* en *str2* overlappen is het gedrag ongedefinieerd.

Geeft het argument *str1* terug.

8.15.2.13 strncpy

Declaratie:

```
char *strncpy(char *str1, const char *str2, size_t n);
```

Kopieert tot *n* karakters van de string waar *str2* naar wijst naar de string waar *str1* naar wijst. Het kopiëren stopt als *n* karakters gekopieerd zijn of als het afsluitende null karakter in *str2* werd ontmoet. Als het null karakter werd ontmoet, worden de null karakters achtereenvolgens gekopieerd naar *str1* totdat er *n* karakters gekopieerd zijn.

Geeft het argument *str1* terug.

8.15.2.14 strcspn

Declaratie:

```
size_t strcspn(const char *str1, const char *str2);
```

Vindt de eerste sequentie van karakters in de string *str1* die geen enkel karakter gevat gespecifieerd in *str2*.

Geeft de lengte terug van de eerst gevonden sequentie van karakters die niet overeenkomen met *str2*.

8.15.2.15 strerror

Declaratie:

```
char *strerror(int foutnum);
```

Zoekt in een intern array naar het foutnummer *foutnum* en geeft een pointer naar een foutboodschap string terug.

Geeft een pointer naar een foutboodschap string terug.

8.15.2.16 strlen

Declaratie:

```
size_t strlen(const char *str);
```

Berekent de lengte van de string *str* tot maar niet inclusief het afsluitende null karakter.

Geeft het aantal karakters in de string terug.

8.15.2.17 strpbrk

Declaratie:

```
char *strpbrk(const char *str1, const char *str2);
```

Geeft het eerste karakter in de string *str1* dat overeenkomt met één van de karakters gespecifieerd in *str2*.

Een pointer naar de locatie van dit karakter wordt teruggegeven. Een null pointer wordt teruggegeven als geen enkel karakter in *str2* bestaat in *str1*.

Voorbeeld:

```
#include<string.h>
#include<stdio.h>
int main(void)
{
char string[]="Hallo daar, Tom!";
char *string_ptr;
  while((string_ptr=strpbrk(string," "))!=NULL)
  *string_ptr='-';
  printf("Nieuwe string is \"%s\".\n",string);
  return 0;
}
```

De uitvoer zou erin moeten resulteren dat elke spatie in de string geconverteerd naar een dash (**-**).

8.15.2.18 strrchr

Declaratie:

```
char *strrchr(const char *str, int k);
```

Zoekt naar het laatste voorkomen van het karakter *k* (een **unsigned char**) in de string waar *str* naar wijst. Het afsluitende null karakter wordt beschouwd als een deel van de string.

De return is een pointer die wijst naar het laatste overeenkomstige karakter, of null als er geen overeenkomst voorkwam.

8.15.2.19 strspn

Declaratie:

```
size_t strspn(const char *str1, const char *str2);
```

Vindt de eerste sequentie van karakters in de string *str1* die één van de karakters gevat gespecifieerd in *str2*.

Geeft de lengte terug van de eerst gevonden sequentie van karakters die overeenkomen met *str2*.

Voorbeeld:

```
#include<string.h>
#include<stdio.h>
int main(void)
{
char string[]="2300 Turnhout";
  printf("De cijferlengte is %d.\n",strspn(string,"1234567890"));
  return 0;
}
```

De uitvoer zou moeten zijn:

```
De cijferlengte is 4.
```

8.15.2.20 strstr

Declaratie:

```
char *strstr(const char *str, const char *str2);
```

Vindt het eerste voorkomen van de volledige string *str2* (niet inclusief het afsluitende null karakter) die voorkomt in de sting *str1*.

Geeft een pointer naar het eerste voorkomen van *str2* in *str1* terug. Indien er geen overeenkomst gevonden werd, dan wordt een null pointer teruggekeerd. Als *str2* naar een string van nul lengte wijst, dan wordt het argument *str1* teruggegeven.

8.15.2.21 strtok

Declaratie:

```
char *strtok(char *str1, const char *str2);
```

Breekt string *str1* in een serie van delen. Als *str1* en *str2* niet null zijn, dan begint volgende zoek sequentie. Het eerste karakter in *str1* dat niet voorkomt in *str2* wordt gevonden. Als *str1* volledig uit karakters bestaat gespecifieerd in *str2*, dan bestaat er geen deel en wordt er een null pointer teruggegeven. Indien dit karakter gevonden wordt, dan markeert dit het begin van het eerste deel. Het begint dan met zoeken voor het volgende karakter volgend op hetgeen dat zich in *str2* bevindt. Wordt dit teken niet gevonden, dan breidt het huidige deel uit tot het einde van *str1*. Als het karakter gevonden wordt, dan wordt dit overshcreven met een null karakter, dat het huidige deel beëindigt. De functie slaat dan de volgende positie intern op en keert terug.

Aansluitende aanroepen met een null pointer voor *str1* zorgen er voor dat de vorige opgeslagen positie wordt hersteld en het zoeken begint vanaf dat punt. Opeenvolgende aanroepen kunnen telkens een andere waarde voor *str2* gebruiken.

De return is een pointer naar het eerste deel in *str1*. Indien er geen deel werd gevonden dan wordt een null pointer teruggegeven.

Voorbeeld:

```
#include<stdlib.h>
#include<string.h>
#include<stdio.h>
int main(void)
{
char zoek_string[]="President Bill Clinton";
char *array[50];
int loop;
  array[0]=strtok(zoek_string," ");
  if(array[0]==NULL)
  {
         printf("Niets om te zoeken.\n");
         exit(0);
  }
  for(loop=1;loop<50;loop++)
  {
         array[loop]=strtok(NULL," ");
         if(array[loop]==NULL)
                break;
  }
  for(loop=0;loop<50;loop++)
  {
         if(array[loop]==NULL)
         break;
         printf("Item #%d is %s.\n",loop,array[loop]);
  }
  return 0;
}
```

Dit programma vervangt elke spatie in een null karakter en slaat een pointer naar elke substring in het array op. Daarna wordt elk item afgedrukt.

8.15.2.22 strxfrm

Declaratie:

```
size_t strxfrm(char *str1, const char *str2, size_t n);
```

Transformeert de string *str2* en plaatst het resultaat in *str1*. Het kopieert ten meeste *n* karakters naar *str1* inclusief het afsluitende null karakter. De transformatie gebeurt zo dat **strcmp** toegepast op twee aparte geconverteerde strings dezelfde waarde teruggeeft als **strcoll** toegepast op dezelfde twee strings. Als overlapping voorkomt, dan is het resultaat ongedefinieerd.

Geeft de lengte van de getransformeerde string terug (exclusief het null karakter).

8.16 time.h

De **time** header heeft verschillende functies nuttig voor het lezen en converteren van de huidige tijd en datum. Het gedrag van sommige functies is gedefinieerd door de **LC_TIME** category van de locatie instelling.

Macro's:

```
NULL
CLOCKS_PER_SEC
```

Variabelen:

```
typedef size_t
typedef clock_t
typedef size_t
struct tm
```

Functies:

```
asctime();
clock();
ctime();
difftime();
gmtime();
localtime();
mktime();
strftime();
time();
```

8.16.1 variabelen en definities

NULL	De waarde van een null pointer constante.
CLOCKS_PER_SEC	Het aantal processor clocks per seconde.

size_t	Het unsigned integer resultaat van het **sizeof** sleutelwoord.
clock_t	Een type geschikt voor het opslagen van de processortijd.
time_t	Een type geschikt voor het opslagen van de kalendertijd.

struct tm is een structure gebruikt om de tijd en datum vast te houden. Zijn leden zijn als volgt:

```
int tm_sec;      /* seconden na de minuut (0 tot 61) */
int tm_min;      /* minuten na het uur (0 tot 59) */
int tm_hour;     /* uren sinds middernacht (0 tot 23) */
int tm_mday;     /* dag van de maand (1 tot 31) */
int tm_mon;      /* maanden sinds januari (0 tot 11) */
int tm_year;     /* jaren sinds 1900 */
int tm_wday;     /* dagen sinds zondag (0 tot 6, zondag = 0) */
int tm_yday;     /* dagen sinds 1 januari (0 to 365) */
int tm_isdst;    /* daglicht besparingen tijd */
```

8.16.2 functies

8.16.2.1 asctime

Declaration:

```
char *asctime(const struct tm *tijdptr);
```

Geeft een pointer terug naar een string die de dag en tijd van de structure *tijdptr* vertegenwoordigt. Het formaat van de string is als volgt:

```
DDD MMM dd hh:mm:ss JJJJ
```

DDD	Dag van de week (Sun, Mon, Tue, Wed, Thu, Fri, Sat)
MMM	Maand van het jaar (Jan, Feb, Mar, Apr, May, Jun, Jul, Aug, Sep, Oct, Nov, Dec)
dd	Dag van de maand (1,...,31)
hh	Uur (0,...,23)
mm	Minuut (0,...,59)
ss	Seconde (0,...,59)
JJJJ	Jaar

De string wordt afgesloten met een nieuwe-lijn karakter en een null karakter. De string is altijd 26 karakters lang (inclusief de afsluitende nieuwe-lijn en null karakters).

Er wordt een pointer naar de string teruggegeven.

Voorbeeld 1:

```
#include<time.h>
#include<stdio.h>
int main(void)
{
time_t timer;
  timer=time(NULL);
  printf("De huidige tijd is %s.\n",asctime(localtime(&timer)));
  return 0;
}
```

Voorbeeld 2:

```
#include <string.h>
#include <time.h>
#include <stdio.h>
int main(void)
{
struct tm t;
char str[80];
  t.tm_sec    = 1; /* Seconden */
  t.tm_min    = 30; /* Minuten */
  t.tm_hour   = 9; /* Uur */
  t.tm_mday   = 22; /* Dag van de maand  */
  t.tm_mon    = 11; /* Maand */
  t.tm_year   = 56; /* Jaar (zonder eeuw) */
```

```
  t.tm_wday   = 4; /* Dag van de week  */
  /* converteert structure naar string */
  strcpy(str, asctime(&t));
  printf("%s\n", str);
  return 0;
}
```

8.16.2.2 clock

Declaration:

```
clock_t *clock(void);
```

Geeft de processor klok tijd terug gebruikt sinds het begin van een gebruiks-gedefinieerd tijdperk (normaal het begin van het programma). De teruggegeven waarde gedeeld door **CLOCKS_PER_SEC** resulteert in het aantal seconden. Als de waarde niet beschikbaar is wordt -1 teruggegeven.

