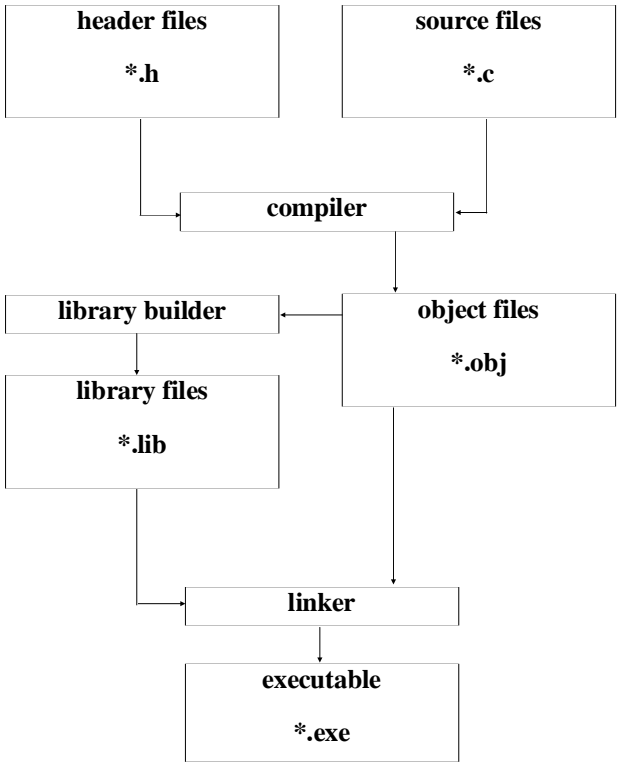
PROGRAMMEREN IN C

CURSUS VOOR STARTERS

J.W. Welleman



PROGRAMMEREN IN C

CURSUS VOOR STARTERS

J.W. Welleman

september 1993

2^{e} druk

INHOUDSOPGAVE

1 INLEIDING

De programmeertaal ***C*** wordt steeds meer toegepast door professionele programmeurs vanwege de flexibiliteit welke de taal biedt. Er zijn inmiddels diverse compilers voorhanden waarmee ***C***-programma's gecompileerd kunnen worden tot objectcode. De werking van de compilers kunnen naar de gebruiker toe nog al eens verschillen qua verschijning en opmaak maar in feite verschillen ze in hun werking niet veel. Sommige gebruikers zullen de voorkeur geven aan de uitgebreide programmeer omgeving welke **TURBO-C** van **Borland** biedt, andere werken liever vanaf de *commandline'* met eenvoudige compileer instructies en een separate editor (ook dit kan met **TURBOC**). Deze laatste manier van werken vertoont veel gelijkenis met de werking van een ***C***-compiler in een **UNIX** omgeving. Andere compilers zijn bijvoorbeeld de **Microsoft C** compiler en de relatief onbekende **WATCOM** compiler welke geschikt is om *protected mode* objectcode te genereren voor 386 en 486 machines. Groot voordeel van ***C*** is de uitwisselbaarheid van de code tussen diverse platforms (**DOS**, **UNIX**) mits er gebruik gemaakt wordt van de **ANSI-C** standaard (American National Standards Institute).

Aan de basis van ***C*** stonden Dennis Ritchie en Brian Kernighan. ***C*** werd ontwikkeld vanuit de **UNIX** wereld en stamt vanuit het begin van de jaren 70. In 1983 werd de **ANSI-C** standaard van kracht waardoor het gebruik van ***C*** een enorme impuls kreeg. De taal ***C*** wordt wel eens aangeduid als een *middle-level language* wat zoiets inhoudt als het midden tussen assembly taal en een hogere programmeertaal. Deze omschrijving zou gebruikers kunnen afschrikken ware het niet dat ***C*** ook heel veel trekjes bezit van hogere talen als **PASCAL** en **FORTRAN**. Het aardige van ***C*** boven deze twee laatste talen is juist dat er op bepaalde momenten ook op een wat lager niveau, lees meer direkt richting hardware toe, geprogrammeerd kan worden. Zo kan er prachtig gemanipuleerd worden met geheugenadressen. Dit geeft aan de gebruiker een grote mate van vrijheid hetgeen ook meteen een gevaar is. Waar in **PASCAL** door de compiler nog wel eens een waarschuwende vinger wordt opgestoken gaat de ***C***-compiler gewoon en binnen de korste tijd kun je in grote problemen verzeild raken. Het *debuggen* van het programma kan dan een gigantische klus worden. Niet zelden zit de fout op een totaal andere plaats dan waar je zoekt.

Belangrijk is dus de manier van werken. In deze korte inleidende cursus zal daar vooral aandacht aan worden besteed. Als de structuur van het programma goed doortimmerd is en er consequent omgesprongen wordt met het declareren, initialiseren en doorgeven van data zal op den duur gemerkt worden dat complete programma's in een keer "foutloos" lopen.

Dit rapport is grofweg op te delen in twee delen, het eerste deel betreft hoofdstuk 1 tot en met 6 waarin een overzicht gegeven wordt van de basis componenten van de taal ***C***. Dit is een soms taaie opsomming van afspraken en regeltjes die nu eenmaal bij het leren van een nieuwe taal horen. Het gaat dan met name om :

- basis componenten van een ***C***-programma
- datatypen, variabelen, constanten, operatoren en expressies
- standaard **IO** (input output)
- programma controle
- subroutines

Daarmee is in hoofdlijnen de taal en de opzet van een ***C***-programma uiteengezet.

In het tweede gedeelte wordt ingegaan op de de leuke kanten van ***C*** :

- complexere data typen zoals structures
- arrays
- pointers
- dynamisch geheugen gebruik

Een aanzet voor ***C++*** is gegeven in hoofdstuk 11. Dit hoofdstuk is meer een kennismaking met ***C++*** en kan zonder problemen overgeslagen worden door beginnende ***C*** *programmers.*

In hoofdstuk12 wordt ingegaan op het maken van een executable welke uit meerdere bronbestanden (*source code*) kan bestaan die elk apart gecompileerd kunnen worden, eventueel zelfs tot *library* bestanden. Hiervoor is het nodig om met *makefiles* om te kunnen gaan en te weten hoe de diverse *object*-bestanden kunnen worden *gelinkt*.

Tot slot wordt in het laatste hoofdstuk een voorbeeld behandeld waarin zoveel mogelijk gebruik wordt gemaakt van eerdere behandelde stof. Het gaat daarbij niet om het op te lossen probleem maar om de wijze van implementatie en de strukturering.

Hoewel vaak gerefereerd wordt aan de **TURBO-C** compiler van **Borland** is dit rapport in eerste instantie gericht op standaard ***C*** zodat de code in de voorbeelden ten alle tijden op diverse platforms en met diverse compilers gebruikt zal kunnen worden. Doel is dan ook om aan het einde van de cursus een goed overzicht te hebben wat je met ***C*** kunt doen en hoe je zelf een ***C***-programma in elkaar zet. In hoofdstuk 12 zal overigens sumier worden aangegeven welke verschillen er zijn tussen de **TURBO-C** compiler van **Borland** en de **gcc** compiler onder LINUX.

2 PROGRAMMA STRUCTUUR

Een goed geschreven computerprogramma zal moeten bestaan uit een zorgvuldig van te voren vastgelegde structuur. Zomaar in het wilde weg programmeren is niet aan te bevelen. Een dergelijk programma is niet te onderhouden. De verschillende manieren van programmeren worden ook wel eens aangeduid met :

- top-down
- bottom-up
- ad hoc

De laatste methode is de eerder genoemde "wilde weg programmeren" methode. De eerste methode is de meest voor de handliggende methode. Er wordt begonnen met het hoofdprogramma dat opgedeeld wordt in kleinere programma onderdelen. Als laatste komen de details aan bod zoals kleine rekenkundige exercities en lees- en schrijfroutines. De *bottom-up* methode is ook een gestructureerde methode. Hier verloopt het proces echter precies in de omgekeerde volgorde. De *top-down* methode dwingt de programmeur om een duidelijke rode lijn door het programma te volgen. Deze structuur kan vooraf al vastgelegd worden in stroomschema's en dergelijke. In deze cursus gaan we daar niet verder op in. Benadrukt wordt echter het belang van een goede structuur al dan niet in de vorm van stroomschema's.

C is bij uitstek geschikt om goede gestructureerde programma's te schrijven. Deze structuur kan op verschillende niveau's worden aangebracht (globaal naar lokaal) :

- opdelen van programma in verschillende bronbestanden : *header files* en *libraries*
- opsplitsen van programma code in verschillende *functions* of subroutines
- gebruik maken van *compount statements* in de code

Deze mogelijkheden zullen tijdens de cursus verduidelijkt worden. We beginnen nu eerst met de hoofdstructuur van een ***C***-programma..

2.1 Minimaal Programma

Zoals bij het leren van elke nieuwe (computer) taal wil je als gebruiker zo snel mogelijk je gedachten kunnen overbrengen in de nieuwe taal. Daarom zal in deze paragraaf een heel kort nietszeggend programma worden behandeld waar echter wel alle structurele componenten van een ***C***-programma in te vinden zijn.

Een ***C***-programma moet tenminstse één zogenaamde *functie* bevatten, de hoofdroutine of ook wel hoofdprogramma. De hoofdfunktie in een ***C***-programma wordt aangeduid met de naam *main()*. Een basis ***C***-programma kan er dus als volgt uitzien :

```
main()
{
}
```

voorbeeld 2.1 : Basis C-programma

De accoladen **{** en **}** geven het begin en einde aan van de functie *main()*. Dit programma doet nog helemaal niets maar kan in principe wel gecompileerd worden. Als we echter meer dan dit willen en gebruik willen kunnen maken van verschillende in ***C*** reeds aanwezige routines zullen we moeten aangeven welke bronbestanden, met daarin de definities van deze functies, we willen gebruiken. Deze bestanden worden *header-files* genoemd en zijn te herkennen aan de extensie *.h* . Er zijn er nogal wat van maar één van de belangrijkste is de file *stdio.h* waarin alle standaard in- en output wordt geregeld. Om de compiler duidelijk te maken dat we de funkties uit *stdio.h* willen gebruiken moeten we door middel van een *#include* statement de stdio.h "laden" in ons programma. Ons programma kan er dan als volgt uit komen te zien :

```
/* vb2_2.c                                                  */
/*                                                          */
/*   <<<  zo geef je dus een commentaarregel weer  >>>      */
/*                                                          */
/* Een eenvoudig basis C programma dat niets doet !         */

#include <stdio.h>

main()
{ printf("Hallo Cursist ");
}
```

voorbeeld 2.2 : Basis C-programma met include

Ons programma is nu in staat om de regel :

```
Hallo Cursist
```

op het scherm te produceren. De afdruk funktie *printf()* hebben we hier gebruikt om de tekst op het scherm te toveren.

Het is een goede gewoonte om boven in een file te zetten welke naam de file bevat en wat erin wordt beschreven. Zeker als een programma opgebouwd is uit verschillende bronbestanden wordt commentaar belangrijk. De hoeveelheid commentaar kan onbeperkt zijn en heeft geen invloed op de uiteindelijke grootte van de objectcode. Er kan niet genoeg commentaar in je programma staan. Gaande weg zal dat duidelijk worden.

Zoals eerder reeds vermeld is het aan te raden een duidelijke structuur in het programma te volgen. Feitelijk wordt de programmeur door ***C*** hiertoe gedwongen. Uit het bovenstaande voorbeeld blijkt dat al een beetje :

- bovenaan in het programma dienen de *header files* aangeroepen te worden
- een eventueel blok met definities van constanten
- daarna kan een blok met nieuwe data-definities volgen
- vervolgens wordt de globale data gedeclareerd en eventueel geinitialiseerd
- waarna bijvoorbeeld diverse ondersteunende funkties volgen
- gevolgd met als laatste de hoofdfunktie *main()*

Strikt gesproken kan er van deze volgorde worden afgeweken maar dat is in deze fase van de cursus nog niet aan de orde. In het hierna volgende voorbeeld wordt het bovenstaande wat verduidelijkt.

```
/* vb2_3.c                                                    */
/*                                                            */
/*                                                            */
/* Een eenvoudig basis C programma dat iets doet !            */

#include <stdio.h>                /* include blok             */
#include <math.h>

#define PI 3.141512               /* constanten blok          */

typedef double guldens;           /* nieuwe data              */
                                  /* definitie                */
                                  /* guldens is een           */
                                  /* nieuw type gelijk        */
                                  /* aan double               */

guldens totaal;                   /* declaratie van           */
                                  /* variabele totaal         */
/*-----------------------------------------------------------*/
/*   SUB PROCEDURES OF FUNCTIES                              */
/*-----------------------------------------------------------*/
nietsnut()
{ printf("dit is de funktie nietsnut\n");
}      /* \n in een string betekent : een nieuwe regel */

print_uitgaven(bedrag) /* functie met variabele welke  */
  guldens bedrag;      /* onveranderd geprint wordt    */
{ printf("bedrag = %lf\n",bedrag);
}      /* %lf is het afdruk formaat van de variabele   */

/*-----------------------------------------------------------*/
/*            H O O F D   P R O G R A M M A                  */
/*-----------------------------------------------------------*/
main()
{ int i;                    /* data initialisatie           */
  totaal=1.0;               /* data initialisatie           */
  printf("Hallo Cursist");
  nietsnut();               /* functie aanroep nietsnut*/
  for (i=0; i < 10; i++)    /* 10 keer een loop van :  */
  { totaal= i * PI * totaal;/* - rekenkundige bewerking*/
    print_uitgaven(totaal); /* - print variabele       */
  }
}
```

voorbeeld 2.3 : Basis-structuur van een programma

Het helemaal snappen van dit programma wordt hier niet geeist. Belangrijk is dat de algemene structuur van het programma, *qua layout*, duidelijk wordt. Gebruikers van **PASCAL** zullen de overstap van **PASCAL** naar ***C*** wellicht niet al te groot vinden aangezien ***C*** min of meer dezelfde opbouw volgt.

Nu we bekend zijn met de uiterlijke verschijning van een ***C***-programma wordt het tijd om iets gestructureerder naar de data-typen, definities en declaraties van variabelen te gaan kijken.

3 DATA IN C

Dit hoofdstuk heeft betrekking op de data welke in ***C*** gebruikt wordt. Aan de orde zal komen hoe data gedeclareerd kan worden, welke typen data er zijn en het verschil tussen *locale* en *globale data*. Voordat hiermee begonnen wordt zal eerst een lijstje worden gegeven van de woorden die we niet mogen gebruiken voor het aangeven van onze data aangezien deze namen reeds in de taal ***C*** gebruikt worden. Het betreft hier de zogeheten *gereserveerde woorden* van de taal **C** (*keywords*).

3.1 Gereserveerde woorden

De gereserveerde woordenlijst maakt deel uit van de syntax beschrijving van een programmeertaal. Voor ***C*** zijn er volgens de **ANSI** standaard 32 *keywords* gereserveerd. **TURBO-C** heeft deze uitgebreid met nog eens 11 woorden tot 43 gereserveerde woorden. Deze zijn in de tabel hieronder weergegeven.

tabel 3.1 : Lijst met gereserveerde woorden

Gereserveerde woorden volgens de **ANSI** standaard			
auto	double	int	struct
break	else	long	switch
case	enum	register	typedef
char	extern	return	union
const	float	short	unsigned
continue	for	signed	void
default	goto	sizeof	volatile
do	if	static	while
Extra gereserveerde woordenlijst van **TURBO C**			
asm	_cs	_ds	_es
_ss	cdecl	far	huge
interrupt	near	pascal	

Het zal duidelijk zijn dat deze woorden niet gebruikt mogen worden voor datatypen definities of voor namen van data.

3.2 Namen voor variabelen en datatypen

De namen welke in ***C*** gebruikt mogen worden voor variabelen, funkties, labels en andere objecten worden ook wel *indentifiers* genoemd. Een indentifier mag bestaan uit maximaal 32 karakters met als eis dat het eerste karakter een letter of een *underscore* (_) moet zijn. **TURBO-C** zal naast deze ***C*** afspraak ook het dollar teken ($) in een indentifier accepteren maar het gebruik hiervan wordt afgeraden. Belangrijk in ***C*** in tegenstelling tot vele andere talen is dat de namen *case-sensitive* zijn. Hiermee wordt bedoeld dat hoofdletters en kleine letters als verschillende karakters worden onderscheiden. Een voorbeeld hiervan kan hieronder worden gevonden.

```
Namen voor Indentifiers :

else    fout want else is een gereserveerd woord
Else    correct
ELSE    correct maar is iet anders dan Else
123     fout
_123    correct
```

voorbeeld 3.1 : Namen voor Indentifiers

3.3 Datatypen

Een datatype is zoals het woord doet vermoeden een aanduiding of een beschrijving van een verzameling van waarden welke aan een variabele van een bepaald type kan worden toegekend. ***C*** kent vijf verschillende basis datatypen zoals is weergegeven in de volgende tabel.

tabel 3.2 : Basis Datatypen

Type	aantal bits	getalswaarde
char	8	0 t/m 255
int	16	-32768 t/m 32677
float	32	3.4E-38 t/m 3.4E+38
double	64	1.7E-308 t/m 1.7E+308
void	0	geen waarde

Het type *char* wordt gehanteerd voor het bewaren van **ASCII** karakters zoals bijvoorbeeld namen. Het integer type *int* wordt gebruikt voor integers waarbij er op de grenzen van dit type moet worden gelet. Dit is met name van belang bij het gebruik van integers in lange array's (waarover later meer). De typen *float* en *double* kunnen voor reële getallen worden gebruikt. Het enige verschil is de maximale waarde welke opgeborgen kan worden. Deze twee laatste typen worden vaak door elkaar gehanteerd hetgeen onjuist is ! Zorg er altijd voor dat een variabele van het juiste type is bij het doorgeven aan funkties omdat anders moeilijk te traceren *bugs* zullen optreden. Het laatste datatype in tabel 3.2 heeft geen inhoud en is als zodanig overbodig. Echter om de compiler te laten weten dat een variabele (b.v. een funktie) geen waarde heeft kan dit datatype gebruikt worden. Het gebruik van het type *void* zal verderop verduidelijkt worden.

