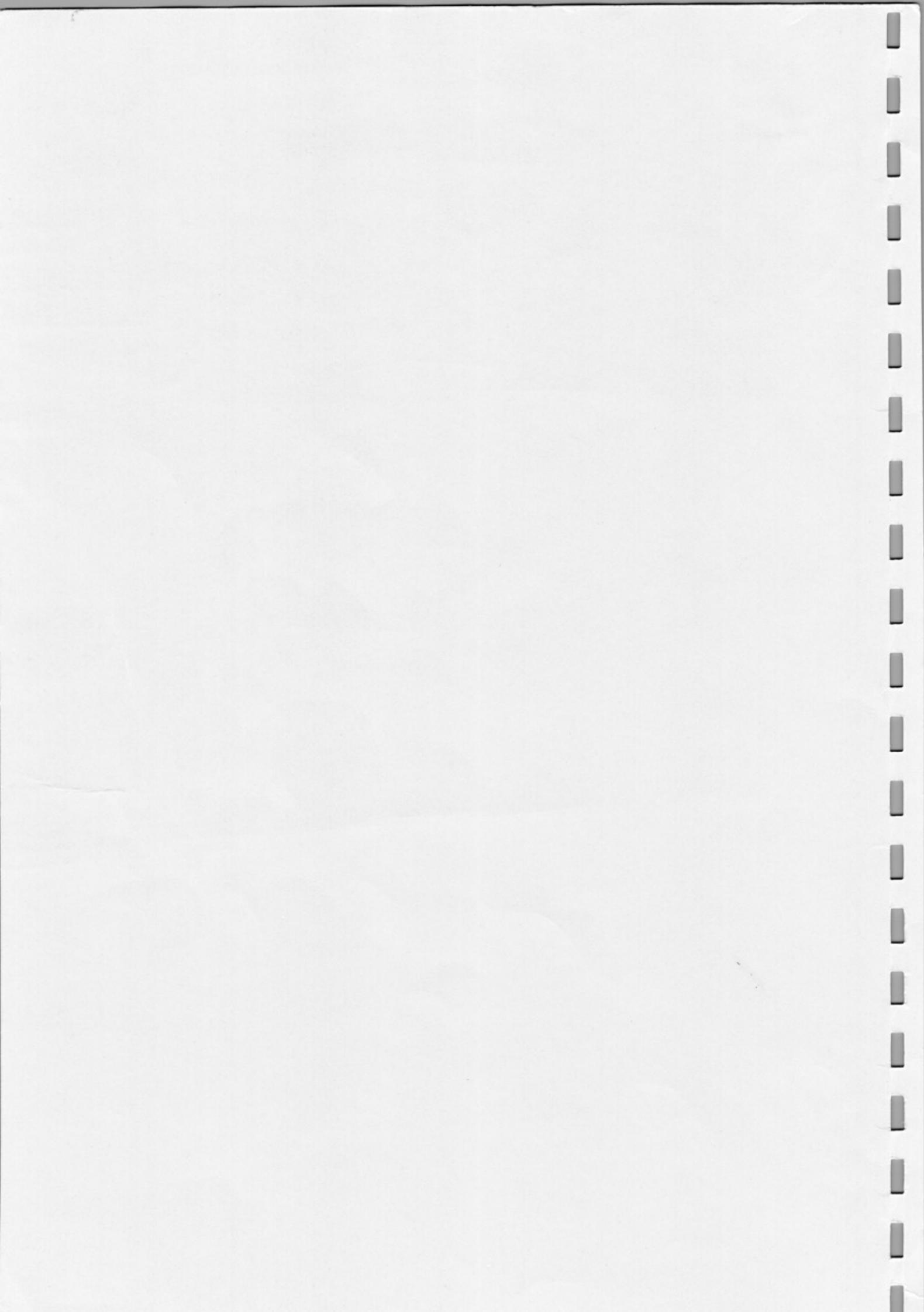


# Wij gaan naar C



ERIK V. EYKELEN

Tweehek Recreatie



*Wij gaan naar C*  
Een cursus C voor beginners

Erik van Eykelen

juni 1990

"TWEEHEK"-RECREATIE  
SCHOONLOËRSTRAAT 4  
9534 PC WESTDORP  
TEL. 05998-34541

© Copyright 1990 by 1001 Software Development  
Noorderkade 2g • 1823 CJ Alkmaar • 072-114541

*computer graphics consultants voor multimedia projecten*

kabelkranten • inhouse communications • presentaties  
graphics t.b.v. televisieprodukties

"TWEEHEK"-Recreatie  
Schoonloërstraat 4  
9534 PC WESTDORP  
Tel. 05998-34541



'Wij gaan naar C' door Erik van Eykelen  
Copyright (C) 1988-'89-'90 1001 Software Development  
1001:Noorderkade 2-G, 1823 CJ Alkmaar, tel/fax 072-114541  
privé:Ilpenwaard 10, 1824 GA Alkmaar, tel 072-612318

Dit document mag niet, gedeeltelijk of geheel, op welke wijze  
dan ook geduplicateerd worden zonder schriftelijke toestemming van  
de auteur.

Lijst van geraadpleegde en/of aanbevolen literatuur:

- \* The C Programming Language  
Brian W.Kernighan and Dennis M.Ritchie  
Prentice-Hall, Inc.  
ISBN 0-13-110163-3
- \* De programmeertaal C  
ir. L. Ammeraal  
Academic Service  
ISBN 90-6233-178-5
- \* De programmeertaal C: grondbeginselen en toepassingen  
Al Kelley & Ira Pohl  
Addison-Wesley Europe  
ISBN 90-6789-068-5
- \* BYTE magazine, August 1988 (Article "The State of C" by  
Kernighan and Ritchie)

Met dank aan:

Rieta (omdat ze een schat is). Pa, Blieb, Arjen, Robbert, Daan,  
Gerjo en Ronald voor het doorlezen en corrigeren van het  
manuscript tijdens het eerste gebruik ervan in het computer-  
vakantiekamp Tweehek in de zomer van 1989.

De auteur:

Erik van Eykelen is 'co-founder' van het Alkmaarse softwarehuis  
1001 Software Development en werkzaam als ontwikkelaar van multi-  
media-applicaties voor bedrijven, openbare instellingen en  
televisieomroepen.

"TWEEHEK"-Recreatie  
Scheeniërstraat 4  
9534 PC WESTDORP  
Tel. 0599-34541



## INHOUDSOPGAVE

<b>Hoofdstuk 1</b>	
1.1 Inleiding . . . . .	1
1.2 Enkele belangrijke termen . . . . .	2
<b>Hoofdstuk 2</b>	
2.1 De eerste programma's . . . . .	4
2.2 Het opnemen van commentaar in een programma . . . . .	6
2.3 Samenvatting . . . . .	6
2.4 Opdrachten . . . . .	7
<b>Hoofdstuk 3</b>	
3.1 Het printen van een getal . . . . .	8
3.2 Samenvatting . . . . .	11
3.3 Opdrachten en vragen . . . . .	11
<b>Hoofdstuk 4</b>	
4.1 Constanten . . . . .	13
4.2 Integers . . . . .	13
4.3 Floats en doubles . . . . .	13
4.4 Karakterconstanten . . . . .	14
4.5 Strings . . . . .	14
4.6 Samenvatting . . . . .	14
<b>Hoofdstuk 5</b>	
5.1 Declareren . . . . .	15
5.2 Assigneren . . . . .	17
5.3 Samenvatting . . . . .	17
5.4 Opdrachten en vragen . . . . .	18
<b>Hoofdstuk 6</b>	
6.1 Optellen, aftrekken, vermenigvuldigen en delen . . . . .	20
6.2 Expressies . . . . .	20
6.3 Opdrachten . . . . .	21
<b>Hoofdstuk 7</b>	
7.1 Relationale en logische operatoren . . . . .	22
7.2 Conditionele statements . . . . .	22
7.3 Een andere vorm van de conditionele expressie . . . . .	24
7.4 Samenvatting . . . . .	24
7.5 Opdracht . . . . .	25
<b>Hoofdstuk 8</b>	
8.1 Herhalingsstatements . . . . .	26
8.2 De while lus . . . . .	26
8.3 Het for statement . . . . .	27
8.4 Het do-while statement . . . . .	28
8.5 Eeuwige lussen . . . . .	28
8.6 Samenvatting . . . . .	29
8.7 Opdrachten . . . . .	30

"TWEEHEK"-Recreatie  
Schoonloërstraat 4  
9534 PC WESTDORP  
Tel. 05998-34541



Hoofdstuk 9		
9.1	De switch en break statements . . . . .	33
9.2	Nogmaals het break statement . . . . .	34
9.3	Het continu statement . . . . .	34
9.4	Samenvatting . . . . .	35
9.5	Opdrachten . . . . .	35
Hoofdstuk 10		
10.1	Increment en decrement . . . . .	36
10.2	Spelen met bits . . . . .	37
10.3	De komma operator . . . . .	38
10.4	Associativiteit . . . . .	39
10.5	Samenvatting . . . . .	40
10.6	Opdrachten . . . . .	41
Hoofdstuk 11		
11.1	Zelf functies maken . . . . .	42
11.2	Het teruggeven van een waarde door een functie .	44
11.3	De ANSI C manier van functies schrijven . . . . .	45
11.4	Het aanroepen van een functie vanuit een functie . . . . .	48
11.5	Het aanroepen van dezelfde functie vanuit een functie . . . . .	49
11.6	Samenvatting . . . . .	50
11.7	Opdracht . . . . .	50
Hoofdstuk 12		
12.1	Geheugenklassen, type, storage class en scope .	51
12.2	De auto storage class . . . . .	51
12.3	De extern storage class . . . . .	52
12.4	De register storage class . . . . .	54
12.5	De static storage class . . . . .	56
12.6	Lokale statische variabelen . . . . .	57
12.7	Externe statische variabelen . . . . .	58
12.8	Samenvatting . . . . .	58
Hoofdstuk 13		
13.1	De diverse constanten . . . . .	59
13.2	Arrays . . . . .	59
13.3	Meerdimensionale arrays . . . . .	61
13.4	Samenvatting . . . . .	62
13.5	Opdrachten . . . . .	62
Hoofdstuk 14		
14.1	Pointers . . . . .	63
14.2	Pointers en adressen . . . . .	65
14.3	Parameteroverdracht via pointers . . . . .	67
14.4	Pointers naar geheugengebieden . . . . .	69
14.5	Pointers en arrays . . . . .	74
14.6	Arrays van pointers . . . . .	75
14.7	Pointers naar functies . . . . .	76
14.8	Samenvatting . . . . .	78

"TWEEHEK"-Recreatie  
Schoonleerstraat 4  
9534 PC WESTDORP  
Tel. 05998-34541



Hoofdstuk 15	
15.1 printf()	80
15.2 scanf()	83
15.3 sprintf() en sscanf()	84
15.4 strcpy(), strcat() en strlen()	84
15.5 strncpy() en strncat()	86
15.6 strcmp() en strncmp()	87
15.7 Opdrachten	87
Hoofdstuk 16	
16.1 Datastructuren in C: structures	89
16.2 Structures	89
16.3 Een array van structures	91
16.4 Structures en functies	91
16.5 Pointers en structures	92
16.6 Dynamische geheugenallocatie	99
16.7 Structuren met bitvelden	107
Hoofdstuk 17	
17.1 Invoer en uitvoer	109
17.2 Karakters lezen en schrijven ( putc(), getc() )	110
17.3 Grote bestanden lezen en schrijven ( fread() en fwrite() )	112
17.4 Regels lezen en schrijven ( fprintf() en fscanf() )	114
17.5 De functies fputs() en fgets()	116
17.6 Random-access lezen en schrijven	116
17.7 De functies perror() en feof()	117
17.8 Programma-argumenten	118
Hoofdstuk 18	
18.1 Typedef	120
18.2 #define	121
18.3 De preprocessor en #include	122
18.4 Voorwaardelijke compilatie	123
18.5 Enkele trucs met de preprocessor	124
Hoofdstuk 19	
19.1 Portability of overdraagbaarheid	126
19.2 Veel gemaakte fouten	127
19.3 Debuggen	128
19.4 Het einde en een nieuw begin	131

"TWEEHEK"-Recreatie  
 Schoenleiderstraat 4  
 9534 PC WESTDORP  
 Tel. 05998-34541



Concerning some aspects of mathematical education

Once children learn to sum it's fun  
To ask them, "What is  $1 + 1$ "  
For when they answer, "Two", we know  
Their knowledge has begun to grow.

Later, they study late at night  
To grasp the rules of logic right,  
And learn, from Boole and other men,  
That  $1 + 1$  is 1 again.

Still later, when they learn to do  
Adding, and higher things, mod 2,  
And drink from wisdom as they ought,  
They find the answer now is, "Naught".

The moral is, as you might guess:  
He who knows more will answer less

Franz Hohn.



## HOOFDSTUK 1

### 1.1 Inleiding

Welkom in het boek 'Wij gaan naar C'. Het boek behandelt de programmeertaal C. De opzet is niet deze taal tot in de kleinste details te behandelen (lees daarvoor het boek van Kernighan & Ritchie, de makers van C) maar zóveel uit te leggen dat snel kleine programma's kunnen worden geschreven. Hiertoe worden de basiscommando's van C (en dat zijn er niet zo erg veel) duidelijk en uitgebreid uitgelegd. Pientere programmeurs hebben een zesde zintuig voor het lokaliseren van programmadelen die efficiënter en kleiner kunnen. Grijp hiervoor naar de reeds genoemde 'C bijbel' van Kernighan & Ritchie, hierin staan tientallen algoritmen en trucs om veel voorkomende problemen kort en snel op te lossen.

Bijna alle boeken over de programmeertaal C beginnen met een stukje geschiedenis. Dit boek dus ook, al is het maar om te tonen dat C weliswaar een onbekende taal is maar toch al bijna 20 jaar veel gebruikt wordt. Wrijf dat de mensen maar onder de neus die zeggen dat ze nog nooit van 'C' gehoord hebben ("...hoe heet die taal?...zee?...of say?...nooit van gehoord!").

Hoewel C oorspronkelijk ontworpen was voor het schrijven van systeemprogrammatuur zoals compilers, operating systems en text editors is in de loop der jaren gebleken dat C toepasbaar is voor veel meer soorten programma's zoals databases, administratieve systemen, wiskundige programma's, grafische toepassingen en tekstverwerkers.

C dateert uit 1972, het tijdperk waarin slechts bedrijven en universiteiten over computers konden beschikken omdat ze zeer duur waren. Dennis Ritchie, werkend voor Bell Laboratories, ontwierp de taal C om een o.a. een operating system (UNIX) te schrijven voor de destijds nieuwe PDP-11, een middelgrote computer. Het nieuwe van de PDP-11 was dat de processor instructies had voor bytes, words en longwords. Aangezien dit niet ondersteund werd door de voorloper van C, de taal B (die weer haar oorsprong vond in de taal BCPL), was een nieuwe taal nodig.

Het resultaat is dat C weliswaar een hogere programmeertaal is maar dat er ook gewerkt kan worden met bytes, words en longwords. Het is zelfs mogelijk om bits te manipuleren (and, or, roteren etc.). Dat C een hogere programmeertaal is, bewijzen de velen typen variabelen, de lusconstructies, verscheidene beslissings-commando's en de uitgebreide I/O mogelijkheden.

Bovenal; C is niet moeilijk om te leren. De schrijver gaat er wel van uit dat de lezer al enige kennis heeft van bijvoorbeeld Pascal, BASIC of machinetaal. Pas wel op dat je geen C commando's door je Pascal programma's gaat gooien, beide talen lijken nogal op elkaar!

Om dit boek goed te kunnen volgen is het nuttig de volgende paragraaf even goed door te lezen. Er staan een aantal termen in die veel worden gebruikt in de rest van het boek en het is van belang dat de lezer deze kent.

Tevens wil ik vragen of de serieuze lezer alle vragen wil beantwoorden die gesteld worden. Zelf houd ik niet van boeken met onzinnige en te-veel-van-hetzelfde-soort vraagstukken dus in dit boek zul je die dan ook niet aantreffen. Dat betekent dat, hoe simpel ze ook lijken(!), de vragen gemaakt moeten worden.

### **1.2 Enkele belangrijke termen**

#### **statement, commando of opdracht**

Een statement is een instructie in een programmeertaal, in BASIC bijvoorbeeld PRINT, INPUT, LIST etc.

#### **variabele**

Een moeilijke definitie: een variabele is een grootheid die uit een gegeven reeks waarden elke willekeurige waarde kan aannemen. Kijkend naar BASIC: A, B zijn getalvariabelen en A\$, B\$ zijn stringvariabelen. Er zijn dus verschillende soorten variabelen om verschillende soorten gegevens in op te slaan (getallen, letters).

#### **karakter, character, teken**

Programmeurs kunnen onderling zo lekker wouwelen over 'controle karakters' of 'rare karaktertjes'. Ze hebben het dan niet over mensen met een knoop in hun geest maar over letters, cijfers en andere tekens die op het beeldscherm kunnen worden afgedrukt.

#### **sourcefile**

Onder een sourcefile wordt verstaan het bestand waarin de programmaregels staan die ingetikt zijn. De sourcefile wordt meestal bewerkt in een editor (een eenvoudige tekstverwerker).

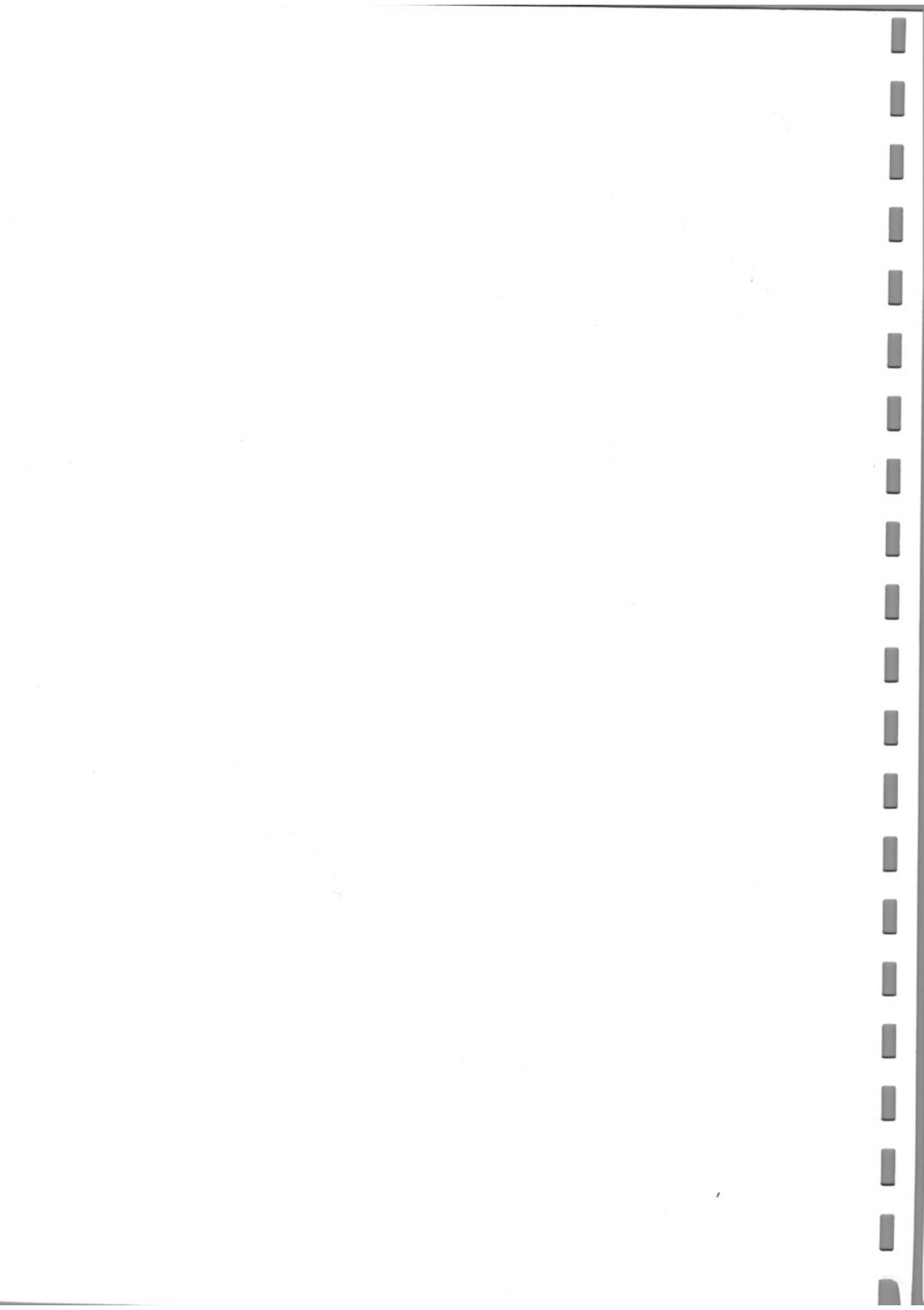
#### **objectcode**

De objectcode is het programma dat draaibaar is. De sourcefile van een programma dat geschreven is in een hogere programmeertaal moet eerst gecompileerd worden voordat het draaibaar is (te 'runnen' in jargon). Het resultaat na het compileren is de objectcode.

## **compiler, assembler en linker**

Een van de voordelen van programmeren in machinetaal (dus echt met een machinetaal monitor) is dat het programma direct klaar is. Al iets geavanceerder is het programmeren in assemblerstaal. Om het programma te draaien moet de sourcecoding geassembleerd worden. Dat assembleren bestaat uit het vertalen en begrijpen van allerlei labels, commentaarregels en andere niet-machinetaal opcodes (machinetaalinstructies) voordat er draaibare machinetaal uitkomt. Bij hogere programmeertalen is er een nog veel groter vertaalprobleem. De assemblermacho's programmeren tenminste nog rechtstreeks in de taal die de processor verstaat n.l. al die drie- of vierletterige afkortingen (STA, RTS, MOVE, MULU etc.) maar in BASIC mag je neerzetten: FOR a=1 TO 10: PRINT "Riets": NEXT a. Er is geen microprocessor die dit rechtstreeks "vreet"! De mensen die in machinetaal programmeren weten dat bijv. een tekst 10 keer op het scherm afdrukken op z'n minst, met een niet al te modern (!) en omvangrijk Operating System, al gauw 10 programmeerregels oplevert. In een hogere programmeertaal vergt dit slechts 1 à 2 regels. Als dus in Pascal, C of andere hogere programmeertaal gewerkt wordt, moet er flink vertaald worden. Bij de meeste talen zijn er drie vertaalstappen te onderscheiden:

- compileren** - Het omzetten van de hogere programmeertaal naar assembler.
- assembleren** - Het omzetten van assembler naar machinetaal.
- linken** - Tijdens het linken wordt er aan het geassembleerde programma nog een aantal reeds eerder geassembleerde functies vastgeplakt. Het programma kan dus functies aanroepen die helemaal niet in het programma staan!  
Die linkbare functies kunnen zelf geschreven zijn maar de fabrikant van de taal heeft er vaak ook al een heleboel gemaakt.



## HOOFDSTUK 2

### 2.1 De eerste programma's

In dit hoofdstuk wordt echt begonnen met C. Als eerste een programma om een regel tekst op het scherm te printen, een zeer belangrijke operatie in elke taal.

```
main()
{
    printf("Hallo allemaal.\n");
}
```

Dit programma print op het scherm:

```
Hallo allemaal.
```

N.B. Als het programma niet werkt, zet dan helemaal bovenaan het programma (dus nog boven 'main()') de volgende regel:

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

Waarom dat moet, zal nog worden uitgelegd.

De taal C is voornamelijk gebaseerd op het gebruik van functies (subroutines). Het gebruik van functies is zelfs zo ver doorgevoerd dat het hoofdprogramma (het 'main'programma) zelf een functie is. Als een C programma gestart wordt, springt de computer eerst naar de main() functie. Van daaruit wordt eventueel naar andere functies gesprongen. Een (simpele) C functie ziet er altijd als volgt uit (later zullen er nog enkele uitbreidingen bij een functie komen):

```
functienaam(parameters)
{
    het programma....
}
```

De overeenkomsten tussen dit voorbeeld en het eerst programma zijn duidelijk. In het programma heet de functie main() en het programma zelf bestaat uit het printen van een tekst. Het afdrukken van een tekst gebeurt ook weer met een functie, n.l. printf(). Ook printf() is weer te herkennen als functie want hij heeft twee ronde haakjes en de parameter van de functie is een string. printf() noemen we een standaardfunctie omdat hij bij de C compiler geleverd wordt en niet zelf gemaakt behoeft te worden.

Let op de puntkomma achter elke functieaanroep!

Als het printf() commando nader bekijken wordt, valt op dat achter de tekst "Hallo allemaal." een '\n' staat. De schuine streep (backslash) met een kleine 'n' erachter zorgt ervoor dat nadat de tekst geprint is, ook een 'newline' geprint wordt. Een newline zorgt er voor dat de cursor op de regel onder "Hallo allemaal." komt.

```
main()
{
    printf("Dit vormt samen ");
    printf("een volledige zin.\n");
    printf("\nDit is\nnook toegestaan.\n");
}
```

De uitvoer van dit programma is:

Dit vormt samen een volledige zin.

Dit is ook  
toegestaan.

Zie ook dat hier weer na elke functieaanroep een puntkomma staat. Deze puntkomma's scheiden de functies en zijn verplicht. Een foutmelding van de compiler is het gevolg als ze vergeten worden.

Naast de newline code '\n' die zojuist gebruikt werd, bestaan er nog een aantal andere handige codes (ook wel escapesequences genoemd):

horizontal tab	\t	slaat een aantal spaties over
backspace	\b	verplaats cursor 1 teken naar links
carriage return	\r	cursor gaat naar de volgende regel
form feed	\f	maakt het scherm schoon
backslash	\\\	nodig als je een \ wilt printen
single quote	'\'	nodig als je een ' wilt printen
double quote	"\"	nodig als je een " wilt printen
null character	\0	nodig bij het werken met stringvariabelen

Het scherm kan dus schoon gemaakt worden door:

```
printf("\f");
```

In Turbo C op de PC werkt dit echter niet, daar kan het scherm worden schoon gemaakt met de functie:

```
clrscr();
```

Als er een hele lange zin geprint moet worden en de zin past niet meer op 1 regel van de tekstverwerker dan kan het splitsingsteken (\) gebruikt worden:

```
printf("Dit is een voorbeeld van \
het splitsingsteken.\n"); /* Zet nooit spaties voor 'het'! */
/* de compiler print die ook! */
```

De uitvoer van dit programma is:

```
Dit is een voorbeeld van het splitsingsteken.
```

## 2.2. Het opnemen van commentaar in een programma

Hoe commentaar in een programma gezet kan worden, kan niet vroeg genoeg uitgelegd worden. Nu zal het nog niet echt nodig zijn om de programma's van tekst en uitleg te voorzien maar bij grote programma's is het VERBODEN om het niet te doen. Tevens kunnen bij het 'debuggen' van programma's stukken coding buiten gebruik worden gesteld worden door er commentaar van te maken.