Voorbeeld:

```
#include<time.h>
#include<stdio.h>
int main(void)
{
clock_t tiks1, tiks2;
  tiks1=clock();
  tiks2=tiks1;
  while((tiks2/CLOCKS_PER_SEC-tiks1/CLOCKS_PER_SEC)<1)
         tiks2=clock();
  printf("Duurde %ld tiks om 1 seconde te wachten.\n",tiks2-
tiks1);
  printf("CLOCKS_PER_SEC is %ld.\n",CLOCKS_PER_SEC);
  return 0;
}
```

8.16.2.3 ctime

Declaration:

```
char *ctime(const time_t *timer);
```

Geeft een string die de locale tijd vertegenwoordigt gebaseerd op het argument *timer* terug. Dit is equivalent met:

```
asctime(localtime(timer));
```

De teruggegeven string staat in het volgende formaat:

```
DDD MMM dd hh:mm:ss JJJJ
```

DDD Dag van de week (Sun, Mon, Tue, Wed, Thu, Fri, Sat)

MMM Maand van het jaar (Jan, Feb, Mar, Apr, May, Jun, Jul, Aug, Sep, Oct, Nov, Dec)

dd Dag van de maand (1,...,31)

hh Uur (0,...,23)

mm Minuut (0,...,59)
ss Seconde (0,...,59)
JJJJ Jaar

De string wordt afgesloten met een nieuwe-lijn karakter en een null karakter. De string is altijd 26 karakters lang (inclusief de afsluitende nieuwe-lijn en null karakters).

Een pointer naar de string wordt teruggegeven.

8.16.2.4 difftime

Declaration:

```
double difftime (time_t tijd1, time_t tijd2);
```

Berekent het verschil in seconden tussen *tijd1* en *tijd2* (*tijd1* - *tijd2*).

8.16.2.5 gmtime

Declaration:

```
struct tm *gmtime (const time_t timer);
```

De waarde van *timer* wordt opgebroken in de structure **tm** en uitgedrukt in Coordinated Universal Time (UTC) ook gekend als Greenwich Mean Time (GMT).

Een pointer naar de structure wordt teruggegeven. Een null pointer wordt teruggegeven als UTC niet beschikbaar is.

8.16.2.6 localtime

Declaration:

```
struct tm *localtime (const time_t timer);
```

De waarde van *timer* wordt opgebroken in de structure **tm** en uitgedrukt in de locale tijdszone.

Een pointer naar de structure wordt teruggegeven.

Voorbeeld:

```
#include<time.h>
#include<stdio.h>
int main(void)
{
time_t timer;
  timer = time(NULL);
  printf("De huidige tijd is %s.\n", asctime(localtime(&timer)));
  return 0;
}
```

8.16.2.7 mktime

Declaration:

```
time_t mktime (struct tm *tijdptr);
```

Converteert de structure waar *tijdptr* naar wijst in een **time_t** waarde naargelang de locale tijdszone. De waarden in de structure zijn niet begrensd tot hun restricities. Als ze hun grenzen te buiten gaan, dan worden ze aangepast zodat ze wel passen. De originele waarden van **tm_wday** waarden werden beperkt. **tm_day** (dag van de maand) wordt niet verbeterd totdat **tm_mon** en **tm_year** verbeterd zijn.

Na aanpassing vertegenwoordigt de structure nog steeds dezelfde tijd.

De gecodeerde **time_t** waarde wordt teruggegeven. Als de kalendertijd niet weergegeven kan worden, dan wordt -1 teruggegeven.

Voorbeeld:

```
#include<time.h>
#include<stdio.h>
#include<stdlib.h>
/* De eerste dag van de 21ste eeuw zoeken */
int main(void)
{
struct tm tijd_struct;
char dagen[7][4]={"Sun", "Mon", "Tue", "Wed", "Thu", "Fri",
"Sat"};
  tijd_struct.tm_year=2001-1900;
  tijd_struct.tm_mon=0;
  tijd_struct.tm_mday=1;
  tijd_struct.tm_sec=0;
  tijd_struct.tm_min=0;
  tijd_struct.tm_hour=0;
  tijd_struct.tm_isdst=-1;
  if(mktime(&tijd_struct)==-1)
  {
         printf("FOUT!!.\n");
         exit(0);
  }
  printf("January 1, 2001 is een
%s.\n",dagen[tijd_struct.tm_wday]);
  return 0;
}
```

8.16.2.8 strftime

Declaratie:

```
size_t strftime(char *str, size_t maxgrootte,
                const char *formaat, const struct tm *tijdptr);
```

Formatteert de tijd vertegenwoordigt in de structure *tijdptr* volgens de formatteerregels gedefinieerd in *formaat* en opgeslagen in *str*. Niet meer dan *maxgrootte* karakters worden opgeslagen in *str* (inclusief het afsluitende null karakter).

Alle karakters in de formaatstring worden gekopieerd naar de *str* string, inclusief het afsluitende null karakter, behalve voor conversie karakters. Een conversie karakter begint met het **%** teken en wordt gevolgd door een ander karakter dat een speciale waarde definieert waardoor het moet vervangen worden.

Conversie karakter	**Wordt vervangen door...**
%a	Afgekorte weekdag naam.
%A	Volledige weekdag naam.
%b	Afgekorte maand naam.
%B	Volledige maand naam.
%c	Juiste datum en tijd weergave.
%d	dag van de maand (01-31).
%H	Uur van de dag (00-23).
%I	Uur van de dag (01-12).
%j	Dag van het jaar (001-366).
%m	Maand van het jaar (01-12).
%M	Minuut van het uur (00-59).
%p	AM/PM bepaler.
%S	Seconde van de minuut (00-61).
%U	Weeknummer van het jaar waar zondag de eerste dag is van week 1 (00-53).
%w	Weekdag waar zondag dag 0 is (0-6).
%W	Weeknummer van het jaar waar maandag de eerste dag is van week 1 (00-53).
%x	Juiste data weergave.
%X	Juiste tijd weergave.
%y	Jaar zonder eeuw (00-99).
%Y	Jaar met eeuw.
%Z	Tijdszone (mogelijk afgekort) of geen karakters als de tijdszone niet beschikbaar is.
%%	%.

Geeft het aantal karakters opgeslagen in *str* terug nexclusief het afsluitende null karakter. Bij een fout wordt nul teruggegeven.

8.16.2.9 time

Declaratie:

```
time_t time(time_t timer);
```

Berekent de huidige kalendertijd en codeert deze in **time_t** formaat.

De **time_t** waarde wordt teruggegeven. Als *timer* geen null pointer is, dan wordt de waarde ook opgeslagen in het object waar het naar wijst. Als de tijd niet beschikbaar is, dan wordt -1 teruggegeven.

8.17 de ASCII tabel

Dec	Oct	Hex	Karakter	Dec	Oct	Hex	Karakter
0	0	00	NUL	64	100	40	@
1	1	01	SOH	65	101	41	A
2	2	02	STX	66	102	42	B
3	3	03	ETX	67	103	43	C
4	4	04	EOT	68	104	44	D
5	5	05	ENQ	69	105	45	E
6	6	06	ACK	70	106	46	F
7	7	07	BEL	71	107	47	G
8	10	08	BS	72	110	48	H
9	11	09	HT	73	111	49	I
10	12	0A	LF	74	112	4A	J
11	13	0B	VT	75	113	4B	K
12	14	0C	FF	76	114	4C	L
13	15	0D	CR	77	115	4D	M
14	16	0E	SO	78	116	4E	N
15	17	0F	SI	79	117	4F	O
16	20	10	DLE	80	120	50	P
17	21	11	DC1	81	121	51	Q
18	22	12	DC2	82	122	52	R
19	23	13	DC3	83	123	53	S
20	24	14	DC4	84	124	54	T
21	25	15	NAK	85	125	55	U
22	26	16	SYM	86	126	56	V
23	27	17	ETB	87	127	57	W
24	30	18	CAN	88	130	58	X
25	31	19	EM	89	131	59	Y
26	32	1A	SUB	90	132	5A	Z
27	33	1B	ESC	91	133	5B	[
28	34	1C	FS	92	134	5C	\
29	35	1D	GS	93	135	5D	]
30	36	1E	RS	94	136	5E	^
31	37	1F	US	95	137	5F	_
32	40	20	SP	96	140	60	`
33	41	21	!	97	141	61	a
34	42	22	"	98	142	62	b
35	43	23	#	99	143	63	c
36	44	24	$	100	144	64	d
37	45	25	%	101	145	65	e
38	46	26	&	102	146	66	f

39	47	27	'	103	147	67	g
40	50	28	(	104	150	68	h
41	51	29	)	105	151	69	i
42	52	2A	*	106	152	6A	j
43	53	2B	+	107	153	6B	k
44	54	2C	,	108	154	6C	l
45	55	2D	-	109	155	6D	m
46	56	2E	.	110	156	6E	n
47	57	2F	/	111	157	6F	o
48	60	30	0	112	160	70	p
49	61	31	1	113	161	71	q
50	62	32	2	114	162	72	r
51	63	33	3	115	163	73	s
52	64	34	4	116	164	74	t
53	65	35	5	117	165	75	u
54	66	36	6	118	166	76	v
55	67	37	7	119	167	77	w
56	70	38	8	120	170	78	x
57	71	39	9	121	171	79	y
58	72	3A	:	122	172	7A	z
59	73	3B	;	123	173	7B	{
60	74	3C	<	124	174	7C	|
61	75	3D	=	125	175	7D	}
62	76	3E	>	126	176	7E	~
63	77	3F	?	127	177	7F	DEL

C-taal voor beginners - hoofdstuk 9

wiskundig

9.1 talstelsels

In dit hoofdstuk behandel ik oppervlakkig de verschillende talstelsels en methodes die u kan tegenkomen tijdens het programmeren. Normaal zou alle informatie correct moeten zijn, maar als u toch onwaarheden denkt gevonden te hebben, laat het mij dan weten. U heeft een browser nodig die machten en subscripts weergeeft.

9.1.1 decimaal

Het decimale talstelsel is het normale stelsel dat u reeds vroeg aangeleerd krijgt. Het bevat tien cijfers: 0, 1, 2, 3, 4, 5, 6, 7, 8 en 9. We zeggen daarom dat de basis 10 is. De regel bij talstelsels is dat een getal geschreven kan worden als de som van machten van de basis, in dit geval 10. Voorbeeld:

$$837{,}526_{10} = 8 * \mathbf{10^{2}} + 3 * \mathbf{10^{1}} + 7 * \mathbf{10^{0}} + 5 * \mathbf{10^{-1}} + 2 * \mathbf{10^{-2}} + 6 * \mathbf{10^{-3}}$$

U ziet ook dat de basis achter het getal geschreven wordt, in subscript.

9.1.2 binair

Het binaire systeem bevat slechts twee cijfers: 0 en 1. Dit is het systeem dat machines en computers gebruiken, omdat elektrische apparaten met bits werken. Een bit is ofwel 0(geen impuls) of 1(impuls). Door de algemene regel toe te passen kunnen we de decimale waarde als volgt vinden:

$$101{,}1101_{2} = 1 * 2^{2} + 0 * 2^{1} + 1 * 2^{0} + 1 * 2^{-1} + 1 * 2^{-2} + 0 * 2^{-3} + 1 * 2^{-4}$$

De volgende tabel toont hoe u tot 16 kan tellen in het binair

decimaal	**binair**
0	00000
1	00001
2	00010
3	00011
4	00100
5	00101
6	00110
7	00111
8	01000
9	01001
10	01010
11	01011
12	01100
13	01101
14	01110
15	01111

16	10000

Ik gebruik hier vijf bits om een binair getal weer te geven. Voorloopnullen mogen echter weggelaten worden, of u mag er bij plaatsen. Aan de rechtse kant mogen geen extra tekens geplaatst worden. Hoe groter de getallen worden, hoe groter het aantal bits geleidelijk aan zal worden.