3.3.1 Type Modifiers

Soms is het nodig om bijvoorbeeld een integer waarde groter dan 32767 op te bergen. In een *int* lukt dat niet en als *float* of *double* zijn we de specifieke kenmerken van een integer kwijt. Daarom bestaan er in ***C*** zogenaamde *type modifiers* waarmee het basis datatype veranderd kan worden. ***C*** beschikt over vier modifiers :

signed
unsigned
long
short

Met deze *modifiers* kunnen de vijf basis datatypen uitgebreid worden tot veertien combinaties die samen de datatype set vormen van **ANSI-C**. Deze veertien typen zijn weergegeven in tabel 3.3.

tabel 3.3 : Alle mogelijke combinaties van *datatypen* en *modifiers*

Type	aantal bits	getalswaarde
char	8	-128 t/m 127
unsigned char	8	0 t/m 255
signed char	8	-128 t/m 127
int	16	-32768 t/m 32767
unsigned int	16	0 t/m 65535
signed int	16	-32768 t/m 32767
short int	16	-32768 t/m 32767
unsigned short int	16	0 t/m 65535
signed short int	16	-32768 t/m 32767
long int	32	-2147483648 t/m
		2147483647
signed long int	32	-2147483648 t/m
		2147483647
float	32	3.4E-38 t/m 3.4E+38
double	64	1.7E-308 t/m 1.7E+308
long double	64	1.7E-308 t/m 1.7E+308

Het verschil tussen *signed* en *unsigned int* zit hem in de interpretatie van het hoogste bit. Een *signed int* (16 bits) met de waarde 32767 heeft de volgende binaire representatie :

0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1

Terwijl -32768 er als volgt uitziet :

1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
sign ↑

Het eerste getal (hoogste bit) (0 of 1) is het zgn. *sign bit*. Als een variabele echter van het type *unsigned int* is levert een *sign bit* 1 met daarachter allemaal eenen het getal 65535.

3.3.2 Afdrukken van data

ANSI-C kent een eenvoudige print-funktie waarmee data van verschillende typen kunnen worden afgedrukt naar het scherm. Verderop in een hoofdstuk over invoer en uitvoer zal deze funktie verder besproken worden maar het gebruik ervan zal hier worden uiteengezet.

De *printf()* funktie is formeel gedeclareerd als :

printf*("control string", argument list);*

In de *control string* zal het datatype moeten worden aangegeven en de wijze waarop een variabele moet worden afgebeeld. In dit gedeelte mag ook tekst staan. In de *argument list* volgen vervolgens de variabelen welke afgedrukt dienen te worden.

Elk *datatype* heeft een vaste code welke voorafgegaan wordt door een % teken. Een complete lijst wordt gegeven in tabel 3.4.

tabel 3.4 : Format Codes van *printf()*

Code	Format
%c	een enkel karakter
%d	decimaal
%i	decimaal
%e	wetenschappelijke notatie
%f	decimaal floating point
%g	gelijk aan kortste notatie van of %e of %f
%o	octaal
%s	string van karakters : bijvoorbeeld printf("%s","hallo");
%u	unsigned decimaal
%x	hexadecimaal
%%	letterlijk %
%p	pointer
%-[]	links uitgelijnd : bijvoorbeeld %-5.1f = links uitgelijnd 5 cijfers, 1 achter de punt

Twee *type modifiers* beschikken over een eigen *format* welke ook wordt aangeduid met behulp van een letter en kunnen worden gecombineerd met de aangegeven formats :

short *h* te combineren met **: d**, **i**, **o**, **u** en **x**.
long *l* te combineren met : **d**, **i**, **o**, **u**, **x**, **e**, **f** en **g**.

Naast de type aanduiding kan ook de verschijning worden beinvloed. Bijvoorbeeld het aantal cijfers dat een getal moet bevatten of het aantal cijfers achter de punt. Ook de plaatsing van het datatype zoals links- of rechts- *(default)* uitgelijnd kan worden opgegeven (zie tabel 3.4). In het volgende voorbeeld wordt de *printf()* funktie verduidelijkt.

```
/* vb3_2.c : Een stukje code voor printf()             */
#include <stdio.h>

void main()            /* main geeft geen waarde terug */
{ double prijs=90.0;
  int n=20;
  unsigned long int ergveel=1111111;
  printf("Hallo %3d cursisten,deze cursus kost u fl %-5.2lf\n",
         n,prijs);
  /* een unsigned long int geeft u weer als */
  printf("groot integer getal %ld\n",ergveel);
}
```

voorbeeld 3.2 : De ***printf()*** funktie

3.3.3 Constanten

Naast de genoemde datatypen bestaan in ***C*** ook *constanten*. Een *constante* kan hetzelfde type voorstellen als eerder beschreven. Zo kan iets een *integer constante* zijn :

aantal = 15;

Daarnaast kent ***C*** ook een *string constante* welke een string van karakters bewaard. In de tabel op de vorige pagina is bij *%s* hier een voorbeeld van gegeven :

"hallo" is een string constante

Naast deze string constante kent ***C*** ook een aantal zogenaamde *karakter constanten*. Deze bestaan uit een enkel karakter tussen quotes. Als voorbeeld :

'a' is een karakter constante

Tevens bestaan er een aantal bijzondere *karakter constanten* voor het gebruik in uitvoer routines. Deze kunnen dus gebruikt worden voor de opmaak van bijvoorbeeld het scherm. In de volgende tabel worden ze weergegeven.

tabel 3.5 : Karakter constanten

Code	Betekenis
\b	backspace
\n	new line
\r	carriage return
\t	horizontale tab
\"	dubbele quotes
\'	enkele quote
\0	null (einde string)
\\	letterlijke backslash
\v	verticale tab
\a	alert
\o	octale constante
\x	hexadecimale constante

Uiteraard kunnen in ***C*** ook *constanten* gedefinieërd worden zoals bijvoorbeeld in **PASCAL**. In ***C*** wordt dan gebruik gemaakt van het *#define* commando. Hiermee kunnen niet alleen *constanten* worden aangeduidt maar complete *macro's* worden gebouwd. In voorbeeld 3.4 is reeds kennis gemaakt met een *constante* **PI** welke als een *#define* is gedefinieërd. **PI** kan nu overal gebruikt worden als een *constante* met waarde 3.141512. Let erop dat er *geen* **;** in de *#define* staat omdat we de **;** in de code zetten zoals bijvoorbeeld in de volgende toekennings opdracht :

*omtrek = 2.0 * PI * r ;*

De macro definities zullen verderop in de cursus nog wat uitgebreider toegelicht worden.

3.4 Declaratie van variabelen

In het hiervoor weergegeven voorbeeld hebben we al kennis kunnen nemen hoe data van een bepaald type gedeclareerd wordt. Formeel ziet een declaratie er altijd als volgt uit :

type_specifikatie lijst_van_variabelen

Alle typen genoemd in tabel 3.3 mogen gebruikt worden. Maar ook eigen datatypen kunnen hier gehanteerd worden (daarover later meer). Achter de type aanduiding volgt de lijst met namen van de variabelen gescheiden door comma's. De lijst wordt afgesloten met een punt comma.

Vanaf het moment dat data gedeclareerd is kan deze met behulp van de variabelenaam een waarde toegekend krijgen. Voor de naam van een variable verwijzen we naar de afspraken zoals eerder gemaakt in paragraaf 3.2.

De declaratie van enkele variabelen kan er als volgt uitzien.

```
/* vb3_3.c : Declaratie van variabelen                */
void main()
{ double prijs, huur, weet_ik_wat;
  short int weinig;
  int i,j,k,l;
  char letter, naam[10];
}
```

voorbeeld 3.3 : Declaratie van variabelen

De declaratie van data kan in een ***C***-programma op drie plaatsen plaatsvinden. Afhankelijk van de plaats van declaratie spreken we over *globale data* (buiten alle funkties), *locale data* (binnen funkties) en *formele data* (in de parameter lijst van de funktie definities). Het is van belang om dit onderscheid goed te onderkennen. Daarom zal van elke vorm een omschrijving worden gegeven omdat juist in ***C*** deze scheiding van data essentieel is. Als voorbeeld wordt voorbeeld 2.3 op pagina 5 gebruikt.

3.4.1 Globale data

Globale data is overal in het programma bekend. De declaratie vindt plaats buiten alle funkties. In voorbeeld 2.3 op pagina 5 is de variabele *totaal* globaal gedeclareerd als *type guldens*. Dit *type guldens* komt overeen met het *type double* aangezien met behulp van het *typedef* commando een eigen typeaanduiding is aangemaakt. De variabele *totaal* kunnen we nu overal in het programma gebruiken. We zien dat gebeuren in de funktie *main()*. Hier wordt *totaal* gebruikt maar is niet gedeclareerd.

3.4.2 Locale data

In *main()* is *i* als locale data gedeclareerd. Deze *i* is dus alleen in *main()* bekend. In een andere funktie kunnen we gerust weer een variabele *i* declareren want de locale variabelen gelden alleen in de funktie waar ze gedeclareerd worden. Deze variabelen worden gecreëerd wanneer de funktie wordt uitgevoerd en worden vernietigd bij het verlaten van de funktie.

3.4.3 Formele data

We zien dat *totaal* in *main()* wordt doorgegeven aan de funktie *print_uitgaven()*. Eigenlijk is dat dus niet nodig in dit geval omdat *totaal* globale data is. Echter als we ook nog andere data van het *type guldens* willen afdrukken is het wel zinvol om aan de funktie de variabele van een bepaald type door te kunnen geven. We spreken dan van *formele data*. De funktie *print_uitgaven()* verwacht dus een variabele van het *type guldens*. In de funktie wordt deze variabele verder aangeduid als *bedrag*. Zodra *print_uitgaven(totaal)* aangeroepen wordt zal in de funktie *print_uitgaven() totaal* de plaats innemen van *bedrag*.

Tussen de wijze van declareren van globale en locale data zit geen verschil. Formele data kan echter op twee verschillende manieren worden aangegeven. Er is een zogenaamde *moderne* versie volgens de **ANSI-C** standaard en een *klassieke* versie volgens de taaldefinitie van *Kernighan en Ritchie*. De keuze tussen de twee versies is afhankelijk van de te gebruiken compiler. In het onderstaande voorbeeld worden beide versies van *formele data* declaratie aangegeven.

```
/* vb3_4.c : Formele declaratie klassiek/modern        */
#include <stdio.h>

/*-------------------------------------------------------*/
/* klassieke notatie volgens Kernighan en Ritchie        */
/*-------------------------------------------------------*/
void doe_iets_1(x,y,z) double x; int y; float z;
{ printf("%f %d %f\n",x,y,z);
}

/*-------------------------------------------------------*/
/* moderne ANSI-C notatie                                */
/*-------------------------------------------------------*/
void doe_iets_2(double x,int y,float z)
{ printf("%f %d %f\n",x,y,z);
}

void main()
{ double getal1=100.0 ;
  int    getal2= 10   ;
  float  getal3=  3.64;
  doe_iets_1(getal1,getal2,getal3);
  doe_iets_2(getal1,getal2,getal3);
}
```

voorbeeld 3.4 : Formele data

In de rest van deze cursus zal de *klassieke* versie worden gehanteerd. De meeste **UNIX** en **DOS** compilers accepteren deze versie.

3.5 Static locale data

In sommige gevallen kan het prettig zijn als een *locale variabele* zijn waarde kan behouden nadat de funktie waarin deze variabele is gedeclareerd is verlaten. De eerst volgende keer dat een dergelijke funktie weer wordt aangeroepen staat de variabele met zijn oude waarde weer tot onze beschikking. Om dit te realiseren kan gebruik gemaakt worden van de aanduiding *static*. In het volgende voorbeeld houden we bijvoorbeeld bij hoe vaak

een funktie wordt aangeroepen. Dit kan uiteraard ook door gebruik te maken van globale data maar voor de overzichtelijkheid van het programma is het beter om *zo min mogelijk globale variabelen* te gebruiken en *zoveel mogelijk* data door middel van *formele declaraties* aan funkties door te geven.

```
/* vb3_5.c : Static data                                          */
#include <stdio.h>

/*-------------------------------------------------------------*/
/* doe_iets is type int en geeft een integer terug             */
/*-------------------------------------------------------------*/
int doe_iets(x,y,z) double x; int y; float z;
{ static int gedaan=0;           /* waarde van gedaan blijft */
  printf("%f %d %f\n",x,y,z);
  return (gedaan++);        * hoog gedaan op, geef waarde af */
}

void main()
{ int i;
  for (i=0; i<3; i++)
  printf("aantal keer gedaan : %d\n",doe_iets(1.0,23,0.14));
}
```

voorbeeld 3.5 : Static data

In de hoofdfunktie *main()* zien we nog een aardigheidje van ***C***. De funktie *doe_iets()* beschikt over *formele data* en krijgt zelf ook een waarde. Daarom kan in de *printf()* in *main()* de waarde van *doe_iets()* worden afgedrukt. Op deze manier kan compact geprogrammeerd worden. Naast *static locale data* bestaat in ***C*** ook de mogelijkheid om *static globale data* te definiëren. Dit is echter alleen van belang bij grote programma's die uit verschillende files bestaan. De *globale static data* is dan bijvoorbeeld alleen globale data in een bepaalde file.

3.6 Register data

Nog een vorm van data manipulatie is het declareren van data met als *storage specifier register*. Dit kan alleen met *locale data* van het type *int* en *char*. De bedoeling van deze manipulatie is om de variabele *niet* in **RAM** te onthouden maar op te bergen in het *register* van de **CPU** (processor). Vooral bij matrix bewerkingen is het interessant om de tellers in de iteratieve statements (bv. *do-while* statement of *for*-statement) als *register int* op te bergen. Hieronder volgt een eenvoudig voorbeeld.

```
/* vb3_6.c : Register int data                                    */
#include <stdio.h>
main()
{ register int i;
  for (i=0; i<1000; i++) printf("aantal keer gedaan : %d\n",i);
}
```

voorbeeld 3.6 : Register data

TURBO-C bijvoorbeeld accepteert maximaal twee *register variabelen* zodat in een produkt funktie voor bijvoorbeeld een matrix met een vector de rij- en kolom-tellers als *register variabelen* kunnen worden gedeclareerd. Hierdoor zal de funktie sneller worden aangezien de tellers direct in de **CPU** beschikbaar zijn.

3.7 Externe data declaratie

Voor grote programma's welke uit meerdere files bestaan is het noodzakelijk om in iedere file aan te geven wat de globale data is en van welk type deze data is. Om dit aan alle onderdelen van het programma duidelijk te maken kan bovenaan in iedere file een lijstje worden opgenomen van de *externe globale data declaraties*. Deze declaraties zien er identiek uit als een gewone *globale data declaratie* voorafgegaan door het woordje *extern*.

Het zal duidelijk zijn dat voor een programma van 30 files dit een vervelende zaak wordt zeker als er veel *globale data* aanwezig is. Daarom is het eenvoudiger om een *header file* te maken met daarin de *externe data declaraties* en vervolgens deze *header file* met behulp van *#include* op te nemen in iedere file. Hieronder volgt een voorbeeld van een programma dat uit *twee* source bestanden bestaat met *globale data declaraties* in de file **vb3_7.c** waarin de *main()* funktie staat en een tweede file **vb3_7lib.c** met funkties welke ook gebruik maken van *globale data*. De *externe data declaraties* zijn vervolgens opgenomen in een *header bestand* **vb3_7.h** welke aangeroepen wordt in **vb3_7lib.c** en **vb3_7.c**.

```
file : vb3_7.c

#include <vb3_7.h>

double getal;

main()
{ int i;
  doe_iets(i);
}

file : vb3_7lib.c

#include <vb3_7.h>

void doe_iets(xyz) int xyz;
{ printf("counter = %d getal = %lf\n",xyz,getal);
}

file : vb3_7.h

#include <stdio.h>

extern double getal;
```

voorbeeld 3.7 : Static data

Door de *extern* declaratie weet de compiler bij het compileren dat *getal* een globaal gedeclareerde *double* is. Als dit niet bekent zou zijn geeft de compiler een foutmelding : *"unknown variable getal"*. Een aardigheidje is om tevens allerlei andere gemeenschapelijke zaken op te nemen in **vb3_7.h** zoals de *#include <stdio.h>*.

3.8 Data-toekenning

In eerdere voorbeelden is al gebruik gemaakt van het toekennen van data aan variabelen omdat er anders niets te doen valt met een programma. Er is in de voorbeelden tot nu toe gebruik gemaakt van verschillende manieren van data toekenning. De eerste is door middel van *initialisatie* en de tweede is met behulp van een *assignment statement*.

3.8.1 Initialisatie

Indien we aan data waarden willen toekennen direkt bij de declaratie dan spreken we van *data initialisatie*. Deze vorm bestaat uit :

type_specifikatie variabele_naam = constante;

Deze vorm hebben we kunnen zien in voorbeeld 3.4 in de hoofdfunktie *main()*.

3.8.2 Assignment statement

Indien een variabele niet is geinitialiseerd of een waarde heeft die veranderd moet worden wordt een zogenaamde *assignment opdracht* gebruikt van de vorm :

variabele_naam = expressie;

In dit geval wordt met *expressie* een waarde of een funktie-aanroep met een *return* waarde bedoeld. Een voorbeeld van deze vorm van datatoekenning is te vinden in voorbeeld 3.2.

3.8.3 Conversie van data type

In sommige gevallen kan het voorkomen dat een datatype niet correspondeert met een verwacht datatype. Met behulp van een *type cast* kan het type dan worden gewijzigd :

(type_specifikatie) expressie

Hieronder wordt een eenvoudig voorbeeldje gegeven.

```
/* vb3_8.c : Conversie van data type                              */

#include <stdio.h>

main()
{ int i;
  for (i=0; i<100; i++)
    printf("%d / 2 = %lf \n",i,(double) i/2);
}
```

voorbeeld 3.8 : Conversie van data type

De waarde *i/2* wordt nu geconverteerd naar een *double* zodat het *format* klopt.

4 OPERATOREN, EXPRESSIES EN STATEMENTS

Een *expressie* is een voorschrift beschreven met *operatoren* en *operanden* waarmee volgens vaste regels een waarde kan worden berekend. Als zodanig vormen expressies met daarin de operatoren het hart van iedere programmeertaal. *Statements* bestaan uit expressies. We hebben al kennis gemaakt met enkele *statements* zoals bijvoorbeeld het *assignment statement* (toekennings opdracht). Belangrijk in ***C*** is dat ieder statement afgesloten wordt met behulp van een **;**. Aangezien het merendeel van ***C*** code uit statements bestaat die het programma besturen zal er een apart hoofdstuk worden gewijd aan *programmabesturing*.