Algemene vorm:      /\* commentaar \*/

```
/*
Dit is een
voorbeeldprogramma
*/

main() /* main() is de hoofdfunctie */
{
    printf("Hallo.\n"); /* printf() is ook een functie */
}
```

Wat niet is toegestaan is het 'nesten' van commentaar, dus:

```
/* /* Dit mag niet */ */
```

## 2.3 Samenvatting

\* Een C-programma heeft altijd de volgende structuur:

```
main()
{
    ...programma regel(s) afgesloten door punt-komma (;)
```

\* printf() is een standaardfunctie. Later gaan we ook zelf functies maken.

\* Commentaar staat tussen /\* en \*/.

#### 2.4 Opdrachten

1. Verbeter het onderstaande programma en compileer het:

```
main()
{
    print"Dit programma is verbeterd door mij")
}
```

2. Verbeter en compileer ook dit programma:

```
main()
{
    printf("Dit is de eerste zin.\n")
    printf("Dit is de tweede zin.\n")
    printf("Dit is de derde zin.\n")
}
```

3. Maak een programma dat jouw adres op het scherm print.

4. Maak een programma dat het scherm schoon maakt en vervolgens op het scherm print (gebruik maar 1 keer het printf() commando!):

```
1
2
3
4
5 dit is de vijfde regel
```

5. Verander het onderstaande programma zó dat het niets op het scherm print. De printf() commando's moeten wel blijven staan en er mag niets aan de teksten worden veranderd die in de printf() commando's staan:

```
main()
{
    printf("Het is niet de ");
    printf("bedoeling dat dit ");
    printf("geprint wordt.\n");
}
```

## HOOFDSTUK 3

### 3.1 Het printen van een getal

Het belangrijkste gebruik van computers is het uitvoeren van berekeningen. Dit hoofdstuk behandelt het werken met variabelen en het afdrukken van variabelen met de printf() functie:

```
main()
{
    int i;
    i = 1001;

    printf("De waarde van de variabele i is %d\n", i);
}
```

De uitvoer is:

```
De waarde van de variabele i is 1001
```

Als we het programma doornemen, komen we iets nieuws tegen:

```
int i;
```

Als we al iets verder kijken in het programma zien we ook dat in de letter die achter int staat ('i') het getal 1001 wordt opgeslagen. Kennelijk heeft int iets te maken met variabelen. De afkorting int staat voor 'integer' wat in het Nederlands 'geheel getal' betekent. Dat klopt want 1001 is een geheel getal, dit in tegenstelling tot bijv. 3.14 dat een gebroken getal is. Let op de punt in 3.14, in het Engels/Amerikaans wordt een punt gebruikt waar wij een komma neerzetten (en voor de goede orde: ze zetten weer een komma neer waar wij een punt zetten, zo zal worden genoteerd \$1,500 en niet \$1.500).

Als in C een variabele gebruikt wordt dan moet hij eerst 'gedeclareerd' worden, dat is: aangeven wat voor een variabele het is (int) en hoe hij moet heten (i). Door aan te geven dat 'i' een integer is, wordt er in het geheugen van de computer een lokatie gereserveerd die groot genoeg is om de integer op te slaan. In vaktaal kan nu gezegd worden dat 'de variabele i is gedeclareerd als het type integer'. Later zal blijken dat er ook nog meer types zijn bijv. om gebroken getallen in op te slaan.

De naam die achter int staat, wordt een identifier genoemd. Een identifier is een verzameling letters en/of cijfers. Wat ook veel gebruikt wordt is de underscore (\_). Om een of andere reden is dit tekentje zeer populair bij de gebruikers van C en het in de inleiding genoemde operating system UNIX. Waar op gelet moet worden is dat C verschil maakt tussen hoofd- en kleine letters, dus pas op bij het overtypen van programma's!

Meestal zijn de eerste 8 tekens van een identifier significant (d.w.z. van betekenis). Dit betekent dus dat het volgende programmafragment problemen zou kunnen opleveren:

```
int horizontale_waarde;  
int horizontale_positie;
```

voorbeelden van correcte namen (identifiers):

```
i  
TweeHek  
x_pos
```

Wat niet als identifier gebruikt mag worden is een keyword. Een keyword is een naam die gereserveerd is voor allerlei C commando's. De onderstaande lijst toont de standaard keywords, let er op dat int dus ook een keyword is. De meeste keywords zullen nog niet bekend zijn, dat komt nog wel:

```
auto, break, case, char, continue, default, do, double, else,  
extern, float, for, goto, if, int, long, register, return,  
short, sizeof, static, struct, switch, typedef, union,  
unsigned, while
```

Het hangt van de C compiler af of er nog meer keywords zijn. De meeste compilers kennen ook nog een 'asm' keyword dat gebruikt wordt om stukken assembler op te nemen tussen de C coding. Een recente toevoeging is het keyword 'enum'. Dit is een nieuw datatype dat de meeste compilers nog niet kennen, het zal dan ook niet in dit boek worden behandeld.

Om het vorige programma verder te bekijken, wordt het nog eenmaal afgedrukt:

```
main()  
{  
    int i;  
    i = 1001;  
  
    printf("De waarde van de variabele i is %d\n", i);  
}
```

De functie printf() werd in hoofdstuk 2 gebruikt om tekst mee af te drukken. Nu wordt hij tevens gebruikt om variabelen te printen.

De printf() functie is in twee delen te splitsen. Het ene deel bevat de tekst die afgedrukt moet worden en bevat tevens aanwijzingen hoe en waar variabelen afgedrukt moeten worden. Die aanwijzing is de %d. De 'd' in '%d' slaat op decimal, het programma drukte het antwoord ook decimaal af. Het tweede deel bestaat uit een verzameling identifiers (hier alleen 'i') of constanten. Naast %d is nuttig:

```
%c single character (enkel teken).  
%s string (wordt geprint totdat \0 wordt bereikt). Dit type variabele wordt behandeld in hoofdstuk 4.  
%f floating point or double (cijfers met een "drijvende komma"). Ook dit type wordt in het volgende hoofdstuk behandeld.  
%u unsigned (positief) decimaal getal.  
%x hexadecimaal getal.
```

Met deze commando's zijn nog diverse mogelijkheden:

```
%-3.7f drukt bijv. af: -3.1415927 (een negatief getal met maximaal 3 cijfers voor en maximaal 7 cijfers achter de komma). DIT IS DUS ANDERS DAN IN PASCAL!  
.5s drukt bijv. af: Tweek als de string is Tweehok. Het afdrukken van strings zal nog behandeld worden.  
%05d drukt bijv. af: 00001 als de variabele gelijk is aan 1.
```

Zie hoofdstuk 15 voor een volledige behandeling van printf().

Het is ook toegestaan om met printf() meerdere getallen te printen in één printf() aanroep:

```
main()  
{  
    int i;  
    int j;  
  

```

De uitvoer is:

```
i is 100 en j is 200
```

### 3.2 Samenvatting

\* Als in C een variabele gebruikt gaat worden, moet hij eerst gedeclareerd worden.

```
int i; /* declareer een integer met naam i */
```

\* De naam die een variabele krijgt heet de identifier.

\* Een keyword is C commando (int is bijvoorbeeld een keyword). Het is niet toegestaan keywords als identifier te gebruiken:

```
main()
{
    int int; /* dit is verboden */
}
```

\* Getallen kunnen worden afgedrukt op de volgende wijze:

```
main()
{
    int getal;

    getal=10;
    printf("Waarde is %d\n", getal);
}
```

### 3.3 Opdrachten en vragen

1. Schrijf een programma dat het decimale getal 255 afdrukt, zowel decimaal als hexadecimaal. De uitvoer moet er uitzien zoals hieronder is afgedrukt:

```
decimaal: 255
hexadecimaal: ff
```

2. Schrijf een programma dat de volgende uitvoer geeft en de onderstaande variabelen gebruikt:

```
int a,b,c;

a=10;
b=20;
c=30;
```

gewenste uitvoer:

a	b	c
000010	020	30
000020	030	10
000030	010	20

3. Kijk wat er gebeurt als je een programma schrijft waarin je een variabele afdrukt met printf() zonder deze eerst op een waarde geïnitialiseerd te hebben.
4. Probeer (op een intelligente wijze) eens uit te vinden hoe groot het getal mag zijn dat in een int past (denk aan 16 bits). Druk dit getal af om te controleren of het getal wel goed afgedrukt wordt. Te grote getallen zullen n.l. 'onzin uitvoer' tot gevolg hebben.



## HOOFDSTUK 4

### 4.1 Constanten

Constanten zijn getallen of een verzameling tekens die een bepaalde eigenschap hebben. Zo zijn getallen te verdelen in gehele en gebroken getallen (de getallen die deel uitmaken van de verzamelingen  $Z$  en  $Q$  in de wiskunde). Bovendien kent de computer de welbekende 'strings'. Alle constanten zullen op een rijtje worden gezet:

### 4.2 Integers

Integers zijn gehele getallen. De grootte van een int is afhankelijk van de processor die in de computer zit. Meestal is het een 16 bits getal, dat betekent dat er een getal in bewaard kan worden tussen -32767 en +32768.

Voorbeelden van integers:	1988	decimaal getal
	0xff (of 0xFF)	hexadecimaal getal
	2001L	een 'long' (dit type zal nog nader besproken worden.)

Let op de L bij longs en de '0x' bij hexadecimale getallen:

```
main()
{
    int i;

    i = ff; /* DIT IS FOUT */
    i = 0xff; /* dit is goed */
    i = 255; /* dit doet hetzelfde als de vorige regel */
}
```

### 4.3 Floats en doubles

Floats zijn getallen die een getal achter de komma hebben (ook 0 geldt als een getal achter de komma!). Het woord float komt van 'floating point', een begrip dat in het Nederlands 'drijvende komma' heet. Eenvoudig gesteld heeft een float altijd een zo klein mogelijk getal voor de komma, een zo precies mogelijk getal achter de komma (die dus heen en weer 'drijft') en een vermenigvuldigingsfactor. Hoe floats intern worden opgeborgen interesseert ons echter niet zo, als er 'gebroken getallen' optreden in een programma moet het float type gebruikt worden.

FLOATS kunnen een veel grotere waarde aannemen dan ints maar het nadeel is dat er tijdens berekeningen onnauwkeurigheden kunnen ontstaan. Een aanverwant type van de float is de double die, zoals zijn naam al doet vermoeden, tweemaal zo precies is als een float. Er wordt dan ook meer geheugen gereserveerd om een double getal op te bergen dan bij een float.

Een notatie voor grote floats en doubles gaat met de z.n. wetenschappelijke notatie die bijv.  $2 * 10\text{-tot-de-derde}$  afkort tot  $2e3$  (of  $2E3$ ). Als de waarde van een float een bepaalde grens overschrijdt, wordt die notatie gebruikt bij het afdrukken van het getal op het scherm:

Geprint wordt:  $2.71828e-3$  wat dus was: 0.00271828

#### 4.4 Karakterconstanten

Een karakterconstante is een enkel karakter, geschreven tussen apostrofs (single quotes in het Engels). De ASCII waarde van de letter A is 65, het getal 65 kan zondermeer vervangen worden door 'A'. Andere karakterconstanten zijn o.a. '\n' en '\0' die al eerder zijn behandeld.

#### 4.5 Strings

Een string is een verzameling karakters tussen aanhalingstekens:

"Dit is een string" maar ook "0" is een string.

"\200\100\String voorafgegaan door twee bytes"

Er is dus een verschil tussen 'a' en "a".

'a' is een karakterconstante en stelt een getal voor.

"a" is een string die een kleine a bevat.

#### 4.6 Samenvatting

- \* **integers** zijn gehele getallen.
- \* **floating point** getallen (kortweg **floats**) zijn getallen met cijfers voor en achter de komma.
- \* **doubles** zijn floats met een dubbele precisie.
- \* **karakterconstanten** zijn tekens tussen apostrofs, die een getal voorstellen.
- \* **strings** zijn verzamelingen van letters en cijfers.

## HOOFDSTUK 5

### 5.1 Declareren

In het vorige hoofdstuk is uitgelegd hoe ints, floating point-getallen, karakterconstanten en strings eruit zien. Wat nog niet behandeld is, is hoe de types heten waarin die constanten bewaard kunnen worden. Een voorbeeld van een type is 'int' die eerder is gebruikt.

```
main()
{
    int i;
    int j;
    float p;
    float q;
    char ch;
}
```

N.B. In TURBO C programma's waar gebruik gemaakt wordt van 'floats' en 'doubles' moet bovenaan (dus nog boven 'main()') de volgende regel worden toegevoegd:

```
#include <float.h>
```

Bovendien moet in de compiler opties de floatingpoint optie op 'emulation' staan als er geen mathematische coprocessor aan boord is. Ask your local computerleider for details.

In dit programma worden de identifiers 'i' en 'j' als int gedeclareerd, 'p' en 'q' als float en 'ch' als char. Dit declareren is nodig omdat de compiler een stuk geheugen moet reserveren om alle variabelen in op te slaan. Aan het begin van het programma staat daarom een lijstje met de 'wensen' van de programmeur ten aanzien van de variabelen die hij/zij wil gebruiken. Het hangt af van het type hoeveel ruimte er wordt gereserveerd, een char (waar een letter in op is te slaan) zal 1 byte 'kosten' en een int (een 16 bits getal) zal 2 bytes in beslag nemen.

Het programmafragment had ook korter genoteerd kunnen worden:

```
main()
{
    int i,j;
    float p,q;
    char ch;
}
```

Het is verboden om een variabele midden in een programma te declareren.

N.B. Toch wordt dit af en toe wel gedaan, voornamelijk om een 'register variabele' te declareren. Dit is een variabele die opgeslagen wordt in de microprocessor en daarom snel werkt. Desondanks is het verboden! (informatica leraren krijgen hier n.l. hartverzakkingen van...).

Voor de type aanduiding kan 'unsigned' gezet worden. Daarmee wordt aangegeven dat de constante die er in bewaard mag worden niet negatief mag zijn (unsigned = zonder minteken):

```
unsigned int i;
```

De meeste compilers kennen verscheidene namen voor de types. Deze namen zijn gedefinieerd in de includefile "types.h" (hoe include-files aangesproken moeten worden, wordt nog uitgelegd). In "types.h" kan bijvoorbeeld staan dat 'char' hetzelfde is als 'BYTE' of 'TEXT' zodat in het programma aan variabelen van het type 'char' suggestievere type-aanduidingen gegeven kunnen worden.

In plaats van:

```
main()
{
char tekstbuffer[10]; /* string van 10 tekens */
char ct; /* telvariabele */

/* ... */
}
```

kan men ook schrijven:

```
#include "types.h"

main()
{
TEXT tekstbuffer[10]; /* string van 10 tekens */
BYTE ct; /* telvariabele */

/* ... */
}
```

Tot slot worden alle elementaire types in C genoemd:

```
char
int
long
short
float
double
```

Voor deze type-aanduidingen mag dus altijd 'unsigned' staan. Het aantal bits dat elke type in beslag neemt is afhankelijk van de machine waarmee gewerkt wordt. Als vuistregel mag men echter uitgaan van de volgende formule:

```
lengte (short) <= lengte (int) <= lengte (long)
```

### 5.2 Assigneren

Assigneren betekent 'het toekennen van een waarde'. Aan een variabele kan op meerdere manieren een waarde toegekend worden:

```
main()
{
    int i;
    i = 1968;
}
```

Het programma had er ook zo uit mogen zien:

```
main()
{
    int i = 1968;
}
```

Het assigneren van een waarde aan meerdere variabelen tegelijk kan ook:

```
main()
{
    int k,l,m;
    k=l=m=1972;
}
```

### 5.3 Samenvatting

- \* **Declareren** is het bekend maken van een variabele aan het programma.
- \* **Assigneren** is het toekennen van een waarde aan een variabele.
- \* Een **unsigned int** is een positieve integer.
- \* **int, float, short etc.** zijn typen.

#### 5.4 Opdrachten en vragen

Kijk voor het maken van de opdrachten nog even naar paragraaf 3.1 (laatste deel) over het printen van de diverse typen (%d, %f etc.)!

1. Maak een programma dat de volgende uitvoer geeft en dat gebruik maakt van variabelen (dus niet letterlijk printen):

```
1  
15.5  
0.707
```

2. Verbeter het onderstaande programma:

```
main()  
{  
integer i;  
  
    i := 20  
    print i  
}
```

3. Maak een programma dat de volgende uitvoer levert (de getallen zijn hexadecimaal):

```
Eerste geheugenadres is: 000000  
512 K verder is:          07ffff  
Weer 512 K verder is:    0fffff
```

4. Waarom is dit programma pertinent FOUT ?

```
main()  
{  
long getal;  
  
    printf("De waarde = %ld", getal);  
}
```

5. Welk(e) programma('s) is/zijn fout of geeft/geven een foute uitvoer ?

A: main()  
{  
j, k;  
  
j = 10;  
k = 20;  
printf("j is %d en k is %d\n", j, k);  
}  
  
B: main()  
{  
int j;  
float k;  
  
j = 10;  
k = 20;  
printf("j is %d en k is %d\n", j, k);  
}  
  
C: main()  
{  
int j, k  
  
j = k = 20  
printf("j is %d en k is %d\n", j, k)  
}  
  
D: main()  
{  
int j;  
  
j = 10;  
k = 20;  
printf("j is %d en k is %d\n", j, k)  
}



## HOOFDSTUK 6

### 6.1 Optellen, aftrekken, vermenigvuldigen en delen

De tekens die gebruikt worden om deze bewerkingen uit te voeren heten operatoren. De + heet bijv. de optellingsoperator:

```
som      = a + b;  
resultaat = a - b;  
quotiënt = a / b;  
produkt   = a * b;
```

Voor delingen is er nog een extra operator, n.l. een operator om de rest van een deling te bepalen, te weten het procentteken (%). Bij de volgende deling ontstaat een breuk:

$$19 / 3 = 6.3333333$$

Dit resultaat is alleen met een float te krijgen. Als er met int's gewerkt wordt, gaat het als volgt:

$$19 / 3 = 6 \quad \text{en} \quad 19 \% 3 = 1$$

De rest kan dan bepaald worden door:  $1/3 = 0.3333333$

### 6.2 Expressies

Een expressie bestaat uit een of meer constanten of variabelen die onderling zijn verbonden door operatoren.

voorbeelden van expressies:

$$x*(y-i)+y*(i-y)$$

$$100/25$$

$$a*b$$

$$i$$

Later zal blijken dat functies die een waarde terug geven, ook te beschouwen zijn als expressies.

Het gebruik van haakjes kan belangrijk zijn in expressies:

$(x-y)*i$  geeft een ander resultaat dan  $x-y*i$  !

TIP: Gebruik liever te veel dan te weinig haakjes

### 6.3 Opdrachten en vragen

1. Maak een programma dat twee floating point getallen optelt, aftrekt, deelt en vermenigvuldigt. Print de resultaten netjes onder elkaar op het scherm.
2. Type het onderstaande programma over en kijk wat de uitvoer is.

```
main()
{
    int i1, i2, res1;
    float f1, f2, res2;

    i1 = f1 = 10;
    i2 = f2 = 3;

    res1 = i1 / i2;
    res2 = f1 / f2;

    printf("res1=%d en res2=%f\n", res1, res2);
}
```

3. In het onderstaande programma zijn geen ronde haakjes geplaatst in de expressie  $x = x*x+y+z$ . Ze horen er wel:

```
main()
{
    int x,y,z;

    x = 2;
    y = 3;
    z = 4;

    x = x*x+y+z;

    printf("Het antwoord = %d\n", x);
}
```

Plaats de haakjes zodanig dat de uitvoer ontstaat:

Het antwoord = 18

## HOOFDSTUK 7

### 7.1 Relationale en logische operatoren

Relationale en logische operatoren worden gebruikt om beslissingen te kunnen nemen:

<b>relationale operatoren</b>	< kleiner dan > groter dan <= kleiner dan of gelijk aan >= groter dan of gelijk aan == gelijk aan != ongelijk aan
<b>logische operatoren</b>	&& and (logische en)    or (logische of) ! not (logische niet)

Let vooral goed op de dubbele == voor 'gelijk aan'!

### 7.2 Conditionele statements

Het komt in een programma vaak voor dat twee waarden vergeleken moeten worden. Met de relationele en logische operatoren uit de vorige paragraaf kunnen zulke conditionele statements gemaakt worden.

```
main()
{
int i,j;

i=10;
j=20;

if (i < j)
    printf("i is kleiner dan j\n");
else
    printf("i is groter dan j\n");
}
```

De uitvoer van het programma is:

i is kleiner dan j

Het conditionele statement ziet er zo uit:

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

of

if (expressie)
{
    ...meerdere statements;
}
else
{
    ...meerdere statements;
}

of

if (expressie)      /* als ... */
{
    ...meerdere statements;
}
else if (expressie) /* anders als ... */
{
    ...meerdere statements;
}
else                /* anders ... */
{
    ...meerdere statements;
}
```

Hieronder volgen voorbeelden met de andere relationele en logische operatoren:

```
if (a == 0)
    printf("a is gelijk aan 0"); /* let op == */

if (a != 0)
    printf("a is ongelijk aan 0");
```

In C is waar (true) gelijk aan 1 en onwaar (false) is 0.  
Hiermee worden de volgende conditionele statements duidelijk:

```
if (a)
    printf("a is gelijk aan 1 dus waar");

if (!a)
    printf("a is ongelijk aan 1 dus onwaar");
```

dit is gelijk aan:

```
if (a == 0)
    printf("a is gelijk aan 0 dus onwaar");
```

Met && en || kunnen meerdere vergelijkingen gemaakt worden:

```
if (x > 0 && y > 0)
    statement; /* beide waar */

if (bit1 == 0 || bit1 == 1)
    statement; /* een van beide waar */
```

LET OP: In het laatste voorbeeld is de 'if' al waar als slechts 1 van de expressies waar is. Als bit0 dus inderdaad gelijk aan 0 is, dan wordt niet meer gekeken of bit1 gelijk is aan 1. Dit is logisch, het betreft hier een 'of' en geen 'en' expressie.

### 7.3 Een andere vorm de conditionele expressie

De regel:

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

kan ook zo geschreven worden:

```
res = a<b ? a+b : a-b;
```

In pseudocode (nep C) staat er:

```
IF a<b
THEN      /* ? */
    res=a+b
ELSE      /* : */
    res=a-b;
```

Dit leidt tot korte coding (die echter minder goed leesbaar is).

### 7.4 Samenvatting

- \* Met de relationele en logische operatoren uit paragraaf 7.1 kunnen conditionele (voorwaardelijke) statements gemaakt worden.
- \* De algemene vorm van een conditionele expressie ziet er als volgt uit:

```
if (expressie)
    statement;
```

### 7.5 Opdrachten en vragen

1. Maak een programma dat om een getal vraagt aan de gebruiker en vervolgens vertelt of het getal groter dan, kleiner dan of gelijk aan 16 is. Gegeven is een voorbeeldprogramma dat een getal laat invoeren. Omdat het invoeren van gegevens nog uitvoerig zal worden behandeld moet de lezer nu het voorbeeld gewoon aanhouden:

```
main()
{
int i;

scanf("%d", &i);/* lees een getal in 'i' */
printf("%d\n", i);
}
```

Maak nu de opdracht.

## HOOFDSTUK 8

### 8.1 Herhalingsstatements

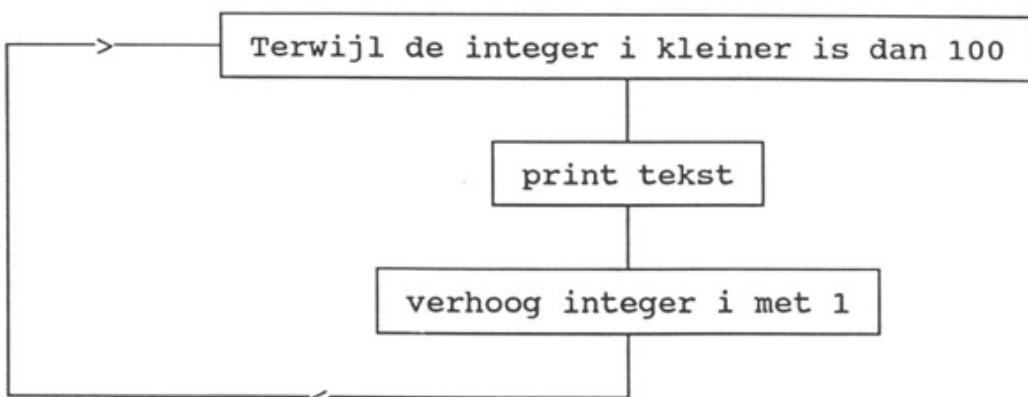
Herhalingsstatements zorgen ervoor dat een bepaalde bewerking of programmadeel meerdere malen herhaald wordt. Het nut van herhalingsstatements zal duidelijk zijn: het bespaart typewerk, het maakt programma's korter en (meestal) overzichtelijker. Het kennen van herhalingsstatements is van groot belang. C kent een aantal manieren om 'lussen' in een programma te maken.

### 8.2 De while lus

```
main()
{
int i=0;

    while(i < 100)
    {
        printf("bever\n");
        i = i + 1;
    }
}
```

While betekent terwijl. Als het programma samengevat wordt in een schema, staat er dit:



### INTERMEZZO: het inspringen van programmaregels

In het programma wordt bij while ingesprongen (while staat drie spaties of een tab naar rechts). Het is een zeer goede gewoonte een vaste manier van inspringen te hanteren. Naast veel commentaar in een programma, maakt een vaste manier van indentatie (inspringen) een programma ook veel leesbaarder.

In de body van while kunnen meerdere commando's worden opgenomen door deze commando's te laten voorafgaan en te laten eindigen door een accolade. Is de while lus echter gebruikt voor de alom bekende wachtlus, dan zijn er geen accolades nodig, zoals onderstaand voorbeeld laat zien:

```
while(i<10000)
    i=i+1;
```

of op 1 regel:

```
while(i<10000) i=i+1;
```

of nog korter:

```
while(i<10000) i++;
```

Dat laatste i++ ziet er vreemd uit, maar is een veel gebruikte verkorting van i=i+1. Dit wordt later uitgebreider behandeld.

### 8.3 Het for statement

Het vorige programma (je weet wel, 100 keer die bever die over het scherm rent) kan ook met het for statement gemaakt worden:

```
main()
{
int i;

for(i=0; i<100; i=i+1)
    printf("bever\n");
}
```

Het for statement is te splitsen in drie delen:

```
for(deel1; deel2; deel3)
```

deel1: Het initialisatiegedeelte: Hier wordt de variabele waarmee geteld wordt op een waarde gezet. In het voorbeeld wordt 'i' op 0 gezet.

deel2: Het vergelijkingsgedeelte: In dit deel wordt vergeleken of de variabele waarmee geteld wordt al een bepaalde waarde bereikt heeft. In het bovenstaande programma wordt gekeken of 'i' kleiner is dan 100, en indien dat waar is, wordt er nogmaals op het scherm geprint.

deel3: Het teldeel: De telvariabele wordt hier veranderd. Dat veranderen kan elke bewerking inhouden (optellen, aftrekken, vermenigvuldigen etc.) zolang er maar in de richting van het getal veranderd wordt dat in het vergelijkingsdeel staat. In het voorbeeld houdt dat in dat i begint op 0, er telkens 1 bij 'i' wordt opgeteld zolang 'i' kleiner is dan 100.

Voor het for statement geldt ook weer dat meerdere commando's uitgevoerd kunnen worden door ze te omsluiten met accolades. In het bovenstaande voorbeeld was dat niet nodig omdat er maar één functie werd aangeroepen (printf()). Wat ook mag is het opnemen van een herhalingsstatement binnen een herhalingsstatement, men spreekt dan van een lus binnen een lus (binnen een lus, binnen een lus, etc.).

#### 8.4 Het do-while statement

Soms is het handig om een bewerking minstens 1 keer uit te voeren alvorens te gaan testen of die bewerking nogmaals moet worden uitgevoerd, dit kan met het do-while statement.