9.1.2.1 binaire rekenregels

Optellen van binaire getallen:

+	**0**	**1**
0	0	1
1	1	10

Voorbeelden:

0 + 0 = 0
0 + 1 = 1
1 + 0 = 1
1 + 1 = 0 met carry 1 = 10
1 + 1 + 1 = 1 met carry 1 = 11

Dus:

$00101001_2 + 01001011_2 = 01110100_2$
$01000110_2 + 11111001_2 = \mathbf{1}00111111_2$

Vermenigvuldigen van binaire getallen:

*****	**0**	**1**
0	0	0
1	0	1

9.1.3 hexadecimaal

Het hexadecimale stelsel heeft als basis 16 en gebruikt de symbolen van 0 tot 9 en van A tot F. De alfabetische tekens worden gebruikt om het aantal unieke symbolen tot zestien te brengen. Nu werken we natuurlijk met machten van de basis 16:

$$302A,1D_{16} = 3 * 16^3 + 0 * 16^2 + 2 * 16^1 + 10 * 16^0 + 1 * 16^{-1} + 13 * 16^{-2}$$

We kunnen onze tabel als volgt uitbreiden:

decimaal	**hexadecimaal**	**binair**
0	0	00000
1	1	00001
2	2	00010
3	3	00011
4	4	00100

5	5	00101
6	6	00110
7	7	00111
8	8	01000
9	9	01001
10	A	01010
11	B	01011
12	C	01100
13	D	01101
14	E	01110
15	F	01111
16	10	10000

9.1.4 octaal

Bij het octale talstelsel gebruiken we een basis van 8. De symbolen gaan van 0 tot 7.

$$17{,}4_8 = 1 * 8^1 + 7 * 8^0 + 4 * 8^{-1} + 8* 8^{-2}$$

Tabel:

decimaal	**hexadecimaal**	**octaal**	**binair**
0	0	0	00000
1	1	1	00001
2	2	2	00010
3	3	3	00011
4	4	4	00100
5	5	5	00101
6	6	6	00110
7	7	7	00111
8	8	10	01000
9	9	11	01001
10	A	12	01010
11	B	13	01011
12	C	14	01100
13	D	15	01101
14	E	16	01110
15	F	17	01111
16	10	20	10000

9.2 machten

Bij het gebruiken en omvormen van deze talstelsels komen een boel machtsverheffingen kijken. Ik geef hier de voornaamste met de bedoeling dat u tenminste de eenvoudigste ervan instudeert.

getal	**10**	**2**	**8**	**16**

macht				
2	100	4	64	256
3	1000	8	512	4096
4	10000	16	4096	65536
5	100000	32	32768	1048576
6	1000000	64	262144	16777216
7	10000000	128	2097152	268435456
-1	0,1	0,5	0,125	0,0625
-2	0,01	0,25	0,015652	0,00390625
-3	0,001	0,125	0,001953125	n.v.t
-4	0,0001	0,0625	n.v.t.	n.v.t
-5	0,00001	0,03125	n.v.t.	n.v.t.
-6	0,000001	0,015625	n.v.t.	n.v.t.

Een lijst met grote binaire cijfers:

decimaal	**binair**
16	10000
32	100000
64	1000000
128	10000000
256	100000000
1024	1000000000
2048	10000000000
4096	100000000000

Als u goed kijkt merkt u een zekere logica op. Bij machten van 2 is het exponent van de macht gelijk aan het aantal nullen bij het binaire getal. Voorbeeld:

$32 = 2^5$, dus komen er 5 nullen bij het binaire getal: 100000

9.3 converties

9.3.1 van 10 naar 2, 8 en 16

Om een decimaal getal naar een binair, hexadecimaal of octaal getal om te vormen kan u éénzelfde mehode toepassen. Het enige dat telkens verschilt is de basis (2, 8 of 16).

Stel, we hebben een decimaal getal **21,375** en we willen dit converteren naar een binair getal (basis 2). We gebruiken de formule: "deel het getal door de basis en onthoud de rest":

0 <-- 1 <-- 2 <-- 5 <-- 10 <-- 21

1 0 1 0 1

We lezen van links naar rechts: 21 gedeeld door 2 is 10, met als rest 1; 10 gedeeld door 2 is 5 met als rest 0; enz... . We hebben nu het binaire gedeelte van het getal dat voor de komma hoort, namelijk **10101**. Wat het gedeelte na de komma betreft passen we een tweede formule toe: "vermenigvuldig het gedeelte

na de komma met de basis en gebruik het gedeelte na de komma van het uitkomstgetal om opnieuw met de basis te vermenigvuldigen":

0,375 * 2 = **0**,<u>750</u> --> 0,75 * 2 = **1**,<u>50</u> --> 0,50 * 2 = **1**,00

Het onderlijnde gedeelte wordt dus telkens gebruikt om opnieuw met de basis te vermenigvuldigen. De bits voor de komma gebruiken we als tweede deel van ons binair getal, namelijk **011**. De volledige uitkomst is dus:

$21{,}375_{10} = 10101{,}011_2$

Deze regels kan u ook toepassen om van basis 10 naar basis 8 en 16 te converteren. De regels blijven hetzelfde, enkel de basis verandert. In plaats van 2 gebruiken we dan 8 of 16.

Let op: als u bij het werken met hexadecimale getallen als uitkomst cijfers krijgt die groter zijn dan 9 moet u deze in het resultaat omvormen naar de juiste letter.

9.3.2 van 2, 8 en 16 naar 10 (methode 2)

Om een getal met basis 2, 8 of 16 naar een decimaal bestand te converteren kunnen we ook telkens eenzelfde formule toepassen. Deze luidt: "Van het gedeelte voor de komma, vermenigvuldig de eerste bit met de basis en tel de tweede bit erbij op. Onthoud de rest, vermenigvuldig deze met de basis en tel de volgende bit erbij op, enz... ". Dit klinkt nogal ingewikkeld maar is het niet. Een voorbeeld van een convertie van basis 2 naar basis 10. Stel, we hebben binaire getal **101,1011**:

1(eerste bit) * 2 + 0(tweede bit) = 2; 2 * 2 + 1(derde bit)= **5**

Het laatste getal is ons deel van het resultaat voor de komma, in dit geval **5**.

Voor het gedeelte na de komma gebruiken we de oude manier:

$1101_2 = 1 * 2^{-1} + 1 * 2^{-2} + 0 * 2^{-3} + 1 * 2^{-4}$

Dit geeft als resultaat **0,8125**, wat het gedeelte na de komma wordt. Dus onze volledige uitkomst is:

$101{,}1101_2 = 5{,}8125_{10}$

Voor de hexadecimale en octale omrekening gebruiken we dezelfde regels, enkel de basis verandert weer.

9.3.3 van 2 naar 8

Dit is één van de zeer simpele omzettingen. Splits het binaire getal in groepen van drie, beginnend bij de komma, en neem de octale waarde per groep. We hebben bijvoorbeeld het binaire getal **11111011,10101**. We berekenen de octale waarde als volgt:

011	111	011	101	010
3	**7**	**3**	**5**	**2**

Het volledige resultaat:

$11111011{,}10101_2 = 373{,}52_8$

U merkt dat het binaire getal uit 13 bits bestaat. Hiermee kunnen we geen volledige groepjes van 3 maken. In dat geval plaats je één of meer nullen voor of achter de reeks, afhankelijk van de linkse en rechtse positie t.o.v. de komma.

9.3.4 van 2 naar 16

De binair-hexadecimale convertie werkt op dezelfde manier, behalve dat we nu groepjes van vier maken en er de hexadecimale waarde van nemen. Bijvoorbeeld: **1110100010111,010101**:

0011	1010	0010	1111	0101	0100
3	**A**	**2**	**F**	**5**	**4**

Het volledige resultaat:

$$\mathbf{1110100010111{,}010101_2 = 3A2F54_{16}}$$

9.3.5 van 8 naar 2

Nu werken we in de omgekeerde volgorde. We nemen octaal getal **3363,7**:

3	3	6	3	7
011	**011**	**110**	**011**	**111**

Het volledige resultaat:

$$\mathbf{3363{,}7_8 = 11011110011{,}111_2}$$

9.3.6 van 16 naar 2

Weer op dezelfde manier. Het hexadecimale getal **3E,7** kunnen we als volgt omrekenen naar een binair getal:

3	E	7
0011	**1110**	**0111**

Het volledige resultaat:

$$\mathbf{3E7_{16} = 111110{,}0111_2}$$

Opmerking bij alle converties: voor omrekeningen die hier niet beschreven werden moet u eerst een tussenstap doen. Om van een hexadecimaal getal naar een octaal getal te gaan, berekent u, bijvoorbeeld, eerst de binaire waarde om dan van het binaire naar een octaal getal te converteren. Bij getallen zonder komma's kan u dezelfde methode's gebruiken als bij getallen met komma's.

9.4 opgaven en oplossingen

Ik maakte een tiental converteeroefeningen in MS-Word en gewoon tekst formaat. U kan dit op twee manieren downloaden:

On-line:

- het Word bestand in één Winzip bestand: converts_word.zip

- het Word bestand zelf: converts_word.doc
- het tekstbestand in één Winzip bestand: converts_tekst.zip
- het tekstbestand zelf: converts_tekst.txt

Het is mogelijk dat uw browser geconfigureerd is om een Word of tekst bestand onmiddelijk te openen, in plaats van te downloaden. In dat geval slaat u het bestand gewoon onder Word op. De Word tekst werd bedoeld om afgedrukt te worden. Ik raad u aan dit te doen, zodat de oefeningen ook beschikbaar zijn als u uw computer niet aan heeft staan.

9.5 voorstelling van negatieve getallen

Een standaardovereenkomst om negatieve getallen voor te stellen, is het plaatsten van een tekensymbool voor het getal. In het decimaal stelsel gebruiken wij hiervoor het telen "**-**". In binaire computers worden getallen voorgesteld in geheugenelementen die slechts twee toestanden kunnen aannemen: 0 of 1. Derhalve moet ook voor het teken één van deze symbolen gebruikt worden. Bij afspraak stelt de meest linkse bit van een binair getal de tekenbit voor.:

0 = Positief getal
1 = Negatief getal

Werkt de computer met registers met een lengte van, bijvoorbeeld, 8 bit, dan is

10001100 gelijk aan het getal **-12**
00010110 gelijk aan het getal **+22**

De absolute waarde van een getal wordt niet altijd voorgesteld in zijn normale vorm. Meestal wordt die bij negatieve getallen voorgesteld door zijn complement. Hierdoor kunnen aftrekkingen herleid worden tot optellingen.

Er zijn twee types van complementen, dit voor elk talstelsel. Wanneer N het grondgetal is van het talstelsel, dan onderscheidt men:

- het N complement
- het N-1-complement

9.5.1 regels voor het complementeren

In elk talstelsel gelden dezelfde regels om het complement van een getal te nemen:

N complement:

elk cijfer aftrekken van N-1, behalve het minst betekenisvolle cijfer dat van N afgetrokken wordt.

N-1-complement:

elk cijfer aftrekken van N-1.