In dit hoofdstuk zal ingegaan worden op de verschillende operatoren in ***C***. Een operator vertelt de compiler wat voor een rekenkundige of logische operatie er verricht moet worden met de operanden. Er wordt onderscheid gemaakt tussen *unaire* en *binaire* operatoren. Een *unaire* operator heeft *één* operand terwijl een *binaire* operator *twee* operanden heeft.

Een standaard voorbeeld is bijvoorbeeld de *min* operator :

```
 y - z      (-) is binaire operator
-x          (-) is unaire operator
```

In ***C*** worden drie *soorten* operatoren onderscheiden :

- *rekenkundige* operatoren
- *relationele / logische* operatoren
- *bit wise* operatoren (waar we niet op in zullen gaan)

In de volgende paragrafen zullen de *rekenkundige* en *relationele/logische* operatoren aan de orde komen.

4.1 Rekenkundige operatoren

C kent dezelfde set rekenkundige operatoren als iedere andere taal. In de volgende tabel zijn deze weergegeven.

table 4.1 : Rekenkundige operatoren

Operator	Aktie
+	optellen
-	aftrekken
*	vermedigvuldigen
/	delen
%	rest van een integer deling (*modulo division*)
--	decrement
+	increment

De eerste vier operatoren behoeven geen nadere uitleg. De laatste drie zijn meer een typische ***C*** eigenschap. In het volgende voorbeeld wordt het een en ander verduidelijkt.

```
/* vb4_1.c : Rekenkundige operatoren                              */
#include <stdio.h>

void main()
{ int i;
  i = 30;
  printf("integer deling 30/4 = %d rest %d\n",i/4,i%4);
  i++;     printf("i is nu met een opgehoogd  : %d\n",i);
  i--;     printf("i is nu met een verminderd : %d\n",i);
  i += 10; printf("i met 10 verhoogd          : %d\n",i);
  i -= 10; printf("i met 10 verlaagd          : %d\n",i);
}
```

voorbeeld 4.1 : Rekenkundige operatoren

De prioriteit van de operatoren is als volgt vastgelegd :

hoogste ++ --
- (unair)
* / %
laagste + -

Operatoren met dezelfde prioriteit worden van links naar rechts verwerkt. Door middel van haakjes kan uiteraard de prioriteit worden veranderd.

4.2 Relationele en logische operatoren

De tweede groep van operatoren wordt gevormd door de relationele en logische operatoren. Relationeel duidt al op een relatie tussen waarden, zoals kleiner of groter of gelijk. De set van relationele operatoren bestaat uit de volgende lijst.

tabel 4.2 : Relationele operatoren

Operator	Aktie
	groter dan
=	groter of gelijk
<	kleiner dan
<=	kleiner of gelijk
==	gelijk
!=	ongelijk

De logische operatoren hebben betrekking op **AND**, **OR** en **NOT** volgens de formele logica. Op de volgende bladzijde worden ze in een tabel weergegeven.

tabel 4.3 : Logische operatoren

Operator	Aktie
&&	en (AND)
||	of (OR)
!	niet (NOT)

In ***C*** is *waar* altijd een waarde ongelijk aan 0 terwijl *niet waar* altijd 0 oplevert. In het onderstaande voorbeeld worden relationele en logische operatoren gebruikt.

```
/* vb4_2.c : Relationele en logische operatoren                */

12 > 6 && !(14 < 3) || 1 <= 8 levert waar op (resultaat is 1)
```

voorbeeld 4.2 : Relationele en logische operatoren

Tot nu toe is alleen kennis gemaakt met enkele taaldefinities voor data en rekenkundige en logische bewerkingen. De belangrijkste hulpmiddelen van een programmeertaal, de *programma besturende statements* zullen in het volgende hoofdstuk aan bod komen. Na dat hoofdstuk zijn we in staat om een feitelijk probleem in programma code te zetten. Afgezien van wat details zoals in- en uitvoer kan men dan met basis ***C*** uit de voeten.

5 PROGRAMMA BESTURING

De hoofdmoot van source code bestaat uit *statements*. In ***C*** kan een *statement* op drie manieren voorkomen :

- een enkel statement
- een *compount* of blok statement
- een leeg statement

Naast deze indeling kan er ook onderscheid gemaakt worden naar de *aard* van een *statement*. Zo onderscheiden vrijwel alle programmeertalen drie hoofdgroepen :

- *conditionele* statements
- *meerkeuze* statements
- *iteratieve* statements

In dit hoofdstuk zal ingegaan worden op deze drie hoofdgroepen van *statements*. Allereerst echter zal een korte formele beschrijving worden gegeven van het *statement* als zodanig.

5.1 Compount statements

Enkele statements zijn tot nu toe in de voorbeelden veelvuldig voorgekomen en zijn ook behandeld in hoofdstuk 3. Statements kunnen echter ook gebundeld worden tot een blok van statements die bij elkaar horen en uitgevoerd dienen te worden indien aan een bepaalde conditie wordt voldaan. In **PASCAL** kan dan met behulp van **begin** en **end** een zogenaamd *compount* statement worden gevormd. In ***C*** gaat dat op een soortgelijke wijze zij het dat **begin** en **end** zijn vervangen door de openings accolade **{** en de sluit accolade **}**. In feite bestaat een funktie uit een *compount* statement zoals uit de voorbeelden blijkt. In het vervolg van dit hoofdstuk zal vaak gebruik worden gemaakt van het *compount statement*.

Voor de duidelijkheid is het van belang de accoladen op een overzichtelijke manier te plaatsen. In de meeste boeken over ***C*** zijn er twee varianten te vinden die hieronder worden weergegeven. Zelf geef ik de voorkeur aan variant twee zodat die verder in deze cursus zal worden aangehouden.

```
variant 1

if ( x > y) {
  statements
}
else {
  statements
  if ( z > x) {
    statements
  }
}
accoladen niet onder elkaar
```

```
variant 2

if (x > y)
{ statements
}
else
{ statements
  if (z > x)
  { statements
  }
}
accoladen onder elkaar
```

voorbeeld 5.1 : Compount statement

5.2 Het conditionele statement

Conditionele statements hebben in het algemeen de volgende vorm :

als (conditie is waar) dan statement
anders statement

In deze omschrijving kan *statement* ook bestaan uit meerdere statements (*compount statement*). ***C*** kent feitelijk twee vormen om de bovenstaande conditie te testen en de bijbehorende statements uit te voeren. De eerste methode is het *if* statement, de tweede methode is het meerkeuze statement *switch* waarover meer in de volgende paragraaf.

Met behulp van het *if* statement ziet het bovenstaande probleem er als volgt uit.

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

De expressie waarop getest wordt is *waar* indien de waarde ongelijk aan nul is. Het *if ~ else statement* kan ook in een getrapte vorm voorkomen waarbij we meerdere condities willen testen :

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

Op deze wijze kunnen er ingewikkelde en meestal ook onoverzichtelijke structuren ontstaan. Het volgende voorbeeld laat een stukje code zien met een *if ~ else* statement.

```
/* zomaar ergens in een stukje code                        */
if ((pressure <= 0.45) && (temperature > 0.5))
  printf("warning : temperature is getting pretty high\n");
else
{ if ((pressure > 5.4) && (!leaking))
    printf("warning : pressure is building up\n");
  else
   if (pressure > 5.4) shutdown();  /* shutdown is een   */
}                                   /* of andere funktie */
```

voorbeeld 5.2 : Conditioneel statement ***if()***

Een meer criptisch gebruik van het *if ~ else* statement is het gebruik van de **?** operator. Deze operator wordt ook wel *ternary operator* genoemd aangezien het drie operanden nodig heeft. De algemene vorm van het op de vorige pagina weergegeven probleem wordt dan heel kort genoteerd door middel van :

expressie 1 ? expressie 2 : expressie 3

Als expressie 1 *waar* is dan wordt expressie 2 uitgevoerd anders expressie 3. De code wordt door deze notatie wel heel erg kort maar kan ook onnodig criptisch worden. Zelf gebruik ik deze notatie nooit maar als je het ergens tegenkomt dan weet je dat je met een *professional* te maken hebt.

5.3 Het meerkeuze statement

Met *if ~ else ~ if* kan als het ware een lange rij van condities worden opgesteld waardoor de structuur verloren gaat. Een duidelijker struktuur wordt geleverd door deze vorm van *meerkeuze statements* te vervangen door het *switch* statement. De syntax van het switch statement ziet er als volgt uit.

```
switch (variabele)
{ case waarde_1  : statement_1 break;
  case waarde_2  : statement_2 break;
  .............    ..............  .........
  case  waarde_n : statement_n break;
  default        : statement_d
}
```

Op deze manier kan gemakkelijk een boom van keuzen worden afgewerkt. Het verschil tussen het *if* commando en *switch* bestaat uit :

- *switch* kan alleen op een gelijkheid testen terwijl *if* logische en relationele expressies can evalueren

- de waarden in een case moeten verschillend zijn

Door het toepassen van een *break* wordt alleen het statement dat bij een bepaalde *case* behoort uitgevoerd. Wordt de *break* weggelaten dan zal het volgende statement ook worden uitgevoerd. Als er niet aan een van de *cases* wordt voldaan zal het *default* statement worden uitgevoerd. In het volgende voorbeeld wordt een stukje code met een *switch* gedemonstreerd.

```
/* in een van de programma header files staat                */

#define COORDINATES 1
#define ELEMENTS    2
#define ILLEGAL     3
#define COMMENT     4
#define EXECUTE     5
#define OLDSYNTAX   6
#define SUPPORTS    7

/* ergens in de code van een programma                       */

table_number = get_table_number(table_name); /* int funktie */
switch (table_number)                        /* conditie    */
{ case COORDINATES : read_coordinates(); break;
  case ELEMENTS    : read_elements();    break;
  case SUPPORTS    : read_supports();    break;
  case EXECUTE     : read_execute();     break;
  case COMMENT     :
  case ILLEGAL     :
  case OLDSYNTAX   : break;
  default          : end_of_reading();
}
```

voorbeeld 5.3 : Meerkeuze statement ***switch***

Aangezien de switch alleen met gelijkheden werkt is zij alleen geschikt voor data van het *type int* en *char*. Immers de gelijkheid tussen *doubles* is vanwege de numerieke onnauwkeurigheid een onmogelijke. Het *switch* statement mag ook getrapt voorkomen let dan echter goed op de *breaks* en de accoladen.

5.4 Het iteratieve statement met loops

C kent drie soorten iteratieve statements :

- *while* statement
- *do ~ while* statement
- *for* statement

Alle drie zullen ze een bepaalde handeling enige malen herhalen om een bepaalde iteratieve bewerking te verrichten. Het verschil zit in de *test conditie* : hoe vaak moet iets herhaald of gedaan worden. Als het een vooraf vast staand aantal keren betreft kan gebruik worden gemaakt van een *for statement*. Is dit niet het geval dan kan er gebruik worden gemaakt van het *do* of *while statement*. De verschillen zullen in de volgende drie paragrafen worden uiteengezet.

5.4.1 De for loop

Een for loop in ***C*** ziet er nogal criptisch uit. De formele syntax luidt :

for (initialisatie; conditie; increment) statement

De drie componenten zorgen voor een zeer flexibel gebruik van het *for* statement.

- *initialisatie* kent een begin-waarde toe aan de zogenaamde *loop-control* variabele ook wel *teller* genoemd

- *conditie* is een relationele expressie welke bepaald of de loop beeindigd moet worden

  increment bepaald hoe de *loop-control* variabele wijzigd gedurende de loop

Een eenvoudig voorbeeld van een *for* statement is te vinden in voorbeeld 3.6. De *loop-control* variabele is in dit geval *i*.

Het leuke van het for statement in ***C*** is dat de teller ook af kan lopen en dat de *conditie* van alles kan bevatten. Ook kunnen er meerdere waarden tegelijk worden *geinitialiseerd* die elk van een eigen *increment* kunnen worden voorzien. Het volgende voorbeeld laat een wild gebruik van het *for* statement zien.

```
/* ergens in de code van een programma                        */

for (i=0, j=100; i < aantal && j > 25; i++, j-=5)
{ doe_iets(i,j)          /* doe iets afhankelijk van i en j */
}
```

voorbeeld 5.4 : ***For*** statement

Het is niet noodzakelijk dat alle componenten van de *for* syntax aanwezig zijn. Zo kunnen bijvoorbeeld alle componenten weggelaten worden. Er zal dan echter een oneidige loop ontstaan. Ook het statement in de *for* mag weggelaten worden. Wat er dan ontstaat is een lege *for* loop die alleen iets aan het tellen is, feitelijk een kleine programma-vertraging.

5.4.2 De while loop

De *while* loop in ***C*** heeft de volgende syntax :

while (conditie) statement

Zolang de *conditie waar* is zal de het *loop-statement* worden uitgevoerd. Uiteraard kan statement hier ook uit meerdere statements bestaan. Van belang is dat de conditie voor het statement wordt geëvalueerd. Hieronder volgt een eenvoudig voorbeeld.

```
/* vb5_5.c                                                    */
#include <stdio.h>

main()
{ char ch = '\0';              /* initialisatie van char ch */
  while (ch!='Q') ch = getchar(); /* stop als Q ingetypt is */
}
```

voorbeeld 5.5 : ***While*** statement

De *getchar()* funktie die hier gebruikt is leest een karakter vanaf het toetsenbord. Deze funktie wordt verderop in het hoofdstuk INVOER en UITVOER behandeld.

Het hiervoor geschreven programma kan verkort worden. Aangezien er niet noodzakelijkerwijs een statement hoeft te volgen in een while constructie mag het volgende ook.

```
/* vb5_6.c                                                    */
#include <stdio.h>

main()
{ char ch='\0';
  while ((ch=getchar()) != 'Q');
}
```

voorbeeld 5.6 : Lege ***while***

5.4.3 De do ~ while loop

In tegenstelling tot de twee eerder genoemde *iterative statements* test de *do ~ while* loop een *conditie* aan het einde van de loop. De syntax ziet er als volgt uit :

do statement while (conditie);

Het gevolg is dat het statement ten minste één keer zal worden uitgevoerd waarna de test conditie word geëvalueerd. In het verdere gebruik is de *do~while* constructie gelijk aan de *while*.

```
/* vb5_7.c                                                    */
#include <stdio.h>

main()
{ char ch = '\0';              /* initialisatie van char ch  */
  do
  { printf("geef een karakter [Q=stop] : ");
    ch = getchar();
    switch (ch)
    { case 'a' : printf("keuze : a\n"); break;
      case 'b' : printf("keuze : b\n"); break;
      case 'c' : printf("keuze : c\n"); break;
      default  : break;
    }
  }
  while (ch!='Q');
}
```

voorbeeld 5.7 : Het ***do ~ while*** statement

Het voorbeeld spreekt verder voor zichzelf.

5.4.4 Het break statement

Het *break* statement zijn we al eerder tegengekomen als de beëindiging van een *case*. *Break* kan echter ook als een beëindiging van een *loop* worden gebruikt. In het volgende voorbeeld wordt dit verduidelijkt.

```
/* vb5_8.c                                                    */
#include <stdio.h>

main()
{ int i;
  for (i=0; i<100; i++)
  { printf("i heeft de waarde : %3d\n",i);
    if (i==45) break;                /* verlaat de loop      */
  }
  printf("buiten de loop\n");        /* dit wordt wel gedaan */
}
```

voorbeeld 5.8 : Het ***break*** statement

Indien meerdere *loops* in elkaar zijn verwerkt zal *break* alleen zorgen voor het stoppen van de *binnenste loop*.

5.4.5 De exit() funktie

Zoals *break* een *loop* beëindigd zo zal *exit()* het programma beëindigen. Aan *exit()* kan ook een getal worden meegegeven waarmee aan het *Operating System* (**OS**) kan worden verteld of het programma goed of niet goed is afgesloten. Een 0 zal in het algemeen aangeven dat het programma op een normale wijze is beëindigd.

```
/* vb5_9.c                                                    */
#include <stdio.h>

main()
{ int i;
  for (i=0; i<100; i++)
  { printf("i heeft de waarde : %3d\n",i);
    if (i==45) exit(9);              /* verlaat het programma */
  }
  printf("buiten de loop\n");        /* dit wordt niet gedaan */
}
```

voorbeeld 5.9 : De ***exit()*** funktie

Hier wordt het programma kennelijk opzettelijk op een abnormale wijze verlaten.

5.4.6 Het continue statement

Soms kan het handig zijn om in *loops* bepaalde statements over te slaan. In ***C*** is hierin voorzien door middel van het *continue* statement. Zodra het woord *continue* in een *loop* wordt geactiveerd zal de volgende stap in de iteratie worden uitgevoerd. In een voorbeeldje wordt dit verduidelijkt.

```
/* vb5_10.c                                                    */
#include <stdio.h>

main()
{ char ch='0';
  do
  { printf("geef karakter [Q=stop, q=continue] : ");
    ch = getchar(); fflush(stdin); /* leeg buffer keyboard  */
    if (ch == 'q') continue;       /* ga naar volgende stap */
    printf("hallo Cursist\n");     /* als ch != 'q'         */
  } while (ch != 'Q');
}
```

voorbeeld 5.10 : Het ***continue*** statement

Deze constructie kan omzeild worden door in plaats van een *continue* het *if* statement anders te definiëren. Het levert precies hetzelfde op waarmee *continue* eigenlijk overbodig wordt.

5.4.7 Het goto en label statement

C heeft ook de beschikking over *spagetti statements* zoals het *goto* statement. Het gebruik wordt ten zeerste afgeraden. Veel gehoorde kreten :

- de code wordt onleesbaar !
- ga dan maar over op **BASIC** !

In sommige situaties kan het echter makkelijk zijn om toch van *labels* en het *goto* commando gebruik te maken. In het algemeen echter kan door *netjes* te programmeren het gebruik altijd vermeden worden.

```
/* vb5_11.c                                                    */
#include <stdio.h>
main()
{ int i=0;
  marker1:
  i++;
  if (i<10)                  /* dit is een ordinaire for loop */
  { printf("i=%d\n",i); goto marker1;}
}
```

voorbeeld 5.11 : Het ***label & goto*** statement

6 FUNKTIES

In het voorafgaande zijn funkties al een paar keer ter sprake gekomen, ze vormen de bouwstenen van een goed gestructureerd programma. In veel talen wordt er onderscheid gemaakt tussen subroutines en funkties. Met subroutines wordt dan vaak een stuk code bedoeld en met een funktie wordt meer een algebraische funktie bedoeld. **PASCAL** bijvoorbeeld maakt onderscheid tussen *procedures* en *functions*. ***C*** maakt dit onderscheid niet, beide noemen we in ***C*** een funktie.