```
main()
{
    int x=0;

    do
    {
        printf("test\n");
        x=x+1;
    }
    while (x < 4);
}
```

Let goed op de puntkomma achter het laatste ronde haakje van while

#### 8.5 Eeuwige lussen

In sommige gevallen is het nodig om iets oneindig keer te laten uitvoeren, men springt dan uit de lus of misschien moet het programma wel eeuwig draaien (?). De volgende varianten van while en for doen dat:

```
for(;;)
    printf("Ik stop als de stoppen stoppen.\n");

while(1)
    printf("Ik stop als de bliksem inslaat.\n");
```

Een andere mogelijkheid (eigenlijk zou dit bij de wet verboden moeten zijn aangezien het tot nare systeem crashes kan leiden) is:

```
main()
{
    printf("eewig....\n");
    main();
}
```

Hier kan een 'stack overflow' de pret verstoren.

N.B. Eeuwigdurende programma's kunnen meestal gestopt worden door CTRL-C in te drukken.

### 8.6 Samenvatting

```
* while (expressie) statement;

    i=0; /* VERGEET DEZE INITIALISATIE NOOIT! */
    while (i<1000)
    {
        printf("Rieta\n");
        ++i;
    }

* for (expressie) statement;

    for (i=0; i<1000; i++)
        printf("Rieta\n");

* do statement while (expressie);

    i=0; /* VERGEET OOK DEZE INITIALISATIE NIET! */
    do
    {
        printf("Rieta\n");
        ++i;
    }
    while (i<1000);
```

## 8.7 Opdrachten

1. Maak een programma dat de tafels van 1 t/m 10 op het scherm afdrukt. Tip: maak gebruik van een lus binnen een lus.
2. Maak een programma dat getallen inleest en deze bij elkaar optelt. Het programma stopt zodra er -1 wordt ingevoerd of wanneer de som van alle ingevoerde getallen groter is dan of gelijk is aan 255. Probeer het zo probleem zo kort mogelijk op te lossen, bijvoorbeeld door het gebruik van |||. Maak weer gebruik van de scanf() functie (zie 7.4).
3. Druk op het scherm de getallen 1 t/m 1000 af.
4. Maak een programma dat telt van 1 t/m 100. Druk 'onderweg' de tientallen af.
5. Maak een programma dat vraagt om een getal onder de 20. Als de gebruiker een verkeerd getal intikt, dan wordt hij/zij gewaarschuwd en stopt het programma. Als het getal wel onder de 20 is, dan moet je je eigen naam zoveel keer laten afdrukken als ingetikt werd. Dus als er 5 ingevoerd was, dan moet je naam 5 maal afdrukkt worden.

Een programma laten stoppen gebeurt door de aanroep van de functie exit(0). exit() met als parameter een ander getal dan 0 betekent (bij de meeste compilers) dat het programma vanwege een foutmelding gestopt is.

```
main()
{
    int i = 0;

    if (i == 1) then
    {
        printf("**** All Systems Ready ***\n");
        exit(0);
    }
    else
    {
        printf("**** General Failure ***\n");
        exit(-1);
    }
}
```

6. Maak een programma dat vraagt om twee getallen. Check of het eerste ingevoerde getal kleiner is dan het tweede. Als dit niet waar is, stopt het programma met een waarschuwing. Als de invoer wel correct is, dan telt het programma alle getallen tussen de twee getallen op. Druk de twee getallen en het resultaat af op het scherm.

7. Maak een programma dat vraagt hoeveel getallen er ingevoerd moeten worden. Laat de gebruiker vervolgens dit aantal getallen daadwerkelijk invoeren. Als alle getallen zijn ingevoerd, moet het gemiddelde op het scherm worden afgedrukt worden en wel als volgt:

Aantal ingevoerde getallen: ...

Totaal: ...

Gemiddelde: ...

## HOOFDSTUK 9

### 9.1 De switch en break statements

Als een variabele vele waarden kan aannemen en het programma op de verschillende waarden anders moet reageren, dan kan een hele rij 'if statements' gebruikt worden maar mooier is het switch statement:

```
main()
{
    int code;

    printf("Voer een getal in >");

    scanf("%d", &code);

    switch(code)
    {
        case 1: printf("\nJe voerde 1 in.\n");
                  break;

        case 2: printf("\nJe voerde 2 in.\n");
                  break;

        default: printf("\nJe voerde geen 1 of 2 in.\n");
                  break;
    }
}
```

Het programma vraagt om een getal, het getal wordt in de variabele 'code' opgeslagen en vervolgens wordt met het switch statement een aantal vragen gesteld. Als 'code' de waarde 1 of 2 heeft dan wordt dat ontdekt en als 'code' een andere waarde heeft, dan wordt dat ook ontdekt. De vertaling van default is standaard: Standaard wordt er geprint "Je voerde geen 1 of 2 in" als code ongelijk is aan 1 of 2.

Denk erom een 'case' af te sluiten met break!

Als je dit vergeet, dan worden de programmaregels achter de volgende case uitgevoerd net zolang totdat een break wordt tegengekomen! Dit kan tot zeer vreemde resultaten leiden. Soms echter kan het toch handig zijn zoals het volgende programma-fragment toont:

```
case KEYPAD_RETURN_TOETS: /* <return> op numerieke padje */

case RETURN_TOETS: printf("\n"); /* gewone <return> */
                    break;
```

## 9.2 Nogmaals het break statement

Het break statement duikt niet alleen op in combinatie met switch maar ook 'los', zoals onderstaand programmafragment illustreert:

```
i=0;
while (i < 10000)
{
    if (i == 1001)
        break;
    printf("zwaardvis\n");
    i=i+1;
}
```

Dit (op zich vrij nutteloze) programmafragment zou tot 10000 tellen en zwaardvis printen als er niet een 'if' in zou staan die de lus onderbreekt als *i* gelijk is aan 1001. De lus wordt gestopt en het programma gaat verder onder de sluitaccoade van het while statement.

## 9.3 Het continue statement

Dit statement wordt alleen behandeld om volledig te zijn. Gebruik het liever niet, het is een tot verwarring leidend commando:

Onderstaand programma berekent de som van ingevoerde getallen mits ze positief zijn. Als de invoer 0 is, stop het programma.

```
main()
{
int som, getal;

som=0;
do
{
    scanf("%d", &getal);
    if (getal <= 0)
        continue; /* ga door onder */
                  /* do-while lus */

    som = som + getal;
}
while (getal != 0); /* let op de ; */

printf("De som = %d\n", som);
}
```

#### 9.4 Samenvatting

```
* switch(waarde)
{
    case 1: statement_uitgevoerd_als_waarde_is_1;
    break;

    case 2: statement_uitgevoerd_als_waarde_is_2;
    break;

    default: statement_indien_waarde_niet_wordt_gecheckt;
    break;
}

* default is niet verplicht, de break wel.

* een programmalus kan onderbroken worden met break
```

#### 9.5 Opdrachten

1. Maak een programma met switch en break dat vraagt om een pincode (de code die o.a. bij de giromaatpas nodig is) en vervolgens de naam en het saldo op het scherm afdrukt dat bij die pincode hoort. Als het saldo negatief is (de klant staat 'rood') dan moet worden afdrukkt "Onvoldoende saldo voor transactie". De volgende gegevens moeten gebruikt worden:

pincode	naam	saldo
1798	J. Blom	f.+ 10,00
8798	Y. Boots	f.- 250,00
7418	E. Bouman	f.+ 1700,00
4991	D. Espinoza	f.- 3600,36
6733	M. Hoedjes	f.+ 250,00
5234	L. Impink	f.+ 7500,20
7897	D. van Winsum	f.- 25600,00

2. Maak een programma met een 'eeuwige lus' (zie paragraaf 8.5) waarin om een getal wordt gevraagd. Alle getallen worden bij elkaar opgeteld totdat een negatief getal wordt ingevoerd. Dan moet het programma stoppen en de som afdrukken.



## HOOFDSTUK 10

### 10.1 Increment en decrement

Om een variabele met 1 te verhogen of te verlagen kan worden genoteerd, zo weten we reeds:

```
i = i + 1;
```

Korter is:

```
i++;
```

Het nut hiervan wordt duidelijk als een lange expressie verhoogd of verlaagd moet worden. De expressie hoeft dan maar 1 maal ingetikt te worden.

Deze expressie...

```
(p*((x*x)/(a*a)) + q*((y*y)/(b*b))) =  
(p*((x*x)/(a*a)) + q*((y*y)/(b*b))) + 1;
```

is langer dan deze, maar doet hetzelfde als...

```
++(p*((x*x)/(a*a)) + q*((y*y)/(b*b)));
```

Er is een verschil tussen `i++` en `++i`, dit heet respectievelijk **post-** en **pre-increment** (**post** betekent 'na', **pre** betekent 'voor' en **increment** betekent 'ophogen').

```
a = 10;
```

```
b = a++;
```

'b' zal nu de waarde 10 hebben en niet 11, dat doet dit programma fragment:

```
a = 10;
```

```
b = ++a;
```

Nu zal eerst 'a' worden opgehoogd en daarna pas zal 'b' gelijk worden aan 'a'.

Hetzelfde geldt voor post- en pre decrement:

```
--a; /* pre decrement */
```

```
a--; /* post decrement */
```

Als een variabele bij een andere variabele opgeteld moet worden, kan dat als volgt:

```
a = a + b;
```

Korter is:

```
a += b;
```

Dit kan ook met de andere operatoren:

```
a -= b; /* trek b van a af */
```

```
a *= b; /* vermenigvuldig a met b */
```

```
a /= b; /* deel a door b */
```

Let ook hier weer op de haakjes:

```
a -= b + c
```

komt overeen met

```
a = a - (b + c)
```

en niet met

```
a = a - b    en vervolgens    a = a + c
```

## 10.2 Spelen met bits

Koren op de molen van de programmeur die tot z'n oksels in de machine graaft, zijn de operatoren om bits te manipuleren.

& bitsgewijze en (and)  
| bitsgewijze of (or)  
^ exclusief of (eor of exor)  
<< schuiven naar links  
>> schuiven naar rechts  
~ 1-complement (golfje; meestal boven links boven op het toetsenbord te vinden of anders naast de RETURN toets).

Om de werking van deze operatoren duidelijk te maken, volgen een aantal voorbeelden:

Voorbeeld: maak alleen de bits die in beide getallen 1 zijn, 1 en zet de rest op 0:

```
getall1 10001101
getall2 10101010
          ----- &
resultaat 10001000
```

In C:

```
resultaat = getal1 & getal2;
```

Voorbeeld: check of bit 3 van een getal 1 is:

```
if (getal & 8)
    printf("bit = 1");
else
    printf("bit = 0");
```

Voorbeeld: zet bit 2 op 1 ongeacht zijn oude waarde:

```
qetal = qetal | 4;
```

of

```
getal := 4;
```

Voorbeeld: vermenigvuldig een getal met 2 door het getal 1 bit naar links te schuiven:

```
qetal = qetal << 1;
```

of

```
getal <<= 1;
```

Voorbeeld: inverteer (keer om) een getal:

```
getal      10101010
~getal     01010101 /* alle 0'en zijn 1'en geworden */
                  /* en vice versa */
```

### 10.3 De komma operator

De komma operator kan expressies onderling scheiden. Een voorbeeld:

```
som=0;  
while (scanf("%d", &getal), getal>0)  
    som += getal;
```

→ deze komma bedoelen we.

Dit programma leest getallen in en houdt het totaal bij in de variabele 'som' totdat een getal wordt ingetikt kleiner of gelijk aan 0. De komma tussen de haakjes van while scheidt de twee expressies `scanf()` (een functie is in C ook een expressie!) en `getal>0`.

#### 10.4 Associativiteit

Al enige malen is er op gewezen dat het gebruik van haakjes in expressies belangrijk kan zijn. Met haakjes is de volgorde waarin een expressie wordt uitgevoerd, te 'sturen'. Een voorbeeld maakt dit duidelijker:

```
getal = x * y + 1;
```

In deze expressie wordt eerst x met y vermenigvuldigd en daarna pas wordt bij het produkt 1 opgeteld. De zaken veranderen als er haakjes ingeplaatst worden:

```
getal = x * (y + 1);
```

Nu zal eerst 1 bij y opgeteld worden en daarna pas de som met x vermenigvuldigd worden.

Het zal nu duidelijk zijn dat de vermenigvuldigingsoperator (\*) een hogere prioriteit heeft dan de optellingsoperator (+). Alle C operatoren zijn te rangschikken naar prioriteit van hoog naar laag:

```
() [] -> .
!
~ ++ -- - (type) * & sizeof /* alle unair */
*
/
-
<< >>
< <= > >=
== !=

&
^
|
&&
||
?:
= += -= *= /= <<= >>= &= ^= |=
,
```

Ondanks dat nog niet alle operatoren besproken zijn. volgt hieronder een korte samenvatting van de betekenis:

```
! logische ontkenning (not)
~ 1-complement
++ increment
-- decrement
- min
+ plus
* maal
/ gedeeld door
% rest bij integer deling
<< schuiven naar links van bits
>> schuiven naar rechts van bits
< kleiner dan
```

```
>    groter dan
<=   kleiner dan of gelijk aan
>=   groter dan of gelijk aan
==   gelijk aan
!=   ongelijk aan
&   bitsgewijze en (and)
|   bitsgewijze of (or)
^   bitsgewijze exclusieve of (eor)
&&  logische en (and)
||  logische of (or)
?:  conditionele expressie
=   assignment
+=  plus-operator
,   komma operator
```

In de tabel staat dat bijv. `++` een unaire operator is. In dit woord zit het woord `un-`, dit komt van het Latijnse 'unus' hetgeen één (1) betekent. Andere zijn binair (bi = twee) of tertiair (tertius is Latijn voor drie).

voorbeelden:

```
unaire operator  ++c;
binaire operator  som = x * y;
tertiaire operator  x = (y == TRUE) ? 1 : 0
```

Deze laatste expressie dient gelezen te worden als:

```
IF y=TRUE
  THEN
    x=1
  ELSE
    x=0;
```

#### 10.5 Samenvatting

- \* `++` is een increment operator, `--` een decrement operator
- \* `++som` is gelijk aan `som=som+1`
- \* `++i` heet pre-increment en `i++` post-increment
- \* met `&, |, ^` zijn bits te manipuleren, bijv.  
`a = a | 7; /* zet bit 0 t/m 2 op '1' */`

```
* if (y == FALSE) x=0;  
else x=1;
```

kan verkort worden genoteerd als:

```
x = (y == FALSE) ? 0 : 1
```

```
* met een komma operator kan een for statement als volgt  
worden 'uitgebreid':
```

```
for (i=0, j=10; i<j; i++; j--)
```

```
* de prioriteit van elke C operator is vastgelegd in een  
tabel.
```

## 10.6 Opdrachten

1. Maak een programma dat vraagt om een 2 getallen, beide kleiner dan 256. Het tweede getal moet 'geAND' worden met eerste en dit antwoord moet worden afgedrukt.
2. Maak een programma dat een floating point getal probeert te schuiven met << of >>. Kijk goed wat de compiler doet!
3. Met de kennis dat bij een negatief getal altijd het laatste (meest linkse) bit '1' is en bij positieve getallen '0', kan je een negatief getal positief maken door:
  - a) Door het getal 1 bit naar links te schuiven en daarna weer naar rechts, het tekenbit is er nu uitgeschoven.  
of (en dit verdient de voorkeur) door:
  - b) Door het getal te ANDen met een 'masker' waarin alle bits die niet op nul gezet moeten worden '1' zijn en het bit (in dit voorbeeld het laatste) '0' is, bijv.:

Maak bit 2 '0', en laat alle andere bits met rust:

	bit 7    bit 0
getal	10011101
masker	11111011
	-----& (AND)
resultaat	10011001    /* bit 2 is '0' en was '1' */

Maak nu een programma waarin de beide methoden toegepast worden (beschreven onder a en b).

4. Schrijf een programma dat een decimaal getal omzet naar zijn binaire equivalent:

Geef decimaal getal: 9  
Binaire equivalent: 1001

## HOOFDSTUK 11

### 11.1 Zelf functies maken

In de vorige hoofdstukken is al enige malen het begrip functie ter sprake gekomen. Zo werd printf() een functie genoemd en main() ook. Om getallen in te voeren werd de functie scanf() gebruikt. Een functie, zo staat in paragraaf 2.1, ziet er altijd als volgt uit:

```
functienaam(parameters)
{
    het programma...
}
```

Tot nu toe zijn in de voorbeeld programma's alleen systeem-functies, zoals printf(), gebruikt. In het onderstaande programma wordt zelf een functie gemaakt:

```
main()
{
int a,b;

    a=10;
    b=20;

    som(a,b);
}

som(s1, s2)
int s1,s2;
{
    printf("som = %d", s1+s2);
}
```

De functie die zelf geschreven is, heet som(). De parameters van deze functie zijn 's1' en 's2'. Onthoudt goed dat de parameters die aan som() worden doorgegeven, hier 'a' en 'b', van hetzelfde type moeten zijn als het type van de variabelen in de functie som() ('s1' en 's2'). Alle variabelen zijn in dit voorbeeld van het type int. Als een functie aangeroepen wordt met andere typen dan in de functie staan dan moeten er eerst maatregelen genomen worden om toch de functie de juiste getallen te leveren. Dit heet 'casten' en dit houdt in dat voordat bijv. floats naar de functie som() gestuurd worden, ze tot int moeten worden 'omgebouwd'.

Stel dat de functie som() met twee floats aangeroepen wordt. Om er voor te zorgen dat de compiler geen fouten maakt (een hele slimme compiler heeft casting eigenlijk niet nodig) wordt het programma als volgt verandert:

```
main()
{
float a,b;

    a=10.0;
    b=20.0;

    som((int)a,(int)b);
}

som(s1, s2)
int s1,s2;
{
    printf("som = %d", s1+s2);
}
```

Het casten bestaat dus uit het laten voorafgaan van (int) aan de floating point variabelen 'a' en 'b'. Van 'a' en 'b' zal een float worden gemaakt, d.w.z. afgerond tot een geheel getal en daarna pas verzonden worden naar de functie som() die graag ints wil hebben.

Casting kan ook gebruikt worden bij het assigneren (toekennen) van waarden aan variabelen:

```
main()
{
int i;
long l;

    i = 10;
    l = (long)i;
}
```

In dit voorbeeld zal de integer 10 omgevormd (gecast) worden tot een long. Bij het gebruiken van systeemfuncties is dus niet alleen van belang te weten hoe de functies heten en wat ze doen maar zeker zo belangrijk is te weten met wat voor typen ze aangeroepen moeten worden. Het aanroepen van een functie die 4 longwords nodig heeft met 4 bytes, kan tot gevolg hebben dan deze functie 4 longwords krijgt met als waarde de bytes die geheel links in het longword liggen:

```
BYTE xpos, ypos, breedte, hoogte;

xpos = 0x10;
ypos = 0x20;
breedte = 0xff;
hoogte = 0x80;

OpenWindow(xpos, ypos, breedte, hoogte);
```

OpenWindow verwacht longwords maar de programmeur maakt hier een onfortuinlijke fout want het Operating System gaat nu een window openen met de volgende parameters (hetgeen in het gunstigste geval zal leiden tot een foutmelding maar vaker tot een 'crash' van het systeem):

```
OpenWindow(0x10000000, 0x20000000, 0xff000000, 0x80000000);
```

De programmeur van dit stuk ellende had het op deze manier moeten aanpakken:

```
OpenWindow((LONG)xpos, (LONG)ypos, (LONG)breedte,  
          (LONG)hoogte);
```

### 11.2 Het teruggeven van een waarde door een functie

Het grote nut, en de schoonheid, van functies is dat een goede functie voor meerdere doeleinden geschikt is en als zodanig op meerdere plaatsen in een programma aangeroepen kan worden. Vaak is het handig dat een functie een aantal parameters toegeleverd krijgt, hiermee aan de slag gaat en na verwerking het antwoord teruggeeft. De vernieuwde functie som() doet dit:

```
main()  
{  
    int a,b, antw;  
  
    a=10;  
    b=20;  
  
    antw = som(a,b);  
  
    printf("De som van %d en %d is %d\n", a,b,antw);  
}  
  
som(s1, s2)  
int s1,s2;  
{  
    som = s1 + s2;  
    return(som);  
}
```

De return in de functie som() levert de berekende waarde terug aan het hoofdprogramma. Dit antwoord wordt toegekend aan de variabele 'antw'.

Met het return statement kan ook zonder een waarde terug te geven uit een functie gesprongen worden:

```
functie()
{
    if (NietGoedGeldTerug)
        return;                                /* spring terug naar */
                                                /* aanroeper van          */
                                                /* functie()              */
```

### 11.3 De ANSI C manier van functies schrijven

Het ANSI (American National Standards Institute) heeft een complete beschrijving van de taal C gemaakt en de makers van C compilers houden zich meestal aan deze gestandaardiseerde C. Dit boek houdt zich daar ook aan echter bij de functies is dit tot nu toe, om niet teveel ineens uit te hoeven leggen, achterwege gelaten. Van nu af aan in het boek zal de ANSI notatie gebruikt worden.

De som() functie, die een int teruggeeft wordt dan zo genoteerd:

```
int som(s1,s2) /* let op de type aanduiding int voor de */
int s1,s2;      /* functienaam           */
{               */
    ...
}
```

Een functie die niets teruggeeft is van het 'void' (Engels voor leeg) type:

```
void print_getal(a)
int a;
{
    ...
}
```

In het oorspronkelijke C, van Brian en Dennis, was het niet nodig om (zoals in Pascal) de functies in de volgorde van aanroep ook in de source code in die volgorde te zetten. Dit betekende dat de compiler niet wist van welke typen de return-waarden waren. Daardoor kon er geen waarschuwing gegeven worden als het return-type verschildde van het type waaraan die waarde toegekend werd. Ook kon er geen waarschuwing gegeven worden als functies met de verkeerde typen aangeroepen werden, simpelweg omdat de typen van de parameters in die functie nog niet bekend waren aan de compiler. Samengevat, een matige compiler zou bij het onderstaande programma geen waarschuwingen gegeven hebben:

```
main()
{
    int a;
    float b;
    long c;

    a = 10;
    b = 3.14;

    c = som(a, b);
}

som(p,q)
int p, q;
{
    int antw;
    antw = p + q;
    return(antw);
}
```

De volgende waarschuwingen zouden gegeven moeten worden:

- \* de functie som() wordt met de een int en een float aangeroepen, terwijl de functie twee integers verwacht.
- \* de functie geeft als return-waarde een int, terwijl in main() deze waarde aan een long wordt toegekend.

Formeel zou dit programma, volgens de ANSI notatie, er zo uit moeten zien. Prent dit goed in het geheugen!

```
int som(p,q)
int p, q;
{
    ...
}

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

Nu weet de compiler, omdat het eerst de functies tegenkomt in het compileer proces en daarna pas het hoofdprogramma, dat som() een int teruggeeft en dat het aangeroepen dient te worden met twee ints. Nu kan de compiler de programmeur waarschuwen voor overtredingen. Uiteraard is het programma nu nog steeds fout, maar de programmeur wordt er nu tenminste op gewezen.

De regel:

```
c = som(a, b);
```

moet zijn (c is een long!):

```
c = (int)som(a, b);
```

want som() levert een int terug en wij willen die zonodig toch in een long proppen. Dat gebeurt door de casting met (int).

Als de programmeur desondanks de functies onder het hoofdprogramma wil hebben maar tevens de ANSI notatie wil gebruiken dan kan dat ook. De functies die aangeroepen worden moeten dan eerst gedeclareerd worden, net als variabelen. Het programma komt er dan als volgt uit te zien:

```
int som(); /* vertel compiler dat som() een int aflevert */

void main()
{
    int a;
    float b;
    long c;

    a = 10;
    b = 3.14;

    c = (int)som(a, b);
}

int som(p,q)
int p, q;
{
    int antw;
    antw = p + q;
    return(antw);
}
```

Dat som() een int aflevert is nu 'globaal' (d.w.z. aan alle functies in het programma bekent. Als alleen één functie, in dit voorbeeld main() dit hoeft te weten, omdat som() alleen daar wordt aangeroepen, dan kan de regel

```
int som();
```

ook in de functie main() gezet worden, in plaats van er boven.

Het verwondert de lezer misschien waarom er over dit onderwerp zo uitvoerig geschreven wordt. Dit is echter van belang omdat een zeer groot deel van de fouten in C programma's ontstaan door het verkeerd heen en weer sturen van variabelen. Zeer belangrijk wordt het als er in een later stadium ook pointers (wijzers naar geheugenadressen) worden verstuurd. Foutieve verzending leidt bijna altijd tot een gecrashed systeem.

LEES DIT HOOFDSTUK NOGMAALS!

#### 11.4 Het aanroepen van een functie vanuit een functie

De functie die de som van twee toegestuurde getallen op het scherm afdrukte, had ook als volgt gesplitst kunnen worden:

```
void druk_getal_af(get)
int get;
{
    printf("%d\n", get);
}

void som(s1, s2)
int s1,s2;
{
int som;
    som = s1+s2;
    druk_getal_af(som);
}
```

Het is dus toegestaan om een functie vanuit een functie aan te roepen. Wat niet toegestaan is, is een functie binnen een functie definiëren:

```
main()
{
int i=10;
    print_getal(i);
    /* de functie print_getal() */
    print_getal(i)      /* dit is VERBODEN */
    int i;
    {
        printf("%d\n");
    }
}
```

### 11.5 Het aanroepen van dezelfde functie vanuit een functie

Hiermee wordt bedoeld dat de aangeroepen functie zichzelf één of meerdere keren aanroept. Een functie die zichzelf aanroept heet een **recursieve** functie. De lezer die het voortgezet onderwijs heeft doorlopen, is bij de wiskundeles vast en zeker lastig gevallen met het begrip faculteit:

$3!$  (lees: 3 faculteit) is de wiskundige notatie van  $3*2*1 = 6$

$0!$  is een uitzondering en levert 1 op

Het berekenen van faculteiten schreewt om in een computerprogramma gegoten te worden en dan met behulp van een recursieve functie:

```
int faculteit(n)
int n;
{
    if (n <= 0)
        return(1);
    else
        return(n * faculteit(n-1));
}

void main()
{
int fact;

fact = 3;
printf("3! = %d\n", faculteit(fact));
}
```

De uitvoer van het programma is:

$3! = 6$

## 11.6 Samenvatting

1. Het is mogelijk om naast systeemfuncties zoals printf() ook **zelf functies** te schrijven:

```
/* voorbeeld van een eigen functie */

void welcome_user()
{
    printf("Hello, welcome to PIN\n");
    printf("The Pentagon Information Network\n");
    printf("\n\nPlease identify yourself...");
}

void main()
{
    welcome_user();

    /* rest van het programma */
}
```

2. Het is van belang dat de functies voorzien worden van een **typeaanduiding** (void, int, float etc.).
3. Het is van belang dat de functies óf gedeclareerd worden zodat het terugkeer type bekend is óf boven de main() functie gezet worden.
4. Functies kunnen andere functies of zichzelf aanroepen (dit zijn **recursieve functies**).
5. Het is verboden een functie binnen een functie te zetten.