Voorbeelden:

N = 10 (decimaal stelsel):

10 complement van 739 is **261.**
9 complement van 739 is **260.**

N = 16 (hexadecimaal stelsel):

16 complement van 3A8 is **C58.**
15 complement van 3A8 is **C57.**

N = 2 (decimaal stelsel)

2-complement van 1011 is **0101.**
1-complement van 1011 is **0100.**

Vermits computers fundamenteel slechts beschikken over de binaire voorstellingswijze, zal verder slechts het 2- en het 1-complementsysteem behandeld worden. In het binaire stelsel vereenvoudigen zich de regels voor het complementeren tot:

2-complement:

elke 0 bit wordt 1 en omgekeerd en er wordt 1 opgeteld bij de MBB (minst beduidende bit).

1-complement:

elke 0 bit wordt 1 en omgekeerd.

Voor gebroken getallen gelden dezelfde regels.

Zo is het 2-complement van $10111{,}10_2$ gelijk aan **$01000{,}10_2$**

9.5.2 overzichtstabel

Decimaal	**Teken en waarde**	**2-complement**	**1-complement**
+7	0111	0111	0111
+6	0110	0110	0110
+5	0101	0101	0101
+4	0100	0100	0100
+3	0011	0011	011
+2	0010	0010	0010
+1	0001	0001	0001
+0	0000	0000	0000
-0	1000	0000	1111
-1	1001	1111	1110
-2	1010	1110	1101
-3	1011	1101	1100
-4	1100	1100	1011
-5	1101	1011	1010

-6	1110	1010	1001
-7	1111	1001	1000
-8	n.v.t.	1000	n.v.t.

Het bereik van de verschillende voorstellingswijzen hangt af van het aantal beschikbare bits. Het is duidelijk dat door het gebruik van een tekenbit het bereik verschoven wordt. Men spreekt dan ook van een binaire offset. Aldus kunnen met behulp van 4 bitposities het getal nul, 7 positieve en 7 negatieve getallen voorgesteld worden. Met 5 bits is het bereik van +15 tot -15 (in 2-complement tot -16).

9.5.3 omzetting van gecomplementeerde getallen

Om getallen die in hun complement uitgedrukt zijn (en dus negatief zijn) in hun gewone decimale vorm om te zetten, zijn een tweetal methodes voorhanden.

- Absolute waarde inverteren en bij het 2-complement 1 MBB bijtellen:

  2-complement: 11011_2

  $0100_2 + 1_2 = 0101_2 = \mathbf{5_{10}}$

  bijgevolg is $1101_2 = \mathbf{-5_{10}}$

  1-complement: 11001_2

  $0110_2 + 1_2 = \mathbf{6_{10}}$

  bijgevolg is $11001_2 = \mathbf{-6_{10}}$

- Optelling van machten. Deze werkzijze gelijkt goed op de reeds besproken conversiemethode, alleen moet de tekenbit vervangen worden door de waarde -1 en moet bij het 1-complement 1 bijgeteld worden.

  2-complement: 11011_2

  $(-1) * 2^4 + 1 * 2^3 + 0 * 2^2 + 1 * 2^1 + 1 * 2^0 = \mathbf{-5_{10}}$

  1-complement: 11001_2

  $1 + (-1) * 2^4 + 1 * 2^3 + 0 * 2^2 + 0 * 2^1 + 1 * 2^0 = \mathbf{-6_{10}}$

9.5.4 bewerkingen

De computer voert optellingen en aftrekkingen uit met dezelfde schakelingen. Alle negatieve getallen staan afhankelijk van de machine, steeds ofwel in 1- of 2-complement.

Aftrekkingen worden herleid tot optellingen door de aftrekker te complementeren. Hierbij is het onverschillig of de aftrekker positief of negatief is, d.w.z. reeds al dan niet gecomplementeerd is.

Er is wel een onderscheid bij de behandeling van de carry bit bij 1- en 2-complement:

- 1-complement: "end around carry", d.w.z. de carry bij het resultaat optellen.

- 2-complement: de carry weglaten

Voorbeelden:

1-complement: $1111\ 1001_2$ - $1111\ 0110_2$

$1111\ 1001_2 + 0000\ 1001_2 = 1\ 0000\ 0010_2 + 1_2 =$ **$0000\ 0011_2$**

1-complement: $1111\ 1110_2$ - $0000\ 1101_2$

$1111\ 1110_2 + 0000\ 1101_2 = 1\ 1111\ 0000_2 + 1_2 =$ **$1111\ 0001_2$**

2-complement: $1111\ 1011_2$ - $1111\ 0100_2$

$1111\ 1011_2 + 0000\ 1100_2 =$ **$0000\ 0111_2$**

2-complement: $1111\ 1100_2$ - $0000\ 0100_2$

$1111\ 1100_2 + 1111\ 1100_2 =$ **$11111\ 10000_2$**

---

opdrachten

1. Probeer functies te schrijven om de berekeningsmethodes in dit hoofdstuk uit te voeren. Schrijf bijvoorbeeld een programma dat een binair getal vraagt en dit omrekent naar de decimale, hexadecimale en octale waarde. Voeg ook de mogelijkheid tot kommagetallen in en u heeft een professioneel programma. De C compiler kan zelf de hexadecimale en octale waarde weergeven, als u dit zo bepaald in uw **printf()** functie, maar het gaat hem om het opdoen van ervaring. Kommagetallen worden op die manier trouwens niet altijd juist weergegeven.

C-taal voor beginners - hoofdstuk 10

aanbevolen C stijl- en coderingsnormen

10.1 inleiding

Dit hoofdstuk bespreekt in het kort de standaardmanier waarop in C bepaalde stijlen en coderingsmethodes toegepast worden. Alhoewel niet verplicht, is het aan te raden deze te volgen wanneer u een programma schrijft. De beste en meest professionele programmeurs kwamen bijeen om deze regels te ontwikkelen. Dit wil zeggen dat een programma dat er gebruik van maakt, er professioneler uitziet en gemakkelijker te begrijpen is. Het doel van dit hoofdstuk is de leesbaarheid verbeteren, het onderhoud verminderen en boven alles de klaarheid van het programma te maximaliseren.

In feite heeft dit weinig te maken met het aanleren van de C-taal, maar ik vond het een onmisbaar concept voor ieder die op een serieuze manier programma's wil schrijven. Denkt of vindt u deze kennis niet nodig te hebben, kan u het hoofdstuk gewoon overslaan.

Opmerking: de voorbeelden uit de cursus zullen niet altijd overeenstemmen met wat ik hier vertel. Op sommige regels maak ik zelf een uitzondering, omdat ik met deze methode vertrouwd was, reeds voordat ik de stijlregels kende.

10.2 bestandsorganisatie

Een bestand bestaat uit verschillende secties die zouden gescheiden moeten worden door verschillende lege regels. Alhoewel er geen maximumlengte voor bronbestanden vastgelegd is, zijn bestanden van meer dan duizend lijnen ingewikkeld om mee te werken. De editor heeft mogelijk niet genoeg tijdelijk geheugen om het bestand te bewerken, compilaties gaan trager, enz... . Vele rijen met asterisks, als commentaar, nemen te veel tijd in beslag om ze te bekijken en worden beter vermeden. Overdrijf nooit met het gebruik van witregels en tabs, zodat een bestand met honderd regels tekst uiteindelijk niet vijfhonderd regels lang is.

10.3 bestandsnamen

Bestandsnamen worden samengesteld uit een basisnaam en een optioneel punt en achtervoegsel (extensie). Het eerste karakter van de naam zou een letter moeten zijn en alle karakters moeten kleine letters of nummers zijn. De basisnaam moet acht of minder karakters breed zijn en de extensie drie of minder karakters. Deze regels worden best toegepast op elk soort bestand.

Sommige compilers en toepassingen vereisen bepaalde extensie conventies voor namen van bestanden:

- bestandsnamen van C bronbestanden moeten eindigen met **.c**
- bestandsnamen van Assembler bronbestanden moeten eindigen met **.s**

10.4 programmabestanden

De voorgestelde volgorde voor secties in een programmabestand is als volgt:

1. Allereerst in het bestand is een proloog die vertelt wat er in het bestand voorkomt. Een omschrijving van het doel van de objecten in het bestand, de externe data declaraties of definities en andere dingen die u graag vermeld. De proloog kan optioneel auteurs, revisieinformatie en referenties bevatten.

2. De header includeringen volgen dan. Als de **#include** voor een niet voor de hand liggende reden dient, dan moet dit met een commentaarlijn verklaard worden. In de meeste gevallen worden systeem **#include** bestanden zoals **stdio.h** geincludeerd voor gebruikers **#include** bestanden.
3. Als volgt komen de **#defines** die van toepassing zijn op het ganse bestand. Plaats constante macro's voor functie macro's. Dan komen de **enums**.
4. Daarna komen de globale (externe) data declaraties, gewoonlijk in de volgorde: extern, niet-statisch, globaal, statisch globaal.
5. De functies komen als laatste en zouden in een logische volgorde geplaatst moeten worden.

10.5 header bestanden

Header bestanden zijn bestanden die in andere bestanden worden ingevoegd door de compiler, voorafgaand aan de compilatie door de C preprocessor. Sommige, zoals **stdio.h**, worden gedefinieerd op systeemniveau en moeten geincludeerd worden door elk bestand dat gebruikt maakt van de standaard I/O bibliotheek. Vermijd privé header bestandsnamen die hetzelfde zijn als bibliotheek header bestandsnamen. Het statement

```
#include "math.h"
```

zal het standaard **math.h** dat bij uw compiler zit laden, als het bedoelde niet gevonden wordt in de huidige directory. Indien u dit toch zo wilt, moet u dat met commentaar uitleggen.

10.6 commentaar

"Als code en commentaar het niet met elkaar eens zijn, zijn beide waarschijnlijk fout."-- Norm Schryer

Commentaar in een programma moet omschrijven wat er gebeurt, hoe het gebeurt, wat de parameters betekenen, welke globalen gebruikt worden en aangepast zijn en eventuele beperkingen of problemen. Vermijd commentaar dat duidelijk is uit de code, omdat dit nogal vlug oud wordt. Commentaar dat het niet eens is met de code is van negatieve waarde. Korte commentaar zou van het *wat* type moeten zijn, zoals "bereken som", i.p.v. het *hoe* type, zoals "som van de waarden gedeeld door n en vermenigvuldigd met y". C is anders dan Assembleertaal; commentaar op de eerste tien lijnen van een programma dat verteld wat het in het algemeen doet, is meestal meer gebruikelijk dan een commentaarlijn op elke regel om de micrologica te beschrijven.

Gebruik commentaar om "lelijke" code te rechtvaardigen. De uitleg zou moeten zijn dat er iets slechts zal gebeuren als er gewone code gebruikt wordt. Code gewoon sneller maken is niet genoeg; de verbetering moet getoond worden. Leg in het commentaar uit waarom u de ongewone methode gebruikt en waarom het een goede oplossing is.