We hebben al gezien dat er aan funkties data kan worden meegegeven (*formele data*). Ook kan een funktie zelf een bepaalde waarde krijgen. Een voorbeeld hiervan is te vinden in voorbeeld 3.5 op pagina 13.

In dit hoofdstuk zal nader worden ingegaan op het *doorgeven* van data aan funkties en aan het *teruggeven* van data uit een funktie.

6.1 Funktie declaratie

Formeel ziet een funktie declaratie er als volgt uit :

```
type-specifikatie funktie-naam (formele data)
{ statements ;
}
```

De formele data kan hier op de eerder aangegeven wijze worden weergegeven (*klassiek* of *modern*). Afhankelijk van de *type-specifikatie* zal de funktie een waarde teruggeven. Standaard zal ***C*** aannemen dat een funktie van het *type int* is. Indien een funktie geen waarde krijgt is het netjes om de compiler dat te laten weten door het *type void* toe te kennen aan de funktie. Hoe een funktie een waarde krijgt toegekend wordt in de volgende paragraaf verteld.

6.2 De return

Ten einde een waarde toe te kennen aan een funktie wordt gebruik gemaakt van het *return* statement :

```
return (variabele) ;
```

Hierdoor zal de funktie afgesloten worden en wordt aan de funktie de waarde van *variabele* toegekend. In het eerder aangehaalde voorbeeld op pagina 13 wordt met *return* de waarde van *gedaan + 1* aan de funktie *doe_iets* toegekend.

Zoals gezegd *return* zal de funktie direkt beëindigen. Van deze eigenschap kunnen we dus handig gebruik maken als we midden in een funktie de funktie willen verlaten. In het volgende voorbeeldje maken we daar gebruik van.

```
/* vb6_1.c : De return                                          */

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

double my_sqrt(x) double x;
{ if (x < 0.0) return (-1.0);
  else return (sqrt(x));
}

main()
{ double getal=4.0;
  printf("Wortel van getal %lf = %lf\n",getal,my_sqrt(getal));
}
```

voorbeeld 6.1 : De ***return***

Met deze funktie wordt voorkomen dat er een wortel getrokken wordt uit een negatief getal. De funktie *sqrt()* is een standaard ***C*** funktie uit de *mathematische library*. De *prototype - definitie* van deze funktie is te vinden in de **math.h** *header-file*. Verderop in dit hoofdstuk zullen we de *prototype* definities behandelen.Vergeet niet de *return* in de funktie want ***C*** geeft default een *0* terug.

6.3 Het doorgeven van parameters

Met behulp van de formele parameter-lijst kunnen we variabelen doorgeven aan funkties. Hierbij kunnen we twee belangrijke verschillen onderscheiden :

- de door te geven data verandert niet van waarde (*call by value*)
- de waarde van de doorgegeven variabele wordt in de funktie gewijzigd en teruggegeven (*call by reference*)

In **PASCAL** bijvoorbeeld zou in het tweede geval de aanduiding **VAR** moeten voorafgaan aan de data in de formele data-lijst van de funktie. In ***C*** gaat dit op een soortgelijke wijze zoals in de volgende paragrafen wordt uitgelegd.

6.3.1 Call by value

Normaal gesproken zal in een funktieaanroep gewerkt worden met een *call by value*. Immers een algebraische funktie is meestal afhankelijk van invoer-variabelen. Deze zullen niet door de funktie veranderd worden. De uitkomst van een dergelijke funktie wordt dan toegekend aan de funktie zelf zoals we dat in voorbeeld 6.1 hebben gezien. De waarde van *x* in dit voorbeeld veranderde niet in de funktie *my_sqrt()*. Leuk is dat we in ***C*** de variabele *x* in de funktie gewoon als een *lokale variabele* mogen gebruiken en wijzigen. Dit heeft geen gevolgen voor de doorgegeven data welke ongewijzigd blijft. In het volgende voorbeeldje wordt dit laatste verduidelijkt.

```
/* vb6_2.c : Call by Value                                        */

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

double sleutel(login_nr) double login_nr;
{ login_nr *= 3.141512;       /* lokaal gebruik van login_nr */
  return(pow(2.14,login_nr)); /* bepaal sleutel              */
}

main()
{ double my_number = 12.34;   /* dit nummer geeft de sleutel */
  printf("sleutel = %lf\n",sleutel(my_number));
  printf("nummer  = %lf\n",my_number);
}                             /* my_number blijft gelijk     */
```

voorbeeld 6.2 : Call by Value

Ook al wordt in de funktie *sleutel()* de waarde van *login_nr* veranderd, de waarde van *my_number* zal niet veranderen aangezien *sleutel()* aangeroepen wordt d.m.v. een *call by value*.

6.3.2 Call by reference

In de voorafgaande voorbeelden konden we de uitkomst van een funktie onderbrengen in de *return* waarde van de funktie. Wat te doen als er bijvoorbeeld drie of vier variabelen wijzigen in een funktie. Zo'n funktie heeft dan meer het karakter van een *subroutine*. We zullen dan toch een manier moeten hebben om deze variabelen terug te geven aan het programma-deel dat deze funktie aanroept. Met een *call by reference* is dit te realiseren.

In feite wordt bij een *call by reference* niet een getalswaarde doorgegeven maar een *adres* verwijzing van de variabele. We werken hier dus feitelijk met *pointers*. Het eigenlijke *pointer* begrip wil ik even uitstellen tot een later hoofdstuk dus ik zal me beperken tot een eenvoudige uitleg speciaal voor de *call by reference*. Als we een *adres* doorgeven aan een funktie kan de funktie de waarde van de variabele bereiken en zo deze ook veranderen. Voor het programma-onderdeel dat de funktie aanroept heeft dit geen gevolgen want ook hier is het *adres* van de variabele bekend. Feitelijk wordt de waardeverandering van de te wijzigen variabele op een min of meer *globale* wijze afgehandeld (op een aangegeven geheugenlokatie). Een funktie met een *call by reference* ziet er als volgt uit (in moderne stijl):

*type-specifikatie funktie-naam (type-specifikatie *variabele)*

Het * verwijst hier (*pointer*) naar de *waarde* van variabele. Een dergelijke funktie zal aangeroepen worden door middel van een adres verwijzing (*&*) van een variabele :

funktie-naam(&variabele)

Dus :

&	geeft het *geheugen-adres* van een variabele
*	is het *complement* van *&* en geeft de *waarde* welke op het adres is te vinden

De *&* en * zijn zogenaamde *pointer operatoren*.

In een voorbeeld zal een *call by reference* worden gedemonstreerd.

```
/* vb6_3.c : Call by Reference                                    */

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

int doe_iets(x,y) double *x, *y;
{ *x = *x + 10.0;
  *y = *y + 11.0;
  if ((*x > 50.0) && (*y > 40.0)) return (1);
  else return (0);
}

main()
{ int i; double var1=0.0, var2=5.0;/* start waarden 0, 5    */
  for (i=0; i<10; i++)             /* loop, 10 x           */
  { if (doe_iets(&var1, &var2))    /* var1 en var2 wijzigen */
      printf("i=%3d var1=%10.1f var2=%10.1f\n",i+1,var1,var2);
  }
}
```

voorbeeld 6.3 : Call by Reference

Vergeet de *&* niet bij het aanroepen van funktie *doe_iets()*. Bij het gebruik van *pointers* moet goed gelet worden op de juiste *syntax* want vaak geeft de compiler geen foutmelding en zal het programma wel iets doen maar in het geheel niet datgene wat de ontwerper verwacht.

6.4 Prototype van een funktie

Het is al een paar keer ter sprake gekomen, hier zal dan verteld worden wat een *prototype* is van een funktie en waar en hoe deze te gebruiken.

Iedere funktie in **C** welke niet van het default *type int* is moet voor het aanroepen aan de compiler bekend gemaakt zijn. Dit kan door de *funktie definitie* zelf maar kan ook door een funktie *prototype definitie*. Deze protoype definitie ziet er net zo uit als de *funktie definitie*. We kunnen hier ook weer onderscheid maken tussen de *moderne* en de *klassieke* stijl :

klassiek *type-specifikatie funktie-naam();*
modern *type-specifikatie funktie-naam(formele data);*

Bij het gebruik van een funktie *voor* de feitelijke *funktie-definitie* zal de compiler aangeven dat deze funktie (van het aangegeven type) niet bestaat. Door nu een *prototype definitie* bovenaan de file op te nemen is de compiler op de hoogte van het bestaan van de bewuste funktie van het aangegeven type. Tevens kan als er gebruik wordt gemaakt van de *moderne* stijl gecontroleerd worden (door de compiler) of het aantal *argumenten* in de funktie-aanroep overeenstemt met de *definitie* en of deze van het juiste type zijn. Deze controle kan *niet* plaats vinden bij de *klassieke* notatie. In het volgende voorbeeld wordt een funktie aangeroepen die nog niet gedefinieerd is. Door gebruik te maken van een *prototype-definitie* kan de *volgorde* van de funkties dus min of meer *vrij* gelaten worden.

```
/* vb6_4.c : Gebruik van een funktie prototype definitie     */

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

double my_sqrt();             /* funktie prototype definitie */

main()
{ double getal=4.0;           /* gebruik my_sqrt() voor def  */
  printf("Wortel van getal %lf = %lf\n",getal,my_sqrt(getal));
}

double my_sqrt(x) double x;   /* funktie definitie           */
{ if (x < 0.0) return (-1.0);
  else return (sqrt(x));
}
```

voorbeeld 6.4 : Prototype Definitie

Bij grote programma's zal het niet altijd mogelijk zijn om funkties voor hun gebruik te definiëren. Funktie *prototype definities* zijn dan onontbeerlijk. Handig is dan om de *funktie prototype definities extern* te maken en op te nemen in een speciale *protoype-header* file.

6.5 De bijzondere funktie main()

De funktie *main()* neemt een bijzondere plaats in in een ***C***-programma. Het is een funktie met een voor de compiler bekende naam. De compiler zal *main()* behandelen als het hoofdprogramma ook wel de hoofdfunktie. Net als iedere funktie kan *main()* beschikken over *formele data*. Deze is echter niet vrij te kiezen. De funktie *main()* ligt per definitie vast :

*type-specifikatie main(argc, argv) int argc; char *argv[];*

Hierin is :

argc	het aantal *strings* op de *commandline*
**argv[]*	een array met daarin de *strings* van de *commandline*

Als we bijvoorbeeld een programma schrijven met de naam *my_program* dan zal onder **DOS** het programma worden gestart met bijvoorbeeld het volgende commando :

my_program invoer.dat uitvoer.dat

Op de *commandline* staan nu drie *strings*. In *my_program* kunnen we deze *strings* opvragen en verder gebruiken in het programma. De variabele *argc* is dus 3. De array met strings **argv[]* is als volgt gevuld :

argv[0] my_program
argv[1] invoer.dat
argv[2] uitvoer.dat

Array's zullen in een volgend hoofdstuk aan de orde komen.

Belangrijk hier is dat er *data* aan het hoofdprogramma kan worden meegegeven in de vorm van *command string parameters*. In het volgende voorbeeld wordt het gebruik van *command string parameters* verduidelijkt.

```
/* vb6_5.c : Gebruik van command line strings in main        */

#include <stdio.h>

main(argc, argv) int argc; char *argv[];
{ if (argc !=3) printf("fout commando\n");
  else
  { printf("invoerfile  = %s\n",argv[1]);
    printf("uitvoerfile = %s\n",argv[2]);
  }
}
```

voorbeeld 6.5 : De funktie ***main()***

Dit programma zal indien het wordt gestart dmv :

vb6_5 test.dat test.out

met de boodschap :

invoerfile = test.dat
uitvoerfile = test.out

reageren. Als er niet drie *strings* opgegeven worden zal het programma een foutmelding geven.

6.6 Mathematische funkties

Het is niet de bedoeling om in detail alle funkties van ***C*** hier te presenteren. Voor meer informatie verwijs ik graag naar betere boekwerken. Ik beperk me hier tot de meest gebruikelijke funkties en zal daar alleen de funktie definitie van ***C*** van gegeven. Inmiddels moet het duidelijk zijn hoe een dergelijke definitie in elkaar zit en hoe de funktie te gebruiken is. De funktie definitie van de mathematische funkties is te vinden in de header file **math.h**. Deze file dient dus opgenomen te worden in de code.

arccosinus	*double acos(double x);*
arcsinus	*double asin(double x);*
arctan	*double atan(double x);*
arctan(y/x)	*double atan2(double y, double x);*
kleinste integer	*double ceil(double num);*
cosinus	*double cos(double x);*
cosinus hyperbolicus	*double cosh(double x);*
e^x	*double exp(double x);*
absolute waarde	*double fabs(double x);*
absolute waarde int	*int abs(int x);*
grootste integer	*double floor(double x);*

rest van x/y	*double fmod(double x, double y);*
natuurlijke logaritme	*double log(double x);*
10 log	*double log10(double x);*
x^y	*double pow(double x, double y);*
sinus	*double sin(double x);*
sinus hyperbolicus	*double sinh(double x);*
wortel	*double sqrt(double x);*
tangens	*double tan(double x);*
tangens hyperbolicus	*double tanh(double x);*

Bij sommige compilers moet er op gelet worden dat de *mathematische libraries* mee worden *gelinkt*. Hoofdstuk 12 zal verder ingaan op deze materie.

7 ARRAYS ALS STATISCHE DATA

In een vorig hoofdstuk is het begrip *array* al eens gevallen. Een *array* is feitelijk een rij van data van *hetzelfde* type. De individuele data in de rij is met behulp van een *index* te bereiken. De meeste talen kennen *arrays* bijvoorbeeld om een vector met reële getallen op te bergen. Zo'n *array* is in ***C*** een *array van doubles*. In **BASIC** wordt voor het declareren van een *array* gebruik gemaakt van bijvoorbeeld DIM A(10) om een rij van 10 getallen te declareren. In ***C*** gaat dit op een soortgelijke manier. We spreken dan over *statische data* waarbij de *array-lengte* in de code wordt vastgelegd. In hoofdstuk 8 zal teruggekomen worden op het begrip *array* in de vorm van *dynamische data* waarbij de *lengte* van de *array* vrij is en er door de code geheugen kan worden gereserveerd voor een *array* van de vereiste lengte. In dit hoofdstuk zal alleen het *array* als *statische data* worden behandeld. Daarbij is onderscheid te maken uit *één-dimensionale arrays* (bijvoorbeeld *vectoren*) en *meerdimensionale arrays* (bijvoorbeeld *matrices*).

7.1 Arrays met dimensie 1

De algemene syntax voor de declaratie van een *array* van een bepaald *type* luidt :

type_specifikatie naam[lengte]

Een *element* van deze *array* kan dan worden aangeduidt met :

naam[i]

De *index i* loopt in ***C*** van *0* tot *lengte*. In het voorbeeld wordt dit gedemonstreerd.

```
/* vb7_1.c : Een array                                          */

#include <stdio.h>

main()
{ int i; double vector[10];
  for (i=0; i<10; i++)
  { vector[i] = 1.0;
  }
}
```

voorbeeld 7.1 : Array

Ieder array is op een bepaald *geheugen-adres* opgeborgen. De naam van de *array* is de aanduiding voor dit *adres* dus :

naam is het *begin-adres* van de *array*

In de al eerder gehanteerde *pointer* notatie kunnen we dit *adres* ook beschrijven als :

adres is gelijk aan *naam* = *&naam[0]*

Alle overige elementen van het *array* staan achter elkaar op rij. Bij het doorgeven van *arrays* aan *funkties* is het dus ook niet nodig om de *gehele array* door te geven, er kan worden volstaan met het *begin-adres* van de *array*. Zo ontstaat dus de *pointer architectuur* van een ***C***-programma. In het volgende voorbeeld wordt hiervan een voorbeeld gegeven.

```
/* vb7_2.c : Arrays en funkties                                    */

#include <stdio.h>

void print_vector(v,n) double *v; int n;
{ int i;
  for (i=0; i<n; i++) printf("v[%3d] = %8.3lf\n",i+1,v[i]);
}

main()
{ int i; double vector[10];
  for (i=0; i<10; i++) vector[i] = 1.0;
  print_vector(vector,10); /* vector is hier het begin-adres */
}
```

voorbeeld 7.2 : Array en funkties

Het resultaat van voorbeeld 7.2 is een rijtje getallen met waarde 1.0. Het voorbeeld toont het doorgeven van het *beginadres* van de *array vector[]* aan de print funktie *print_vector()*. Belangrijk is in ***C*** dat de gebruiker zelf de *grenzen* van een *array* in de gaten houdt. De compiler geeft geen foutmelding als we een *index* opvragen groter dan *lengte*. Wat er dan echter gebeurt is niet te beschrijven (avonden zoeken naar .....) ! Dus let op, de index *moet* liggen in het interval :

[0, ... , lengte >

Tot nu toe is het begrip *array* gebruikt voor het opbergen van getallen. In ***C*** is echter een *string* ook een *array*, namelijk een *array van karakters* (*char*). Hoewel ***C*** het *type string* niet kent gebruik ik deze aanduiding als een afkorting van een *array van characters*. Een string in ***C*** wordt dus als een *array van het type char* gedeclareerd. Zo'n *string* heeft een bijzonderheidje en dat is de *string-beëindiging* d.m.v. een *null character* '\0' als *string terminator*. Deze *terminator* hoeft de gebruiker zelden te plaatsen, echter is wel aanwezig. Om een aantal bewerkingen op *strings* te vereenvoudigen kent ***C*** een aantal standaard *string-funkties* waarvan de *prototype definities* te vinden zijn in o.a. de **string.h** *header file*. Een kleine opsomming van de *belangrijkste funkties* volgt hieronder :

- *strcat(char *str1, char *str2)* plakt *str2* achter *str1*
- *strcmp(char *str1, char *str2)* vergelijkt *str1* met *str2*
- *strcpy(char *str1, char *str2)* maakt een kopie van *str2* in *str1*
- *int strlen(char *str)* bepaald de lengte van *str*
- *strupr(char *str)* zet *str* om naar hoofdletters
- *strlwr(char *str)* zet *str* om naar kleine letters

In het volgende voorbeeld worden *strings* gebruikt.

```
/* vb7_3.c : String als array van char                          */

#include <stdio.h>

void print_string(s) char *s;
{ printf("%s\n",s);
}

main()
{ int i,j,k;
  char ch,
       string[21]="hallo Cursist",  /* geen terminator nodig */
       tmp[21]  ="niets";           /* character constante   */
  for (i=0; i< strlen(string); i++)
    printf("letter %3d string : %c   tmp : %c\n",
           i+1,string[i],tmp[i]);
  strcpy(tmp,string);               /* tmp = "hallo Cursist" */
  if (strcmp(string,tmp))
    printf("string en tmp zijn nu hetzelfde\n");
  strcat(string,tmp);
  printf("string = %s\n",string);   /* 2x hallo Cursist      */
  strupr(string);                   /* alles in KAPITALEN    */
  printf("string = %s\n",string);
  strlwr(string);
  printf("string = %s\n",string);
  print_string(string);             /* funktie aanroep       */
}
```

voorbeeld 7.3 : String als array van ***char***

Tot nu toe zijn alleen *één-dimensionale* arrays behandeld. In ***C*** kunnen ook echter *meerdimensionale* arrays gedeclareerd worden. Dit zal worden uitgelegd in de volgende paragraaf.