## 11.7 Opdracht

1. Maak een programma dat de gebruiker het volgende menu laat zien:

1. Optellen
2. Aftrekken
3. Vermenigvuldigen
4. Delen
5. Stoppen

Na dat de gebruiker een juiste (controleren!) keuze heeft gemaakt, moeten er twee getallen worden ingevoerd. De gekozen bewerking wordt op de twee getallen uitgevoerd. Nadat dit is gedaan, moet het scherm worden schoon gemaakt, het antwoord bovenaan het scherm geprint worden en het menu opnieuw geprint worden. Maak voor elke bewerking een apart functie.



## HOOFDSTUK 12

### 12.1 Geheugenklassen, type, storage class en scope

In C heeft elke variabele en functie twee kenmerken die betrekking hebben op de gegevens die er respectievelijk in bewaard en door teruggegeven kunnen worden. Deze twee kenmerken heten formeel 'type' en 'storage class' (=geheugenklasse).

In hoofdstuk 4 is al uitgelegd dat met **type** wordt aangeduid of een variabele een int, float, double, long of char is. Met **geheugenklasse** wordt echter iets nieuws bedoeld. Het heeft met de mogelijkheden te maken waarmee variabelen gebruikt kunnen worden door diverse programmadelen. Sommige variabelen, zo zal blijken, mogen gebruikt worden door alle functies in een bepaald programma en andere variabelen (met een andere geheugenklasse) kunnen alleen in één functie gebruikt worden. De geheugenklasse definiëert het **bereik** van een variabele, dit heet in het Engels de 'scope' van een variabele. Om het vocabulaire van de gemiddelde informaticus ook in dit boek te gebruiken zal gesproken worden over scope en storage class en niet over bereik en geheugenklasse.

De volgende storage classes kunnen worden gebruikt:

auto    extern    register    static

### 12.2 De auto storage class

De auto storage class is al gebruikt in eerdere programmavoorbeelden zonder dat te weten! Het onderstaande programma declareert een variabele van het type int met de storage class auto:

```
void main()
{
    int getal; /* auto int getal is ook toegestaan maar is */
               /* hoogst ongebruikelijk */

    getal = 1;
}
```

De variabele 'getal' wordt nu 'lokaal voor de functie main()' genoemd. De twee belangrijkste eigenschappen van lokale variabelen zijn dat zij alleen bekend zijn in de functie waarin zij zijn gedeclareerd en dat hun waarde onherroepelijk verloren gaat bij het verlaten van de functie:

```
void print_getal()
{
    printf("%d", getal);
}

void main()
{
int getal = 10;

    print_getal();
}
```

Dit programma levert al tijdens het compileren een foutbericht op in de trant van:

```
"undefined symbol 'getal' used in function print_getal()"
```

Dat is niet zo verwonderlijk want de variabele 'getal' van het type int en storage class auto is lokaal voor de functie main(). De functie print\_getal() heeft helemaal geen idee van 'getal' en daarom klaagt de compiler terecht. Als 'getal' toch bekend moet zijn in print\_getal() dan zal hij doorgegeven moeten worden aan de functie. Lees in het begin van hoofdstuk 11 hoe dat moet.

Let op: auto variabelen hoeven niet op 0 gezet worden door de compiler. Er kan troep in staan, dus moeten zij voor gebruik geinitialiseerd worden!

### 12.3 De extern storage class

In deze paragraaf worden de z.g. 'globale' variabelen behandeld. Globale variabelen zijn van het externe storage class, de aanwijzing 'extern' is echter alleen in bijzondere gevallen nodig. Globale variabelen zijn variabelen die bekend zijn aan alle functies in een programma en verliezen ook niet hun waarde bij het verlaten van een functie.

```
int getal = 10;

void print_getal()
{
    printf("%d", getal);
}

void main()
{
    print_getal();
}
```

Dit programmavoorbeeld is wel correct, al lijkt het erg op het programma in de vorige paragraaf. Nu is de variabele 'getal' echter een globale variabele die zowel in de functie main() alswel in print\_getal() bekend is. 'getal' is hier na de declaratie al op 10 gezet maar kan later door alle functies worden veranderd.

Het gebruik van globale variabelen lijkt erg aantrekkelijk omdat men zich nooit zorgen hoeft te maken of een variabele nu wel of niet gebruikt kan worden in een functie. Er kleven echter bezwaren aan het overmatig gebruik van globalen:

- 1 - Omdat globalen overal van waarde kunnen worden veranderd leidt dat al snel tot onoverzichtelijke programma's.
- 2 - Lokale variabelen mogen dezelfde naam hebben in afzonderlijke functies. Van globale variabelen mag maar 1 versie bestaan. De naamgeving wordt bemoeilijkt bij heel veel globalen.

Als een waarde moet worden doorgegeven aan een functie en in de functie van waarde kan veranderen dan zullen pointers moeten worden gebruikt (zie hoofdstuk 14). Die zijn voor veel beginners nogal lastig en daarom worden dan maar globale variabelen gebruikt. Niet doen.

Een tweede gebruik van de externe storage class is het gebruik van variabelen in z.g. programmamodulen. Grote programma's zijn vaak opgebouwd uit een aantal programmamodulen. Op deze wijze kunnen dan meerdere programmeurs aan één programma werken en later alle modulen aan elkaar plakken. Goede C-compilers kunnen zelfs modulen die al klaar en gecompileerd zijn vastplakken aan nog te compileren modulen zodat de compileertijd drastisch verkort wordt. Een eis bij losse modulen is natuurlijk dat onderling variabelen kunnen worden uitgewisseld.

In het bestand 'hoofdmodule.c' staat:

```
int getal;  
  
void main()  
{  
    int kw;  
  
    getal=10;  
  
    kw = kwadraat(getal);  
  
    printf("10 in het kwadraat is %d", kw);  
}
```

In het bestand 'rekenmodule.c' staat:

```
int kwadraat()
{
extern getal;

    return(getal*getal);
}
```

Deze twee modulen kunnen nu gescheiden worden gecompileerd en zal hetzelfde resultaat hebben als het volgende programma:

```
int getal;

int kwadraat()
{
    return(getal*getal);
}

void main()
{
int kw;

    getal=10;

    kw = kwadraat(getal);

    printf("10 in het kwadraat is %d", kw);
}
```

Het opsplitsen van programma's in modulen is een zeer goede gewoonte. Niet alleen omdat het overzichtelijker is maar zeker ook omdat het compileertijd bespaart. En dat is pas fijn!

#### 12.4 De register storage class

Variabelen worden normaal gesproken ergens in het geheugen van de computer bewaard. Om twee getallen op te tellen moet de microprocessor de volgende stappen nemen:

```
haal 'getall1' op uit geheugen
stop 'getall1' in microprocessor register 'd0'
haal 'getall2' op uit geheugen
stop 'getall2' in microprocessor register 'd1'
tel 'd0' op bij 'd1'
stop 'd1' (de som) in een geheugenplaats waar de som moet komen
```

Dit process wordt in een for-lus met 1000 stappen 1000 keer uitgevoerd. Het is veel efficiënter en dus sneller als de registers van de microprocessor ook worden gebruikt om tussen-tijdse resultaten op te bergen.

```
main()
{
register int i;

    for(i=0; i<1000; i++)
        printf("%d\n", i);
}
```

In dit voorbeeld zal de variabele 'i' in een register worden opgeteld en de functie printf() zal de tussentijdse waarden ook uit het register halen i.p.v. uit een geheugenplaats.

Men dient zich wel te bedenken dat de aanduiding 'register' slechts een advies is aan de compiler. De microprocessor registers zijn druk bezet en kunnen niet altijd worden gebruikt. Declareer daarom register variabelen altijd zo dicht mogelijk bij de plaats waar zij gebruikt worden. Stel dat onze microprocessor 6 registers over heeft per 'blok', dan is het eerste programma trager dan het tweede:

```
langzaam()
{
register int a,b,c,
            x1,x2,x3,
            y1,y2,y3; /* nu al is er een tekort aan */
/* registers want er zijn al 6 */
/* gedeclareerd... */

for(a=x1, b=x2, c=x3; a<100; a++, b--, c=c+10)
;

for(a=y1, b=y2, c=y3; b<1000; a++, b++, c++)
}
}
```

```

snel()
{
register int a,b,c;

{ /* open blok */
register x1, x2, x3;

for(a=x1, b=x2, c=x3; a<100; a++, b--, c=c+10)
;
} /* sluit blok */

{ /* open blok */
register y1, y2, y3;

for(a=y1, b=y2, c=y3; b<1000; a++, b++, c++)
;
} /* sluit blok */
}

```

Deze programmavoorbeelden schenden de regel dat variabelen moeten worden gedeclareerd vlak na de functiedeclaratie! Het declareren midden in een functie mag alleen als een blok wordt aangegeven, dit gebeurt met de accoladen ( { en } ).

Voor de assemblerfreaks is het interessant om eens een kijkje te nemen in de assemblerlisting die de C-compiler genereert om uit te vinden hoe de coding wordt geooptimaliseerd door register variabelen te gebruiken. De assemblerlisting kan soms worden bekeken door na het compileren het assembleren en linken niet uit te voeren. Na het assembleren wordt namelijk door veel compilers dit bestand weer gewist. Overigens kan het handig zijn enige kennis te hebben van machinetaal zodat in de assemblerlisting gecontroleerd kan worden of de compiler zijn werk wel goed heeft gedaan. Sommige expressies waarvan de programmeur denkt dat zij goed zijn, worden soms verkeerd vertaald zodat het programma iets anders doet dan verwacht werd. Als ten einde raad de assemblerlisting wordt bekeken komt de kromme vertaling aan het licht en kan de programmeur de expressie anders opschrijven. Wees niet ongerust als kennis over assembler ontbreekt, goede compilers maken geen vertaalfouten en ze worden gelukkig steeds beter.

### 12.5 De static storage class

Deze storage class heeft twee gebruiken, ten eerste is het mogelijk om met 'statics' lokale variabelen hun waarde te laten behouden ook nadat de functie verlaten is. Dit was onmogelijk met lokale variabelen van het auto storage class. Het tweede gebruik heeft betrekking op externe declaraties. De uitleg is gesplitst in 2 paragrafen.

## 12.6 Lokale statische variabelen

Als eerste zal het gebruik van static worden gebruikt in een functie die een reeks schijnbare willekeurige waarden produceert. Het algoritme is gebaseerd op lineaire congruentie methoden, maar dat mag de pret niet drukken.

In het bestand 'toeval.c' staat:

```
#define FACTOR      25173
#define MODULUS     65536
#define OPHOGING    13849
#define INITIEEL_GETAL 17

int toeval()
{
    static int getal = INITIEEL_GETAL;

    getal = (FACTOR * getal + OPHOGING) % MODULUS;
    return(getal);
}
```

In het bestand 'hoofdmodule.c' staat:

```
void main()
{
int i;

for(i=0; i<100; i++)
    printf("%d\n", toeval());
}
```

Tekens als de functie toeval() wordt aangeroepen zal bij de variabele 'getal' een willekeurige waarde worden opgeteld. De waarde van 'getal' wordt via 'return' teruggegeven maar wordt ook bewaard in de functie toeval(). De declaratie:

```
static int getal = INITIEEL_GETAL;
```

wordt dus maar 1 maal uitgevoerd, anders zou 'getal' steeds weer op de waarde 'INITIEEL\_GETAL' worden gezet.

Het gebruik van een statische variabele verhindert in dit programma het gebruik van een globale (=externe) variabele. De naam 'getal' kan nu nog andere plaatsen gebruikt worden en dat zou onmogelijk zijn geweest als 'getal' een 'global' geweest zou zijn.

## 12.7 Externe statische variabelen

In de al eerder genoemde modulen is het handig als bepaalde variabelen 'privébezit' zijn van die module. De volgende module heeft een externe statische variabele 'getal'. Deze variabele kan alleen in deze module gebruikt worden, andere modulen kunnen er dus niet bij, ook niet door in die modulen een externe declaratie te geven met naam 'getal'.

```
static int getal; /* externe statische variabele */

func1()
{
    /* 'getal' is hier bekend */
}

func2()
{
    /* 'getal' is ook hier bekend */
}
```

De variabele 'getal' is nu privébezit van de module.

## 12.8 Samenvatting

1. Variabelen kunnen op diverse wijzen worden opgeborgen. De **storage class** geeft aan hoe het programma de variabele behandelt en bepaalt tevens de levensduur en bereikbaarheid van de variabele.
2. De **auto** storage class maakt een variabele lokaal voor de functie waarin de variabele wordt gedeclareerd.
3. De **extern** storage class maakt een variabele globaal voor het gehele programma. De variabele is overal bruikbaar en verliest nooit zijn inhoud.
4. De **static** storage class kent twee toepassingen:
  - a) Gedeclareerd binnen een functie is een static variabele te gebruiken als variabele die zijn inhoud niet verliest bij het verlaten van de functie en kan bij de type declaratie eenmalig op een initiële waarde gezet worden (net als iedere andere variabele).
  - b) Buiten een functie gedeclareerd is een static variabele globaal voor de module waarin hij gedeclareerd is maar lokaal voor andere modulen.
5. De **register** storage class probeert bij het declareren van een variabele een microprocessorregister te gebruiken als opslagplaats.

## HOOFDSTUK 13

### 13.1 De diverse typen

In hoofdstuk 4 zijn de diverse constanten behandeld, het waren:

**integers**, gehele getallen.

**floating point getallen**, gebroken getallen met een drijvende komma.

**doubles**, floats met een tweemaal zo hoge precisie.

**karakterconstanten**, karakters tussen apostrofs die een numerieke waarde vertegenwoordigen.

**strings**, geheugengebieden met 1 of meerdere tekens.

Hoe integers, floats, doubles en karakterconstanten worden opgeslagen is reeds uitgelegd. De strings komen nu aanbod, bij het begrip 'array'.

### 13.2 Arrays

Via de declaratie:

```
int verz[10];
```

wordt in het geheugen van de computer een 'worst' van ints gereserveerd en wel:

```
verz[0], verz[1], verz[2] ... verz[9]
```

Het getal tussen de rechte haken ([10]) geeft het aantal elementen aan dat gereserveerd wordt, lopend van 0 t/m 9. Een voorbeeld programma met een array:

```
main()
{
    int i[4];

    i[0] = 1986;
    i[1] = 1987;
    i[2] = 1988;
    i[3] = 1989;

    printf("%d %d %d %d\n", i[0], i[1], i[2], i[3]);
}
```

Arrays kunnen worden gebruikt om ints, floats etc. op te bergen maar ook om strings te bewaren:

```
main()
{
    char WachtWoord[6]; /* reserveer 6 elementen */
    /* te weten 0 t/m 5 */

    printf("Tik uw wachtwoord in: .....\\b\\b\\b\\b\\b");
    scanf("%s", WachtWoord);

    printf("Uw wachtwoord is %s\n", WachtWoord);
}
```

Dit programma verdient uitgebreide aandacht aangezien er vele interessante zaken in gebeuren. Er wordt een array in gebruikt van het type char. Er worden 6 chars gereserveerd dus er kunnen 6 letters, cijfers of andere tekens in bewaard worden (een array van chars is te vergelijken met een \$ variabele in BASIC).

Vervolgens wordt er op het scherm een tekst geprint met aan het einde 5 puntjes. De 5 \b's zorgen ervoor dat de cursor op het eerste puntje komt. Zo weet de gebruiker dat een woord van vijf letters (of minder) verwacht wordt. In dit voorbeeld is het (helaas) nog mogelijk om meer dan 5 tekens in te voeren aangezien de functie scanf() pas klaar is als er op 'return' gedrukt wordt. Later kan een veiliger invoerroutine gemaakt worden want nu kan de arrayruimte overschreden worden en dit kan leiden tot een crash.

In de aanroep van de routine valt het (hopelijk) op dat de & bij de variabele naam WachtWoord mist. Deze is echter alleen nodig bij getalvariabelen en niet bij arrays. Dit zal later uitgelegd worden maar heeft te maken met het feit dat arrays een soort pointers zijn en ints niet.

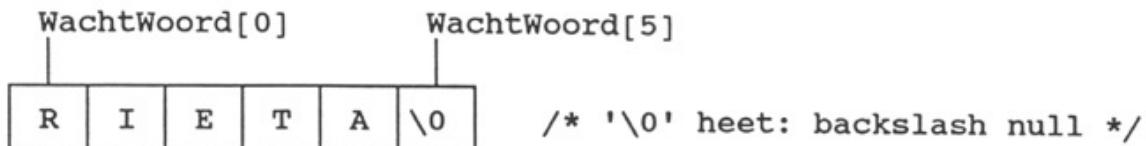
De "%s" in scanf() en printf() geven aan dat een string respectievelijk ingevoerd en afgedrukt moet worden.

Het is misschien opgevallen dat de array niet 5 maar 6 elementen bevat, terwijl er toch maar 5 gevraagd worden. Dit komt omdat in C alle strings afgesloten worden met een zogenoemd NULL teken.

Als ingetikt was als wachtwoord:

RIETA <RETURN>

dan wordt dat als volgt opgeslagen:



Als dus een string zelfgebouwd wordt en afgedrukt moet worden dan zal altijd de karakterconstante '\0' de string moeten afsluiten. Gebeurt dit niet en de string wordt geprint met de functie printf() dan zal printf() net zolang tekens printen totdat het een '\0' tegenkomt. Als er dus in het geheugengebied achter de gereserveerde string zonder '\0' allemaal 'troep' staat dan zal die troep ook geprint worden!

```
main()
{
char naam[5];

    naam[0] = 'E';
    naam[1] = 'r';
    naam[2] = 'i';
    naam[3] = 'k';
    naam[4] = '\0'; /* de afsluitende NULL */

    printf("%s\n", naam); /* print array tot '\0' */
}
```

### 13.3 Meerdimensionale arrays

Net als in BASIC is het mogelijk om in C een array te definiëren die tweedimensionaal is. Een tweedimensionale array is op te vatten als een array van arrays met gelijke lengte. Een tweedimensionale array is het equivalent van de matrix uit de wiskunde:

	0	1	2	3	4	5	6	7
0								
1								
2			*					
3								

De cel waarin het sterretje staat is te omschrijven als cel 2,3. De verticaal geplaatste getallen zijn de rijnummers en de horizontaal geplaatste de kolomnummers. De declaratie van deze matrix gaat in C als volgt:

```
int matrix[4][8]; /* 4 rijen en 8 kolommen */
```

#### 13.4 Samenvatting

1. Een array wordt gedeclareerd door de type aanduiding te noteren, gevolgd door het aantal elementen omsloten door rechte haken.
2. Karakterstrings moeten afgesloten worden door een backslash NULL teken ('\0').
3. Arrays zijn al een soort pointers en daarom is het niet nodig in scanf() de & 'handle' (= adreswijzer) voor een arraynaam te zetten. Pointers worden nog behandeld.

#### 13.5 Opdrachten

1. Maak een programma dat 10 cijfers inleest in een array en vervolgens deze array van voren naar achter afdrukt en van achter naar voren.
2. Maak een programma dat in een tweedimensionale array de tafels van 1 t/m 10 opslaat. Vervolgens moet de gebruiker een tafel kunnen opvragen. Nu kan zonder verdere berekeningen de gewenste tafel geprint worden vanuit de tabel. Het programma stopt door bij de tafelkeuze een 0 in te tikken.

## HOOFSTUK 14

Now all the criminals in their coats and  
their ties  
Are free to drink martinis and watch the  
sun rise  
While Rubin sits like Buddha in a ten-foot  
cell  
An innocent man in a living hell

Bob Dylan

### 14.1 Pointers

In de voorgaande hoofdstukken is al regelmatig het begrip pointer ter sprake gekomen. De variabelen die in een scanf() functie werden gebruikt hadden een handle (&) nodig zodat scanf() het adres kon vinden van die variabele. Bij arrays bleek dit weer niet nodig te zijn. Pointers zijn voor de lezers die alleen in BASIC konden programmeren iets nieuws en vaak moeilijk te begrijpen. Veel mensen hebben de neiging om maar helemaal geen pointers te gebruiken omdat ze het onduidelijk vinden of ze niet begrijpen. Het gebruik van pointers leidt echter tot compactere en snellere coding en soms kan een bewerking alleen met pointers worden uitgevoerd. Stap voor stap zal het raadsel worden onthuld...

INTERMEZZO: In dit hoofdstuk en de komende hoofdstukken zal in grote mate gebruik worden gemaakt van termen als 'window', 'muis', 'slider', etc. De programmeervoorbeelden hebben vaak betrekking op stukken coding die (aangepast) gebruikt zouden kunnen worden op grafisch georiënteerde systemen zoals de Apple Macintosh. Voor de lezers die geen ervaring hebben met dergelijke systemen volgt een korte uitleg over de diverse termen (tip: toch eens achter een Mac kruipen!).

1	Titel	2
	before, how are our brains cheated, who - see thing new - bring forth something already in memory could look back five hundred years to picture in some antique book, since thought w writing. So that I might see what an earlier perfection of your frame : wheather we have i better, or whether the cycle of years has bro point. Of one thing I am sure : the wits of f	3 4

fig.1 - een window

Het bovenstaande figuur is een z.g. window of raam. Het window is een veelvuldig gebruikte manier om op grafische computers informatie te tonen. Een window kan beeldvullend zijn of -zoals hier- slechts een deel van het scherm vullen. Vaak kunnen windows elkaar overlappen zodat sommige ramen geheel of gedeeltelijk onzichtbaar zijn.

Een window kan verplaatst worden door de titelbalk met de muispijl aan te klikken en een muisknop ingedrukt te houden tijdens het slepen en los te laten zodra het raam op de plaats van bestemming is.

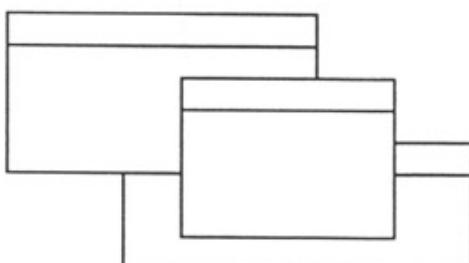


fig.2 - elkaar overlappende windows

Met 1 (zie fig.1, linker hoek) kan het raam gesloten worden door in het vierkantje te klikken met de muispijl. Met 2 kan het raam in één klap beeldvullend of piepklein gemaakt worden. Met 3 kan de inhoud van het raam (hier is dat een tekst) omhoog of omlaag geschoven worden door er met de muisknop op te klikken en de rechthoek waarin de 3 staat te verplaatsen. Het verschuifbare rechthoekje wordt een slider genoemd. Met 4 kan het raam naar keus groter of kleiner gemaakt worden.

De windows kunnen naast tekst en tekeningen ook objecten bevatten die aangeklikt kunnen worden om een bepaalde keuze te kunnen maken:

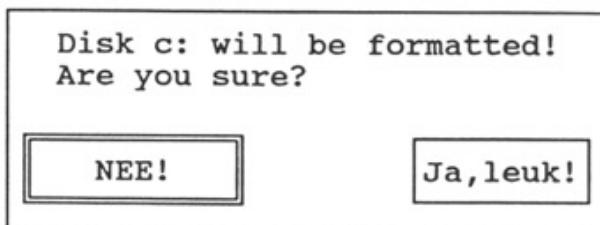


fig.3 schermobjecten

In fig.3 zijn twee objecten die aangeklikt kunnen worden te weten de rechthoek met "NEE!" en de rechthoek met "Ja, leuk!".

Naast de besproken grafische elementen zal gesproken worden over computers waarop tegelijkertijd meerdere programma kunnen draaien (multi-tasking). Computers waarop dit alles mogelijk is zijn o.a. de Apple Macintosh, de Commodore Amiga en de IBM PS/2.

## 14.2 Pointers en adressen

Een pointer is een variabele van een bepaald type (bijv. int) die wijst naar een adres van een andere variabele van hetzelfde type:

Een programma waarin pointers worden gebruikt:

```
main()
{
    int i, j;
    int *ptr_naar_i;

    ptr_naar_i = &i;

    *ptr_naar_i = 20;

    printf("De waarde van i = %d\n", i);

    j = *ptr_naar_i;

    printf("De waarde van j = %d\n", j);
}
```

De uitvoer van dit programma is:

```
De waarde van i = 20
De waarde van j = 20
```

De declaratie:

```
int *ptr_naar_i;
```

moet gelezen worden als 'declareer een pointer met de naam ptr\_naar\_i die naar een integer wijst'.

In de regel:

```
ptr_naar_i = &i;
```

wordt via de handle (&) aan de pointer het adres toegekend van de variabele 'i'.

- \* 'i' is een variabele van het type int.
- \* 'ptr\_naar\_i' is een pointer (wijzer) naar een variabele van het type int en bevat nu het adres waar de waarde van variabele 'i' zich bevindt.

De regel:

```
*ptr_naar_i = 20;
```

moet gelezen worden als 'maak de waarde die zich op het adres dat in de pointer 'ptr\_naar\_i' staat, gelijk aan 20'. Dat betekent dat de variabele 'i' indirect op de waarde 20 wordt gezet. Er had dus ook kunnen staan, via de directe methode:

```
i=20;
```

De regel:

```
j = *(ptr_naar_i);
```

moet gelezen worden als "stop in de variabele j de waarde die te vinden is op het adres dat in de pointer 'ptr\_naar\_i' staat". Ook dit is een indirecte methode om aan 'j' de waarde van 'i' toe te kennen. Er had ook genoteerd kunnen worden:

```
j=i;
```

Kort samengevat, na de declaraties:

```
int i,j;    en    int *ptr_naar_i;
```

kan gesteld worden:

- a) `ptr_naar_i = &i` zorgt ervoor dat het adres waar de waarde van 'i' bewaard wordt, in 'ptr\_naar\_i' gezet wordt.
- b) `*ptr_naar_i = 20` zorgt ervoor dat op het adres die in de pointer 'ptr\_naar\_i' de waarde 20 gezet wordt:

```
*ptr_naar_i = 20    IS GELIJK AAN    i = 20
```

- c) `j = *ptr_naar_i` maakt 'j' gelijk aan de waarde die te vinden is op het adres dat 'ptr\_naar\_i' herbergt:

```
j = *ptr_naar_i    IS GELIJK AAN    j = i
```

De lezer zal (hopelijk!) afvragen wat een pointer nu eigenlijk voor een type is. Het bevat een adres maar is ook van een bepaald type...