Commentaar om datastructuren, algorithmen, enz... te beschrijven, zou in blokvorm met het openende ***/**** in kolom 1-2, een ***** in kolom 2 voor elke lijn met commentaartekst en het sluitende ****/*** in kolom 2-3 moeten staan. Een alternatief is om ****** in kolom 1-2 te plaatsen en het sluitende ****/*** ook in 1-2.

```
/*
 *        Hier staat een blokcommentaar.
 *        De commentaartekst zou tabs en witruimtes moeten
gebruiken.
 *        Het openende slash-ster en sluitende ster-slash staan
alleen op één lijn.
 */
/*
** Alternatief formaat voor blokcommentaar.
```

```
*/
```

Blokcommentaar binnen een functie is passend en zou op dezelfde positie moeten staan als de code waarover de commentaar gaat. Commentaar dat alleen op een lijn staat zou de tabpositie moeten gebruiken van de volgende code.

```
do
{
    c = fgets(eenwoord, 100, fp1);
    /* Controleren of het bestand op zijn einde is. */
    if (c != NULL)
        printf("%s", eenwoord);
} while (c != NULL);
```

Zeer korte commentaar mag op dezelfde lijn staan als de code die ze beschrijven en er zou een zekere witruimte tussen moeten staan. Als er meer dan één commentaar voorkomt in eenzelfde blok code, dan moeten deze op dezelfde tabpositie te staan.

```
do
{
    c = fgets(eenwoord, 100, fp1);
    if (c != NULL)      /* Controleren of het bestand op zijn
einde is. */
        printf("%s", eenwoord);
} while (c != NULL);                    /* Doorgaan totdat c NULL
is */
```

Het is mogelijk dat uw browser dit niet helemaal juist weergeeft, maar de uitleg is duidelijk. U kan de voorbeeldprogramma's bij deze cursus bekijken voor het juiste gebruik van commentaar.

10.7 declaraties

Bij een lange lijst declaraties van hetzelfde type variabele, zoals

```
char string1[20], string2[50], string3[320], string4[5],
string5[10], string6[20];
```

worden de variabelen beter elk op een lijn geplaatst:

```
char string1[20],
     string2[50],
     string3[320],
     string4[5],
     string5[10],
     string6[20];
```

In Europa is deze methode niet echt geliefd. Het is dan ook weinig moeite om het bovenstaande als volgt te declareren en de leesbaarheid te verhogen.

```
char string1[20];
char string2[50];
char string3[320];
char string4[5];
char string5[10];
```

```
char string6[20];
```

10.8 witruimte

Gebruik verticale en horizontale witruimte overvloedig. Inspringingen moeten de blokstructuur van de code weergeven. Er zouden, bijvoorbeeld, tenminste twee witregels moeten staan tussen het einde van een functie en het commentaar van de volgende.

Een lange rij van conditionele statements kan u beter over afzonderlijke lijnen verdelen:

```
if(cijfer1 <  cijfer2 && cijfer1 > cijfer3 && cijfer2 > cijfer4)
{...
```

kan u misschien beter schrijven als

```
if(cijfer1 <  cijfer2
        && cijfer1 > cijfer3
        && cijfer2 > cijfer4)
{       ....
```

Op dezelfde manier kunnen **for()** loops gescheiden worden over meerdere regels:

```
for(i = 0, y = 100;
        y < 50;
        i++, y + 2 * x)
{       ....
```

Andere complexe operaties, zoals degene die de **? :** gebruiken, kan u ook best scheiden:

```
c = (a == b)
            ? d + f(A)
            : f(b) - d;
```

Het is wel aan te raden om de **? :** constructie zoveel mogelijk te vermijden. Sleutelwoorden die gevolgd worden door een expressie tussen ronde haken zouden gevolgd moeten worden door een spatie (behalve de **sizeof** operator). Blanco's zouden ook na komma's moeten komen in de argumentenlijst om de argumenten visueel beter te kunnen onderscheiden. Macrodefinities met argumenten mogen geen witruimte hebben tussen de naam en het linkerhaakje, anders zal de C preprocessor het niet herkennen.

10.9 compound statements

In deze cursus werden voorwaarden met meer dan één opdracht als volgt gecodeerd:

```
if(getal == 250)
{
    printf("Het getal is 250\n");
    printf("Dus is het niet 300\n");
}
```

Dit is de Amerikaanse stijl:

```
if(getal == 250) {
```

```
        printf("Het getal is 250\n");
        printf("Dus is het niet 300\n");
    }
```

Als u een heleboel geneste compound statements gebruikt in een programma is de tweede methode niet echt aan te raden. Indien bij een **if() else** statement één van de twee secties samengesteld is, zouden zowel de **if()** als **else** voorwaarde(n) tussen accolades moeten geplaatst worden [ref. C FAQ - Sectie 3, Punt 4]:

```
if(getal == 250)
{
    printf("Het getal is 250\n");
}
else
{
    printf("Het getal is dus niet 300\n");
    printf("Ook is het kleiner dan 400\n");
}
```

De haakjes zijn ook nodig bij **if() if() else** sequenties, zonder een tweede **else**. Anders zal de preprocesser dit niet juist coderen voor de uiteindelijke compilatie:

```
if(getal <= 250)
{
    if(getal == 250)
    {
        printf("Het getal is 250\n");
    }
}
else
{
    printf("Het getal is kleiner dan 250\n");
}
```

Dit is een mooi voorbeeld om duidelijk te maken dat de Amerikaanse manier van compound statements nogal ingewikkeld kan werken. Het laatste voorbeeld zou er dan zo uitzien:

```
if(getal <= 250) {
    if(getal == 250) {
        printf("Het getal is 250\n");
    }
}
else {
    printf("Het getal is kleiner dan 250\n");
}
```

10.10 operators

Als u denkt dat een expressie moeilijk te lezen zal zijn, overweeg dan om ze op te breken over verschillende lijnen. De scheiding maken bij de laagste-prioriteit operator naast de onderbreking is het beste. Omdat C een paar vreemde regels heeft, kan u expressies met gemixte operators beter tussen haken plaatsten. Het gebruik van extra ronde haken kan in veel gevallen, bij elk soort expressie fouten vermijden. Overdreven gebruik, echter, maakt een lijn moelijker, aangezien mensen niet goed zijn in het met elkaar vergelijken van linkse en rechtse ronde haken.

10.11 tabs

In de voorbeelden in deze cursus en de programma's die erbij zitten gebruik ik steeds tabs om de leesbaarheid te verhogen. Zolang u op zichzelf werkt is dat geen probleem, maar in groepverband moet u opletten. Een tab kan namelijk op verschillende manieren ingesteld staan, bijvoorbeeld op positie 1, 2, 3, 4 of 8. Zorg voor een goede afspraak omtrent het gebruik van inspringingen. Tabs kunnen ook vreemde resultaten geven bij het afdrukken.

10.12 geheugenallocatie

Een paar aanwijzingen:

- Initialiseer alles (vooral pointers) direct bij de declaratie op een bepaalde waarde, desnoods op NULL. Dus in de plaats van:

  ```
  int *mijn_var;
  ```

  is het volgende veiliger:

  ```
  int *mijn_var = NULL;
  ```

- Gebruik een pointer die je vrij geeft daarna nooit meer. Stel dat u een pointer naar structure **X** met daarin een pointer **Y** heeft. Codeert u dan het volgende dan geeft dit zonder twijfel run-time en compilerfouten:

  ```
  free(X);
  free(X->Y);
  ```

  Het laatste kan niet meer omdat het geheugen van **X** al is vrijgegeven. Beide regels moeten dus omgedraaid worden.

10.13 conclusie

Een standaard werd gepresenteerd voor de C programmeerstijl. De meest belangrijke punten zijn:

- Het juiste gebruik van witruimte en commentaar, zodat de structuur van het programma evident is vanuit de layout van de code.
- In gedachten houdend dat u of iemand anders waarschijnlijk gevraagd zal worden om code aan te passen of om het op verschillende machines te laten lopen, ooit in de toekomst. Maak uw code zo dat ze overdraagbaar is naar andere types van machines. Localiseer optimalisaties omdat deze deze dikwijls verwarrend zijn.
- Veel stijlkeuzes zijn betwistbaar. Een stijl gebruiken die consequent is (in het bijzonder met een groepstandaard) is belangrijker dan het volgen van de absolute stijlregels. Het mixen van stijlen is erger dan één stijl, hoe slecht ook.

10.14 tenslotte

Zoals met elke standaard, moet deze gevolgd worden als het nuttig is. Heeft u problemen met het toepassen van de aanbevelingen in dit hoofdstuk, laat ze dan niet links liggen. Spreek met uw locale guru of ervaren programmeur of leerkracht. De opinies in dit hoofdstuk, of zelfs de ganse gids, zijn niet de opinies van alle auteurs. Sommige schrijvers over de C-taal denken nu éénmaal anders over hoe we zouden moeten programmeren. Alles blijft dan ook een kwestie van eigen smaak.

De belangrijkste regel bij het schrijven van programma's:

"een degelijk programma wordt eerst op papier gezet, daarna op de computer"

Deze lijfspreuk zal ongetwijfeld een hele reeks problemen vermijden. U zal waarschijnlijk denken dat de tussenstap niet nodig is, maar door het programma gewoon in uw compiler beginnen te typen, zal u dingen over het hoofd zien. Op papier ziet iets er meer overzichtelijk uit.

Ik wil u in geen geval verplichten om ook maar iets in deze cursus als gedwongen leerstof te ervaren. Mijn enige doel bij het schrijven ervan was een degelijk, duidelijk en vooral uitgebreid en zo weinig mogelijk ingewikkeld werk te leveren. Denkt u dat ik hierin geslaagd ben en heeft u iets gehad aan deze cursus, dan zou ik het erg op prijs stellen als u me dat laat weten. Indien u constructieve opmerkingen heeft, die kunnen bijdragen tot het verbeteren van deze cursus, dan zou ik ook willen dat u me dat vertelt. Wacht niet met reageren als u vindt dat ik iets op een verkeerde manier uitlegde of fouten gemaakt heb. Een werk bestaande uit zoveel tekst zal ongetwijfeld onregelmatigheden bevatten, zelfs na een paar grondige nazichten. Het meest waarschijnlijk zal u spellingsfouten ontdekken, omdat ik geen spellingstest gebruikte. Daarom werk ik ook in versies. Geleidelijk aan zullen er nieuwere en betere versies verschijnen. Dan begrijpt u ook meteen waarom ik uw inbreng verwelkom. Raadpleeg de inleiding voor informatie om mij te contacteren.

C-taal voor beginners - hoofdstuk 11

structures, unions en typedefs

11.1 inleiding

[ref. C FAQ - Sectie 2]

Dit hoofdstuk is grotendeels een vertaling van het gelijkaardige hoofdstuk uit "Programming in C". Een Engelse C-gids door Peter Burden. De auteur gaf mij de wettelijke toestemming tot het vertalen en publiceren van dit materiaal. Ik probeer me telkens zoveel mogelijk aan de Nederlandse taal te houden, maar dit hoofdstuk bevat een aantal begrippen die zich niet laten vertalen.

11.2 structures

Een C structure is een samengesteld (compound) data object. Zulk een object bestaat uit een collectie van data objecten van, mogelijk, verschillende types. Het kan bekeken worden als een privé of gebruikers-gedefinieerd data type, apart van de standaard data types die voorzien worden in de programmeertaal C.