7.2 Meerdimensionale arrays

Nu bekend is hoe een *ééndimensionale array* eruit ziet zal het niet al te moeilijk voor te stellen zijn hoe een *meerdimensionale array* eruit ziet. De syntax voor de declaratie ziet er als volgt uit :

type_specifikatie naam[lengte1][lengte2]...[lengteN]

In een volgend voorbeeld wordt een *vierdimensionaal array* gedeclareerd.

```
/* vb7_4.c : Een meerdimensionaal array                         */
#include <stdio.h>

main()
{ int i,j,k,l,big[12][5][7][13];
  for (i=0;i<12;i++) for (j=0; j<5;j++)
  for(k=0; k<7; k++) for (l=0; l<13; l++) big[i][j][k][l]=1;
}
```

voorbeeld 7.4 : Meerdimensionale Arrays

Het zal duidelijk zijn dat *meerdimensionale arrays* een flinke hoeveelheid geheugen in beslag kunnen nemen. Immers een array van *120 x130 x15* getallen van het *type double* (*64 bits = 8 bytes [tabel 3.2 pag 7]*) heeft ongeveer 1.8 Mb aan geheugenruimte nodig. Daarom zal het gebruik van *meerdimensionale arrays* meestal beperkt blijven tot *twee-* of *driedimensionale arrays* van beperkte omvang. Het doorgeven van *meerdimensionale arrays* aan *funkties* gaat op een soortgelijke manier als voor *ééndimensionale arrays*. De *eerste* afmeting van de *array* mag weggelaten worden. Het volgende voorbeeld laat dit zien.

```
/* vb7_5.c : Meerdimensionale arrays en funkties             */

#include <stdio.h>

void print_array(mat,n) int mat[][5][7][13]; /* n is vrij    */
{ int i,j,k,l;
  for (i=0;i<n;i++)
    for (j=0; j<5;j++)
      for(k=0; k<7; k++)
        for (l=0; l<13; l++)
          printf("big[%d][%d][%d][%d]=%d\n",
                  i+1,j+1,k+1,l+1,mat[i][j][k][l]);
}

main()
{ int i,j,k,l,big[12][5][7][13], count=1;
  for (i=0;i<12;i++)
    for (j=0; j<5;j++)
      for(k=0; k<7; k++)
        for (l=0; l<13; l++)
        { big[i][j][k][l]=1;
          count++;
        }
  print_array(big,12);
  printf("%d getallen\n",count-1);
}
```

voorbeeld 7.5 : Meerdimensionale Arrays en Funkties

Op zich is deze manier van werken onhandig. Alleen de eerste index is op deze manier algemeen door te geven, de andere liggen vast. De funkties voor dergelijke arrays zijn dus *niet flexibel* qua lengte. Daarom is het in het algemeen handiger om arrays *dynamisch te alloceren*. De lengte kan dan vrij gekozen worden. In het volgende hoofdstuk over *pointers* zal hierop worden ingegaan.

8 POINTERS EN DYNAMISCHE DATA

Pointers zijn in de voorgaande hoofdstukken al terloops ter sprake gekomen. In dit hoofdstuk zal iets dieper ingegaan worden op het gebruik van *pointers*. Het gebruik van *pointers* in ***C*** is een van de sterkste punten van deze taal. Door direkt naar het *adres* van data te verwijzen worden progamma's veel sneller. Het heeft echter ook een keerzijde (*behoud van ellende*), veel *fatale* fouten worden veroorzaakt door het gebruik en misbruik van pointers. In de inleiding werd hier al op gewezen. Het vervelende van deze *bugs* is dat opsporen moeizaam is en dat de gevolgen van *pointer-crashes* vaak een *hangend system* tot gevolg hebben. Geen prettig vooruitzicht ?? Door een vaste routine (die goed werkt) aan te houden kan het werken met pointers probleemloos verlopen. Daar zal dus in dit hoofdstuk het accent op liggen.

8.1 Het begrip pointer

Een *pointer* (wijzer) is niets meer dan een variabele waarin een geheugenadres is opgeslagen. Op dit adres "woont" een bepaalde variabele. Belangrijk is dus om het *adres* en de *waarde* van een variabele te onderscheiden. Als we dus een *pointer* naar het adres van een variabele laten verwijzen dan kunnen we ook de *waarde* op dit adres opvragen. Dit klinkt nog redelijk eenvoudig maar de verwijzingen kunnen over vele tussenstations lopen waardoor zeer ingewikkelde konstrukties kunnen ontstaan. (*pointer naar pointers*)
De declaratie van een *pointer* variabele gaat op dezelfde wijze als dat voor een andere variabele met één verschil en dat is het *pointer* (*) teken dat aan de naam moet voorafgaan :

*type-specifikatie *naam;*

Hiermee wordt bekend gemaakt naar welk *type* variabele de *pointervariabele naam* wijst. Deze vorm van declaratie zijn we al tegen gekomen in de *formele data definitie* van funkties met een *call by reference* (paragraaf 6.3.2 op pagina 29).

8.2 Operaties op pointers

De *pointer operatoren &* en * zijn we al eerder tegengekomen in paragraaf 6.3.2. Het *adres* wordt aangeduid met de *&* operator terwijl de * operator de waarde van de variabele op het *adres* oplevert. In het volgende voorbeeld wordt dit gedemonstreerd.

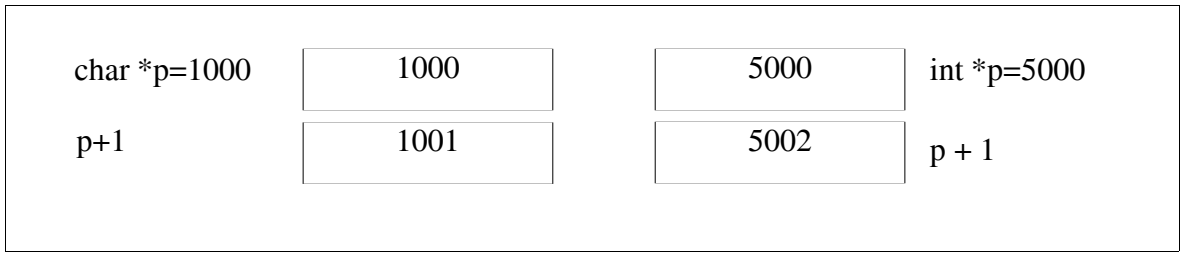
```
/* vb8_1.c : Pointer operatoren                                    */

#include <stdio.h>

main()
{ int var=4, *int_pointer;
  int_pointer = &var;  /* adres van var staat in int_pointer */
  printf("var heeft de waarde  : %d\n",*int_pointer);
  printf("het adres van var is : %p\n",int_pointer);
  /* zie tabel 3.4 voor de %p format code voor pointers      */
}
```

voorbeeld 8.1 : Pointer operatoren

Aangezien pointers wijzers zijn naar bepaalde adressen zijn er niet zo veel bewerkingen mogelijk op pointers. Rekenkundige bewerkingen als optellen en aftrekken zijn de enige bewerkingen op pointers welke zijn toegestaan. Bij het *verhogen* van een *pointer* met de waarde 1 zal het *adres* niet met 1 toenemen maar met *1 x aantal bytes* van het bewuste *type* worden opgehoogd. Dus :

int_pointer++ wijst naar het *adres* van de volgende *int*

Als we dus een *char* pointer en een *int* pointer met de basis adressen 1000 en 5000 nemen dan zullen de opvolgende pointer-adressen er verschillend uitzien :



voorbeeld 8.2 : Pointer bewerkingen

Al eerder is laten zien dat pointers veelal gebruikt kunnen worden in combinatie met arrays. Als we bijvoorbeeld een string hebben van lengte 10 dan kunnen we het adres op verschillende manieren noteren.

char p[10];
p
&p[0]

We kunnen dus ook een *pointer vergelijking* toepassen :

p == &p[0] is waar

Ook in het geval van *array-indexering* is een andere (*pointer*) notatie te hanteren. Het volgende voorbeeld laat dit zien. Tevens is hier te zien dat alle data in een array achter elkaar

```
/* vb8_3.c : Pointers en arrays                                  */
#include <stdio.h>
main()
{ int *int_pointer, int_array[10], int_matrix[4][4];
  int_pointer = int_array;   /* adres wijst naar int_array   */
  int_pointer[5]   = 100;    /* dmv index notatie            */
  *(int_pointer+5) = 100;    /* dmv pointer operator notatie */
  printf("%d %d \n",*(int_pointer+5),int_array[5]);
  int_matrix[0][3] = 10.0;   /* dmv index notatie            */
  int_pointer      = &int_array[0][0];
  *(int_pointer+3) = 10.0;   /* dmv pointer operator notatie */
  /* int_matrix[i][j] = *(int_matrix+(i*rij_lengte)+j)       */
  printf("%d %d \n",*(int_pointer+3),int_matrix[0][3]);
}
```

voorbeeld 8.3 : Pointers en arrays

is opgeborgen. Dit geldt overigens alleen voor *statische arrays*. *Dynamisch* gealloceerde arrays zijn meestal anders opgebouwd waardoor de data verspreid kan zijn opgeborgen. In de volgende paragraaf zal hier verder op in worden gegaan.

8.3 Dynamisch geheugen

Tot nu toe is alleen *statische data* behandeld. De *globale* en *lokale* data welke behandeld is zijn voor zo ver het *globale of statisch lokale* data betreft vast qua geheugen *positie* en *omvang*. Met behulp van *dynamische geheugen allocatie* kan geheugen *op maat* worden gereserveerd voor data. Met name bij het gebruik van *arrays* is het zinvol om slechts zoveel geheugen te reserveren als dat er nodig is. Daarmee wordt overbodige data voorkomen. Afmetingen van *matrices* en *vectoren* worden immers vaak door de probleemgrootte bepaald.

C biedt standaard funkties waarmee geheugen kan worden gereserveerd en worden vrijgegeven de belangrijkste twee zijn *malloc()* en *free()*. De funktie *malloc()* alloceert geheugen en geeft een pointer terug naar het beginadres van het gereserveerde geheugen. Met de funktie *free()* kan vervolgens dit geheugen vrij worden gegeven. Naast deze twee funkties zijn er nog andere funkties waarmee het *dynamische* geheugen kan worden gemanipuleerd. De **ANSI** standaard geeft vier funkties voor *dynamische* geheugen allocatie :

calloc() alloceert geheugen
malloc() alloceert geheugen
free() geeft geheugen vrij
realloc() realloceert geheugen (rekt bestaand geheugen op)

De hierbij horende *header file* is **stdlib.h** of **alloc.h** welke dus meegenomen dient te worden in de bronbestanden. Het voordeel van deze standaard is dat op alle platforms er op dezelfde manier geheugen kan worden aangevraagd. Dit geldt dus ook voor WINDOWS waarbij al het beschikbare RAM en deels de *swapfile* van WINDOWS beschikbaar is. In de volgende paragrafen zal uiteengezet worden hoe geheugen gealloceerd, gerealloceerd en vrijgegeven kan worden.

8.3.1 Geheugen alloceren, malloc() en calloc()

Met *calloc()* en *malloc()* wordt zoals eerder aangekondigd geheugen gealloceerd. De beide funkties zijn vrijwel indentiek en geven een *pointer* naar het beginadres van het geheugen. Het verschil tussen de funkties zit in de *formele data* zoals uit de syntax blijkt :

```
void *malloc(unsigned size)
void *calloc(unsigned num, unsigned size)
```

Hierin is bij *malloc() size* het aantal *bytes* dat gealloceerd moet worden terwijl bij *calloc() num * size* het aantal te alloceren *bytes* voorstelt. De *pointer* welke terug komt zal altijd door middel van een *type cast* geconverteerd moeten worden naar het gewenste *type*. Indien de allocatie faalt zal er een *NULL pointer* terug gegeven worden. Door hierop te testen kan dus de geheugen allocatie worden beveiligd tegen het gebruik van ongeldige *pointers*. In het volgende voorbeeld wordt het gebruik van de allocatie funkties gedemonstreerd. Let in dit voorbeeld vooral op de *pointer initialisatie* voor gebruik. Een *pointer* zal namelijk voor gebruik een *geldige waarde* of de *waarde NULL* moeten hebben. Zo niet dan weten we niet *waar* de *pointer* naar *wijst* en kan het *programma* crashen. Wellicht wijst de pointer naar een geheugenadres van het **OS** zodat ook het **OS** crashed.

```
/* vb8_4.c : Allocatie met malloc() en calloc()              */
#include <stdio.h>
#include <stdlib.h>

main()
{ int *vi,i,j,n,m; double *vd; char s[100];
  vi = NULL;            /* pointer geinitialiseert naar NULL */
  vd = NULL;
  printf("geef n : "); gets(s); n = atoi(s);
  printf("geef m : "); gets(s); m = atoi(s);
  vi = (int *) calloc((unsigned)n, sizeof(int));
  if (vi)   printf("integer array[0..%d] is gealloceerd\n",n);
  else    { printf("calloc() error\n"); exit(1); }
  vd = (double *) malloc((unsigned)(m * sizeof(double)));
  if (vd)   printf("double array[0..%d] is gealloceerd\n",m);
  else    { printf("malloc() error\n"); exit(1); }
  for (i=0; i<n; i++) vi[i] = i;
  for (i=0; i<m; i++) vd[i] = (double) i;
  for (i=0; i<n; i++) printf("vi[%d] = %d\n",i+1,vi[i]);
  for (i=0; i<m; i++) printf("vd[%d] = %f\n",i+1,vd[i]);
}
```

voorbeeld 8.4 : ***calloc()*** en ***malloc()***

Het programma zal in dit voorbeeld twee arrays van de op te geven lengte alloceren. Indien dit niet lukt volgt een foutmelding en wordt het programma beëindigd. *Sizeof()* levert het aantal bytes dat voor een bepaald type is gereserveerd en zorgt er zodoende voor dat de code op verschillende machines probleemloos draait. Een *integer* bijvoorbeeld hoeft niet op iedere machine te bestaan uit 2 *bytes*.

8.3.2 Geheugen realloceren met realloc()

In het voorgaande voorbeeld gaven we de lengte van een *array* op waarmee de geheugen allocatie werd uitgevoerd. Stel dat we van te voren opgeven dat onze *arrays* een lengte 10 hebben. We alloceren deze vervolgens op lengte 10 en vervolgen dan de rest van het programma. Indien de opgegeven *n* en *m* groter zijn dan de reeds bestaande dan rekken we de *arrays* een beetje op. We *realloceren* dan geheugen. De syntax van *realloc()* is als volgt :

*void *realloc(void *ptr, unsigned newsize)*

Het aardige van *realloc()* is dat de bestaande data netjes wordt overgenomen. Het volgende voorbeeld laat dit zien. Let op : de *pointer* van de te *realloceren* data moet bestaan.

```
/* vb8_5.c : Reallocatie realloc()                                  */
#include <stdio.h>
#include <stdlib.h>

main()
{ char *p;
  p = NULL;
  p = (char *) malloc(11 * sizeof(char));
  if (p) printf("string p is gealloceerd\n");
  else { printf("malloc error\n"); exit(1); }
  strcpy(p,"1234567890"); /* p bestaat uit 10 karakters      */
  p = (char *) realloc(p,(unsigned) (20 * sizeof(char)));
  if (p) printf("reallocatie op p uitgevoerd\n");
  else { printf("realloc error\n"); exit(1); }
  strcat(p," langer");
  printf("p=%s\n",p);
}
```

voorbeeld 8.5 : ***realloc()***

Overigens mag een *realloc()* ook worden toegepast om overbodige ruimte vrij te geven. Het resultaat zal zijn dat de string *p* uiteindelijk bestaat uit :

1234567890 langer

Dit is in essentie hetgeen *realloc()* doet.

8.3.3 Geheugen vrijgeven met free()

Het grote voordeel van *dynamisch* geheugenbeheer is, dat geheugen dat niet meer gebruikt wordt vrijgegeven kan worden en gebruikt kan worden voor nieuwe data. Het vrijgeven van geheugen wordt verzorgd door de funktie *free()*. De syntax van *free()* is als volgt :

*void free(void *ptr)*

Heel in het kort wordt in het volgende voorbeeld *free()* gedemonstreerd.

```
/* vb8_6.c : Reallocatie realloc()                                  */
#include <stdio.h>
#include <stdlib.h>

main()
{ char *p=NULL;
  p = (char *) malloc(11 * sizeof(char));       /* alloceer  */
  free(p);                                      /* geef vrij */
}
```

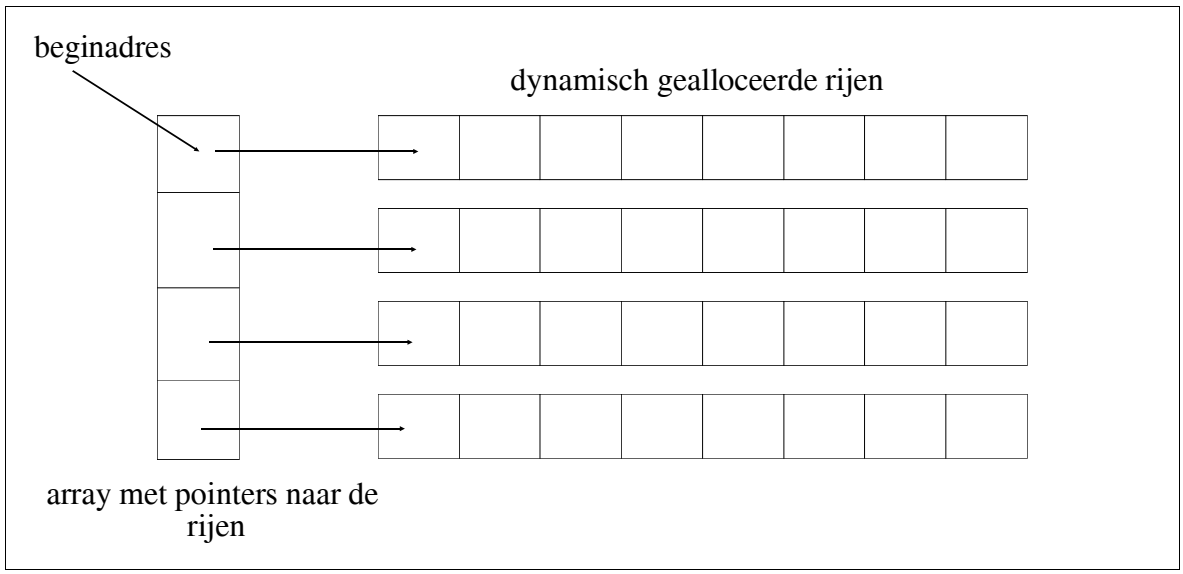
voorbeeld 8.6 : ***free()***

Verderop zal *free()* vaker worden toegepast.