- a) De lengte van een pointer is voor alle typen gelijk! Op een PC zal een pointer een 16 bits getal zijn, ongeacht of het wijst naar een int, float of long. Dit komt voort uit het feit dat om een adres te kunnen bewaren minstens 16 bits nodig zijn. Op mini-computers of computers met bijv. Motorola 68000 microprocessor zal een pointer (meestal) 32 bits zijn.
- b) Een pointer heeft een type aanduiding bij zich, niet om zijn eigen grootte aan te geven maar om de grootte aan te geven van het element waar hij naar wijst! Dit dient goed in de oren te worden geknoopt!

### 14.3 Parameteroverdracht via pointers

Een beginnende C programmeur wil een functie schrijven die de waarde van 2 variabelen verwisselt. De eerste variabele krijgt de waarde van de tweede en vice versa. Het onderstaande programma wordt gemaakt:

```
void verwissel();

void verwissel(v1, v2)
int v1, v2;
{
    int temp;
    temp = v1;
    v1 = v2;
    v2 = temp;
}

main()
{
    int a=10, b=20;
    verwissel(a, b);
    printf("%d %d\n", a, b);
}
```

Dit ziet er, voor een bijna-leek, prima uit. Het is alleen jammer dat in C een functie de waarden van variabelen uit een andere functie niet kan veranderen. In dit voorbeeld betekent dit dat de functie `verwissel()` niet bij de ints 'a' en 'b' kan komen.

Met een z.g. geheugenschema kan worden bewezen dat het programma structureel fout is.

a	b	v1	v2	temp	
10	20	10	20	10	initialisatie
		20			tijdens aanroep van <code>verwissel()</code>
		*	*	*	<code>temp = v1</code>
					<code>v1 = v2</code>
					<code>v2 = temp</code>

De '\*'tjes geven aan dat vanaf dat punt de variabelen niet meer bestaan aangezien het 'lokale variabelen' zijn (variabelen die ALLEEN in de functie gebruikt mogen worden). De variabelen 'v1', 'v2' en 'temp' zijn lokale variabelen in de functie `verwissel()`. De variabelen 'a' en 'b' zijn 'lokalen' in de functie `main()`.

De enige juiste manier is geen waarden door te geven maar pointers:

```
void verwissel();

void verwissel(v1, v2)
int *v1, *v2; /* Let op de *v1 en *v2 */
{
    int temp;
    temp = *v1;
    *v1 = *v2;
    *v2 = temp;
}

main()
{
    int a=10, b=20;
    verwissel(&a, &b); /* Let op de handles (&a en &b) */
    printf("%d %d\n", a, b);
}
```

Nu worden aan de functie `verwissel()` geen waarden doorgegeven maar adressen waar de functie `verwissel()` de waarden van 'a' en 'b' kan vinden:

```
temp = *v1; /* geef 'temp' de waarde die staat op het */
             /* adres van de pointer 'v1' */

*v1 = *v1; /* zet in het adres die de pointer 'v1' bevat */
            /* de waarde die staat op het adres die de */
            /* pointer 'v2' bevat */

*v2 = temp; /* zet in het adres die de pointer 'v2' bevat */
            /* de waarde die de variabele 'temp' bevat */
```

Het doorgeven van adressen i.p.v. waarden gebeurt door voor de variabelen een & te plaatsen. De & noemt men een 'handle'.

Variabelen waarnaar met handles wordt gewezen (in dit voorbeeld de variabelen 'v1' en 'v2' moeten altijd pointers zijn!) Dit wordt geregeld door voor deze 'adres ontvangers' een '\*'tje te zetten. Daardoor worden het pointers. Zorg er tevens altijd voor dat pointers van hetzelfde type (int, float etc.) zijn als de variabelen waarvan het adres wordt doorgegeven. De variabelen 'a' en 'b' en de pointers 'v1' en 'v2' zijn in dit voorbeeld van hetzelfde type.

#### 14.4 Pointers naar geheugengebieden

Iets wat voor veel programmeurs die afkomstig zijn van relatief eenvoudige machines zoals de MSX en de '64 een onbekend verschijnsel is, is het reserveren van geheugengebieden. Kijken we echter naar bijv. een IBM PS/2, een Macintosh of een Amiga waarop meerdere programma's tegelijkertijd kunnen draaien dan doemt al snel een probleem op. Als bijv. een tekenprogramma op één van deze machines gestart wordt dan zal dat programma geheugen nodig hebben om het plaatje op te slaan, misschien zelfs twee van deze geheugengebieden voor de 'UNDO' functie. Het programma trekt zich niets aan van andere programma's en plaatst het plaatje vanaf adres \$50000. Nu start de gebruiker ook nog een communicatiepakket omdat hij plaatjes wil 'downloaden' van een BBS (Bulletin Board Service) en deze bekijken in het tekenprogramma. Dit communicatiepakket heeft 32 kB nodig als ontvangstbuffer en plaatst deze (de programmeur was net zo eigenwijs als de programmeur van het tekenprogramma) op adres \$52000.

Ja, dat gaat fout. Het tekenpakket zal of crashen of zal een tekenblad tonen dat vol met troep staat, n.l. de troep die het communicatiepakket op \$52000 schrijft. De programmeurs zullen zich enige huishoudelijke regels eigen moeten maken, te beginnen met het reserveren van geheugengebieden. Het Operating System van de genoemde machines draagt er zorg voor dat gereserveerde geheugengebieden niet gereserveerd kunnen worden door andere programma's. Het is zelfs zo dat nieuwe microprocessoren zoals de Motorola 68030 en de Intel 80386 z.g. MMU's (Memory Management Units) aan boord hebben die er hardwarematig voor zorgen dat het gereserveerde geheugen bij 1 eigenaar blijft en ook nooit overschreven kan worden door andere programma's.

Wat de programmeur van het tekenprogramma had moeten doen is het volgende:

1. Reserveer 32 kB
2. Controleer of het OS (Operating System) dit ook daadwerkelijk nog 'over' heeft. Indien dit niet het geval is, print dan 'Out of memory' of iets dergelijks.
3. Vul het gebied met nullen (de gebruiker heeft graag een wit tekenvelletje).

Omdat het bovenstaande een veelvuldig terugkerende operatie is in vele programma's zal een volledig uitgewerkt voorbeeldprogramma worden behandeld. Er zullen twee stukken geheugen worden gereserveerd, 1 stuk voor het tekenblad en 1 stuk voor het 'undo' tekenblad. Let vooral ook op de uitgebreide foutdetectie. Hierover straks meer!

```

#include <stdio.h>
#include <memory.h> /* dit KAN verschillen per compiler */

ULONG vlaggen = 0L;
#define TEKENBLAD 0x00000001L /* maak er een LONG van */
#define UNDOBLAD 0x00000002L /* idem dito */

UBYTE *teken_blad,
      *undo_blad; /* pointers naar geheugengebieden */

***** functiedeclaraties *****

UBYTE *reserveer_geheugen();
void geef_geheugen_terug();
void vul_geheugen();
void ruim_op();

***** functies *****

UBYTE *reserveer_geheugen(grootte)
ULONG grootte;
{
    return((UBYTE *)malloc(grootte));
}

void geef_geheugen_terug(ptr)
UBYTE *ptr;
{
    free(ptr);
}

void vul_geheugen(ptr, grootte)
UBYTE *ptr;
ULONG grootte;
{
    ULONG i;
    for(i=0; i<grootte; i++)
        *(ptr+i) = 0;
}

void ruim_op()
{
    if (vlaggen & TEKENBLAD)
        geef_geheugen_terug(teken_blad);
    if (vlaggen & UNDOBLAD)
        geef_geheugen_terug(undo_blad);
}

```

```

***** hoofdprogramma ****/
void main()
{
    /* reserveer geheugengebieden */

    teken_blad = reserveer_geheugen(32768L);
    if(teken_blad == NULL)
    {
        printf("Geen geheugen voor tekenblad!\n");
        exit(FALSE);
    }
    vlaggen |= TEKENBLAD;

    undo_blad = reserveer_geheugen(32768L);
    if(undo_blad == NULL)
    {
        printf("Geen geheugen voor undoblad!\n");
        ruim_op();
        exit(FALSE);
    }
    vlaggen |= UNDOBLAD;

    /* maak geheugengebieden schoon */

    vul_geheugen(teken_blad, 32000);
    vul_geheugen(undo_blad, 32000);

    /* hier is plaats voor de rest van het tekenprogramma */

    .....

    /* beëindig programma netjes */

    ruim_op();
}

```

Waarschijnlijk zal deze en de voorgaande bladzijde ezelsoren krijgen en na verloop van tijd ook wat vieze vette vingertjes vertonen als de lezer een vervente C programmeur wordt, het is namelijk een standaard stukje programmatuur. De uitleg:

1. **ULONG vlaggen = 0L;**

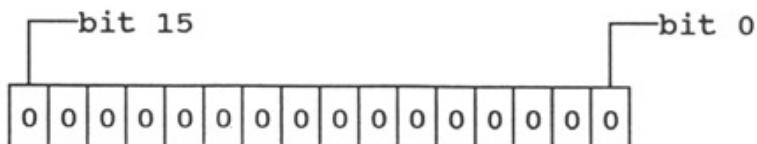
De ULONG 'vlaggen' is een globale variabele en wordt op 0 gezet. De 'L' zorgt ervoor dat er 32 bitjes op 0 worden gezet, ook bij slappe compilers.

2. **#define TEKENBLAD 0x00000001L  
#define UNDOBLAD 0x00000002L**

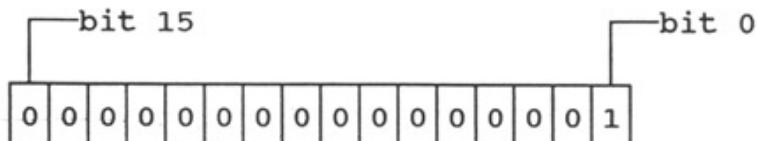
Deze twee defines geven aan '1' en '2' suggestieve namen.

3. `vlaggen |= TEKENBLAD;`

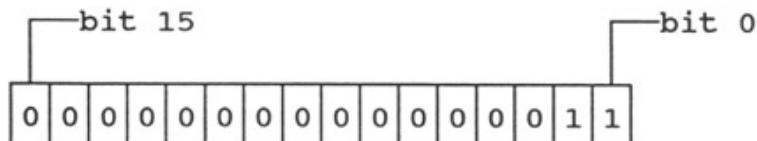
Voordat deze assignment plaats vond, had 'vlaggen' de volgende inhoud:



Na 'vlaggen |= TEKENBLAD' is dat:



En na 'vlaggen |= UNDOBLAD' is dat:



4. De functie `ruim_op()` zorgt er voor dat alle gereserveerde gebieden teruggegeven worden. Wat de functie niet weet is welke gebieden wel en niet gereserveerd zijn. Het kan zijn dat `ruim_op()` werd aangeroepen omdat na enige reserveringen toch niet genoeg vrij geheugen over was. `ruim_op()` moet dan alleen de wel gereserveerde gebieden vrij geven (het vrijgeven van een niet gereserveerd gebied kan n.l. een leuke crash veroorzaken!).

```
if (vlaggen & TEKENBLAD)
    geef_geheugen_terug(teken_blad);
```

'vlaggen & TEKENBLAD' filtert door middel van '& TEKENBLAD' alle andere bits uit de variabele 'vlaggen' en wordt 'TRUE' als het TEKENBLAD bit (bit 0) '1' is. De functie `geef_geheugen_terug()` wordt dus alleen aangeroepen als de expressie 'vlaggen & TEKENBLAD' waar (TRUE) is.

5. `teken_blad = reserveer_geheugen(32768L);`

Deze programmaregel zorgt ervoor dat de pointer die de functie `reserveer_geheugen()` oplevert in de pointer 'teken\_blad' wordt bewaard. Dat `reserveer_geheugen()` een pointer teruggeeft kan men zien aan de functiedeclaratie:

```
UBYTE *reserveer_geheugen();
```

Men dient er dan nog wel voor te zorgen dat dit ook daadwerkelijk gebeurt (en dat gebeurt hier ook via een cast):

```
return((UBYTE *)malloc(grootte));
```

De functie malloc() reserveert een geheugengebied ter grootte 'grootte' en geeft een pointer terug naar het eerste byte van dat gebied. Deze pointer wordt door reserveer\_geheugen() via de return() aan de aanroeper doorgegeven. Er had derhalve ook in het hoofdprogramma main() kunnen staan:

```
teken_blad = malloc(32768L);
```

Omdat dit voorbeeldprogramma het doorgeven van pointers behandeld en omdat het nuttig is alvast te wennen aan het gebruiken van een collectie standaardfuncties die men in ieder programma zal moeten hebben is de functie reserveer\_geheugen() te rechtvaardigen.

6. **\*(ptr+i) = 0;**

Dit stukje coding staat in de functie vul\_geheugen(). Het is hopelijk duidelijk wat er gebeurt. Het '\*'tje zorgt er voor dat de 0 wordt geplaatst op het adres die de pointer 'ptr' bevat. Dit adres is aan vul\_geheugen() doorgegeven via de aanroep:

```
vul_geheugen(teken_blad, 32768);
```

Dat 'teken\_blad' een pointer was, dat was al duidelijk na de declaratie:

```
UBYTE *teken_blad
```

De haakjes rond 'ptr+i' staan er niet vanwege kunstzinnighedsverhogende redenen! Als er zou staan:

```
*ptr+i = 0;
```

dan wordt de waarde die staat op het adres die 'ptr' bevat plus 'i' gelijk gemaakt aan 0. Dat kan dus de volgende bewering opleveren:

```
Stel: *ptr = 5 /* de waarde op het adres in 'ptr' is 5 */
      i = 50;
```

```
*ptr+i = 0 <==> 5+50 = 0 ==> Dat klopt dus niet...
```

Dit is fout. Het moet dus zijn: \*(ptr+i) = 0;

N.B. Voor de functiedeclaratie staat overigens 'void' omdat vul\_geheugen() geen waarde via een return() statement teruggeeft.

```
7. free(ptr);
```

Nette mensen geven iets terug als ze iets geleend hebben. Wij zijn nette mensen dus passen wij deze deugd ook toe tijdens het programmeren. Het voorbeeldprogramma heeft in totaal 65536 bytes gereserveerd en geeft die bij het beëindigen van het programma terug aan het OS. Gebeurt dat niet dan zullen tot aan het afschakelen van de machine of het geven van een 'reset' deze 65536 bytes 'verloren' zijn. Het OS weet namelijk dat dit geheugengebied niet meer aan andere gebruikers (lees: programma's) gegeven mag worden.

De functie free() verwacht een pointer naar een gereserveerd gebied. Intern is een lijstje bijgehouden van gereserveerde gebieden zodat deze pointer voldoende informatie voor free() bevat. De grootte van het gebied haalt hij uit het lijstje.

N.B. Verwacht niet dat het OS een aanval overleeft met een free() naar een niet bestaand geheugengebied. free() dus alleen nadat vast is gesteld dat de pointer ook werkelijk naar iets zinnigs wijst. In het voorbeeldprogramma gebeurd dat als volgt:

```
if (vlaggen & TEKENBLAD)
    geef_geheugen_terug(teken_blad);
```

#### 14.5 Pointers en arrays

Arrays zijn behandeld in hoofdstuk 13, paragraaf 13.2. Daar bleek dat na de declaratie:

```
int verz[10];
```

het mogelijk was om 10 integers op te slaan, in verz[0] t/m verz[9]. In deze paragraaf zal worden aangetoond dat pointers en arrays nogal op elkaar lijken, zo zijn:

```
verz
```

en

```
&verz[0]
```

beide pointers naar het eerste element.

De functie vul\_geheugen() uit de vorige paragraaf kan dus op twee manieren worden aangeroepen:

```
vul_geheugen(verz, 10);
```

of

```
vul_geheugen(&verz[0], 10);
```

De '\*'tjes kunnen bij arrays ook worden gebruikt:

```
*verz = 10      doet hetzelfde als    verz[0] = 10
*(verz+1) = 20  doet hetzelfde als    verz[1] = 20
*(verz+2) = 30  doet hetzelfde als    verz[2] = 30
*(verz+3) = 40  doet hetzelfde als    verz[3] = 40
```

Nu volgt een zeer belangrijke opmerking:

```
Stel: ULONG coords[10]; /* 10 coördinaten */
      int i;
```

Er is een array van longs gedeclareerd. Een long neemt 4 bytes in beslag en de int 2 bytes. Men zou kunnen denken dat men 2 zou moeten optellen om bij de volgende long te komen. Het volgende is echter NIET waar:

```
i = 2;
*(coords+i) = 10L  doet hetzelfde als  coords[1];
```

DIT IS NIET GOED !

'coords' is namelijk gedeclareerd als array van longs. Door bij de pointer naar het eerste element (\*coords) 1 op te tellen, komt men automatisch terecht bij de tweede long van de array. Het programmafragment dient er dan ook als volgt uit te zien:

```
i = 1;
*(coords+i) = 10L  doet hetzelfde als  coords[1];
```

DIT IS WEL GOED !

DUS: Zelfs als een array elementen bevat die groter zijn dan 1 byte, dan geldt de equivalentie \*(array+i) = array[i].

#### 14.6 Arrays van pointers

Moet dat nou? Ja hoor, arrays van pointers kunnen heel handig zijn, al klinkt het wat onheilspellend. Het volgende programma slaat in een array de pointers op van 3 gereserveerde geheugen-gebieden en gebruikt deze array om het gereserveerde geheugen later weer terug te geven:

```
void FreeMem(ptr)
char *ptr[];
{
int i;
for(i=0; i<3; i++)
    free(ptr[i]);
}
```

```
void main()
{
    char *ptr[3];

    ptr[0] = (char *)malloc(8192);
    ptr[1] = (char *)malloc(16384);
    ptr[2] = (char *)malloc(32768);

    FreeMem(ptr);
}
```

In werkelijkheid zou dit programma enige foutcontrole zou moeten doen! De uitleg:

1. De declaratie 'char \*ptr[3]' bouwt een array op waarin 3 pointers bewaart kunnen worden. De 'char' type-aanduiding vertelt de compiler dat de pointers wijzen naar een geheugengebied waar 8 bits tekens (bytes) worden opgeslagen.
2. Na de assignment 'ptr[0] = (char \*)malloc(8192)' is in het eerste arrayelement een pointer opgeslagen die wijst naar het 8192 bytes grote gebied dat is gereserveerd. Hier moet eigenlijk gechecked worden of malloc() geen NULL teruggeeft.
3. De functie FreeMem() gaat in een for-lus alle arrayelementen af en geef de opgeslagen pointers naar geheugengebieden door aan de 'standaard' functie free().

#### 14.7 Pointers naar functies

Het startadres van een functie kan in C ook worden gebruikt om een functie, net als variabelen, door te geven aan een functie die vervolgens iets met die functie, of variabele, doet. Een (hopelijk) aanschouwelijk voorbeeld komt voort uit de windowomgevingen zoals op Macintosh, Amiga of PS/2. Bij deze machines zit naast de windows soms een z.g. 'slider' of 'scroll bar' waarmee de inhoud van een window (bijvoorbeeld een brief) heen en weer kan worden geschoven omdat de tekst langer is dan past in het window. Stel dat de programmeur twee functies heeft geschreven, 1 om de tekst heen en weer te schuiven en 1 om de 'slider' te bewegen. Zodra het hoofdprogramma ziet dat de gebruiker z'n muispijltje op de slider zet dan wordt de functie aangeroepen om de slider heen en weer te bewegen en aan deze functie wordt een pointer meegegeven van een functie die moet worden uitgevoerd tijdens het heen en weer schuiven. De brief moet namelijk de slider 'volgen' tijdens het schuiven.

Om het programma overzichtelijk te houden, wordt eerst een functie geschreven die in staat is een andere functie aan te roepen..

```
int CallFunction(func)
int (*func)();
{
int result;
    result = (*func)();
    return(result);
}
```

De functie CallFunction() krijgt als argument een pointer naar een functie die een variabele van het type int teruggeeft. Deze variabele wordt via CallFunction() aan de aanroeper van CallFunction() teruggegeven. De opdracht:

```
result = (*func)();
```

heeft als resultaat dat gesprongen wordt naar de functie die staat op het adres dat zich bevindt in de pointer naar een functie. Die pointer heet 'func'.

In programmafragmenten wordt nu het slider programma gegeven:

Het hoofdprogramma:

```
switch(muisactie()) /* muisactie() leest muis uit */
{
    case LINKER_MUISKNOP:
        switch(object()) /* object() kijkt welke slider is */
        {
            /* aan geklikt */

            case SLIDER_1: SchuifSlider(SchuifBrief);
                break;

        }
        break;
}
```

Het SchuifSlider() programma:

```
void SchuifSlider(func)
void (*func)();
{
int positie;

switch(muisactie())
{
    case MUISBEWOGEN: positie = TekenenSlider();
        CallFunction(func(positie));
        break;
}
}
```

1. De opdracht 'SchuifSlider(SchuifBrief)' geeft aan SchuifSlider() een pointer door naar SchuifBrief(). De functie SchuifBrief() is van het type void, dus in de functie-declaratie van SchuifSlider() wordt het functieargument ook als zodanig behandeld.
2. De functie TekenSlider() wist de oude slider op het scherm en tekent hem op een nieuwe positie. De slider beweegt namelijk als de gebruiker de linkermuistoets ingedrukt houdt en de muis naar voren en achter schuift. De nieuwe positie die de slider in neemt op het grafische scherm stelt de positie voor van de cursor in het window waarin de brief staat. De functie TekenSlider() geeft daarom een regelnummer terug die gebruikt wordt om de brief te positioneren.

Het voordeel van het doorgeven van een pointer naar een functie is dit geval evident. Bij een ander window zal met de slider bijvoorbeeld door de lijst met bestanden uit een directory geschoven kunnen worden. Dan zal i.p.v. de functie SchuifBrief() wellicht de functie SchuifDirectory() worden gebruikt als argument voor SchuifSlider(). De functie SchuifSlider() hoeft dan echter niet aangepast te worden en kan dus opgenomen in een bibliotheek met grafische routines waar nooit meer aan gesleuteld hoeft te worden.

#### 14.8 Samenvatting

1. Het volgende voorbeeld moet geheel duidelijk zijn en in het geheugen gegrift worden (de uitvoer is 20):

```
main()
{
    int p, *q;

    p = 10;
    q = &p;           /* q vraagt adres op van p */
    *q = 20;          /* waarde op adres in q wordt 20 */
    printf("%d", p)
}
```

2. Arrays en pointers:

```
verzameling[5] = 1001;
```

mag ook geschreven worden als pointer:

```
*(<verzameling + 5) = 1001;
```

3. Variabelen doorgeven aan een functie en aldaar van waarde laten veranderen:

```
void maak_positief(getal)
int *getal;
{
    if (*getal < 0)
        *getal = *getal * -1;
}

main()
{
int i;

    i = -10;
    maak_positief(&i); /* let op de handle & */
    printf("%d", i);
}
```



## HOOFDSTUK 15

### 15.1 printf()

Het printf() commando lijkt wat ouderwets in een wereld vol grafische schermen met windows en icons maar blijkt nog steeds een waardevolle functie. Vele programmeeromgevingen, of dit nu een geïntegreerde omgeving zoals Turbo C, een terminal omgeving zoals UNIX of een window omgeving op een Amiga is, tijdens het bouwen van software is het vaak wenselijk om de waarden van variabelen af te drukken tijdens het draaien van het programma. Op een PC zal dit printen op hetzelfde scherm gebeuren als waarop het programma zijn uitvoer print en op een Amiga naar een apart window voor 'karaktermode-achtige uitvoer' (geen puntjes of lijnen, alleen letters en cijfers). Dit laatste heeft als voordeel dat de programma-uitvoer (de icons en andere grafische elementen) niet verwoest worden door tussentijds afgedrukt waarden van variabelen.

printf() is al regelmatig gebruikt om zinnen en getallen af te drukken op het scherm en heeft een besturingsstring en een argumentenlijst:

```
printf("elementnummer %d met naam %s", num, naam);  
besturingsstring: "elementnummer %d met naam %s"  
argumentenlijst: num, naam;
```

De % geeft aan dat daar achter 1 of meerdere tekens komen die aangeven hoe de argumenten afgedrukt dienen te worden. De %d en %s worden 'conversietekens' genoemd. De volgende conversietekens staan tot onze beschikking:

Conversieteken	Betekenis	Voorbeeld
d	decimale integer	31
o	octale integer (integer afgedrukt in 8-tallig stelsel i.p.v decimaal (=10-tallig) stelsel.)	37=31 decimaal
x	hexadecimale integer (16-tallig stelsel)	1f=31 decimaal
u	unsigned decimale integer	31
e	floating point getal met exponent	6.67e-2
f	floating point getal zonder exponent	0.06670000

vervolg conversietekenlijst:

Conversieteken	Betekenis	Voorbeeld
g	korste floating point uitvoer wordt gekozen (met of zonder exponent)	0.0667
c	enkel teken	a
s	string	Riets

Met %c, %d, %o en %x is de volgende uitvoer mogelijk:

Stel: **kar = 'a'**  
**getal = 1001**

Conversieteken	Expressie	Uitvoer	Uitleg
%c	kar	a	1 teken geprint
%2c	kar	a	veld 2 tekens breed en rechts opgelijnd
%-4c	kar	a	veld 4 tekens breed en links opgelijnd
%d	getal	1001	1 getal geprint
%6d	getal	1001	veld 6 tekens breed en rechts opgelijnd
%-6d	getal	1001	veld 6 tekens breed en links opgelijnd
%-d	getal	-1001	min voor getal
%06d	getal	001001	opgevuld met nullen z.g. leading zeros
%-06d	getal	-01001	negatief getal, opgevuld met nullen

De '| 'en geven de linker- en rechtergrens aan van de uitgevoerde tekens.

longs kunnen worden geprint door voor de 'd', 'x' etc. een 'l' te zetten:

```
groot_getal = 78934567L; /* grote L maakt een long */
printf("druk een long af %ld", groot_getal);
```

Met %e, %f, en %q is de volgende uitvoer mogelijk:

Stel: g = 9.8066 /\* standaard valversnelling \*/  
 me = 0.00054858 /\* rustmassa electron in u \*/

Conversieteken	Expressie	Uitvoer	Uitleg
%f	g	9.8066	standaard floating point uitvoer
.1f	g	9.8	1 teken achter de punt ('komma')
%2.3f	g	9.807	2 tekens voor de punt en 3 achter de punt
%-03.1f	g	-09.8	min en nul(len) voor de punt en totaal 3 tekens breed en 1 cijfer achter de punt
%f	me	0.000549	standaard float uitvoer: precisie van 6 cijfers
%e	me	5.49e-04	korte wetenschappelijke notatie

Met %s is de volgende uitvoer mogelijk:

Stel: str[] = "Hello"

Conversieteken	Expressie	Uitvoer	Uitleg
%s	str	Hello	hele string geprint
%7s	str	Hello	rechts uitgelijnd en 7 karakters breed
%-7s	str	Hello	links uitgelijnd en 7 karakters breed
.4s	str	Hell	4 tekens geprint

vervolg %s opties:

Conversietekens	Expressie	Uitvoer	Uitleg
%7.4s	str	Hell	7 tekens breed en 4 geprint, rechts opgelijnd
%-7.4s	str	Hell	7 tekens breed en 4 geprint, links opgelijnd

## 15.2 scanf()

Ook de functie scanf() heeft een besturingsstring en een argumentenlijst, net als printf():

```
scanf("%s %d %d", str, &getal1, &getal2);
```

De functie scanf() verwacht pointers naar de argumenten dus in de argumentenlijst hebben de integers 'getal1' en 'getal2' een handle (&). De string 'str' is al een pointer (het is een array) en behoeft dus niet voorafgegaan te worden door een handle (&).