C voorziet mogelijkheden om zulke objecten te declareren, d.w.z. het definiëren van hun interne structuur via een template (vorm) en het declareren van een tag (label, etiket) om geassociëerd te worden met zulke obecten zodat het niet nodig is om de definitie telkens te herhalen. Gegeven de declaratie, de structuur en de bijbehorende tag, is het enkel nodig om de tag te gebruiken bij het declareren van eigenlijke voorkomens van structures. Een simpel voorbeeld van een structure declaratie en definitie wordt hier getoond. Het sleutelwoord **struct** wordt gebruikt bij het declareren en definiëren van structures.

```
struct datum     /* de tag */
{
  /* begin van het template */
  int dag;       /* een lid */
  int maand;     /* een lid */
  int jaar;      /* een lid */
  char naam;     /* een lid */
} datums[MAXDAT], vandaag, *volgende;   /* gevallen */
```

Hierbij is **datums** een groepering van gevallen (instances) van **struct datum**, **vandaag** is een simpel geval van **struct datum** en **volgende** is een pointer naar **struct datum**. Eénmaal de code hierboven is voorgekomen in het programma, kunnen verdere gevallen van deze structure als volgt gedeclareerd worden:

```
struct datum mijn_verjaardag;
struct datum einde_termijn;
```

Tag namen vallen onder dezelfde regels als namen van variabelen, maar behoren tot een apart "universum". Dit heeft tot gevolg dat een tag en een variabele in eenzelfde programma eenzelfde naam kunnen hebben. Het is daarom legaal als u het volgende schrijft:

```
struct datum datum;
```

Het template vertelt de compiler hoe de structure in het geheugen geplaatst wordt en geeft details door over de lidnamen (members). Een template reserveert geen gevallen van de structure, het laat de

compiler enkel weten wat het betekent. Declaraties van leden van structures vallen onder dezelfde syntax als gewone variabel declaraties.

Deze vreemde en verwarrende code is perfect legaal:

```
struct u
{
  int u;
  int v;
} v;
struct v
{
  char v;
  char u;
} u;
```

Pre-ANSI compilers hadden dikwijls meer striktere regels wat betreft de afzonderlijke benamingen.

Leden van structures kunnen van elk data type zijn, inclusief andere structures, groeperingen en pointers, alsook pointers naar structures en pointers naar functies. Een structure mag, voor duidelijke redenen, geen gevallen van zichzelf bevatten, maar wel pointers naar gevallen van zichzelf.

Structures mogen geïnitialiseerd worden op dezelfde manier als groeperingen, gebruik makend van initialisers. Bijvoorbeeld:

```
struct datum Kerstmis = {25, 12, 1998, 3};
```

Naar individuele leden van een structure kan als volgt worden verwezen:

```
datums[k].jaar
vandaag.maand
(*volgende).dag
```

De **.** (punt) operator selecteert een bepaald lid van een structure. Het valt onder dezelfde voorrangsregels als ( ) en [ ], welke voorgaan op de unaire of binaire operator. Evaluaties gebeuren van links naar rechts. De eigenlijke syntax is:

```
structure naam.lidnaam
```

Het is vereist dat het eerste component een structure is dus

```
datums.jaar[k]   /* FOUT */
```

zou verkeerd zijn omdat **datums** een groepering is van structures en **jaar** is dat niet, zo zou ook

```
*volgende.dag    /* FOUT */
```

verkeerd zijn omdat de **.** operator een hogere prioriteit heeft dan de ***** operator. Dit foutief gebruik probeert **volgende** als een structure te gebruiken en toegang te verkrijgen tot het object waarvan het adres zich bevindt in **dag**.

De correcte manier om te verwijzen naar een lid van een structure waarvan het adres is gegeven is typisch:

```
(*volgende).dag
```

Dit wordt zo frequent gebruikt dat er een speciale operator voorzien werd. De alternatieve syntax is

```
volgende->dag
```

De operator bestaat uit het minteken en het groter dan teken. De voorrangsregels zijn dezelfde als die van de punt operator.

Structures kunnen toegekend worden, gebruikt worden als normale functieparameters en teruggegeven worden als functionele waarden. Zulke operaties laten de compiler sequenties genereren met laad en opslag instructies die mogelijk efficiëntie problemen opleveren. C programmeurs die bezorgd zijn over de snelheid van een programma zullen deze dingen meestal vermijden en enkel met pointers naar functies werken.

Er zijn een paar operaties die toegepast kunnen worden op structures. De enige geldige operators die geassocieert mogen worden met structures zijn **=** (toekenning) en **&** (neem het adres). Het is niet mogelijk om structures op gelijkheid te testen met **==**, noch is het mogelijk om er rekenkunde op toe te passen. Zo'n operaties moeten expliciet gecodeerd worden via bewerkingen op de leden van de structure.

11.2.1 simpele voorbeelden van structures

De volgende simpele strucuture declaraties kan u tegenkomen in een grafische omgeving.

```
struct punt
{
  double x;
  double y;
};

struct cirkel
{
  double straal;
  struct punt cen;
};
```

Gegeven bovenstaande declaraties kan volgende functie geschreven worden voor het berekenen van de oppervlakte.

```
double opp(struct cirkel cirkel)
{
  return (PI * cirkel.straal * cirkel.straal);
}
```

Dit voorbeeld gaat ervan uit dat PI op de juiste manier gedefinieerd werd (met **#define**). Om te bepalen of een gegeven punt binnen de cirkel ligt kan u deze functie gebruiken:

```
incirkel(struct punt punt, struct cirkel cirkel)
{
  double dx,dy;
  dx = punt.x - cirkel.cen.x;
  dy = punt.y - cirkel.cen.y;
  return (dx*dx+dy*dy <= cirkel.straal*cirkel.straal);
}
```

Verdere grafische structure declaraties zoals

```
struct lijn
{
  struct punt start;
  struct punt einde;
};
```

en

```
struct driehoek
{
  struct punt pt[3];
};
```

kunnen gebruikt worden op een natuurlijke en nuttige manier.

11.2.2 in de praktijk

Dit hoofdstuk kan nogal ingewikkeld lijken, wat het ook wel is. Toch mag u zich niet laten intimideren. De grote kracht van structures zit hem in het verwerken van databestanden. Als voorbeeld wordt in deze paragraaf een simpele toepassing besproken.

Telkens iemand deze C-gids downloadt van mijn site, wordt er een internet-formulier ingevuld. Deze gegevens komen in mijn elektronische postbus terecht en alle data wordt in een database geplaatst. Vanuit de database kan er een textbestand gegenereerd worden met de velden gescheiden door een puntkomma, bijvoorbeeld:

```
Smit;Jan;jsmit@village.uunet.bet#http://jsmit@village.uunet.bet#;1;17/8/1998 0:00:00
```

Dit is de manier waarop elk record in het tekstbestand voorkomt. Alle recordes staan onder elkaar, zonder lege regels. Het eerste veld is de voornaam, het tweede de achternaam. Het volgende veld is het emailadres (u merkt dat het er ook nog eens in http formaat achter staat). Het eencijferig getal dat volgt is ofwel 1 of 0, naargelang de persoon updates wil ontvangen of niet. Dit veld zullen we niet gebruiken, maar het staat wel in het record. Dan volgt de datum, waarvan we enkel het eerste deel nodig hebben (dus niet de tijdsweergave).

We hebben als invoerbestand dus een verzameling van records, onder elkaar geplaatst. We willen nu de informatie per record extracteren en als volgt op het scherm plaatsen. Het cijfer bovenaan is het recordnummer.

```
 ---
|001|
 ---
NAAM: Smit
VOORNAAM: Jan
EMAIL: jsmit@village.uunet.bet
DATUM: 17/8/1998
```

De eerste lijnen van ons programma zijn natuurlijk bestemd voor de header includes:

```
#include<stdio.h>        /* voor schermuitvoer enz... */
#include<stdlib.h>       /* voor het gebruiken van EXIT_FAILURE
enz... */
```

```
#include<string.h>      /* voor het toepassen van stringfuncties
*/
```

Daarna declareren we onze structure, waarvan de leden de veldinhouden zullen bevatten.

```
struct downloaden
{
  char naam[50];
  char voornaam[50];
  char email[50];
  char datum[50];
} c_tutor;
```

Nu komen we aan de **main()**, waarin we om te beginnen de nodige declaraties doen:

```
int main(void)
{
FILE *in, *uit;
char lijn_uit_bestand[150];     /* volledig record uit het
tekstbestand */
char rommel[100];       /* om onnodige delen uit het record te
filteren */
int record_nr = 1;      /* teller van het aantal records */
```

De bestandspointer **in** zal voor het tekstbestand met de records gebruikt worden. We willen ook een apart tekstbestand aanmaken waarin alle emails bijgehouden worden, bij wijze van mailinglist. Volgend stuk code zou duidelijk moeten zijn (als u dit te ingewikkeld of te lelijk vindt kan u het natuurlijk ook wat uitgebreider coderen):

```
if((in = fopen("c_gids_downloaden.txt", "r")) == NULL) {
  printf("Inputbestand niet geopend!!\n");
  getchar();
  return EXIT_FAILURE;
}
if((uit = fopen("mailing_list.txt", "w")) == NULL) {
  printf("Uitvoerbestand niet geopend!!\n");
  getchar();
  return EXIT_FAILURE;
}
```

Eerst openen we dus het bestand met de records, daarna maken we een bestand aan om de emails in te plaatsen.

Het hoofddeel van ons programma filtert telkens een record en plaatst de gewenste gegevens in de juiste leden van de structure. Voor het filteren gebruiken we de **string.h** functie **strtok()**. Deze wordt volledig besproken in het hoofdstuk over de 15 ANSI-C headers. Ook **feof()** is nieuw. Dit test gewoon of een bestand op zijn einde is (een beetje zoals wij vroeger op NULL testten).

```
do
{
  fgets(lijn_uit_bestand, 150, in);
          if(!feof(in)) {

          /* recordnummer tonen */
          printf(" ---\n");
```

```
          printf("|%03d|\n", record_nr);
          printf(" ---\n");
          /* filterproces */
          strcpy(c_tutor.naam, (strtok(lijn_uit_bestand, ";")));
          strcpy(c_tutor.voornaam, (strtok(NULL, ";")));
          strcpy(c_tutor.email, (strtok(NULL, "#")));
          strcpy(rommel, (strtok(NULL, ";")));
          strcpy(rommel, (strtok(NULL, ";")));
          strcpy(c_tutor.datum, (strtok(NULL, " ")));
          /* de gefilterde gegevens op het scherm tonen */
          printf("NAAM: %s\n",c_tutor.naam);
          printf("VOORNAAM: %s\n", c_tutor.voornaam);
          printf("EMAIL: %s\n", c_tutor.email);
          printf("DATUM: %s\n\n", c_tutor.datum);
          /* wachten op enter alvorens het volgende record te
verwerken */
          printf("        <ENTER> VOOR HET VOLGENDE RECORD\n");
          getchar();
          /* teller verhogen */
          record_nr++;
          /* het emailadres naar het uitvoerbestand schrijven */
          fprintf(uit, "%s\n", c_tutor.email);
  }
/* herhalen tot er geen records meer zijn */
} while(!feof(in));
```

Om het programma te eindigen sluiten we de bestanden en tonen we een EOF boodschap op het scherm. We wachten dan op enter alvorens af te sluiten.

```
fclose(in);
fclose(uit);
printf("         END OF FILE! - <ENTER>\n");
getchar();
return 0;
}
```