8.4 Pointers en arrays als dynamische data

We hebben hiervoor al gezien dat bij de indexering van arrays gebruik gemaakt kan worden van pointers. In de hieraan voorafgaande voorbeelden hebben we ook gezien hoe een één-dimensionaal array dynamisch kan worden gealloceerd met behulp van een pointer. Het bleek dat een *dynamisch* gealloceerd array in het verdere gebruik net zo werkt als een *statisch* array. Indien meerdimensionale arrays dynamisch gealloceerd worden zal er gebruik gemaakt moeten worden van *pointers naar pointers.* Dit zal uitgelegd worden aan de hand van een *n x m* matrix.

Een matrix (*n x m*) kan *dynamisch* worden gealloceerd door de rijen van de matrix als dynamische arrays te alloceren. We willen echter niet *n* rijen los ergens in het geheugen hebben hangen dus bergen we de beginadressen van alle rijen op in een array met daarin de *pointers* naar die beginadressen. Dit *array van pointers* heeft ook een startadres en dit adres is op te vatten als een *pointer naar pointers* en vormt als het ware het beginadres van de matrix. In de hieronder afgebeelde illustratie wordt dit verduidelijkt.



voorbeeld 8.7 : Pointer naar pointer

Als we een dergelijke matrix willen alloceren beginnen we met de allocatie van de *array van pointers*. Vervolgens alloceren we de afzonderlijke rijen waarbij de *pointers* naar deze *rijen* worden opgeborgen in het eerder gealloceerde *array van pointers*. Het volgende voorbeeld laat dit zien.

```
/* vb8_8.c : Pointers en Arrays                                    */
#include <stdio.h>
#include <stdlib.h>

main()
{int i,j,n,m; double **A;  /* matrix A m-rijen en n-kolommen */
 char s[10]; A = NULL;
 printf("geef n : "); gets(s); n = atoi(s);
 printf("geef m : "); gets(s); m = atoi(s);
 /* eerst het array van pointers alloceren                         */
 A = (double *(*))calloc((unsigned int)(m),sizeof(double *));
 /* daarna per rij een dynamische allocatie (vectoren)             */
 for (i=0; i<n; i++)
 { A[i] = NULL;
   A[i] = (double *) calloc((unsigned int)(n), sizeof(double));
 }
 for (i=0; i<m;i++)
   for (j=0; j<n; j++)
     A[i][j] = 10.0;
 for (i=0; i<m;i++)
 { for (j=0; j<n; j++)
     printf("%e ",A[i][j]);
   printf("\n");
 }
}
```

voorbeeld 8.8 : Pointers en arrays

Ook hier blijkt dat de *dynamisch* gealloceerde array net zo te gebruiken is als zijn *statische* collega.

8.5 Doorgeven van pointers aan funkties

Het doorgeven van pointers aan funkties verschilt niet veel van het doorgeven van de tot nu toe gehanteerde datatypen. Het is zelfs al ter sprake gekomen bij de uitleg over de *call by reference*. Hieronder wordt een eenvoudig voorbeeldje gegeven.

```
/* vb8_9.c : Pointers en funkties                                  */
#include <stdio.h>
#include <stdlib.h>

void v_iclear(v,n) int *v,n;
{int i; for (i=0; i<n; i++) v[i]=0;}

void v_iprint(v,n) int *v, n;
{ int i; for(i=0; i<n;i++) printf("v[%d]=%d\n",i+1,v[i]);}

main()
{ int *vi=NULL;
  vi = (int *) malloc(10 * sizeof(int));        /* alloceer  */
  v_iclear(vi,10); v_iprint(vi,10);  /* funkties met pointer */
}
```

voorbeeld 8.9 : Pointers en funkties

9 GEAVANCEERDE DATATYPEN

Met de *standaard* datatypen in ***C*** kunnen we nu overweg. Variabelen kunnen *individueel* gedeclareerd worden. Deze *"losse"* variabelen kunnen we echter ook nog groeperen tot meer gestruktureerde data, wat bij elkaar hoort plaats je bij elkaar. In dit hoofdstuk zal uiteen worden gezet hoe je met dergelijke geavanceerde strukturen van data (*structures*) omgaat en hoe je je eigen datatype creëert. Daarom wordt eerst begonnen met het *typedef* commando waarmee eigen datatypen gedefinieerd kunnen worden.

9.1 Nieuwe datatypen, typedef

In feite kun je *geen* eigen datatype definiëren in ***C***, alleen de naam van een bestaand type kun je wijzigen. Het lijkt echter alsof je iets nieuws hebt gemaakt daarom de aanduiding *nieuw datatype*. Laten we als voorbeeld nemen het *type guldens* uit voorbeeld 2.3 op pagina 5. Hier is in feite een nieuwe *typedefinitie* gegeven aan een *double*. In dit programma kan dus iets gedefinieerd worden van het *type guldens* wat in feite hetzelfde is als *double*. Door middel van het commando *typedef* wordt deze type-aanduiding aangemaakt. De syntax voor *typedef* is hieronder aangegeven :

typedef type-specifikatie naam;

Een voordeel van *typedef* is dat je een meer sprekende naam aan een bepaald *type* variabele kan geven. Tevens kan het uitkomst bieden indien de code op een ander platform moet kunnen draaien. Er hoeft dan immers maar op één plaats (in de *typedef*) iets gewijzigd te worden.

9.2 Verzamelde data in structs

Naast het creëren van nieuwe *namen* voor *datatypen* is het ook prettig om losse data te *groeperen* in een nieuw datatype. Ook hier is niet echt sprake van een *nieuw type* maar meer van een *nieuwe aanduiding* voor bestaande datatypen.

Stel als voorbeeld een groep van data welke bij elkaar horen. Te denken valt aan *window* data :

vensternaam
coordinaten linker bovenhoek
coordinaten rechter benedenhoek
kleur
aktief

We kunnen nog veel meer bedenken dat bij een *window* hoort maar laten we het voorlopig maar hier op houden. Als we bijvoorbeeld vier windows op het scherm willen toveren is het lastig om vier variabelen voor vensternamen te definieren. Beter is om een *datatype window* te definieren welke de bovenstaande variabelen in zich *"draagt"*. Vervolgens kunnen we vier variabelen van het *type window* declareren en al onze data is nu beschikbaar.

Een *struct* wordt gedefinieerd met behulp van het sleutelwoord *struct* volgens de volgende syntax :

struct naam { lijst van type-specifikatie variabelenaam } ;

Hierbij wordt de lijst van de variabelen gescheiden door comma's :

struct voorbeeld
{ int var1;
double var2;
};

Een *variabele* kan nu van een *struct type* worden voorzien door een standaard variabele declaratie :

struct voorbeeld vb1, vb2, vb3;

Op deze manier zijn drie variabelen (vb1, vb2 en vb3) van het *type struct voorbeeld* gedeclareerd. Om de variabelen *binnen* de *struct* te kunnen aanspreken wordt de volgende syntax gehanteerd :

struct_variabele_naam.variable

Als we een variabele *binnen* een *struct* willen gebruiken gebruiken we de *dot* (**.**) operator. De *variabele* in een *struct* mag ook een eerder gedefinieerde *struct* zijn. Er is dan sprake van *nesting*. In het voorbeeld wordt het *window* voorbeeld met een *struct* gedefinieërd.

```
/* vb9_1.c : Structs                                                  */
#include <stdio.h>
#define ROOD 5

struct window
{ char     vensternaam[80];
  double   x_boven, y_boven;
  double   x_onder, y_onder;
  int      kleur;
  int      aktief;
};

main()
{ struct window win1, win2, win3, win4;
  strcpy(win1.vensternaam,"Window nr 1"); win1.kleur  = ROOD;
  win1.x_boven = 0.0; win1.y_boven = 0.0; win1.aktief = 0;
  win1.x_onder = 0.0; win1.y_onder = 0.0;
  printf("win1 titel : %s     \n",win1.vensternaam);
  printf("     x1,y1 : %lf %lf\n",win1.x_boven,win1.y_boven);
  printf("     x2,y2 : %lf %lf\n",win1.x_onder,win1.y_onder);
  printf("     kleur : %d     \n",win1.kleur);
  printf("     aktie : %d     \n",win1.aktief);
}
```

voorbeeld 9.1 : ***Struct***

Variabelen in een *struct* kunnen ook d.m.v. een *call by reference* worden doorgegeven. De *& pointer operator* wordt dan voor de *struct-naam* geplaats, niet voor de *variabele-naam* :

&naam.variabele

9.3 Arrays van structs en pointers naar naar structs

We hebben in het voorgaande voorbeeld gezien dat er vier variabelen van het *type struct window* werden gedeclareerd. Ook deze losse data kunnen we verzamelen in een *array van het type struct window*. We hebben dan slechts één variabele waarin *alle window- data* is opgeborgen. Als we dit dan ook nog eens combineren met een *dynamisch gealloceerd array van het type struct window* dan zijn we in staat om een van te voren onbepaald aantal windows de definiëren. We maken dan echt gebruik van de geavanceerde datatypen in ***C***. In deze paragraaf zal worden uiteengezet hoe dit aangepakt moet worden. We beginnen eerst met *statische arrays* van *structs*. Daarna zal het *pointer* begrip worden uiteengezet in *combinatie met structs* waarna ten slotte een voorbeeld volgt met een *dynamische* geheugenallocatie voor een *array van structs*.

9.3.1 Statische arrays van structs

Statische *arrays van structs* worden op dezelfde wijze gedeclareerd als normale statische arrays zoals behandeld in paragraaf 7.1. In het voorbeeld hieronder wordt dit getoond :

```
/* vb9_2.c : Statische arrays van structs                    */
#include <stdio.h>
#define ROOD 5

struct window
{ char     vensternaam[80];
  double   x_boven, y_boven;
  double   x_onder, y_onder;
  int      kleur;
  int      aktief;
};

main()
{ struct window win[4]; /* vier windows win[0...4]          */
  strcpy(win[0].vensternaam,"Window nr 1");
  win[0].kleur   = ROOD;
  win[0].x_boven = 0.0; win[0].y_boven = 0.0;
  win[0].aktief = 0;
  win[0].x_onder = 0.0; win[0].y_onder = 0.0;
  printf("win1 titel : %s     \n",win[0].vensternaam);
  printf("     x1,y1 : %lf %lf\n",win[0].x_boven,
                                  win[0].y_boven);
  printf("     x2,y2 : %lf %lf\n",win[0].x_onder,
                                  win[0].y_onder);
  printf("     kleur : %d     \n",win[0].kleur);
  printf("     aktie : %d     \n",win[0].aktief);
}
```

voorbeeld 9.2 : Statische arrays van ***structs***

Nu we vier variabelen hebben welke in één array van structs zijn opgeborgen is het makkelijker om funkties te definiëren waaraan de *window data* kan worden doorgegeven. Als we tevens in deze funkties gebruik willen maken van *call by reference data transport* dan hebben we *pointers naar structs* nodig. De volgende paragraaf toont het gebruik van *pointers* naar *structs*.

9.3.2 Pointers en structs

Het gebruik van *pointers* bij *struct variabelen* wijkt een beetje af van wat we tot nu toe hebben gezien. Wel werken we hier met de *pointer operatoren* * en *&* maar de *verwijzing* naar de *struct data* wijkt een beetje af. De syntax van een *struct pointer* luidt :

*struct naam *struct_variabele_naam;*

Deze *declaratie* volgt dus de gebruikelijke weg. Als we echter nu de waarde van een variabele binnen de struct willen bereiken moeten we de volgende notatie hanteren :

*(*struct_variabele_naam).variabele_naam*

In oude ***C*** programmatuur is de bovenstaande manier de enige manier om een variabele binnen een *struct pointer* te bereiken. In de huidige taal-definitie is een alternatief voorhanden waardoor het gedoe met haakjes , * en . vereenvoudigd wordt tot :

*struct_variabele_naam***->***variabele_naam*

De pijl (*arrow*) operator vervangt dus de punt (*dot*) operator indien het een *structure pointer* betreft. In het volgende voorbeeld wordt het gebruik van de *arrow* operator gedemonstreerd.

```
/* vb9_3.c : Pointers naar structs                              */
#include <stdio.h>
struct window
{ char     vensternaam[80];
  double   x_boven, y_boven;
  double   x_onder, y_onder;
  int      kleur;
  int      aktief;
};

void win_ini(w) struct window *w;     /* waarden veranderen  */
{ strcpy(w->vensternaam,"noname");    /* call by reference   */
  w->x_boven = 0.0;   w->y_boven = 0.0;   w->kleur = 5;
  w->x_onder = 100.0; w->y_onder = 100.0; w->aktief= 0;
 }

void win_set(w) struct window w;      /* waarden onveranderd */
{ printf("titel : %s     \n",w.vensternaam);
  printf("x1,y1 : %lf %lf\n",w.x_boven,w.y_boven);
  printf("x2,y2 : %lf %lf\n",w.x_onder,w.y_onder);
  printf("kleur : %d     \n",w.kleur);
  printf("aktie : %d     \n",w.aktief);
}

main()
{ struct window win1, win2;
  win_ini(&win1); win_ini(&win2);     /* call by reference   */
  win_set(win1);  win_set(win2);      /* call by value       */
}
```

voorbeeld 9.3 : Pointers naar ***structs***

9.3.3 Dynamische arrays van structs

Tot nu toe zijn alleen *statische structs* behandeld. Net als arrays van de *standaard datatypen* kunnen ook *arrays van structs dynamisch* worden gealloceerd. Wat we dan nodig hebben is een *struct pointer* die we op NULL zetten en daarna een geheugengebied toewijzen met een *malloc()* of *calloc()*. De hoeveelheid ruimte in bytes die nodig is bepalen we door middel van de *sizeof()* funktie zoals we al eerder hebben gezien. In het onderstaande voorbeeld zal weer de *window struct* worden gehanteerd maar nu voeren we een *dynamische geheugenallocatie* uit en vullen we de array van *structs* met initiële waarden.

```
/* vb9_4.c : Dynamische arrays van structs                    */
#include <stdio.h>
#include <stdlib.h>

struct window
{ char     vensternaam[80];
  double   x_boven, y_boven;
  double   x_onder, y_onder;
  int      kleur;
  int      aktief;
};

void win_ini(w,string)
  struct window *w; char *string;     /* waarden veranderen  */
{ strcpy(w->vensternaam,string);      /* call by reference   */
  w->x_boven = 0.0;   w->y_boven = 0.0;   w->kleur = 5;
  w->x_onder = 100.0; w->y_onder = 100.0; w->aktief= 0;
 }

void win_set(w) struct window *w;     /* waarden onveranderd */
{ printf("titel : %s     \n",w->vensternaam);
  printf("x1,y1 : %lf %lf\n",w->x_boven,w->y_boven);
  printf("x2,y2 : %lf %lf\n",w->x_onder,w->y_onder);
  printf("kleur : %d     \n",w->kleur);
  printf("aktie : %d     \n",w->aktief);
}

main()
{ int i,n; struct window *win, *w; char s[10],naam[80];
  win = NULL;
  printf("hoeveel windows ? : "); gets(s); n = atoi(s);
  win = (struct window *) calloc((unsigned int)(n),
                                 sizeof(struct window));
  if (!win) { printf("calloc error\n\n"); exit(1); }
  for (i=0; i<n; i++)
  { sprintf(naam,"window nr %d",i+1);
    w = &win[i];                      /* een hulp pointer om */
    win_ini(w,naam);                  /* naar de rij data te */
    win_set(w);                       /* te wijzen           */
  }
  printf("ready\n");
}
```

voorbeeld 9.4 : Dynamische arrays van ***structs***

9.4 Unions

Sterk lijkend op de *struct* kent ***C*** nog een verzameltype voor data : *union*. Het verschil tussen *structs* en *unions* is dat in een *union* de data hetzelfde *geheugen-beginadres* heeft. Dit klinkt een beetje criptisch maar als in een *union* een *int* en een *char* zijn gedefinieërd dan reserveert ***C*** een stukje geheugen voor de *union* ter grootte van de *int* aangezien deze de meeste bytes inneemt. De data is dus *niet* gelijktijdig toegankelijk. Daarmee is de *union* alleen geschikt voor die situaties waarbij het zeker is dat slecht één van de variabelen in de *union* aktief gebruikt wordt. De syntax voor de union definitie luidt :

union naam { lijst van type-specifikatie variabelenaam } ;

Hierbij wordt de lijst van de variabelen gescheiden door comma's :

```
union voorbeeld
{ int var1;
  double var2;
};
```

Een *variabele* kan nu van een *union type* worden voorzien door een standaard variabele declaratie :

union voorbeeld vb1, vb2, vb3;

Dit is dus geheel overeenkomstig de gang van zaken bij *structs*. Deze manier van data-bundeling neemt dus aanzienlijk minder ruimte in beslag, echter dedata is niet gelijktijdig toegankelijk. Het volgende standaard voorbeeldje laat dit zien.

```
/* vb9_5.c : Unions                                              */
#include <stdio.h>

union voorbeeld
{ int    i;
  double d;
};

main()
{ union voorbeeld v;
  v.i = 10;                        /* i staat op adres       */
  v.d = 32.3;                      /* d staat nu op plaats i */
  printf("integer i : %d\n",v.i);  /* dit levert niets       */
  printf("double  d : %f\n",v.d);  /* dit levert d           */
  printf("let op union <> struct\n");
  printf("dus wat we hier willen kan niet met een union\n");
}
```

voorbeeld 9.5 : ***Unions***

In het verdere gebruik is werkt de *union* net zo als een *struct*.