De besturingsstring kan de tekens %d, %o, %x, %u, %e, %f, %c, %s bevatten, zie paragraaf 15.1 voor de betekenis van deze conversietekens.

De argumenten moeten bij het intikken gescheiden worden door een spatie (of een tab of een return).

De besturingsstring:

```
scanf("%[abcd \n", str);
```

leest in de array 'str' alleen a's, b's, c's, d's en een return.

```
scanf("%[^q]", str);
```

leest een string in die wordt beëindigd door een 'q'.

### 15.3 sprintf() en sscanf()

Met de functie `sprintf()` kan een integer in een string worden geplaatst:

```
Stel: int i=1234;  
      char str[5];  
  
      sprintf(str, "%d", i);
```

heeft als resultaat dat de string 'str' als volgt gevuld wordt:

1	2	3	4	\0
---	---	---	---	----

Met de functie `sscanf()` gebeurt het tegenovergestelde:

```
sscanf(str, "%d", &i);
```

Dit heeft als resultaat dat de string 'str' wordt omgezet naar een integer.

### 15.4 strcpy(), strcat() en strlen()

In deze paragraaf worden twee strings gebruikt die op de volgende wijze zijn gedeclareerd:

```
char a[20],  
     b[20];
```

#### voorbeeld 1

Met de opdracht:

```
strcpy(a, "Hello");
```

wordt het woord "Hello" in de string 'a' gezet.

Let op: Achter het woord Hello (en de afsluitende '\0' die `strcpy()` altijd achter een string plakt) kan troep staan!. Strings hoeven door de compiler dus niet schoongemaakt te worden (bijvoorbeeld gevuld met '\0').

Met de opdracht:

```
strcpy(b, a);
```

wordt de string 'a' in 'b' gekopieerd.

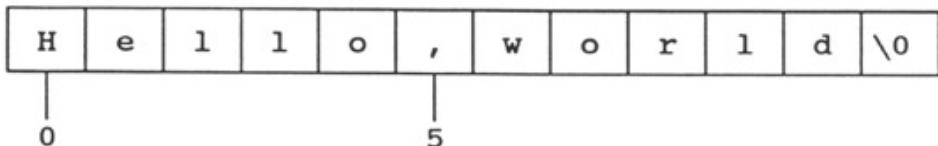
### voorbeeld 2

```
strcpy(a, "Hello");
strcpy(b, ",world");
```

Na de opdracht:

```
strcpy(&a[5], b);
```

ziet de string 'a' er zo uit:



","world" wordt achter "Hello" geplakt doordat via een handle (&) de functie strcpy() een pointer doorkrijgt die wijst naar het zesde teken van de string 'a'. De handle was bij 'b' niet nodig omdat:

b

naar hetzelfde wijst als:

```
&b[0]
```

### voorbeeld 3

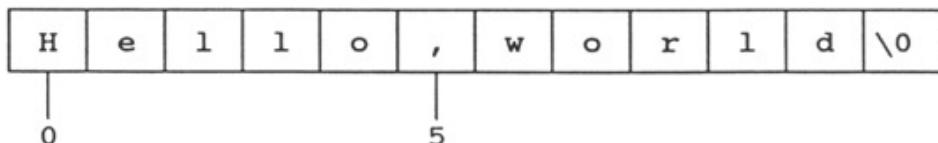
Het vorige resultaat kan eenvoudiger verkregen worden met de functie strcat() (string concatenate):

```
strcpy(a, "Hello");
strcpy(b, ",world");
```

Na de opdracht:

```
strcat(a, b);
```

ziet de string 'a' er zo uit:



#### voorbeeld 4

Met de functie `strlen()` kan de lengte van een string worden bepaald (de afsluitende '\0' wordt niet meegerekend!).

```
strcpy(a, "Bever");
printf("string %s is %d tekens lang", a, strlen(a));
zal printen:

string Bever is 5 tekens lang
```

#### 15.5 strncpy() en strncat()

Het enige verschil met de in paragraaf 15.4 behandelde functies is dat aan deze varianten van `strcpy()` en `strcat()` een maximale lengte meegegeven kan worden.

#### voorbeeld 1

```
strncpy(a, "Steve Jobs", 5);
```

Deze opdracht heeft als resultaat dat alleen "Steve" wordt gekopieerd in de string 'a'.

#### voorbeeld 2

```
strcpy(a, "Hello");
strncat(a, ",world. Are you ok?", 6);
```

Deze opdrachten hebben als resultaat dat de string 'a' de tekst "Hello,world" zal bevatten.

#### voorbeeld 3

```
strncpy(a, "\8\15\test\0\test2\test3\1", 12);
```

Deze opdracht behoort als resultaat te hebben dat de string 'a' er als volgt uit komt te zien:

1	0x8	0xf	t	e	s	t	0x0	t	e	s	t	2	\0
hex hex				hex				null					

Helaas zijn sommige C-compilers zo eigenwijs om te stoppen met kopiëren zodra ze de 0 tegenkomen achter "test".

## 15.6 strcmp() en strncmp()

Met de functie strcmp() kan gezocht worden naar het voorkomen van 1 of meerdere karakters in een string. Het resultaat kan kleiner dan 0, gelijk aan 0 of groter dan 0 zijn.

### voorbeeld 1

```
strcmp("a", "b");           /* resultaat < 0 */
strcmp("test", "test");     /* resultaat = 0 */
strcmp("b", "a");           /* resultaat > 0 */
```

### voorbeeld 2

De functie strncmp() vergelijkt de eerste 'n' karakters in een string.

```
strncmp("Renault 21", "Renault 25", 7); /* resultaat = 0 */
```

Kijk in de handleiding van de compiler of de fabrikant wellicht nog meer handige functies standaard mee heeft geleverd!

## 15.7 Opdrachten

1. Maak een programma dat vraagt om een voor- en achternaam en deze twee strings met strcpy() aan elkaar plakt (gebruik strlen() maar niet strcat()).
2. Gegeven:

```
char str1[] = "Aap....Mies";
char str2[] = "Noot";
```

Maak het programma zodanig af, m.b.v. slechts 1 strcpy() aanroep, dat 'str1' de woorden "AapNootMies" bevat. Druk de string ook af.

3. Maak de volgende functies:

```
right_string(str, lengte);  
mid_string(str, start, eind);  
left_string(str, lengte);
```

De functie right\_string() kopieert 'lengte' karakters vanaf array positie 0 in de string 'str'.

De functie mid\_string() kopieert alle karakters vanaf 'start' t/m 'eind' in de string 'str'.

De functie left\_string() kopieert 'lengte' karakters vanaf de laatste array positie in de string 'str'.



## HOOFDSTUK 16

### 16.1 Datastructuren in C: structures

In databases en administratieve systemen wordt veel gebruik gemaakt van datastructuren die een collectie variabelen bij elkaar houden, dikwijls van diverse datatypen, maar wel met één gemeenschappelijke eigenschap: ze horen bij elkaar. In C heet zo'n samenhangende collectie variabelen een **structure**.

### 16.2 Structures

Als voorbeeld wordt een structure getoond waarin enkele gegevens over een persoon bewaard kunnen worden:

```
struct
{
    char naam[30];
    int leeftijd;
    int aantal_parkeerbonnen;
    char naam_van_hond[10];
} persoon;
```

De variabelen die zijn gedeclareerd in de structure 'persoon' kunnen alleen worden gebruikt als wordt vermeld dat het de variabelen zijn die bij de 'persoon' structure horen. De variabelen kunnen, mits de structure static of extern wordt gedeclareerd, ook geïnitialiseerd worden:

```
static struct
{
    char naam[30];
    int leeftijd;
    int aantal_parkeerbonnen;
    char naam_van_poes[10];
} persoon = { "Joost", 23, 17, "Hier" };
```

N.B. De poes van Joost heet 'Hier' omdat -ie dan alleen "Kom Hier" hoeft te roepen. Joost studeerde economie, vandaar.

In een programma kunnen, uiteraard ook na eventuele initialisatie, de waarden in een structure aangepast worden. Dit gebeurt op de volgende wijze:

```
strcpy(persoon.naam, "Erik");
persoon.leeftijd = 22;
aantal_parkeerbonnen = 0;
strcpy(persoon.naam_van_poes, "Backslash null");
```

N.B. De naam van Erik's poes komt voort uit ernstige mate van z.g. 'beroepsdeformatie' ("Achter de string "Kom!" hoort toch een backslash null", bracht hij eens in verweer...).

Het handige van structures is dat we een eenmaal goed doordachte datastructuur kunnen gebruiken als 'vormpje' (weet u nog wel, zo'n plastic dingetje waarmee u speelde op het strand). Met zo'n vormpje kunnen kopieën worden gegoten, de kopieën hebben dan een andere naam maar dezelfde vorm.

```
struct Window
{
    int xpos, ypos, breedte, hoogte;
    char naam[80];
    int binnenkleur, randkleur, tekstkleur;
};
```

Het vormpje heet in dit voorbeeld 'Window'. Van dit vormpje kunnen op de volgende wijze duplicaten worden gemaakt:

```
struct Window Wdw1, Wdw2;
```

De structure met de naam 'Wdw2' kan nu gevuld worden:

```
Wdw2.xpos = 0;
Wdw2.ypos = 0;
Wdw2.breedte = 1024;
Wdw2.hoogte = 512;
strcpy(Wdw2.naam, "MyVeryOwnWindow");
etcetera...
```

De twee Window structures hadden ook in één declaratie gemaakt kunnen worden:

```
struct Window
{
    int xpos, ypos, breedte, hoogte;
    char naam[80];
    int binnenkleur, randkleur, tekstkleur;
} Wdw1, Wdw2; /* twee declaraties! */
```

De term 'vormpje' nemen informatici liever niet in de mond. Ze gebruiken meestal de Engelse term 'template'.

### 16.3 Een array van structures

De Window structure kan 10 keer herhaald worden door een array van structures te declareren:

```
struct Window
{
    int xpos, ypos, breedte, hoogte;
    char naam[80];
    int binnenkleur, randkleur, tekstkleur;
} Wdw[10];
```

Het vijfde element in de array van structures kan nu op de volgende manier worden bereikt:

```
Wdw[4].xpos = 320;
Wdw[4].ypos = 640;
```

etcetera...

Bouw geen arrays van structures met tienduizenden elementen, de ervaring heeft geleerd dat sommige compilers daar een beetje zenuwachtig van worden. Is toch een dergelijke grote datastructuur nodig, doe dat dan via (dynamische) geheugenallocatie, zoals nog zal worden behandeld.

### 16.4 Structures en functies

Structures kunnen net als variabelen worden doorgegeven aan functies. De Window structure uit de vorige paragraaf zal worden doorgegeven aan een functie OpenWindow():

```
struct Window
{
    int xpos, ypos, breedte, hoogte;
    char naam[80];
    int binnenkleur, randkleur, tekstkleur;
} Wdw1, Wdw2;
```

De functie OpenWindow ziet er gedeeltelijk zo uit:

```
int OpenWindow(wdw)
struct Window wdw;
{
    TekenKader(wdw.xpos, wdw.ypos, wdw.breedte, wdw.hoogte);
    etcetera...
}
```

De functie TekenKader() gebruikt de waarden van de structuur die is doorgegeven om een kader te tekenen op het scherm.

### 16.5 Pointers en structuren

In de vorige paragraaf werd een Window structuur doorgegeven aan een functie die de waarden gebruikte om een kader te tekenen. Deze functie behoefde de waarden van de doorgegeven structuur-elementen niet te wijzigen en dat is maar goed ook want dat zou niet lukken. Om de elementen van een doorgegeven structuur te kunnen veranderen dient een pointer naar die structuur te worden doorgegeven. Een functie die de waarden van de Window structuur moet kunnen aanpassen is bijvoorbeeld de functie MaxWindow() die een window over de volle hoogte en breedte van het scherm uitrekt:

```
struct Window
{
    int xpos, ypos, breedte, hoogte;
    char naam[80];
    int binnenkleur, randkleur, tekstkleur;
};

void MaxWindow(wdw)
struct Window *wdw;
{
    wdw->xpos = 0;
    wdw->ypos = 0;
    wdw->breedte = 1024; /* maximale scherm breedte */
    wdw->hoogte = 1024; /* maximale scherm hoogte */

    ResizeWindow(wdw);
}

void main()
{
    struct Window Wdw;

    Wdw.xpos = 100;
    Wdw.ypos = 100;
    Wdw.breedte = 320;
    Wdw.hoogte = 320;
    /* de overige elementen worden in dit voorbeeld niet
     * ingevuld... */

    OpenWindow(Wdw);

    Delay(10); /* wacht 10 seconden */
}
```

```
    MaxWindow(&Wdw); /* blaas window op tot */
                      /* maximale grootte */

    Delay(10);

    CloseWindow(wdw);
}
```

De uitleg stap voor stap:

1. Extern wordt een structuur gedeclareerd met de naam Window.
2. In de functie main() wordt uit de Window template een kopie gegoten met de naam Wdw. Vervolgens worden de nieuwgebakken structuur-elementen gevuld met enige waarden.
3. Vervolgens wordt met de opdracht:

```
OpenWindow(Wdw);
```

een window geopend op het grafische scherm met de dimensies (de hoogte en breedte etc.) die waren opgegeven.

4. Na de Delay(10) waarbij 10 seconden wordt gewacht om het pas geopende raam te bewonderen, staat de opdracht:

```
MaxWindow(&Wdw);
```

De handle (&) toont dat aan de functie MaxWindow() een pointer wordt doorgegeven naar de Wdw structuur.

5. De header (het kopje) van de functie MaxWindow() zag er zo uit:

```
void MaxWindow(wdw)
struct Window *wdw;
{ ...
```

De handle in de MaxWindow() aanroep was juist want deze functie verwacht inderdaad een pointer naar de Window structuur.

6. Vervolgens gebeurt er iets geheel nieuws. De elementen van de structure worden niet bereikt met:

```
structure_naam.element_naam
    └─LET OP DE PUNT
```

maar met:

```
structure_naam->element_naam
    └─LET OP DE PIJL
```

Blijkbaar moeten elementen van pointers naar structures bereikt worden met een pijltje. Nu is pointer Engels voor pijltje, dus dit valt wel te onthouden.

7. De rest van de functie MaxWindow() doet niets meer dan de maximale breedte en hoogte van het grafische scherm in de structure elementen stoppen en de functie ResizeWindow() aan te roepen. ResizeWindow() past een reeds geopend raam aan aan de waarden die in de window structure staan.

Functies kunnen ook een pointer naar een structure terugleveren. Een markant voorbeeld is het teruggeven van z.g. 'objecten' in de vorm van structures. Veel grafische interfaces, zoals de Presentation Manager op een IBM PS/2, de Finder op een Mac of Intuition op een Amiga hebben schermobjecten die reageren nadat er met de muis op geklikt te is. Deze objecten zijn bijvoorbeeld rechthoeken met tekst (OK, Cancel, Stop etc.), de schuifbare objecten (waarmee bijvoorbeeld een tekst heen en weer kan worden geschoven), en de rechthoeken waarin één regel tekst kan worden ingetikt.

Als voorbeeld voor het teruggeven van een pointer naar een structuur zal een functie worden gebruikt die de muis uitleest, het aangeklikte object opzoekt en een pointer teruggeeft naar dat object:

```
struct Object
{
    int x, y, breedte, hoogte,
    type,
    nummer;
} obj_list[10];

struct Object *WhichObject(WO_obj_list)
struct Object WO_obj_list[];
{
int mx, my,
i;

WhereIsMouse(&mx, &my);

for(i=0; i<10; i++)
    if (mx > WO_obj_list[i].x &&
        mx < (WO_obj_list[i].x + WO_obj_list[i].breedte &&
        my > WO_obj_list[i].y &&
        my < (WO_obj_list[i].y + WO_obj_list[i].hoogte)
    return(WO_obj_list[i]);

return(NULL);
}
```

Stap voor stap zal het programmafragment 'ontleedt' worden:

1. De declaratie van de 'Object' structuur is reeds gesneden koek. De regel 'obj\_list[10]' declareert een array van 10 'Object' structures.
2. De functie WhichObject() is algemeen van opzet, niet alleen 'Object' structures kunnen worden verwerkt maar andere (uitgebreidere objecten!) óók. Aan de functie moet namelijk het 'hoofd' van de array van objecten worden doorgegeven zodat WhichGadget() weet in welke array en vanaf welk object er gezocht moet worden:

```

...programma

struct Object obj;

obj = (struct Object)WhichObject(obj_list);

printf("Object met dimensies %d, %d, %d, %d geraakt\n",
      obj.x, obj.y, obj.breedte, obj.hoogte);

...programma

struct Object *WhichObject(WO_obj_list)
struct Object WO_obj_list[];
{
    ...etc.
}

```

Als de bovenstaande functiedeclaratie de vraag oproept wat nu eigenlijk de functienaam is, het mag het ook zo genoteerd worden:

```

struct Object *
WhichObject(WO_obj_list)
struct Object WO_obj_list[];
{
}

```

Nu blijkt dat de functie 'WhichObject()' heet en dat hij een pointer naar een structure als 'return value' heeft.

Tevens blijkt uit de functiedeclaratie dat de functie een structure als argument wil hebben. Dit argument is meestal het hoofd van een array van objecten. De naam 'WO\_obj\_list' is gekozen omdat deze array lokaal is aan de functie WhichObject(), vandaar de WO\_... Er kan bij een dergelijke naamgeving geen verwarring ontstaan over lokalen en globalen. Een tip misschien?

3. De functie WhereIsMouse() stopt de x- en y-positie van het muispijltje in de variabelen 'mx' en 'my'.
4. De for-lus doorzoekt vervolgens alle objecten en kijkt of de muis wellicht in het rechthoekje van het object staat, zo ja: dan geeft de functie een pointer terug naar dat object, zo nee: dan wordt een NULL 'gereturned'. De 'if' is wel wat aan de lange kant maar zal geen hoofdpijn bezorgen over hoe hij functioneert.

Er is nog een manco aan de functie WhichObject() en dat is de constante '10' in de for-lus. Om het voorbeeld 'clean' te houden is daarvoor gekozen maar een echte versie moet natuurlijk in staat zijn om Objectlijsten van bijvoorbeeld een halve k tot 1.5 Gigabyte te doorzoeken. Het aantal elementen van een lijst kan berekend worden met:

```
object_grootte = sizeof(struct Object);  
lijst_grootte = sizeof(WO_obj_list);
```

Het aantal elementen is dan:

```
aantal_objecten = lijst_grootte / object_grootte;
```

De grootte kan aldus in de functie WhichObject() berekend worden, wel zal dan naast het doorgeven van het hoofd van de objectenlijst ook 1 object structuur moeten worden doorgegeven. Bijvoorbeeld:

```
obj = (struct Object)WhichObject(obj_list, Object);
```

The diagram illustrates the memory layout of the variable 'obj'. It shows a bracket spanning from the opening parenthesis '(struct Object)' to the closing parenthesis ')'. Two arrows point downwards from this bracket: one to the right pointing to the text 'object structuur', and another pointing to the right at the bottom pointing to the text 'objectlijst'.

Dit is natuurlijk niet nodig als de structuur globaal gedeclareerd is, hetgeen veelal het geval zal zijn.

Het heen- en weer overdragen van structuren kan tot heel duidelijke en herbruikbare functies leiden. Bovendien kan met de functie zoals boven is beschreven niet alleen de 'Object' structuur gewerkt worden maar ook met veel uitgebreidere structures; zolang de variabelen 'x', 'y', 'breedte' en 'hoogte' maar aanwezig blijven in dat object. Door een familie van goed doordachte structuren te bouwen en functies zoals WhichObject() kan in het hoofdprogramma met objecten gewerkt worden en wordt het zicht niet vertroebeld door muiscoordinaten, objectcoördinaten en objectdimensies (breedte en hoogte). Wil je meedoen aan de laatste trend, zeg dan dat je aan Object Oriented Programming doet. Maakt indruk!

Als laatste wordt een voorbeeld gegeven van zo'n 'afgeleide' van de 'Object' structure. Het is de structure die een 'schuifje' declareert:

```
/* het standaard object */

struct Object
{
    int x, y, breedte, hoogte,
    type,
    nummer;
} obj_list[10];

/* het van het standaard object afgeleide schuifje object */

struct Schuifje_Object
{
    int x, y, breedte, hoogte,
    type,
    nummer,
    stapgrootte, huidige_positie;
} schuif_obj_list[10];
```

De variabelen 'stapgrootte' en 'huidige\_positie' zijn toevoegingen aan de 'Object' structure.

Voor 'echt' Object Oriented Programming zijn diverse talen gemaakt waarin dit standaard in is verwerkt. Zo bestaat onder andere de taal C++ (die veel lijkt op C) en Turbo Pascal (vanaf versie 5.5) waarin 'OOP' mogelijk is.

## 16.6 Dynamische geheugenallocatie

Als voorbeeld voor deze paragraaf zal de opbouw van een teksteditor genomen worden. De teksteditor is van het snelle en efficiënte type: zoveel geheugen reserveren als nodig is en geen dataverplaatsingen bij het tussenvoegen of verwijderen van regels. Het eerstgenoemde slaat op het feit dat sommige teksteditors maar eventjes 100 k reserveren zodat er genoeg ruimte is voor de meeste bestanden. Bij kleine bestanden is 100 k echter zwaar overdreven (het tegelijkertijd laten draaien van 3 of 4 van zulke editors in een multitasking omgeving levert al snel problemen op!). Een efficiënte editor reserveert dus precies genoeg en reserveert bij het groeien van het bestand de benodigde extra ruimte, dit noemt men dynamische geheugenallocatie.

De tweede eigenschap van onze editor, het niet verplaatsen van data, maakt de editor snel. Stel eens voor dat midden in een bestand van 80 k een regel verwijderd wordt: de editor zou dan 40 k moeten aansluiten aan de eerste 40 k! Zulke editors gaan dus echt direct het raam uit.

Er vanuit gaande dat onze tekst geen foto's en (vector)tekeningen bevat dus enkel 'platte tekst' dan bestaat de tekst uit een regel tekst die weer wordt voorafgegaan en wordt gevolgd door een regel tekst. Alleen de eerste en laatste regel van de tekst wijken van deze definitie af. Een veelvuldig gebruikte constructie om alle regels bij elkaar te houden is deze:

```
struct knoop
{
    struct knoop *vorige_regel;
    struct knoop *volgende_regel;
    char regel[80];
};
```

Het lijkt misschien wat vreemd dat in de declaratie van de structure ook verwijzingen naar dezelfde structure staan maar dit is toegestaan zolang het verwijzingen naar pointers zijn en niet declaraties van dezelfde structure.

Aldus is de regel:

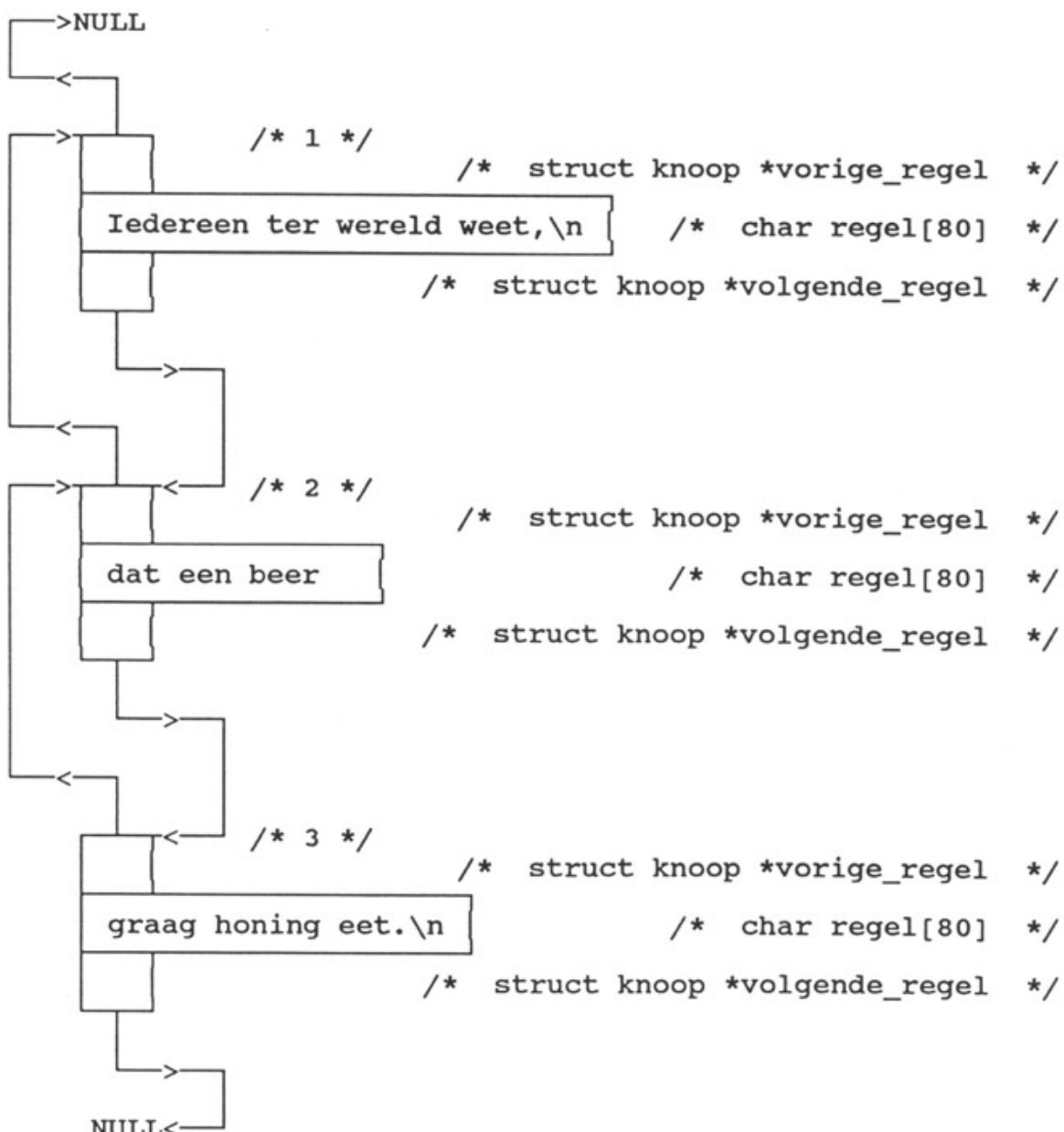
```
struct knoop *vorige_regel;
```

een pointer wijzende naar een structure 'knoop'.