Ons programma is nu volledig. Mits enige aanpassingen kan dit uitgebreid worden tot een zeer functionele toepassing door bijvoorbeeld een zoekfunctie in te bouwen enz... waarbij u beter zelfgemaakte functies gebruikt. Ter vervollediging het programma in één stuk:

```
/* 11.2.2.1.c */
#include<stdio.h>       /* voor schermuitvoer enz... */
#include<stdlib.h>      /* voor het gebruiken van EXIT_FAILURE
enz... */
#include<string.h>      /* voor het toepassen van stringfuncties
*/
struct downloaden
{
  char naam[50];
  char voornaam[50];
  char email[50];
  char datum[50];
} c_tutor;
int main(void)
{
FILE *in, *uit;
```

```
char lijn_uit_bestand[150];     /* volledig record uit het
tekstbestand */
char rommel[100];       /* om onnodige delen uit het record te
filteren */
int record_nr = 1;      /* teller van het aantal records */
if((in = fopen("c_gids_downloaden.txt", "r")) == NULL) {
  printf("Inputbestand niet geopend!!\n");
  getchar();
  return EXIT_FAILURE;
}
if((uit = fopen("mailing_list.txt", "w")) == NULL) {
  printf("Uitvoerbestand niet geopend!!\n");
  getchar();
  return EXIT_FAILURE;
}
do
{
  fgets(lijn_uit_bestand, 150, in);
          if(!feof(in)) {

          /* recordnummer tonen */
          printf(" ---\n");
          printf("|%03d|\n", record_nr);
          printf(" ---\n");
          /* filterproces */
          strcpy(c_tutor.naam, (strtok(lijn_uit_bestand, ";")));
          strcpy(c_tutor.voornaam, (strtok(NULL, ";")));
          strcpy(c_tutor.email, (strtok(NULL, "#")));
          strcpy(rommel, (strtok(NULL, ";")));
          strcpy(rommel, (strtok(NULL, ";")));
          strcpy(c_tutor.datum, (strtok(NULL, " ")));
          /* de gefilterde gegevens op het scherm tonen */
          printf("NAAM: %s\n",c_tutor.naam);
          printf("VOORNAAM: %s\n", c_tutor.voornaam);
          printf("EMAIL: %s\n", c_tutor.email);
          printf("DATUM: %s\n\n", c_tutor.datum);
          /* wachten op enter alvorens het volgende record te
verwerken */
          printf("        <ENTER> VOOR HET VOLGENDE RECORD\n");
          getchar();
          /* teller verhogen */
          record_nr++;
          /* het emailadres naar het uitvoerbestand schrijven */
          fprintf(uit, "%s\n", c_tutor.email);
  }
/* herhalen tot er geen records meer zijn */
} while(!feof(in));
fclose(in);
fclose(uit);
printf("         END OF FILE! - <ENTER>\n");
getchar();
return 0;
}
```

11.3 unions

De leden van een structure worden in het geheugen geplaatst, meestal de ene na de andere. Een union is synthetisch identiek, behalve dat het sleutelwoord **union** wordt gebruikt in plaats van **struct**. Het verschil tussen een union en een structure is dat de leden van een union elkaar overlappen. In een union beginnen alle leden bij dezelfde geheugenlocatie, bij een structure niet. Unionleden kunnen op zichzelf structures zijn en structureleden kunnen op zichzelf unions zijn.

Een typische toepassing wordt geïllustreerd door het volgende code fragment. Als data, in de vorm van floating point getallen, opgeslagen wordt in een bestand dan is dat bestand moeilijk te lezen, aangezien alle standaard C bestandsoperaties karakter per karakter te werk gaan. Het fragment beneden omzeilt die moeilijkheid door gebruik te maken van een union waarvan de twee leden bestaan uit een karakter array en een floating point nummer. Hier wordt verondersteld dat een floating point nummer 8 karakters bezet.

```
union ibf
{
  char c[8];
  double x;
} ibf;
double waarden[...];
for(i=0; i<8; i++)
  ibf.c[i] = getc(ifp); /* ifp is een pointer naar een geopend
bestand */
waarden[j] = ibf.x;
```

11.4 typedefs

Het gebruik van **typedef** is een simpele macro-achtige mogelijkheid voor het declareren van nieuwe namen voor data types, inclusief gebruikers-gedefinieerde data types. Typische voorbeelden:

```
typedef long GROOTINT;
typedef double ZEERGROOT;
typedef struct punt
{
  double x;
  double y;
  PUNT;
}
```

Met bovenstaand gegeven zou het mogelijk zijn om volgende declaraties te schrijven:

```
PUNT a, b, c;
ZEERGROOT a1, a2;
```

Dit creëert geen nieuwe types, het hernoemt alleen bestaande types. De interpretatie van typedefs wordt gedaan door de compiler (niet door de pre-processor) zodat controles over de correctheid ervan mogelijk zijn.

11.5 gelinkte lijsten

Een linked list (gelinkte lijst) is een data structure die velden voor het opslaan van data bevat alsook één of meer pointer velden. Zulk een data structure noemt men een node. De pointerveld(en) wijzen naar andere nodes in de lijst. Door nodes op deze manier met pointer aan elkaar te hangen, kunnen we de lijst traverseren door de pointers in de nodes te volgen. [ref. C FAQ - Sectie 1, Punt 14]

11.5.1 enkel gelinkte lijsten

Een enkel gelinkte lijst vereist dat elk informatie item een link naar het volgende element in de lijst bevat. Elk data item bestaat gewoonlijk uit een strutucre die informatievelden bevat samen met een link pointer.

Eigenlijk zijn er twee manieren om een enkel gelinkte lijst op te bouwen. De eerste is simpelweg door elk nieuw item op het einde van de lijst te plaatsen. De andere is om items toe te voegen op specifieke plaatsen in de lijst, bijvoorbeeld in oplopende volgorde. Hoe de lijst opgebouwd wordt bepaalt de manier waarop de opslagfunctie wordt gecodeerd.

Laten we beginnen met een zeer eenvoudig geval van het creëeren van een gelinkte lijst door items aan het einde toe te voegen. De items opgeslagen in een gelinkte lijst bestaan meestal uit structures omdat elk item een link naar het volgende item in de lijst alsook naar de data op zich moet bevatten. Daarom zullen we een structure moeten definiëren die gebruikt zal worden in de voorbeelden die volgen. Aangezien mailing lijsten gewoonlijk opgeslagen worden in een gelinkte lijst, is een adres structure geen slechte keuze. De data structure voor elk element in de mailing lijst wordt hier gedefinieerd:

```
struct adres
{
  char naam[40];
  char straat[40];
  char stad[20];
  char staat[3];
  char zip[11];
  struct adres *volgende;
}info;
```

De **slopslaan()** functie, als volgende getoond, bouwt een enkel gelinkte lijst door elk nieuw karakter op het einde te plaatsen. Het moet doorgegeven worden als een pointer naar een structure van type **adres** alsook als een pointer naar het laatste element in de lijst. Als de lijst leeg is, dan moet de pointer naar het laatste element in de lijst null zijn.

```
void slopslaan(struct adres *i, struct adres **laatste)
{
  if(!*laatste)
          *laatste = i;  /* eerste item in de lijst */
  else (*laatste) -> volgende = i;
  i -> volgende = NULL;
  *laatste = i;
}
```

Alhoewel we de lijst gecreëerd met de functie **slopslaan()** kunnen sorteren als een afzonderlijke operatie, is het gemakkelijker om de lijst te sorteren tijdens het bouwen ervan door elk nieuw item op de juiste volgorde in de ketting in te voegen. Als de lijst reeds gesorteerd is, dan zou het voordelig zijn om hem gesorteerd te houden door nieuwe items op de juiste locatie in te voegen. Dit kan gedaan worden door het sequentieel scannen van de lijst totdat de juiste locatie werd gevonden en het nieuwe adres in te voegen op dat punt en indien nodig ook de links te herschikken.

11.5.2 dubbel gelinkte lijsten

Dubbel gelinkte lijsten bestaan uit data en links naar het volgende item alsook naar het voorgaande item.

Er zijn drie manieren om een nieuw element in te voegen in een dubbel gelinkte lijst: een nieuw eerste element invoegen, in het midden invoegen en als laatste element invoegen.

Een dubbel gelinkte lijst bouwen lijkt op het bouwen van een enkel gelinkte lijst behalve dat er twee links onderhouden worden. Daarom moet de structure ruimte hebben voor beide links. We gebruiken het vorige voorbeeld en passen het als volgt aan:

```
struct adres
{
  char naam[40];
  char straat[40];
  char stad[20];
  char staat[3];
  char zip[11];
  struct adres *volgende;
  struct adres *vorige;
} info;
```

Gebruik makend van **adres** als het basis data item bouwt de volgende functie een dubbel gelinkte lijst:

```
void dlopslaan(struct adres *i, struct adres **laatste)
{
  if(!*laatste)
          *laatste = i;  /* eerste item in de lijst */
  else (*laatste) -> volgende = i;
  i -> volgende = NULL;
  i -> vorige = *laatste;
  *laatste = i;
}
```

De functie **dlopslaan()** plaatst elke nieuwe toevoeging aan het einde van de lijst. De functie moet aangeroepen worden met een pointer naar de data die opgeslagen moet worden alsook een pointer naar het einde van de lijst, wat null moet zijn bij de eerste aanroep.

C-taal voor beginners - hoofdstuk 12

dynamische geheugenallocatie

12.1 inleiding

[ref. C FAQ - Sectie 7]

Als we precies weten welke opslagelementen we nodig hebben wanneer we een programma schrijven, dan kunnen we ze simpelweg allemaal definiëren en gebruiken. Er zijn echter momenten dat we niet precies weten hoeveel opslag we nodig hebben om een programma uit te voeren, dus alloceren we zoveel als we nodig hebben terwijl het programma uitgevoerd wordt. Wanneer we klaar zijn met het gebruiken van het dynamisch gealloceerd geheugen, dan dealloceren we het voor hergebruik door een ander deel van het programma. Voor het alloceren van geheugen zijn een paar standaardfuncties beschikbaar, uit de **stdlib.h** header (zie ook hoofdstuk 8 en uw compilerdocumentatie):

12.2 malloc

De functie **malloc** alloceert het gevraagde geheugen en geeft een pointer naar dat geheugen terug. De waarde van de ruimte is onbeslist. Bij succes wordt een pointer tot de gevraagde ruimte teruggegeven. Bij mislukking wordt er een null pointer teruggegeven.

Voorbeeld:

```
/* 12.2.1.c */
#include <stdio.h>
#include <string.h>
#include<stdlib.h>

int main(void)
{
    char *str;

    /* alloceer geheugen voor een string */
    str = (char *) malloc(10);

    if(str == NULL)
    {
       printf("Niet genoeg geheugen!\n");
       return EXIT_FAILURE;
    }

    /* kopieer "hallo" naar de string */
    strcpy(str, "Hallo");

    /* string weergeven */
    printf("String is %s\n", str);

    /* geheugen leegmaken */

    free(str);

    return 0;
}
```

12.3 free

In het voorbeeldprogramma ziet u ook duidelijk waarvoor de functie **free** gebruikt wordt, namelijk om het vooraf gealloceerde geheugen terug vrij te maken voor ander gebruik. U maakt er best een goede gewoonte van om dit telkens op het einde van het programma te doen als u een allocatie functie gebruikte, net zoals u ook altijd alle geopende bestanden moet sluiten. Deze functie levert geen return.