9.5 Opsomming enum

Een handig type in ***C*** is het *enum type*. Dit type is een opsomming (*enumeration*) van waarden welke een variabele van dit type kan hebben. De syntax ziet er als volgt uit :

enum naam { lijst van waarden };

De declaratie van een variabele van het type *enum naam* gaat op de gebruikelijke wijze :

enum naam variabele-naam;

Stel we hebben een *variabele verfkleur* die verschillende waarden kan aannemen. De kleurwaarden (*kleuren*) zetten we in een *enum type* met naam *kleur*. Vervolgens declareren we de *variabele verfkleur* die een waarde kan krijgen uit *kleur*. Het voorbeeld zal dit verduidelijken.

```
/* vb9_6.c : Opsomming enum                                    */
#include <stdio.h>

enum kleur { black, blue, green, cyan, red, magenta, brown,
             lightgray, darkgrey, lightblue, lightgreen,
             lightcyan, lightred, lightmagenta, yellow,
             white
           };
main()
{ enum kleur verfkleur;
  verfkleur = lightcyan;                  /* verfkleur = 11 */
  printf("verfkleur = %d\n",(int)verfkleur);
}
```

voorbeeld 9.6 : Opsomming ***enum***

Default begint de integer telling in een *enum* bij 0. Als we dit niet willen kunnen we aan een of meerdere elementen in de enum een ander getal toekennen de daarop volgende elementen hebben een opvolgend nummer. Als we dus de kleurnummers willen laten beginnen bij 100 zullen we de *enum kleur* definitie moeten aanpassen tot :

```
enum kleur { black=100, blue, green, cyan, red, magenta, brown,
             lightgray, darkgrey, lightblue, lightgreen,
             lightcyan, lightred, lightmagenta, yellow,
             white
           };
```

voorbeeld 9.7 : Opsomming ***enum***

Duidelijk is dat het gebruik van *enum* de leesbaarheid van de code ten goede komt. Feitelijk is het echter niets meer dan een *naamgeving* aan integer waarden voor variabelen. Let wel op de *type cast* in voorbeeld 9.6 want een *enum* is niet (altijd) een *int*.

9.6 Macro's, #define

In hoofdstuk 3 zijn we de macro definitie *#define* al tegen gekomen. Daar werd *#define* gebruikt voor het definiëren van een constante **PI**. Echter de macro definitie kan veel meer dan alleen constanten definiëren. We kunnen er zelfs complete subroutines mee bouwen. Een aardig voorbeeld voor een macro is een algemene allocatie-routine.

Zoals we tot nu toe gezien hebben gaat de *dynamische allocatie* van *arrays* steeds op dezelfde wijze. We zetten een *pointer* naar het *beginadres* van een te alloceren gebied met *malloc()* of *calloc()* en testen of het beginadres niet-NULL oplevert. Het vervelende echter bij deze allocatie is dat de *typen* van de te alloceren variabelen steeds anders kunnen zijn. Een echte *funktie* met vaste *formele* data (van een *vast type*) is nu dus niet te gebruiken. Tot nu hebben we bijvoorbeeld de volgende type data *dynamisch* gealloceerd :

vb8_4.c *int vi*
vb8_4.c *double vd*
vb8_5.c *char p*
vb8_8.c *double *A*
vb8_8.c *double A*
vb8_9.c *int vi*
vb9_4.c *struct window win*

Door nu een macro te definiëren kunnen we wel ieder type data doorgeven aan de *malloc()* of *calloc()*. In het volgende voorbeeld wordt dit macro gegeven.

```
/* vb9_8.c : Macro's met #define                                    */
#include <stdio.h>

#define ERROR(x,s)  if ((x)) printf("%s\n",s); exit(x)

#define ALLOC(t,p,n) \
        if ((p)==null \
          (p)=(t *)calloc((unsigned int)(n),sizeof(x));\
        else \
          (p)=(t *)realloc((char *)(p),\
                   (unsigned int)((n)*(sizeof((t)))));\
        if ((p))==NULL) ERROR(99,"ALLOC error")
#define FREE(p)
        if ((p)!=NULL)\
        { free((char *)(p)); (p)=NULL;\
        }


main()
{ int *iv=NULL; double *dv=NULL;
  ALLOC(iv, 100, int); ALLOC(dv, 100, double);
  printf("allocatie gelukt\n");
  FREE(iv); FREE(dv);
  printf("data vrijgegeven\n");
  ERROR(1,"einde programma");
}
```

voorbeeld 9.8 : Macro's met ***#define***

9.7 Conditionele compilatie

Vooral indien programma's op verschillende machines gecompileerd worden, of als er verschillende compilers gebruikt worden kan het handig zijn om bepaalde stukken code uit te sluiten van compilatie. In ***C*** kan dat met behulp van *conditional compiler directives*. Dit zijn als het ware aanwijzingen voor de compiler hoe met bepaalde delen van de code om moet worden gegaan tijdens compilatie.

Met de *directives #ifdef, #ifndef* en *#endif* kunnen dit soort konstrukties worden gemaakt. Stel we compileren met de **TURBO-C** compiler en we willen graag de niet **ANSI-C** funktie *clrscr()* gebruiken. Indien we de source met een andere compiler compileren volgt een foutmelding op de *clrscr()* funkties. Om dit te omzeilen maken we een eigen schermfunktie waarin de **TURBO-C** funktie opgenomen is. Door het plaatsen van *#ifdef* en *#endif* is de *clrscr()* funktie geisoleerd. Het voorbeeld laat dit zien.

```
/* vb9_9.c : Conditionele compilatie                              */
#include <stdio.h>

#define TURBO

void cls()
{
#ifdef TURBO
   clrscr();
#endif
}

main()
{ printf("een testje met #ifdef\n");
  cls();
}
```

voorbeeld 9.9 : Conditionele compilatie

Laat de *#define* TURBO eens weg om te zien welk effect dat heeft op de *executable*.

Naast de *#ifdef* bestaat ook de *#ifndef* directive. Deze test of een bepaalde *define* niet bestaat. Met *#if* en *#else* kunnen soortgelijke konstrukties worden gebouwd als we al kennen uit het hoofdstuk over operatoren.

Een blok beginnend met *#ifdef*, *#ifned*, *#if* of *#else* moet altijd worden afgesloten met *#endif*.

10 INVOER EN UITVOER

Tot nu toe is in dit cursusboek slechts incidenteel gebruik gemaakt van invoer-funkties. Dat is met opzet gebeurd om niet te veel nieuwe onderwerpen op hetzelfde moment te introduceren. In dit hoofdstuk zal worden uitgelegd hoe in ***C*** invoer verkregen kan worden vanaf het toetsenbord of vanuit files en hoe uitvoer naar het scherm en files gestuurd kan worden.

10.1 Invoer vanaf het toetsenbord, gets(), scanf(), getche() en getchar()

In de voorgaande voorbeelden is steeds gebruik gemaakt van een standaard ***C***-funktie *gets()* om een *string* van het *toetsenbord* in te kunnen lezen. Daarna werd de gelezen *string* met bijvoorbeeld de funktie *atoi()* naar een *integer* getal geconverteerd (met *atof()* naar een *float*). Uit de voorbeelden hebben we al gemerkt dat de invoer wordt afgesloten door op de ENTER toets te drukken.

De string welke ingelezen wordt moet wel een *beginadres* hebben dus of een *statisch gedeclareerde array van char* of een *gealloceerde dynamische array van char*. De syntax definitie van *gets()* luidt aldus :

*char *gets(char *string);*

De *gets()* funktie levert dus een funktie-waarde op. Dit is kennelijk het beginadres van de ingelezen string. We kunnen hier ook weer net als bij *malloc()* en *calloc()* een test inbouwen of de aanroep van *gets()* een *niet* NULL *adres* opleverd. Op deze manier kan faal-vrije invoer gecreëerd worden waardoor er een robust programma ontstaat.

Met *gets()* kunnen we alleen *strings* inlezen. Er bestaat in ***C*** echter ook een funktie waarmee data op maat kan worden ingelezen. Het in te lezen *type* wordt van te voren aan de funktie doorgegeven. Deze funktie is *scanf()*. Met *scanf()* is het mogelijk om meerdere getallen die door een spatie zijn gescheiden met één *scanf()* aanroep in te lezen. De syntax van *scanf()* luidt :

*int scanf(char *format, arg-list);*

Deze funktie heeft enorm veel mogelijkheden. In de *format* wordt aangegeven welk *type* data gelezen moet worden terwijl in de *arg-list* de *variabele namen* worden opgenomen. In het hieronder weergegeven voorbeeld wordt het gebruik van *scanf()* verduidelijkt. De invoerdata moet nu gescheiden door spaties ingetypt worden.

```
/* vb10_1.c : Scanf()                                                */
#include <stdio.h>

main()
{ int aantal=0,i=0; double d=0.0; char s[20];
  printf("geef int double string gescheiden dmv spaties : ");
  aantal = scanf("%d%lf%s",&i,&d,s);
  printf("integer %d double %lf string %s\n",i,d,s);
  printf("aantal velden : %d\n",aantal);
}
```

voorbeeld 10.1 : ***Scanf()***

De *return* waarde van *scanf()* is het aantal velden dat is ingelezen. Let erop dat de variabelen in *scanf()* als een *call by reference* moeten worden doorgegeven, immers de waarde van de variabele moet veranderd worden ! Een *array van char* wordt niet voorafgegaan door een *&* want de *naam* van de array is het *beginadres* van de array waardoor het gebruik van de *adres operator &* overbodig wordt.

Naast de *scanf()* funktie bestaan in ***C*** ook nog de *getche() , getchar()* en *getch()* funkties voor het inlezen van een enkel karakter. Het verschil tussen *getche()* en *getchar()* is dat de laatste altijd op een ENTER wacht terwijl de eerste direkt reageert op een toetsaanslag. Het verschil tussen *getche()* en *getch()* is dat de eerste funktie het karakter op het *scherm* toont terwijl *getch()* dit niet doet.

```
/* vb10_2.c : Karakter input                                         */
#include <stdio.h>

main()
{ char c;
  c = '0';
  printf("geef karakter : "); c=getche();
  printf("getche  -> c=%c\n",c);
  printf("geef karakter : "); c=getchar();
  printf("getchar -> c=%c\n",c);
  printf("geef karakter : "); c=getch();
  printf("getch   -> c=%c\n",c);
}
```

voorbeeld 10.2 : Karakter input

10.2 Uitvoer naar het scherm

De uitvoer naar het scherm levert veelal geen problemen op. In voorgaande paragrafen is al veelvuldig gebruik gemaakt van de uitvoer funktie *printf()*. Deze funktie biedt alle mogelijkheden voor een nette uitvoer en is reeds besproken in paragraaf 3.3.2. Formeel is de syntax van *printf()* :

*int printf(char *format, arg-list);*

Hieruit blijkt een sterke overeenkomst met de *scanf()* funktie. De funktie heeft als *return* waarde het aantal karakters dat wordt weggeschreven.

10.3 File controle routines

C beschikt over een groot aantal funkties waarmee het lezen en schrijven naar *files* kan worden geregeld. De belangrijkste daarvan zullen in deze paragraaf aan de orde komen. Het gaat hierbij dan om :

fopen	*voor het openen van files*
fclose	*voor het sluiten van files*
remove	*voor het wissen van files*
rename	*voor het hernoemen van files*
fflush	*voor het legen van buffers (lezen en schrijven)*

Deze funkties zullen hieronder kort worden beschreven. Alvorens we daarmee kunnen beginnen zal eerst het begrip *file-pointer* worden uitgelegd.

10.3.1 File pointer

Alle in- en uitvoer in ***C*** verloopt op een *standaard* wijze. We spreken dan ook van *standaard I/O*. Zo is de toegang tot *files* geregeld door middel van een *file-pointer*. Deze pointer wordt in ***C*** een *stream* genoemd. Een *stream* hoeft niet een *file* te zijn maar dat kan wel. We kunnen met een *stream* ook het scherm bedoelen of het toetsenbord. ***C*** maakt dus fysiek geen onderscheid tussen de verschillende invoermogelijkheden, alles wordt door middel van een *stream* behandeld. De *file-pointer* is dus een *stream* die we gebruiken voor *fysieke files*. De *file-pointer* is in de *header file* **stdio.h** als een *struct* gedefinieërd met daarin gegevens over de file. We zullen op die gegevens hier niet verder ingaan. Belangrijk is hoe we zelf een *file-pointer* kunnen declareren. Dit doen we op de volgende wijze :

```
FILE *file-pointer-naam;
```

Let erop dat het woordje FILE in *hoofdletters* staat en vergeet niet de *pointer operator* * voor de naam van de *file-pointer*.

10.3.2 Openen van files

Als we over een *file-pointer* beschikken kunnen we met de *fopen()* funktie een file *openen*. Daarbij kunnen we aangeven wat de *naam* van de te openen *file* is en welke *aktie* we willen uitvoeren op de file (lezen, schrijven of toevoegen). De syntax van *fopen()* luidt :

```
FILE *fopen(char *file-naam, char *mode);
```

We zien dat de funktie *fopen()* een return waarde heeft van het *type* pointer naar FILE. De *aktie* welke kan worden ondernomen wordt aangegeven met een string *mode* :

"r"	lezen van een file (*read*)
"w"	schrijven naar een file (*write*)
"a"	toevoegen aan een file (*append*)
"rb"	lezen van een binaire file
"wb"	schrijven naar een binaire file
"ab"	toevoegen aan een binaire file

In het voorbeeld 10.3 wordt getoond hoe een *file geopend* kan worden. Tevens wordt gecontroleerd dat de *file-pointer* niet NULL is waardoor fouten kunnen worden vermeden.

10.3.3 Sluiten van files

Nadat een *file geopend* is met *fopen()* heeft de *file-pointer* een waarde welke *niet* NULL is. We kunnen de file *sluiten* met *fclose()*. De syntax van *fclose()* luidt :

*int fclose(FILE *stream);*

Indien de funktie met succes is uitgevoerd levert *fclose()* een *return* waarde 0. In het voorbeeld wordt het openen, lezen uit en het sluiten van een file **cursist.dat** gedemonstreerd.

```
/* vb10_3.c : Openen en sluiten van een file                */
#include <stdio.h>

FILE *fp;

void lees_file()                        /* lege lees routine */
{
}

main()
{ fp=NULL;
  if ((fp = fopen("cursist.dat","r")) == NULL)  /* open file */
  { printf("file cursist.dat niet gevonden\n"); exit(1);
  }
  else                                          /* file bestaat */
  { lees_file();
  }
  if (fclose(fp))                               /* file sluiten */
    printf("fout bij sluiten van file\n");
  else
    printf("foutloos\n");
}
```

voorbeeld 10.3 : ***fopen()*** en ***fclose()***

Standaard zal ***C*** ervoor zorgen dat bij het beëindigen van het programma de *files* gesloten worden.

10.3.4 Wissen van files, remove() en unlink()

Het wissen van bestanden gaat erg simpel in ***C***. Met *remove()* of *unlink()* wordt de filenaam doorgegeven welke gewist moet worden. De syntax van *remove()* luidt :

*int remove(char *file-naam);*

De funktie levert een 0 op indien de file met succes is verwijderd en een -1 indien een fout is opgetreden. *Unlink()* werkt op precies dezelfde wijze en is afkomstig vanuit UNIX. Als *unlink()* gebruikt wordt moet het *header bestand* **dos.h** in de code worden opgenomen.

10.3.5 Hernoemen van files, rename()

Naast de funktie *remove()* bestaat er ook de funktie *rename()* voor het hernoemen van *file-namen*. De syntax van *rename()* luidt :

*int rename(char *oude-naam, char *nieuwe-naam);*

Ook hier levert de funktie een 0 op indien met succes een file-naam is *gewijzigd*.

10.4 Legen van de buffer, fflush()

De syntax van *fflush()* luidt :

*int fflush(FILE *stream);*

Een aanroep van *fflush()* bij het lezen zal de *input-buffer* legen. Bij schrijven wordt de *uitvoer-buffer* geleegd. De data wordt dan direkt weggeschreven. Bij succes levert de funktie een 0 als *return-waarde*. De funktie zal slechts *incidenteel* gebruikt worden.

10.5 Schrijven naar files, fprintf(), fputc() en fputs()

Bij het schrijven naar files wordt vrijwel dezelfde routine gebruikt als voor het schrijven naar het beeldscherm. De algemene schrijf-funktie *fprintf()* is als volgt gedefinieerd :

*int fprintf(FILE *stream, char *format, arg-list);*

De gelijkenis met *printf()* is duidelijk te zien. De werking is ook identiek aan die van *printf()* zij het dat een *file-pointer* moet worden meegegeven. De *return-waarde* is het aantal karakters dat is weggeschreven. Indien een fout optreedt zal *fprintf()* een negatief getal teruggeven.

Naast de *fprintf()* funktie bestaan er ook nog de funkties *fputc()* en *fputs()*. De syntax van *fputc()* waarmee karakters kunnen worden weggeschreven luidt :

*int fputc(int ch, FILE *stream);*

Het karakter is hier als *int* gedefinieerd maar als een *char* wordt aangeboden wordt door de funktie een *type conversie* uitgevoerd. De *return-waarde* van *fputc()* is de waarde van het karakter. Indien er een fout optreedt wordt een constante EOF (end of file) als *return-waarde* afgegeven.

De funktie *fputs()* wordt gebruikt voor het wegschrijven van een *string*. In tegenstelling tot *fprintf()* kan hier geen *argumenten-lijst* worden meegegeven. De syntax van *fputs()* luidt :

*int fputs(char *string, FILE *stream);*

Indien *fputs()* met succes is afgesloten is de *return-waarde* 0. In het andere geval wordt een niet-nul getal teruggegeven.