In de teksteditor wordt de tekst:

```
Iedereen ter wereld weet,  
dat een beer  
graag honing eet.
```

getikt. De regels wijzen schematisch als volgt naar elkaar:



Elke structure wijst naar de vorige structure en de volgende. De eerste wijst alleen naar de volgende en de laatste structure wijst alleen naar de vorige. De pointers die naar niets wijzen, hebben de waarde NULL. De geheugenadressen van de vorige en volgende structures worden in de 'knoop' pointers bewaard.

De term 'knoop' komt van knooppunt en wordt ook vaak 'node' genoemd (Engels voor knoop). De winst die bereikt wordt met een dergelijk ingewikkelde constructie is het volgende:

1. Bij het verwijderen van een regel hoeft alleen de pointer van de volgende regel die wijst naar de te verwijderen regel verandert te worden zodat hij gaat wijzen naar de regel die voor de te verwijderen regel komt. De pointer van de vorige regel die wijst naar de te verwijderen regel moet verandert worden zodat hij gaat wijzen naar de regel die komt na de regel die verwijdert gaat worden.

Met andere woorden, stel dat de regel "dat een beer" verwijderd wordt, dan zal de pointer in de structure /\* 3 \*/ die wijst naar structure /\* 2 \*/ gaan wijzen naar structure /\* 1 \*/. De pointer in structure /\* 1 \*/ die wijst naar structure /\* 2 \*/ zal gaan wijzen naar structure /\* 3 \*/. De structure /\* 2 \*/ kan leeg gemaakt worden voor hergebruik of het ervoor gereserveerde geheugen kan teruggegeven worden. Voor het verwijderen van een regel hoeven dus maar twee pointers veranderd te worden en dat gaat razendsnel!

2. Bij het tussenvoegen van een regel moet eerst een nieuwe knoop worden gemaakt, via het reserveren van geheugen of het opzoeken van een lege knoop in een verzameling knopen. Daarna moeten de pointers van deze nieuwe knoop gaan wijzen naar respectievelijk de voorafgaande en de opvolgende knoop.

Hier is de snelheidswinst ook aanzienlijk. Het tussenvoegen van een regel is geen operatie waarbij data verplaatst hoeft te worden maar alleen maar een kwestie van 4 pointers (1 pointer van de voorafgaande knoop, 2 van de nieuwe en 1 van de volgende knoop) op de juiste waarden zetten.

Het naar elkaar laten wijzen van knopen zal worden behandeld nadat is uitgelegd hoe knopen dynamisch worden gebouwd:

Dezelfde 'knoop' structuur zal worden gebruikt:

```
struct knoop
{
    struct knoop *vorige_regel;
    struct knoop *volgende_regel;
    char regel[80];
};
```

N.B. Dit soort structuren, die naar elkaar gaan wijzen, moeten op sommige computers (bijv. met een Motorola 68000) z.g. 'word aligned' zijn. Dat betekent dat de totale structuur een even aantal bytes moet bevatten. Was de string 'regel' bijvoorbeeld 17 bytes groot geweest dan had de structuur aangevuld moeten worden met een z.g. 'pad byte' om de structuur op te vullen ('to pad' is Engels voor opvullen) tot een even aantal bytes.

```
struct knoop *MaakKnoop(oude_knoop, tekst)
struct knoop *oude_knoop;
char *tekst;
{
    struct knoop *nieuwe_knoop;

    /* reserveer geheugen voor nieuwe knoop */
    nieuwe_knoop = malloc(sizeof(struct knoop));

    /* plak de nieuwe knoop aan de oude knoop */
    oude_knoop->volgende_knoop = nieuwe_knoop;
    nieuwe_knoop->vorige_knoop = oude_knoop;
    nieuwe_knoop->volgende_knoop = NULL;

    /* kopieer regel in nieuwe knoop */
    strcpy(nieuwe_knoop->regel, tekst);

    return(nieuwe_knoop);
}
```

```

void PrintKnopen(k)
struct knoop *k;
{
    /* de knoop 'k' is de laatste knoop en is dus de knoop */
    /* die de laatste regel bevat. De bedoeling is natuur- */
    /* lijk dat vanaf de eerste regel geprint wordt dus */
    /* moet eerst het 'hoofd' van de knopenlijst gevonden */
    /* worden. Het hoofd kenmerkt zich door het bezit van */
    /* een NULL pointer naar de 'vorige_knoop'. */

    /* zoek hoofd van knopenlijst */
    while(k->vorige_knoop != NULL)
        k = k->vorige_knoop;

    /* print regels totdat staart van knopenlijst is bereikt */
}
do
{
    printf("%s\n", k->regel);
    k = k->volgende_knoop;
}
while (k != NULL);

void LeegKnopen(k)
struct knoop *k;
{
    /* loop de knopen van achter naar voren af en geef */
    /* het gereserveerde geheugen van elke knoop terug */
    while(k->vorige_knoop != NULL)
    {
        k = k->vorige_knoop;
        free(k->volgende_knoop);
    }
}

void main()
{
char t[80];
struct knoop *k;

    /* maak knoop schoon */
    k->vorige_regel = NULL;
    k->volgende_regel = NULL;

    /* lees eerste regel */
    scanf("%s", t);
}

```

```

/*lees volgende regels en stop bij invoer van een punt*/
while(t[0] != '.')
{
    k = MaakKnoop(k, t);
    scanf("%s", t);
}

PrintKnopen(k);

LeegKnopen(k);
}

```

Bestudeer dit programma aandachtig! Als je het niet begrijpt, ga dan een stukje hardlopen of je vriend/vriendin een knuffel brengen en probeer het dan opnieuw...

De uitleg stap voor stap:

1. De regels:

```

struct knoop *MaakKnoop(oude_knoop, tekst)
struct knoop *oude_knoop;
char *tekst;
{
    struct knoop *nieuwe_knoop;

```

moeten zolangzamerhand geen problemen meer opleveren. De eerste regel vertelt de compiler dat de functie MaakKnoop() een pointer teruggeeft naar een 'knoop' structure. Als argumenten heeft de functie een pointer naar een 'knoop' structure en een pointer naar een tekstregel. De declaratie:

```
    struct knoop *nieuwe_knoop;
```

declareert een lokale pointer naar een structure. Deze structure zal bij het verlaten van de functie worden doorgegeven met een return() statement.

2. De functie MaakKnoop() heeft een tweeledige functie. Ten eerste reserveert hij geheugen voor een nieuwe knoop en ten tweede plakt hij de nieuwe knoop aan de knoop die aan de functie als argument is doorgegeven:

```
nieuwe_knoop = malloc(sizeof(struct knoop));
```

Hierdoor gaat de pointer-naar-een-structure 'nieuwe\_knoop' wijzen naar een geheugengebied ter grootte van een 'knoop' structure. Eigenlijk hoort hier een controle of malloc() geen NULL teruggeeft hetgeen gebeurt bij een tekort aan geheugenuimte.

```
oude_knoop->volgende_knoop = nieuwe_knoop;
```

Deze regel zorgt ervoor dat de nieuwe knoop aan de vorige regel geplakt wordt. Dit gebeurt door de pointer van de vorige knoop die wijst naar 'een' volgende knoop te laten wijzen naar de zojuist gebouwde nieuwe knoop. Dit is een fundamentele wijze van werken in C met pointers!

```
nieuwe_knoop->vorige_knoop = oude_knoop;
```

Op deze wijze wordt de pointer van de nieuwbakken knoop die wijst naar de vorige regel vertelt waar de vorige regel zich bevindt. In de vorige twee regels werd dus eerst aan de oude knoop verteld waar de nieuwe zich bevindt en vervolgens aan de nieuwe waar de oude zich bevindt. Nu rest ons alleen nog de staart van de knopenlijst met een NULL pointer te vullen zodat de staart herkend kan worden (de kop van de knopenlijst heeft een NULL pointer naar de vorige knoop en de staart dus een NULL pointer naar de volgende knoop). Het maken van de staart gebeurt zo:

```
nieuwe_knoop->volgende_knoop = NULL;
```

De ingetikte tekst moet nog worden bewaard in de verse knoop en dat gaat zo:

```
strcpy(nieuwe_knoop->regel, tekst);
```

De nieuwe knoop moet terug worden gegeven aan main() want daar zal hij na het intikken van weer een nieuwe regel gaan fungeren als oude knoop!

```
return(nieuwe_knoop);
```

3. De functie PrintKnopen() heeft een probleem, lees daarvoor het commentaar zoals verwerkt in het programma. Het hoofd van de lijst moet gevonden worden:

```
while(k->vorige_knoop != NULL)
    k = k->vorige_knoop;
```

Aan de functie PrintKnopen() wordt een pointer meegegeven die wijst naar de laatste knoop. De while-lus controleert of het hoofd van de lijst is gevonden door te kijken of de pointer die wijst naar de vorige knoop NULL is. Als dit niet het geval is dan gaat de pointer 'k' wijzen naar de vorige knoop in de lijst net zo lang totdat het hoofd is gevonden. Ook dit is een zeer fundamenteel stukje C!

Nu het hoofd gevonden is kunnen vanaf daar alle regels worden geprint totdat een knoop wordt tegengekomen die een pointer die naar de volgende knoop wijst met de waarde NULL. Zoals eerst het hoofd van de knopenlijst gezocht werd, zo zal nu de staart gezocht worden terwijl de regels worden afdrukt die onderweg worden tegengekomen:

```
do
{
    printf("%s\n", k->regel);
    k = k->volgende_knoop;
}
while (k != NULL);
```

Voor elke knoop die werd gecreëerd, was een stukje geheugen gereserveerd. Bij het beëindigen van het programma moet die teruggegeven worden. Het teruggeven gebeurt van achter naar voren (zodat niet eerst het hoofd gezocht hoeft te worden) maar dit is niet verplicht:

```
while(k->vorige_knoop != NULL)
{
    k = k->vorige_knoop;
    free(k->volgende_knoop);
}
```

Merk op dat eerst de pointer naar de vorige regel in 'k' wordt gezet alvorens de huidige knoop te 'free-en'. Als namelijk eerst de knoop wordt doorgegeven aan free() en daarna de pointer naar de vorige knoop nog eruit wordt gehaald is de kans erg groot dat free() de pointers al op NULL gezet heeft.

4. In de functie main() gebeuren niet echt wereldschokkende zaken. De aanroep:

```
k = MaakKnoop(k, t);
```

heeft tot gevolg dat de pointer 'k' doorgegeven wordt aan de functie MaakKnoop() en aldaar tot 'oude knoop' bestempeld wordt. Een nieuwe knoop zal de regel 't' gaan bevatten en deze knoop zal via een return() statement aan de pointer 'k' worden doorgegeven. Dit gaat voort todat de gebruiker een punt intikt. Vervolgens worden alle ingetikte regels afgedrukt en vrij gemaakt.

Het woord fundamenteel is enkele malen gebruikt in deze paragraaf en aldus is het behandelde programma ook te bestempelen. Merk op dat in het gehele programma geen for- of while-lussen zijn gebruikt met tellers! Alle lussen werken met pointers zodat een 'variabelen huishouding' (het bijhouden van allerlei waarden van variabelen) overbodig is. Tevens is het programma sneller en efficiënter dan een vergelijkbaar programma met een array van regels (een array van welke grootte?) en een teller die bijhoudt met welk arrayelement gewerkt wordt.

### 16.7 Structuren met bitvelden

Informatie die beschreven kan worden door een simpele 'ja' of 'nee' of 'vrouw' of 'man' kan zeer efficiënt worden opgeslagen door aan één van de mogelijkheden de waarde 0 toe te kennen en aan de overblijvende de waarde 1. Op deze wijze is het n.l. mogelijk om de informatie in 1 bit op te slaan. Als er drie mogelijkheden zijn, dan zijn 2 bits voldoende om alle mogelijkheden te bewaren maar het blijft efficiënter dan een complete 'char' of 'int' uit de kast rukken:

```
#define VROUW 0
#define MAN 1

#define GEHUWD      0
#define ONGETROUWD  1
#define GESCHEIDEN  2
#define OVERLEDEN   3

struct
{
    unsigned sexe : 1;
    unsigned staat : 2;
    char naam[30];
} persoon_info;
```

De structuur kan op dezelfde wijze als een gewone structuur worden gevuld met waarden:

```
persoon_info.sex = VROUW;  
persoon_info.staat = GEHUWD;
```

Voor het bitveld 'staat' zijn 2 bits nodig zodat er '2 tot de 2de is 4' mogelijkheden zijn om een getal te bewaren (0, 1, 2 en 3 met respectievelijk de bitpatronen 00, 01, 10 en 11).

## HOOFDSTUK 17

### 17.1 Invoer en uitvoer

Een Pascal programmeur keek op een dag eens in een C leerboek en bladerde wat heen en weer. Toen hij de bladzijde bestudeerde waarop alle gereserveerde woorden stonden afgedrukt schrok hij enorm! C kent geen standaardcommando's voor het lezen of schrijven van een bestand! Hij krabte zijn hoofd en smeet het boek in een hoek en mompelde: "Of het is een laf boek of C is een budget taal".

Wij weten dat geen van beide waar is en de oplossing van het ogenschijnlijke probleem zit 'em in de structuur van C waarin alles om functies draait. Elke C compiler kent een set functies om bestanden te lezen, te schrijven (in zijn geheel of per karakter). Veel compilers (vooral Turbo C is zeer genereus) kennen daarnaast een enorme collectie extra functies, bijvoorbeeld om een directory binnen te lezen, bestanden te wissen, bestandsgroottes op te vragen etc.

Wel dient te worden opgemerkt dat er tamelijk veel coding vereist is om, geheel foutbestendig, bestanden te bewerken. Bestanden moeten zelf geopend, gelezen en gesloten worden. Bij elke stap dient gecontroleerd te worden of er geen fouten zijn opgetreden. Niets is frusterender dan er achter te komen (als het te laat is!) dat een programma toch niet een bestand heeft weg geschreven omdat (ja ja) de disk vol was. Iets dat gecontroleerd had kunnen worden (en wel voordat er geschreven gaat worden!). Bedenk dat de waardering voor een programma door een gebruiker voor 90% afhangt van de zorgvuldigheid waarmee zijn/haar gegevens worden behandeld en bewaard. De lees- en schrijffuncties dienen dus puntgaaf te zijn. Altijd.

N.B. De komende programma voorbeelden zullen bestanden gaan lezen en schrijven van en naar diskette. Als een filename tussen "..." staat, bijvoorbeeld:

**"bestand.asc"**

dan mag daar altijd een 'device'naam voor staan, bijvoorbeeld:

**"C:bestand.asc"**

Op een PC zal dan het bestand van de harddisk worden gelezen (mits C: een bestaand device is).

## 17.2 Karakters lezen en schrijven (putc() en getc())

Ooit wel eens een bestand-kopieerprogramma geschreven in 5 minuten? Het onderstaande programma "does just that":

```
#include <stdio.h>
#include <dos.h>      /* dit KAN verschillen per compiler */

void main()
{
FILE *source,
    *dest;
char teken;

    source = fopen("A:test.asc", "r");
    if (source == NULL)
    {
        printf("Kan A:test.asc niet openen\n");
        exit(FALSE);
    }
    dest = fopen("B:test.bak", "w");
    if (dest == NULL)
    {
        printf("Kan B:test.bak niet openen\n");
        fclose(source);
        exit(FALSE);
    }

    while((teken=getc(source)) != EOF)
        putc(teken, dest);

    fclose(dest);
    fclose(source);
}
```

Dit programma kopieert het bestand 'test.asc' vanaf de A: drive naar het bestand 'test.bak' op de B: drive.

Allereerst valt op dat er twee pointers worden gedeclareerd van een nieuw en onbekend type, n.l. FILE. Dit type is een structure en is (waarschijnlijk) gedefinieerd in 'stdio.h' (neem maar eens een kijkje in dit bestand). Nadat de twee bestanden zijn geopend, wijzen 'source' en 'dest' naar twee gebieden die beide allerlei informatie bevatten over vrije diskruimte, bestandsgroottes misschien zelfs cylinderpositie etc. Hiermee houden wij ons echter helemaal niet bezig, wat we moeten onthouden is dat het openen van een bestand altijd nodig is voordat er iets mee gedaan kan worden. Deze pointers worden vaak 'file pointers' genoemd.

```
source = fopen("A:test.asc", "r");
if (source == NULL)
{
    printf("Kan A:test.asc niet openen\n");
    exit(FALSE);
}
```

De aanroep van fopen() levert een pointer naar een FILE structure op. Als er ergens iets fout gaat (drive deurtje nog open, bestand verminkt of gewoon niet aanwezig) dan levert fopen() een NULL terug. De "r" in fopen() betekent dat het bestand 'test.asc' alleen gelezen gaat worden. De meeste compilers kennen de volgende opties:

- "r" Lees een bestand (read).
- "w" Schrijf een bestand (write). Er zijn twee mogelijkheden:
  - 1) Het bestand bestaat reeds: het oude bestand zal overschreven worden.
  - 2) Het bestand bestaat nog niet: de naam zal in de directory worden opgenomen en een plaats krijgen op de disk.
- "a" Plak (append) iets aan het einde van een bestand. Als het bestand nog niet bestaat zal het worden gecreeerd.

De while-lus is het eigenlijke kopieerprogramma:

```
while((teken=getc(source)) != EOF)
    putc(teken, dest);
```

De expressie die tussen de haakjes van het while statement staat is TRUE zolang 'teken' niet gelijk wordt aan EOF (End Of File). EOF is gedefinieerd in 'stdio.h' en is meestal 0 of -1. De functie getc() leest een karakter uit het bestand binnen en plaatst dat karakter in de variabele 'teken'. Dit karakter wordt daarna met de functie putc() weer weggeschreven. Merk op dat getc() en putc() beide een pointer naar een FILE structure nodig hebben. Zo kunnen zij bepalen op welke disk en op welk bestand de lees- en schrijf-acties plaats vinden.

De bestanden dienen na gebruik weer gesloten worden! Gebeurt dit niet dan is alle geschreven informatie meestal weer verdwenen als de computer uitgezet wordt! Veel (Disk) Operating Systems schrijven namelijk eerst naar een stuk RAM en pas bij het sluiten van het bestand naar diskette. De functie fclose() sluit het bestand waarnaar de FILE structure wijst.

N.B. Merk op dat bij het tweede bestand dat geopend wordt in de foutdetectie ook het sluiten van het eerste bestand is opgenomen. Die is dan namelijk al wel succesvol geopend!

### 17.3 Grote bestanden lezen en schrijven (fread() en fwrite())

Het kopieerprogramma zoals beschreven in de vorige paragraaf las 1 karakter binnen en schreef deze weg naar een ander bestand. Veelal is het efficiënter om bestanden in zo groot mogelijke brokken binnen te lezen of weg te schrijven, dit gaat n.l. vele malen sneller. Het volgende programma leest een bestand in 1 keer binnen en schrijft het vervolgens naar een ander bestand ook weer in 1 keer weg:

```
#include <stdio.h>
#include <dos.h> /* dit KAN verschillen per compiler */

#define BUFGROOTTE 1024

void main()
{
FILE *source,
    *dest;
char buffer[BUFGROOTTE];
int echte_grootte;

source = fopen("A:test.asc", "r");
if (source == NULL)
{
    printf("Kan A:test.asc niet openen\n");
    exit(FALSE);
}
dest = fopen("B:test.bak", "w");
if (dest == NULL)
{
    printf("Kan B:test.bak niet openen\n");
    fclose(source);
    exit(FALSE);
}

echte_grootte = fread(buffer, 1, BUFGROOTTE, source);
if (echte_grootte > 0)
    fwrite(buffer, 1, echte_grootte, dest);
else
    printf("Bronbestand is te groot of heeft leesfout\n");

fclose(dest);
fclose(source);
}
```

1. Als eerste valt op dat er een buffer wordt gedeclareerd om het bestand in op te slaan. De #define bovenaan het programma zorgt er voor dat overal in het programma de naam BUFGROOTTE door de compiler geïnterpreteerd wordt als 1024.
2. De fopen() statements zijn weer precies gelijk aan die van het vorige programma. De nieuwe statements zijn fread() en fwrite(), zij hebben de volgende parameters:

```
fread(pointer,elementgrootte,aantal_elementen,file-pointer)
fwrite(pointer,elementgrootte,aantal_elementen,file-pointer)
```

**pointer** = pointer naar een bestandsbuffer.

**element\_grootte** = grootte in bytes van de elementen in de bestandsbuffer, dus:

```
element_grootte = 1 bij char (1 byte) buffers
element_grootte = 2 bij int (2 bytes) buffers
element_grootte = 4 bij long (4 bytes) buffers
```

**aantal\_elementen** = aantal maal dat element\_grootte in de buffer past. Bij een long buffer van 1024 elementen is aantal\_elementen gelijk aan  $1024/4 = 256$ .

**file-pointer** = pointer naar file control block.

Beide functies leveren een 0 als er iets fout is gegaan of als het einde van het bestand is bereikt. In alle andere gevallen leveren ze het aantal gelezen elementen. Hiervan wordt dankbaar gebruik gemaakt door het voorbeeldprogramma. Aan de variabele 'echte\_grootte' wordt het werkelijk aantal gelezen bytes toegekend zodat bij het schrijven ook dat bestand weer even groot wordt. De fread() in dit voorbeeld probeert maximaal 1024 bytes binnen te lezen, lukt dit echter niet (het gelezen bestand is kleiner) dan zal 'echte\_grootte' een andere waarde bevatten, n.l. de werkelijke bestandsgrootte. Dit vermijdt het probleem dat kopieën altijd 1024 bytes worden.

Even een verhaaltje over 'portability'. Met portability wordt bedoeld de mate waarin een programma zonder veel herschrijven gedraaid kan worden op een ander computersysteem dan waarop het ontworpen was. Wij nemen aan dat een char, een int en een long respectievelijk 1, 2 en 4 bytes in beslag nemen. De schrijver van dit boek werkt echter met een computer waarop ints 4 i.p.v. 2 bytes in beslag nemen. Om portabiliteitsproblemen te voorkomen had de volgende regel:

```
fwrite(buffer, 1, echte_grootte, dest);
```

er zo uit moeten zien:

```
fwrite(buffer, sizeof(char), echte_grootte, dest);
```

De 1 is vervangen door de sizeof operator die het werkelijk aantal bytes dat een char in neemt berekend. Lees over portability (of overdraagbaarheid in Nederlands) in paragraaf 19.1.

#### 17.4 Regels lezen en schrijven (fprintf() en fscanf())

De veel gebruikte functies printf() en scanf() die gebruikt worden om respectievelijk iets uit te voeren naar het beeldscherm en in te voeren van het toetsenbord, kennen equivalenten functies die met de disk te maken hebben. Na een bestand geopend te hebben kan met:

```
fprintf(fp, "Hello, world");
```

de string "Hello, world" naar het geopende bestand op disk geschreven worden. fp is een file-pointer die de functie fopen() oplevert. Dat printf() en fprintf() veel op elkaar lijken, toont het volgende voorbeeld:

```
i = 1001;  
  
printf("%d\n", i); /* print 1001 op het beeldscherm */  
  
fprintf(fp, "%d\n", i); /* schrijf 1001 naar een bestand */
```

Zolang het bestand waarnaar geschreven wordt met fclose() niet wordt gesloten, zal elke string die geschreven wordt aan de reeds geschreven zinnen 'geplakt' worden. Als er ook steeds een '\n' wordt geschreven aan het einde van de string, onstaan in het bestand zinnen onder elkaar.

De functie scanf() heeft, en dat zal geen verbazing wekken, een 'disk' broertje (of zusje) dat fscanf() heet. Hiermee kunnen bestanden stukje bij beetje worden binnengelezen zoals het volgende voorbeeld toont. Het is een stukje coding uit een programma dat een z.g. 'scriptfile' regel voor regel leest om te kijken welk plaatje op het scherm moet worden getoond en voor welke tijdsduur. De scriptfile ziet er als volgt uit:

##### voorbeeld 1 - een scriptfile

```
foto1,150  
foto2,30  
foto3,60  
enz.
```

De programmaregel die de scriptfile regel voor regel aftast:

```
fscanf(fp, "%s,%d\n", naam, aantal_seonden);
```

met de variabelen:

```
char naam[12];
int aantal_seonden;
```

De "%s,%d\n" zorgt er voor dat fscanf() gaat zoeken naar een string ("%s") en zal spaties die voor de string overslaan. Na de komma wordt gezocht naar een integer ("%d") en de leesactie stopt zodra de "\n" bereikt is. De scriptfile kan eenvoudig worden aangemaakt in een tekstverwerker omdat na een druk op de <ENTER> toets een "\n" wordt geplakt achter de ingetikte zin.

#### voorbeeld 2 - een adressenbestand

Een adressen bestand is als volgt opgebouwd:

```
Voornaam:Robert
Achternaam:Peck
Straatnaam:500-Westdrive
Postcode:CA-32523
Woonplaats:Cupertino
Voornaam:R.J.
Achternaam:Michal
Straatnaam:12624-Greenlane
Postcode:CA-95540
Woonplaats:Sacramento
```

Een waarschuwing is hier op zijn plaats: Het splitsingsteken (\) dat behandeld is in paragraaf 2.1 zal gebruikt moeten worden als met fscanf() een hele rij elementen wordt binnengelezen en de 'formatstring' (de "%s,%d\n") niet meer op 1 tekstverwerkingsregel past! Zo zal:

```
fscanf(fp, "Voornaam:%s\nAchternaam:%s\nStraatnaam:%s\n
Postcode:%s\n Woonplaats:%s\n", vn, an, sn, pc, wp);
```

problemen geven omdat de compiler denkt dat er voor Postcode een hele serie spaties staan. De regel dient er dan ook zo uit te zien:

```
fscanf(fp, "Voornaam:%s\nAchternaam:%s\nStraatnaam:%s\n\
Postcode:%s\n Woonplaats:%s\n", vn, an, sn, pc, wp);
```

Een andere veel gebruikte toepassing voor fprintf() en fscanf() is het respectievelijk schrijven en lezen van z.g. configuratiefiles. Deze files bevatten allerlei door gebruikers ingestelde waarden van een programma. Om deze files voor mensen leesbaar te houden kunnen (zoals in het adressenbestandvoorbeeld) de waarden voorafgegaan worden door zinningen en suggestieve labels.

Zo is:

```
XOFFSET=-5  
YOFFSET=-10  
FONTSIZE=24  
FONTNAME=times
```

toch echt veel beter leesbaar (en aanpasbaar!) dan de hexdump:

```
FB F6 18 74 69 6d 65 73 00
```

die dezelfde informatie bevat. Niet waar?

### 17.5 De functies fputs() en fgets()

De functies puts() en gets() kennen net als printf() en scanf() hun disk equivalenten, te weten fputs() en fgets(). De functies hebben de volgende parameters:

```
fputs(str, fp);
```

```
fgets(str, maximale_lengte, fp);
```

str is gedeclareerd als een array van karakters:

```
char str[80];
```

maximale\_lengte is de lengte die fgets() maximaal in str mag binnentrekken. Een veilige manier is dan ook:

```
fgets(str, strlen(str), fp);
```

fp is een file-pointer.