12.4 calloc

De functie **calloc** alloceert het gevraagde geheugen en geeft een pointer naar dat geheugen terug. De ruimte wordt geïnitialiseerd tot allemaal nul bits. Bij succes wordt een pointer tot de gevraagde ruimte teruggegeven. Bij mislukking wordt er een null pointer teruggegeven. **calloc** alloceert het hoofdgeheugen en neemt toegang tot de heap.

Voorbeeld:

```
/* 12.4.1.c */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main(void)
{
    char *str = NULL;

    str = (char *) calloc(10, sizeof(char));

    strcpy(str, "Hallo");

    printf("String is %s\n", str);

    free(str);

    return 0;
}
```

Hier alloceert de functie een geheugenblok van **10 * sizeof(char)**, u moet de **10** namelijk zien als het aantal items en **sizeof(char)** als de gewenste grootte (**calloc** vereist dus een extra argument ten opzichte van **malloc**). De declaratie van deze functie is namelijk:

```
void *calloc(size_t nitems, size_t grootte);
```

12.5 realloc

Declaratie:

```
void *realloc(void *ptr, size_t grootte);
```

Pobeert het geheugenblok waar *ptr* naar wijst en dat voordien werd gealloceerd met een aanroep tot **malloc** of **calloc** van grootte te veranderen. De inhoud waar *ptr* naar wijst blijft onverandert. Als de waarde van *grootte* groter is dan de vorige grootte van het blok, dan hebben de extra bytes een onbesliste waarde. Als de waarde *grootte* kleiner is dan de vorige grootte van het blok, dan worden de overige bytes aan het einde van het blok geledigd. Indien *ptr* null is, dan gedraagt deze zich als **malloc**. Wijst *ptr* naar een geheugenblok dat niet gealloceerd werd met **calloc** of **malloc**, of het is een ruimte die werd

gedealloceerd, dan is het resultaat niet gedefinieerd. Wanneer de nieuwe ruimte niet gealloceerd kan worden, dan wordt de inhoud waar *ptr* naar wijst niet veranderd. Als *grootte* nul is, dan word het geheugenblok volledig geledigd.

Bij succes wordt een pointer tot het geheugenblok teruggegeven (dit kan in een andere locatie staan als voordien). Bij mislukking of als *grootte* nul is, dan wordt een null pointer teruggegeven.

Voorbeeld:

```
/* 12.5.1.c */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main(void)
{
   char *str;

   str = (char *) malloc(10);

   strcpy(str, "Hello");

   printf("String is %s\n  Adres is %p\n", str, str);
   str = (char *) realloc(str, 20);
   printf("String is %s\n  Nieuw adres is %p\n", str, str);

   free(str);

   return 0;
}
```

12.6 tenslotte

Onthoud dat u de grootte van het te alloceren geheugen op meerdere manieren kan aanduiden:

```
getal = (double *)malloc(sizeof(double));
```

is even geldig als bijvoorbeeld:

```
getal = (double *)malloc(8);
```

Het gebruik van typecasting is bij de dynamische geheugenallocatie niet vereist volgens de ANSI-C standaard, maar het kan zoals u ziet wel toegepast worden.

U mag niet vergeten dat er ook ruimte moet gereserveerd worden voor de 0x00, zoals dit stukje code aangeeft:

```
char *str = NULL;
char *hallo = "Hallo daar";
str = (char *)malloc(sizeof(char)*(strlen(hallo)+1));
```

Het is ook heel belangrijk om goed rekening te houden met het feit dat je het gealloceerde geheugen ook zelf weer vrij moet geven. Als je namelijk een functie hebt die een stukje geheugen niet vrijgeeft en die functie wordt zeer dikwijls aangeroepen, dan zorgt dit voor problemen.

C-taal voor beginners - hoofdstuk 13

ASCII

13.1 overzicht

Char	Dec	Oct	Hex	Char	Dec	Oct	Hex	Char	Dec	Oct	Hex	Char	Dec	Oct	Hex
(nul)	0	0000	0x00	(sp)	32	0040	0x20	@	64	0100	0x40	`	96	0140	0x60
(soh)	1	0001	0x01	!	33	0041	0x21	A	65	0101	0x41	a	97	0141	0x61
(stx)	2	0002	0x02	"	34	0042	0x22	B	66	0102	0x42	b	98	0142	0x62
(etx)	3	0003	0x03	#	35	0043	0x23	C	67	0103	0x43	c	99	0143	0x63
(eot)	4	0004	0x04	$	36	0044	0x24	D	68	0104	0x44	d	100	0144	0x64
(enq)	5	0005	0x05	%	37	0045	0x25	E	69	0105	0x45	e	101	0145	0x65
(ack)	6	0006	0x06	&	38	0046	0x26	F	70	0106	0x46	f	102	0146	0x66
(bel)	7	0007	0x07	'	39	0047	0x27	G	71	0107	0x47	g	103	0147	0x67
(bs)	8	0010	0x08	(	40	0050	0x28	H	72	0110	0x48	h	104	0150	0x68
(ht)	9	0011	0x09	)	41	0051	0x29	I	73	0111	0x49	i	105	0151	0x69
(nl)	10	0012	0x0a	*	42	0052	0x2a	J	74	0112	0x4a	j	106	0152	0x6a
(vt)	11	0013	0x0b	+	43	0053	0x2b	K	75	0113	0x4b	k	107	0153	0x6b
(np)	12	0014	0x0c	,	44	0054	0x2c	L	76	0114	0x4c	l	108	0154	0x6c
(cr)	13	0015	0x0d	-	45	0055	0x2d	M	77	0115	0x4d	m	109	0155	0x6d
(so)	14	0016	0x0e	.	46	0056	0x2e	N	78	0116	0x4e	n	110	0156	0x6e
(si)	15	0017	0x0f	/	47	0057	0x2f	O	79	0117	0x4f	o	111	0157	0x6f
(dle)	16	0020	0x10	0	48	0060	0x30	P	80	0120	0x50	p	112	0160	0x70
(dc1)	17	0021	0x11	1	49	0061	0x31	Q	81	0121	0x51	q	113	0161	0x71
(dc2)	18	0022	0x12	2	50	0062	0x32	R	82	0122	0x52	r	114	0162	0x72
(dc3)	19	0023	0x13	3	51	0063	0x33	S	83	0123	0x53	s	115	0163	0x73
(dc4)	20	0024	0x14	4	52	0064	0x34	T	84	0124	0x54	t	116	0164	0x74
(nak)	21	0025	0x15	5	53	0065	0x35	U	85	0125	0x55	u	117	0165	0x75
(syn)	22	0026	0x16	6	54	0066	0x36	V	86	0126	0x56	v	118	0166	0x76
(etb)	23	0027	0x17	7	55	0067	0x37	W	87	0127	0x57	w	119	0167	0x77
(can)	24	0030	0x18	8	56	0070	0x38	X	88	0130	0x58	x	120	0170	0x78
(em)	25	0031	0x19	9	57	0071	0x39	Y	89	0131	0x59	y	121	0171	0x79
(sub)	26	0032	0x1a	:	58	0072	0x3a	Z	90	0132	0x5a	z	122	0172	0x7a
(esc)	27	0033	0x1b	;	59	0073	0x3b	[	91	0133	0x5b	{	123	0173	0x7b
(fs)	28	0034	0x1c	<	60	0074	0x3c	\	92	0134	0x5c	|	124	0174	0x7c
(gs)	29	0035	0x1d	=	61	0075	0x3d	]	93	0135	0x5d	}	125	0175	0x7d
(rs)	30	0036	0x1e	>	62	0076	0x3e	^	94	0136	0x5e	~	126	0176	0x7e
(us)	31	0037	0x1f	?	63	0077	0x3f	_	95	0137	0x5f	(del)	127	0177	0x7f

ASCII Naam	Omschrijving	Escape Sequenties
nul	null byte	\0
bel	bell karakter	\a
bs	backspace	\b
ht	horizontale tab	\t
np	formfeed	\f
nl	newline	\n
cr	carriage return	\r
vt	vertical tab	
esc	escape	
sp	space	

13.2 wat is ASCII?

ASCII staat voor American Standard Code for Information Interchange. Dit is een in Amerika ontwikkelde code om het doorgeven van informatie tussen verschillende computers mogelijk te maken. Op een paar uitzonderingen na kunnen alle computers informatie in ASCII formaat lezen.

Er zijn 128 verschillende karakters in de standaard ASCII-code. 96 ervan zijn de normale afdrukbare karakters: grote en kleine letters, numerals en een aantal speciale tekens. De andere 32 worden controle codes genoemd aangezien zijn de computer (of printer) eerder controleren dan visueel zichtbaar te zijn. Voorbeelden zijn de carriage return en line feed codes die terug naar het begin van een lijn gaan en een lijn naar beneden bewegen. Er zijn ook minder duidelijke codes: om naar een nieuwe pagina te gaan, om het einde van een pagina aan te duiden, om een geluid te laten horen, om andere karakters te verwijderen enz... .

13.3 ASCII-bestanden

Wat wij kennen als een ASCII-bestand is een computerbestand dat geen codes bevat behalve de 128 gedefinieerd in de standaard. Een vervelend gevolg van het feit dat het een Amerikaanse standaard is is dat geaccentueerde karakters en tekens als het Engelse Pond teken niet weergegeven kunnen worden. Er bestaat ook een ASCII codetabel met 256 codes, maar deze is niet standaard en kan mogelijk niet op alle computers toegepast worden.

Een speciaal type ASCII-bestand is het tekstbestand. Dit is simpelweg een ASCII-bestand dat enkel de controle codes bevat die betrekking hebben tot afdrukken en natuurlijk de gewone letters, figuren en dergelijke.

Bestanden die andere codes bevatten noemt men binaire bestanden, die speciaal kunnen hercodeerd worden als 7-bit bestanden, zodat computers denken dat ze ASCII zijn terwijl ze zeker geen gewone tekstbestanden zijn.

Het verlies van zulke extra's wordt meer dan gecompenseerd door het feit dat ASCII bestanden, op één computer aangemaakt, kunnen gebruikt worden door een compleet ander systeem, gebruik makend van andere software en kan over elke vorm van data link gestuurd worden. Daarom hebben de grote hoeveleheid toepassingsprogramma's die bestaan informatie schrijven in ASCII formaat en zo door andere bestanden gebruikt worden. Zulke bestanden kunnen van computer naar computer gestuurd worden.

Alhoewel er vele types van computer bestanden zijn, zijn er twee termen die gebruikt worden om degene die niet ASCII zijn te onderscheiden: binaire bestanden en 7-bit bestanden.

13.4 problemen

ASCII-bestanden kunnen soms wel voor problemen zorgen als gevolg van bepaalde karakters. Zo wordt het '\n' karakter als twee tekens beschouwd onder het Dos systeem, maar als één teken onder het UNIX systeem. Hier moet u op letten als u tussen deze besturingssystemen bestanden wil overdragen.