### 17.6 Random-access lezen en schrijven

Met de functies uit de voorgaande paragrafen kon óf per karakter óf per verzameling karakters gelezen en geschreven worden. Deze lees- en schrijffacties werden sequentieel (karakter na karakter) uitgevoerd. Soms kan het handig zijn als we heen en weer kunnen 'bewegen' in een bestand, om hier en daar een byte van waarde te veranderen. Random-access betekent dan ook 'willekeurige toegang', dit slaat dan op het heen en weer springen in het bestand. Dit heen en weer bewegen kan met fseek() (to seek is Engels voor zoeken):

```
fseek(fp, positie, mode);
```

fp = file-pointer.

**positie** = een (long!) getal dat aangeeft hoeveel bytes verschoven moet worden.

Het beginpunt van de zoektocht wordt aangegeven door **mode**.  
**mode** = 0 beginpunt = vanaf begin van bestand  
= 1 beginpunt = vanaf huidige positie  
= 2 beginpunt = vanaf einde van bestand

fseek() geeft een 0 terug als alles goed is gegaan en een andere waarde als er onderweg iets is mis gelopen. Controleer dit!

### 17.7 ferror() en feof()

**ferror()** - Bij de functies fread() en fscanf() is er reeds op gewezen dat de waarden die deze functies teruggeven, getest dienen te worden op hun waarde (een 0 betekende dat een fout was opgetreden en dat niets geschreven of gelezen was).

```
ferror(fp); /* fp is een file-pointer */
```

kan worden aangeroepen na fread(), fscanf(), fgetc() etc. ferror() geeft als alles goed is gegaan een 0 (dus net andersom als bij b.v. fread()!) en een ander getal als een fout is opgetreden. Dit 'andere getal' is bij sommige compilers een foutmeldingscode (bijv. 1 voor "disk full" en 2 voor "bad track"). Zie hiervoor de handleiding van de compiler.

Bijvoorbeeld:

```
if (ferror(fp))
{
    /* er is een fout opgetreden */
}
```

**feof()** - Pascal programmeurs weten dat voordat gelezen gaat worden vanaf disk eerst gecontroleerd moet worden of EOF(fp) niet aangeeft dat het einde (End) van het (Of) bestand (File) is bereikt. In C gebeurt dat nadat geprobeerd is te lezen, en wel met de functie feof():

```
feof(fp); /* fp is een file-pointer */
```

De waarde die feof() teruggeeft is 0 zolang het einde van het bestand niet is bereikt.

Bijvoorbeeld:

```
if (feof(fp))
{
    /* het einde van het bestand is bereikt */
}
```

## 17.8 Programma-argumenten

In het kopieerprogramma zoals beschreven in de paragraaf 17.2 stonden de te kopiëren bestandsnamen in de programmaregels. Met twee scanf() aanroepen zou eventueel na het opstarten van het programma om deze namen gevraagd kunnen worden. Het is echter veel handiger als direct achter de programmanaam de namen van de twee te kopiëren bestanden staan. Men zou dan met ons kopieerprogramma 'cp' een bestand 'test1' naar 'test2' kunnen kopiëren en wel als volgt:

```
cp test1 test2
```

Hoe programma-argumenten na het opstarten van het programma kunnen worden onderschept, toont het volgende programma dat twee getallen optelt:

```
#include <stdio.h>

void main(argc, argv)      /* NIEUW */
int argc;                  /* NIEUW */
char *argv[];               /* NIEUW */
{
    int getal1,
        getal2;

    if (argc != 3)
    {
        printf("Aanroep moet zijn: %s getal1 getal2\n", argv[0]);
        exit(0);
    }
    else
    {
        getal1 = sprintf(argv[1], "%d", &getal1);
        getal2 = sprintf(argv[2], "%d", &getal2);
        printf("som=%d\n", getal1+getal2);
    }
}
```

1. argc (argCOUNT) geeft het aantal argumenten aan. Als wordt ingetikt 'som 10 20' dan is 'argc' gelijk aan 3 (de programma-naam 'som' wordt dus meegeteld).
2. argv is een array van pointers. Deze array wijst naar een aantal strings ('argc' stuks om precies te zijn). Na de aanroep 'som 10 20' zijn deze strings als volgt gevuld:

```
argv[0] = "som"
argv[1] = "10"
argv[2] = "20"
```

en argc = 3.

3. Als er geen 2 programma-argumenten zijn ingetikt achter 'som' dan moet er een waarschuwing worden gegeven:

```
printf("Aanroep moet zijn: %s getal1 getal2\n", argv[0]);
```

Er staat niet 'De aanroep moet zijn: som getal1 getal2' omdat een gebruiker het programma 'som' gerenamed kan hebben naar 'telop'. Dit verklaart waarom de programmaam ook wordt onthouden door 'argv[0]'. Door met renamen rekening te houden, krijgt de gebruiker nooit onzinnige foutberichten. Een goed gebruik dus.

4. Dat de argumenten als strings worden opgeslagen is lastig als we integers nodig hebben. De functie sprintf() is echter in staat snel een string om te bouwen naar een integer:

```
sprintf(argv[1], "%d", &getal1);
```

De string waarnaar argv[1] wijst wordt als integer geplaatst in de variabele getal1. De "%d" vertelt sprintf() dat een integer gebouwd moet worden en geen float ("%f") of long ("%ld").

Vooral bij DOS commando's worden vaak programma-argumenten gebruikt, denk maar eens aan 'diskcopy A: B:'. De 'A:' en 'B:' zijn in dit geval de argumenten. Een goed gebruik is dat wanneer de gebruiker de vereiste argumenten vergeet in te tikken, het programma een uitleg geeft hoe de argumentenvolgorde er formeel uit hoort te zien EN de gebruiker een tweede kans geeft door nogmaals te vragen de argumenten in te tikken. Dit voorkomt gevallen waarin een gebruiker tien keer een als het graf zwijgend commando moet intikken om alle combinaties te proberen en als enige respons de opmerking 'no such argument' krijgt. Een vraagteken als enige argument zou alleen de uitleg moeten ontlokken aan het commando. Bovendien, nu we toch bezig zijn, moet het mogelijk zijn om zowel 'diskcopy A: B:' als 'diskcopy A: to B:' in te mogen tikken. Wees tolerant en gedraag je intelligent!



## HOOFDSTUK 18

### 18.1 Typedef

Terwille van de leesbaarheid is het mogelijk om de standaard typen van C een andere naam te geven. De oude naam blijft dan ook nog bruikbaar maar de nieuwe naam mag er voor vervangen worden. Op een computer waarop met 32 bits integers gewerkt kan worden, maar ook met 16 bits en 8 bits integers is het wellicht handig om de volgende 'typedefs' bovenaan elk programma te zetten (of beter: in een includefile op te nemen):

```
typedef unsigned int ULONG;
typedef int LONG;
typedef unsigned char UBYTE;
typedef char BYTE;
```

uiteraard is het ook mogelijk een 'pointer type' te definiëren:

```
typedef char *STRPTR;
```

Het is nu mogelijk om:

```
char *str;
```

te vervangen door:

```
STRPTR str;
```

hetgeen duidelijk een pointer naar een string declareerd.

Een andere mogelijkheid is het gebruiken van typedef voor structures:

```
typedef struct
{
    int x, y, breedte, hoogte;
} WINDOW;
```

Nu kan worden gedeclareerd:

```
WINDOW wdw1, wdw2;
```

Het declareren van functies om het terugkeertype te declareren, kan ook met `typedef` in een nieuw jasje worden gestoken. Na de declaratie:

```
typedef int *SOM();
```

kan worden geschreven:

```
SOM s1, s2;
```

in plaats van:

```
int *s1(), *s2();
```

`s1()` en `s2()` zijn functies die een pointer afleveren die wijst naar integer.

## 18.2 #define

Met `define` kan een z.g. `macro` worden gedefinieerd. De eenvoudigste macro is een enkel getal of verzameling karakters:

```
#define BTW 18.5
```

```
#define Error_1 "Harddisk is crashed, buy a new one."
```

deze defines kunnen nu overal in het programma gebruikt worden:

```
printf("Huidige BTW percentage = %f", BTW);
```

```
printf("Error message --->%s", Error_1);
```

Sommige mensen maken er een sport van in een macrodefinitie een zo groot mogelijke expressie te proppen:

```
#define max(a, b) ((a) > (b) ? (a) : (b))
```

Nu kan genoteerd worden:

```
antwoord = max(getal_1, getal_2);
```

Het voordeel is dat voor 'max' geen functie hoeft te worden geschreven en dat de variabelen 'a' en 'b' in de macrodefinitie automatisch worden gedeclareerd door de compiler. Om de compiler niet in de war te brengen is het veelvuldig gebruik van haakjes niet onverstandig. Zet om alle identifiers haakjes alsmede om de gehele expressie en test macrodefinities uit alvorens ze te gebruiken.

De macrodefinitie 'max' kan weer ongedaan worden gemaakt door:

```
#undef max
```

Het gebruik van #define wordt sterk aanbevolen voor de z.g. 'constanten' die in veel programma's voorkomen. Een constante, de naam zegt het al, heeft overal in het programma dezelfde waarde. Mocht nu de waarde van de constante veranderen dan hoeft alleen de #define aangepast te worden als systematisch in het hele programma de #define gebruikt is.

Een ongeschreven wet schrijft voor dat constanten die gedefinieerd worden met een #define met hoofdletters worden geschreven, ze kunnen dan onderscheiden worden van de gewone variabelen die meestal met kleine letters worden geschreven.

### 18.3 De preprocessor en #include

C heeft een krachtig hulpmiddel die de preprocessor wordt genoemd. De preprocessor zorgt ervoor dat tijdens het compileren allerlei taken worden uitgevoerd, bijvoorbeeld het geven van waarden aan constanten met #define zoals werd uitgelegd in de vorige paragraaf. Preprocessor commando's beginnen altijd met een hekje (het # teken) en zijn vanaf de plaats van definitie geldig voor de rest van het programma, tenzij de betekenis wordt opgeheven met #undef.

Met #include wordt de compiler gedwongen tijdens het compileren het bestand dat achter #include staat binnen te lezen en te compileren alvorens verder te gaan met compileren van het hoofdprogramma. Dit wordt genoteerd als:

```
#include "hulp_bestand"
```

De compiler zal als hij deze opdracht tegenkomt het bestand "hulp\_bestand" gaan zoeken op de disk in de directory waarin de 'sourcecode' staat (de huidige directory) en als hij het bestand daar niet vindt zal hij zoeken in speciaal aangewezen 'include' en 'library' directories. Dit is echter afhankelijk van het systeem waarop gewerkt wordt. See your local handboek of de systeembeheerder (waarom heten die altijd Onno?).

De vorm:

```
#include <hulp_bestand>
```

wordt toegepast voor includefiles die altijd in de bovenvermeldde systeemdirectories staan. Het zoeken in de eigen directory wordt dan overgeslagen. In de includefiles mogen trouwens ook weer #includes staan, zo kan je de compiler dus een flink uitstapje laten maken alvorens hij terugkomt bij het hoofdprogramma.

Includefiles worden bijvoorbeeld gebruikt om hele rijen #defines apart op te bergen en om bij elkaar horende functies in losse modulen te zetten. In het algemeen is het verstandig om een programma in overzichtelijke brokken op te splitsen en #include geeft daartoe de mogelijkheid. zie ook hoofdstuk 12.

#### 18.4 Voorwaardelijke compilatie

Voorwaardelijke, of conditionele, compilatie geeft de mogelijkheid om de compiler tijdens het compileren 'te besturen'. Zo is een veel gebruikte toepassing het wel of niet laten compileren van 'diagnostics', dit zijn allerlei tussentijds geprinte berichten en waarden die tijdens de ontwikkeling van een programma erg handig zijn maar die in de 'final version' natuurlijk niet op het scherm mogen verschijnen. De beslissing om echter alle diagnostics uit een programma te slopen is een moeilijke omdat bij een latere uitbreiding van het programma ze er weer ingezet moeten worden. Natuurlijk kan een 'verkoop versie' en een 'sleutel versie' worden gemaakt, maar dan moeten later alle verbeteringen in de 'verkoop versie' worden gezet, waarbij altijd regels worden vergeten. Conditionele compilatie heeft dus echt voordelen.

Als eerste voorbeeld wordt de voorwaardelijke compilatie gebruikt voor een programma waarvan twee versie gemaakt kunnen worden, namelijk een versie voor machines met 1 Mb (Mega byte) geheugen en een versie voor 'beesten' met 4 Mb:

```
#define GEHEUGEN 1 /* 1 Mb */

#if GEHEUGEN >= 4
    void 3d_animatie() /* alleen voor 4 Mb machines */
{
    ...
}
#endif
```

Door nu op één plaats in het programma de #define aan te passen kunnen hele lappen coding in- of buiten werking worden gesteld.

Net als de 'if-else' constructie in programma's, kan else ook gebruikt worden voor voorwaardelijke compilatie:

```
#if GEHEUGEN >= 4
    void 3d_animatie() /* alleen voor 4 Mb machines */
{
    ...
}
#else
    void 3d_animatie() /* alleen voor 1 Mb machines */
{
    printf("Sorry, u heeft minstens 4 Mb nodig.");
}
#endif
```

Het is ook mogelijk om te controleren of een identifier is gedeclareerd met een #define:

```
#define PRINT_DIAGNOSTICS

#ifndef PRINT_DIAGNOSTICS
    printf("De variabele 'ct' heeft de waarde %d\n", ct);
#endif
```

Het is ook mogelijk om te checken of de identifier niet is gedefinieerd:

```
#ifndef PRINT_DIAGNOSTICS
    printf("De diagnostics zullen niet geprint worden!\n");
#endif
```

Deze regel zou in het begin van het programma kunnen staan. Alle #ifdef's zullen worden overgeslagen behalve deze #ifndef (voor de slechte lezers: er staan dus een 'n' tussen 'if' en 'def', de n van NOT).

### 18.5 Enkele trucs met de preprocessor

1. #define kan ook rekenen!

```
#define SECONDEN_PER_UUR (60*60)
```

2. Een veel gemaakt fout in C, namelijk een enkele = schrijven i.p.v. == kan worden vermeden door te definiëren:

```
#define EQU ==
```

Hetgeen de assemblerprogrammeurs bekend zal voorkomen. Men kan nu noteren:

```
if (a EQU b)
    ...
```

3. Pascalfanaten kunnen schrijven:

```
#define then /* definiëer niets ! */
#define begin {
#define end   ;}
```

en dan:

```
if (x > 640) /* helaas: de haakjes moeten wel */
then
    begin
        x = 640;
        y = 0
    end
```

4. Stijlfreaks definiëren:

```
#define EEUWIG ;
```

en dan:

```
for(EEUWIG) { ... }
```

Overigens, stijlfreaks gebruiken geen eeuwigdurende lussen.  
Dat is vies.

## HOOFDSTUK 19

### 19.1 Portability of overdraagbaarheid

Portabiliteitsproblemen treden niet alleen op bij lees- en schrijfacties en verschillen in variabelegroottes maar zeker ook bij uitvoer naar het scherm. Denk alleen al het verschil tussen de IBM PC die een z.g. 'karaktermode' heeft en de Apple Macintosh waar de letters puntje voor puntje (pixels) worden getekend.

Het overdragen of 'porten' naar andere machines levert, zeker als ze in andere talen dan C, zijn geschreven meestal flinke problemen op. C wordt dan ook gebruikt om z'n relatief kleine portabiliteitsproblemen en omdat voor bijna elk computertype een compiler vorhanden is.

Bij het schrijven van programmatuur die ook op andere systemen moet gaan draaien is het van belang enkele zaken in het oog te houden:

1. Scheid de programmadelen die de algemene algoritmen en functies bevatten van de systeem-afhankelijke functies.
2. Roep Operating System functies aan vanuit een eigen functie. Verweef dus geen OS functies in de algemene modulen.
3. Verwerk punt 1 en 2 in aparte modulen zodat diverse teams zich met het porten kunnen bezig houden. Een team past de algemene modulen aan, zij hoeven relatief weinig af te weten van de machine en zijn het snelste klaar. Het tweede team verzorgt het omschrijven van de OS aanroepen. Dit zijn de 'hardware hackers' en zullen meestal een tijdje zoet zijn.
4. Houd vooral rekening met de verschillen in grootte van typen. Een long is niet altijd 32 bits maar soms ook maar 16 bits groot. Gebruik sizeof() om hier achter te komen.
5. Klein detail maar toch belangrijk: gebruik voor wachtlussen geen optellussen maar echt aan tijd gerelateerde lussen. Op machines met een snellere microprocessor heb je vaak zo weinig tijd om welkomsberichten te lezen: de 10 seconden die de programmeur je toebedeeld had, blijken opeens 2 seconden te zijn op jouw supersnelle machine!

Het vreselijke gezegde 'bezint eer ge begint' is hier toch echt van toepassing!

## 19.2 Veel gemaakte fouten

Deze paragraaf is bedoeld als hulpje bij het programmeren en is uiteraard niet compleet omdat het veelvuldig maken van specifieke fouten natuurlijk een persoonlijke kwestie is! Desondanks is het een aanzet tot een soort foutenlijstje die aangevuld kan worden met de persoonlijke fouten top 5 (top 10 ?). Niet voor niets staat deze paragraaf op de laatste bladzijden, de kaft kan gebruikt worden als notitieblad (niet met pen BARBAREN!).

1. Geen '==' maar '=' bij een if-statement, dus:

```
if (a=10) { ... } i.p.v. if (a==10) { ... }
```

2. Verkeerd geneste if-else-statements:

```
if (a==1)
    if (b==2)
        statement_1;
else
    statement_2;
```

Het 'statement\_2' wordt dus niet uitgevoerd als 'a' ongelijk is aan 1 maar als 'a' gelijk is aan 1 en 'b' ongelijk aan 2! Als de programmeur dat wel gewild zou willen hebben dan had het als volgt genoteerd moeten worden:

```
if (a==1)
{
    if (b==2)
        statement_1;
}
else
    statement_2;
```

De fout is er ingeslopen omdat in het eerste programmafragment verkeerd is ingesprongen, het had er zo uit moeten zien:

```
if (a==1)
    if (b==2)
        statement_1;
else
    statement_2;
```

Nu is goed te zien wat het programmafragment doet.

3. De grootte van een variabele is te klein om het getal te kunnen bevatten dat er in wordt gestopt.
4. Het type van een variabele die als argument wordt doorgegeven aan een functie komt niet overeen met het type dat in de functie het argument ontvangt.

5. Een negatief getal wordt in een unsigned type gestopt. Een heel groot getal is het resultaat.
6. '&' wordt gebruikt waar '&&' nodig is en vice versa.
7. '\*' wordt gebruikt i.p.v. '->' en vice versa.
8. Een pointer wordt niet door de vereiste '\*' voorafgegaan.
9. De handle (&) wordt vergeten bij scanf("%d", &i).
10. Bij het lezen en schrijven van bestanden wordt fclose() vergeten. De meeste Operating Systems accepteren ca. 20 openstaande bestanden en zullen dus na een tijdje gaan piepen als systematisch fclose() vergeten wordt. Soms openbaart dit piepen zich echter door een misleidende 'file not found' foutmelding.
11. Het lezen van int's uit een array van char's gaat fout.
12. Programmeer niet na half vijf 's nachts (tenzij het 's ochtends voor 150 man gepresenteerd moet worden...)

### 19.3 Debuggen

Bijna niemand schrijft programma's zonder fouten te maken. De fouten die gemaakt kunnen maken zijn onder te brengen in twee categorieën. De eerste categorie betreft de syntaxfouten. Dit zijn fouten die te maken hebben met een verkeerde spelling, bijvoorbeeld printf() i.p.v. printf() of het vergeten van een punt-comma. Dit soort fouten zijn de minst erge: de compiler ontdekt ze en waarschuwt de programmeur tijdens het compileren. Er zijn zelfs teksteditors die tijdens het intypen van de programmaregels de C keywords herkennen en (bijvoorbeeld) een kleurtje geven. Bij het saven tellen ze ook nog even de haakjes en accolades en geven een waarschuwing als ze een oneven aantal van één van beiden tegenkomen.

De tweede categorie fouten zijn de semantische fouten. Deze fouten hebben betrekking op een verkeerd gebruik van de beschikbare woorden. Zo is de zin "het is 's avonds kouder dan buiten" syntactisch volkomen correct echter semantisch is het wartaal. Ook in C is het mogelijk om programma's te schrijven die de compiler en de linker helemaal goed vinden maar die nooit zullen draaien. Deze categorie fouten zijn het moeilijkst te vinden maar kunnen wel sneller gevonden worden door de volgende aanwijzingen op te volgen:

- 1) **Modulen** - Splits elk programma groter dan een paar schermpagina's op in modulen. Berg in de modulen de functies op die bij elkaar horen: de schermopbouwroutine's, de in- en uitvoer routine's, de routine's die de invoer verwerken etcetera. De modulen mogen best groot zijn als ze maar bij elkaarhorende functies bevatten.

- 2) **De modulen uitwerken** - Ga de modulen uitwerken in volgorde van belangrijkheid. Maak van elke functie een waanzinnig mooie 'black box' d.w.z. je kunt er informatie in stoppen, de functie kauwt er op en geeft de verwerkte informatie terug zonder dat de aanroeper hoeft te weten hoe de functie werkt. In C is het zó eenvoudig om functies te maken dat zelfs stukken coding die maar 1 maal worden aangeroepen in een functie worden opgeborgen enkel en alleen voor de overzichtelijkheid.

Modulen die nog niet nodig zijn, bijvoorbeeld in de eerste fase van het programmeren de in- en uitvoer, kunnen bestaan uit 'functierompen' die alleen een boodschap printen:

```
void SavePicture()
{
    printf("SavePicture aangeroepen");
}
```

- 3) Geef alle functies de volgende beschrijving mee:

```
***** functienaam() *****/
/*
 * input(s):
 *
 * output(s):
 *
 * description:
 *
 */
int functienaam()
{
    ...
}
```

- 4) Voeg aan een nieuwe functie altijd enige printf() aanroepen toe die waarden en/of strings afdrukken. Het is erg nuttig de beginwaarde te printen ("is de beginwaarde goed doorgegeven aan de functie?"), de tussentijdse waarden ("worden alle for-lussen goed doorlopen, blijft een while-lus niet hangen, zijn alle if-statements goed?") en is de eindwaarde correct ("is de eindwaarde goed of moet er nog 1 bij opgeteld of afgetrokken worden?").

Bijvoorbeeld: In het onderstaande programmafragment is op een strategische positie een printf() opgenomen. De printf() drukt alle waarden af die de functie genereert tijdens z'n aanroep:

```

int faculteit(n)
int n;
{
    printf("n = %d\n", n);

    if (n <= 0)
        return(1)
    else
        return(n * faculteit(n-1));
}

```

Ook met strings kunnen vervelende dingen gebeuren. Het missen van de '\0' zal bij het printen of aanroepen van bijvoorbeeld strcpy() nare effecten tot gevolg hebben. Bedenk ook dat spaties slecht zichtbaar zijn op het scherm, print 'debug strings' dan ook zo:

```
printf("string is [%s]\n", str);
```

De afgedrukt string ziet er dan bijvoorbeeld zo uit:

```
string is [c:\dos\dir      ]
```

De programmeur kan nu zien dat er 5 verdwaalde spaties in de string zitten.

- 5) Ga bij het debuggen er niet van uit dat de nieuwe functie persé fout is. Als in de nieuwe functie oude functies worden aangeroepen die vroeger alleen elders werden aangeroepen dan is het verstandig die oude functies nog eens grondig te inspecteren. Veel fouten ontstaan door het aannemen dat die oude functie bijv. een long wil hebben. Laadt hem nog eens binnen en zie dat hij een int wil! Tsja...
- 6) Bouw direct in elke functie een goede controle op fouten. Functies die geen waarden teruggeven in de vorm van variabelen kunnen altijd TRUE (1) of FALSE (0) teruggeven (definiëer zelf met typedef een BOOL type!). Functies die wel waarden teruggeven, kunnen bijvoorbeeld -1 teruggeven als er in de functie een fout optreedt. Ga er bij in- en uitvoer vanuit dat er geen diskette in de drive zit of dat de harddisk vol is. Ga er bij geheugenallocatie vanuit dat het programma draait op de kleinste versie van een machine. Ga er bij het tekenen op het scherm vanuit dat de gebruiker een A3 paginaopmaakscherm heeft of geen kleuren. Ga er altijd vanuit dat alles fout kan gaan. Een goede manier om een programma te testen is te checken (in de source coding) of in het geval dat alles mis gaat het programma stopt (met een foutmelding) en dat alles in de staat is waarin het werd aangetroffen (geheugen vrij etc.).
- 7) Probeer foute coding te isoleren. Als een programma vastloopt in een bepaalde functie en het is onduidelijk waar precies dan kan door middel van het buiten werking stellen van gegarandeerd correcte coding (door er commentaar van te maken) de foute coding geïsoleerd worden.

Corrigeer de fout en voeg stukje bij beetje de 'eruit gecommentaarde' coding weer toe. Controleer steeds of de gecorrigeerde coding nog steeds werkt.

- 8) Debuggen doe je zelf, bugs vinden niet! Programmeurs hebben een perfect ontwikkeld gevoel voor het omzeilen van hun eigen fouten maar gebruikers vinden ze gegarandeerd. Wees niet bang een programma voor de leeuwen te gooien, het wordt alleen maar beter van!
- 9) Als het printen van diagnostics (printen van variabelen, strings etc.) bemoeilijkt wordt door bijv. het openen van een grafisch scherm zodat de geprintte uitvoer niet meer zichtbaar is dan zijn er nog enkele mogelijkheden om de diagnostics te bekijken:
  - Sluit de ontwikkelcomputer aan op een tweede computer of een terminal en stuur alle diagnostics via een seriële kabel daar naar toe.
  - Stuur alle diagnostics naar een bestand op disk en lees de file als het programma beeindigd is.
  - Stuur alle diagnostics naar de printer.

#### 19.4 Het einde en een nieuw begin

Ik hoop dat de lezer van dit boek de taal C nu net zo mooi vindt als ik. Het investeren van tijd in het leren van een programmeertaal levert naast de 'fun', volgens mij, ook speciale mensen op met speciale gedachten en oplossingen voor alledaagse dingen. Als je niet aan het programmeren bent, heb ik nog wat leestips (wel Engelse boeken):

Be Here Now, Baba Ram Dass  
Lama Foundation, 1971

Odysssey: Pepsi to Apple, John Sculley  
ISBN 0 00 637284 8

Steve Jobs, The journey is the reward, Jeffrey S. Young  
ISBN 1 85181 190 7

The soul of a new machine, Tracy Kidder  
ISBN 0 14 00 6249 1

Zen mind, beginner's mind, Shunryu Suzuki  
Weatherhill, 1970

Erik van Eykelen  
Alkmaar, juni 1990