Vanwege de flexibiliteit geef ik de voorkeur aan de *fprintf()* funktie. Op de volgende bladzijde wordt het gebruik van *fprintf()*, *fputc()* en *fputs()* gedemonstreerd.

```
/* vb10_4.c : fprintf(), fputc() en fputs()                    */
#include <stdio.h>

FILE *fp;

main()
{ char string[10]="hallo", ch='A'; double d=11.11; int i=4;
  if ((fp = fopen("vb10_4.out","w"))==NULL)
  { printf("fout : vb10_4.out niet geopend\n"); exit(1); }
  else
  { fprintf(fp,"int %d double %lf string %s char %c\n",
            i,d,string,ch);
    fflush(fp);                    /* leeg de uitvoer buffer */
    fputs("nu met fput.. routines\n",fp);
    fputc(ch,fp); fputs(string,fp);
  }
}
```

voorbeeld 10.4 : ***fprint(), fputc(), fputs()***

10.6 Lezen vanuit files, fgets(), fscanf() en fgetc()

Ook bij het lezen vanuit een file worden bijna dezelfde funkties gebruikt als die we al tegen kwamen bij het inlezen van data vanaf het toestenbord. Vanwege deze sterke overeenkomst zal ik hier weinig uitleg geven.

De funktie *fgets()* leest een *string* in vanuit een file. De syntax luidt :

*char *fgets(char *string, int lengte, FILE *stream);*

Er worden nu *lengte-1* karakters ingelezen in de string. De funktie geeft het *adres* naar de string als *return-waarde* terug. Indien dit adres NULL is is er iets fout gegaan.

De funktie *fscanf()* lijkt sprekend op *scanf()*. De syntax luidt :

*int fscanf(FILE *stream, char *format,arg-list);*

Bij succes zal *fscanf()* het aantal ingelezen *velden* als *return-waarde* teruggeven. Indien EOF wordt teruggegeven dan is het einde van de file bereikt.

De funktie *fgetc()* ten slotte leest een enkel karakter uit een file. De syntax van deze funktie is als volgt :

*int fgetc(FILE *stream);*

De return-waarde van *fgetc()* is dus het karakter. Dat een karakter van het *type char* is maakt niet uit, de conversie wordt automatisch uitgevoerd.

Een voorbeeld van gelezen invoer uit een ASCII-file is in het volgende voorbeeld weergegeven.

```
/* vb10_5.c : fgets() fscanf() fgetc()                          */
#include <stdio.h>

main()
{ char string[101]; int i; double d; char ch; FILE *fp;
  if ((fp = fopen("cursist.dat","r"))==NULL)
  { printf("file cursist.dat niet gevonden\n"); exit(1); }
  else
  { fgets(string,100,fp);
    printf("string : %s",string);
    fscanf(fp,"%d%lf %ch",&i,&d,&ch);
    printf("i=%d d=%lf ch=%c\n",i,d,ch);
    do
    { ch = fgetc(fp);
      printf("%c",ch);
    } while (!feof(fp));                  /* tot einde file */
  }
}
```

voorbeeld 10.5 : ***fgets(), fscanf() fgetc()***

10.7 Einde file, feof() en EOF

Als het *einde van de file* bereikt wordt, bij lezen of bij schrijven, kunnen er fouten optreden. Om deze af te vangen kan met *feof()* getest worden of het einde van de file bereikt is :

*int feof(FILE *stream);*

De waarde van *feof()* kan in een logische expressie worden opgenomen. Als *feof()* waar is dan is het einde van de file bereikt. Deze toepassing is te zien in het bovenstaande voorbeeld. Naast deze funktie geven sommige *file-funkties* een constante EOF af indien er bij een *file-operatie* iets fout gaat. Bij ASCII files is dit geen probleem aangezien EOF niet een bestaand ASCII-karakter is. Echter bij *binaire files* kan een EOF ingelezen worden terwijl het einde van de file nog niet bereikt is. Bij *binaire files* dient dus altijd door middel van *feof()* op het einde van de file te worden gecontroleerd.

10.8 Positie in een file, rewind(), fseek() en ftell()

Soms kan het nodig zijn om na een aantal leesopdrachten de *file-pointer* weer aan het begin van de file te zetten en opnieuw van voren af aan te beginnen met lezen. De funktie *rewind()* in ***C*** zet de *file-pointer* weer aan het begin van de file.

Tevens kan het voorkomen dat een file zodanig van opzet is dat er vanaf een bepaalde positie in de file gelezen moet worden. Te denken valt bijvooorbeeld aan tekstfiles. Stel we hebben een file met tekst onderverdeeld in pagina's. Als we pagina 3 willen zien dan willen we eigenlijk over de pagina's 1 en 2 heen springen. We kunnen dit doen door de locatie van pagina 3 op te slaan in een *locatie-pointer*. Daarna kunnen we de positie opvragen en er naar toe springen. Met *ftell()* slaan we de positie op en met *fseek()* gaan we er naar toe. Door de locaties van het begin van iedere pagina in een *array* op te slaan kunnen we heel snel van pagina naar pagina springen. Allerlei toepassingen zijn nu mogelijk. In het volgende voorbeeld wordt dit getoond. Zorg dat de file **vb10_6.dat** in dezelfde directory staat als de executable.

```
/* vb10_6.c : ftell() en fseek()                                    */
#include <stdio.h>
#define NMAX 100
FILE *fp;

int aantal_paginas(array,nmax) long int *array; int nmax;
{ int n=0; char name[100];
  do
  { if ((fscanf(fp,"%s",name)!=0) && (!feof(fp)))
    if (strcmp(name,"#pagina")==0)
    { array[n] = ftell(fp);
      n++;
    }
  } while ((!feof(fp)) && (n<nmax));
  array[n+1] = ftell(fp);                      /* einde file */
  return (n);
}

void show_locatie_data(array,n) long int *array; int n;
{ int i;
  printf("aantal pagina's : %3d\n",n);
  for (i=0; i<n; i++)
  { printf("#pagina %3d locatie %ld\n",i+1,array[i]);
  }
}

void lees_pagina(array,pagina) long int *array; int pagina;
{ int i; char s[101]; long int filepointer=0;
  if (fseek(fp,array[pagina-1],0))
  { printf("fseek fout\n"); exit(1); }
  else
  { clrscr(); /* TURBO-C maak het scherm schoon NIET ANSI-C !*/
    do
    { fgets(s,100,fp);                  /* lees   vb10_6.dat */
      filepointer=ftell(fp);            /* ga niet voorbij   */
      if (filepointer >= array[pagina]) /* voorbij volgende  */
        break;                          /* pagina            */
      printf("%s",s);
    }
    while (!feof(fp));
  }
}

main()
{ int pagina=0,n=0; long int loc_array[NMAX+1]; char s[20];
  if ((fp=fopen("vb10_6.dat","r"))==NULL)
  { printf("bestand vb10_6.dat niet gevonden\n"); exit(1); }
  n = aantal_paginas(loc_array,NMAX);
  show_locatie_data(loc_array,n);
  do
  { printf("geef pagina nummer [-1=stop] : "); gets(s);
    pagina = atoi(s);
    if ((pagina > 0) && (pagina <= n))
      lees_pagina(loc_array,pagina);
  }
  while (pagina != -1);
}
```

voorbeeld 10.6 : ***ftell() en fseek()***

De syntax van de funkties *rewind()*, *fseek()* en *ftell()* wordt hieronder weergegeven :

*void rewind(FILE *fp);*

Deze funktie geeft geen waarde terug en zet de l*ocatie-pointer* aan het begin van de file.

*long ftell(FILE *fp);*

Deze funktie geeft de *locatie* terug in een *long int*.

*int fseek(FILE *fp, long int locatie, int mode);*

Hierin is *locatie* de met *ftell()* verkregen positie in de file (of een zelf opgegeven positie). *Mode* geeft aan of het zoeken vanaf het *begin* (0), vanaf de *huidige positie* (1) of vanaf het *einde* (2) van de file moet beginnen. Als de funktie *fseek()* met succes is afgesloten wordt een 0 teruggegeven.

10.9 Standaard IO streams, stdin en stdout

Al eerder is opgemerkt dat in ***C*** de data verwerking naar files door middel van *streams* verloopt. Het beeldscherm en het toestenbord zijn ook *streams*. We kunnen ons dus nu een *file-pointer* voorstellen die naar het toetsenbord wijst. Dit is dus niet een echte *file-pointer* maar een meer algemene pointer, een *stream*. Zo is er ook een *stream* te verzinnen voor het beeldscherm. ***C*** levert deze twee *streams* standaard. Voor het toetsenbord is dit ***stdin*** en voor het beeldscherm is dit ***stdout***. De meer algemene routines waarbij dus een *file-pointer* moet worden meegegeven zijn ook te gebruiken voor het lezen van *stdin* (toetsenbord) en het schrijven naar *stdout* (beeldscherm). In feite kun je dan alle *invoer* en *uitvoer* regelen met behulp van *fscanf()* en *fprintf()*. Een boodschap naar het *scherm* kan dus ook met *fprintf()* :

fprintf(stdout,"hallo Cursist\n");

Dit levert de volgende boodschap op het *scherm* :

```
hallo Cursist
```

10.10 Standaard uitvoer naar de printer (TURBO-C)

Tot slot is het in **TURBO-C** mogelijk om een printer aan een parallelle poort direkt aan te sturen. Hiervoor bestaat dus ook een *standaard stream*. Deze is echter alleen te gebruiken indien de compiler van **TURBO-C** gebruikt wordt. Voor andere compilers kan dus iets anders gelden. In **TURBO-C** kunnen we dus iets naar de printer sturen door gebruik te maken van de *standaard printer stream* : ***stdprn***. De *fprintf()* funktie kunnen we daarvoor dus prima gebruiken :

fprintf(stdprn,"hallo Cursist\n");

Dit levert de volgende boodschap op de *printer* :

```
hallo Cursist
```

11 AANZET VOOR *C++*

In dit korte hoofdstuk zal een aanzetje worden gegeven voor het programmeren in ***C++***. Dit onderwerp kan feitelijk in een aparte cursus worden ondergebracht echter vanwege de sterke relatie met ***C*** zal ik hier een kort overzichtje geven met een eenvoudig voorbeeldje. Het is niet meer dan een introductie. ***C++*** zal echter in de toekomst de taal worden om in te programmeren. In de volgende paragrafen zal in het kort een aantal nieuwe eigenschappen worden behandeld die niet in ***C*** aanwezig zijn. Tot slot worden in een voorbeeld deze kenmerkende eigenschappen gedemonstreerd. Uiteraard is voor ***C++*** een aparte compiler vereist, **Borland** levert een zeer bruikbare welke tevens is voorzien van een prima programmeer-omgeving.

11.1 Inkapseling

In de voorgaande hoofdstukken is regelmatig gewezen op het belang van gestructureerd programmeren. Structuur kan op verschillende manieren en op verschillende plaatsen in het programma worden aangebracht, zie pagina 3. Met name door het introduceren van *structs* is het mogelijk om een overzichtelijke datastructuur op te bouwen. Daarnaast kunnen we door het bij elkaar plaatsen van funkties overzichtelijke bibliotheken bouwen. De missende schakel in dit geheel is dat vele funkties vaak werken op bepaalde data-typen. Het mooie van ***C++*** is nu dat de datastruktuur en de funkties die betrekking hebben op deze struktuur bij elkaar geplaatst kunnen worden (*inkapseling of encapsulation*). De *struct* is nu uitgebreid met funkties. Een eenvoudig voorbeeldje is hieronder weergegeven.

```
/* vb11_1.c : C++ met Standaard C                                    */

#include <stdio.h>
#include <string.h>

struct files
{ FILE *fp;                /* data                                   */
  char name[80];           /* data                                   */
  char iotype[2];          /* data                                   */
  void open_file();        /* funktie                                */
  void close_file();       /* funktie                                */
};                         /* vergeet de ; niet !!!!!                */

/* funktie-inkapseling door middel van :                             */
/* type structnaam::funktienaam(formele data in nieuwe stijl)*/

void files::open_file()  { if (!fp) fp = fopen(name,iotype);}

void files::close_file() { if (fp) fclose(fp); }

main()
{ struct files inp;
  strcpy(inp.name,"noname"); strcpy(inp.iotype,"r");
  inp.open_file(); inp.close();
}
```

voorbeeld 11.1 : Uitgebreide *struct*

Het zal duidelijk zijn dat de voordelen nu niet bepaald van dit voorbeeld afdruipen. De essentie echter dat funkties en data gekoppeld worden wordt wel duidelijk. Object georienteerd programmeren (**OOP**) is nu geintroduceerd want de variabele *inp* mogen we nu niet meer een gewone variabele noemen maar een *object* behorend tot nu nog een *struct* maar deze naamgeving zullen we ook overboord gooien en de *struct* heet dan vanaf nu *class*. Hoewel het voorgaande voorbeeld prima werkt moeten we het eigenlijk als volgt opzetten.

```
/* vb11_2.c : Eenvoudig C++ programma                          */

#include <stdio.h>
#include <string.h>

class files
{ // DATA MEMBERS
  private :
    FILE *fp;
    char name[80];
    char iotype[2];
  // MEMBER FUNKTIES
  public :
    files();                      /* constructor              */
    ~files();                     /* destructor               */
    void set_filename(char *);    /* wijzig de filenaam       */
    char *get_filename();         /* haal de filenaam op      */
    void open_file();             /* funktie voor het openen  */
    void close_file();            /* funktie voor het sluiten */
};

files::files() {fp=NULL; strcpy(name,"x"); strcpy(iotype,"r");}
files::~files(){}
void files::set_filename(char *string){strcpy(name,string);}
char *get_filename()    {return(name); }
void files::open_file() {if (!fp) fp = fopen(name,iotype);}
void files::close_file(){if (fp) fclose(fp);}

main()
{ class files inp;
  inp.set_filename("hallo.txt"); inp.open_file();
  printf("huidige naam : %s\n",inp.get_filename());
  inp.close();
}
```

voorbeeld 11.2 : OOP's nieuwe structuren

Nieuw hierboven zijn de termen ***private*** en ***public***. In tegenstelling tot de *struct* kunnen *data* en *funkties* van een *class* afgeschermd worden. Zo zijn bij het toepassen van een *struct* de data en de funkties ***public*** (onbeschermd) terwijl bij een *class* deze ***private*** (beschermd) zijn. In het hierboven weergegeven voorbeeld is de data van de *class private* en kan dus alleen veranderd worden door een *member-funktie*. De volgende toekenning is met *private data* dus niet mogelijk :

```
strcpy(inp.name,"hallo.txt"); /* dit mag niet,name is private */
```

De data moet met een *member-funktie* gewijzigd worden. Hierdoor kan data afgeschermd worden voor onbevoegden (funkties e.d.).

Nieuw zijn ook de termen *constructor* en *destructor*. Dit zijn feitelijk twee standaard funkties die bij elke *class-definitie* horen. De *constructor* regelt de *data-initialisatie* direct bij het creëren van een *object* terwijl de *desctructor* de data opruimd bij eventuele verwijdering van een *object*. De naam van de *constructor-funktie* is altijd identiek aan de naam van de *class*. Dit geldt ook voor de *destructor*, echter hier wordt de naam voorafgegaan door een ~.

De hierboven weergegeven *class* moet als louter illustratief gezien worden aangezien ***C***$^{++}$ zelf over standaard *classes* beschikt voor IO. Deze wil ik hier echter niet gebruiken aangezien dat teveel afwijkt van de eerder geintroduceerde *io-funkties*. We doen dus net alsof we niet weten dat er een *ios*, *istream* en *ostream class* bestaat.

11.2 Overerving

Naast *inkapseling* behoort *overerving* ook tot een min of meer nieuwe eigenschap van ***C***$^{++}$. Hoewel in ***C*** ook overerving voorkomt (echter op een andere manier) in de vorm van data uit *structs* geldt dit in ***C***$^{++}$ ook voor de *member-funkties*. Bij *overerving* kunnen we een nieuwe *class* definieren die alle eigenschappen van een andere *class* erft. Dit kan eenvoudig door achter de *klasse definitie* de verwijzing naar de *basisklasse* op te nemen :

```
class <klassenaam> : public <basisklassenaam>
```

Op deze manier erft *klassenaam* alles wat al in *basisklassenaam* zit. Tevens kunnen er nu extra *data* en *funkties* aan *klassenaam* worden toegevoegd. Dubbele code wordt zo dus voorkomen en een wijziging van een hoofdeigenschap is alleen in de *basisklasse* vereist. Een voorbeeld van overerving is te zien in het voorbeeld van paragraaf 11.4. Voordat dit voorbeeld wordt behandeld zal eerst nog een andere nieuwe eigenschap van ***C***$^{++}$ worden behandeld.

11.3 Polymorfie

Bij het *overerven* van data en funkties kan het zo zijn dat bepaalde funkties vervangen moeten worden door een nieuwe definitie. Stel dat we van iedere klasse de naam willen weten. We creëren dan in iedere klasse een funktie die de klassenaam terugggeeft :

```
char *klassenaam(void) { return("afgeleide klasse 1");
```

Als we daar steeds dezelfde naam voor kiezen zal na overerving de verkeerde klassenaam worden teruggegeven. We krijgen dan steeds de naam van de *basisklasse* terug. Er is hier dus een probleem bij de *binding* van *data* en *funktie*. De funktie die we in de diverse klassen gebruiken komt steeds in een net iets andere gedaante naar voren. Dit noemen we polymorfie. Het probleem van de *binding* is op te lossen door in alle klassen de funktie die de klasse-naam teruggeeft virtual te maken :

```
virtual char *klassenaam(void) { return ("bla bla bla");
```

Nu zal de compiler snappen dat de naam welke teruggegeven moet worden die is van de geldende klasse en niet die van de basisklasse. In het voorbeeld van paragraaf 11.4 is een voorbeeld weergegeven waarin deze problematiek verduidelijkt wordt. Zeker indien overerving plaatsvindt van funkties met dezelfde naam uit meerdere klassen kunnen er problemen optreden. Het sleutelwoord voor deze problemen luidt dan ook vaak *virtual*. Het voert te ver voor deze korte introduktie om hier verder op in te gaan. In het volgende voorbeeld zal naast *inkapseling* en *overerving* ook *polymorfie* worden verduidelijkt.

11.4 Een eenvoudig *C++* voorbeeld

In deze paragraaf zal een eenvoudig voorbeeld worden gebruikt om de nieuwe eigenschappen van ***C++*** te verduidelijken. Als probleem wordt een adressenbestand gekozen. In een dergelijk bestand vinden we personen met personalia en adressen. De eigenschappen van de database kunnen we splitsen in een naam, adres en geboortedatum gebied. Zo kunnen we een boom van *klassen* definiëren waarbij elke volgende klasse de eigenschappen van de vorige *erft* en tevens nieuwe eigenschappen toevoegt. Voor de database gebruiken we hier de volgende struktuur :

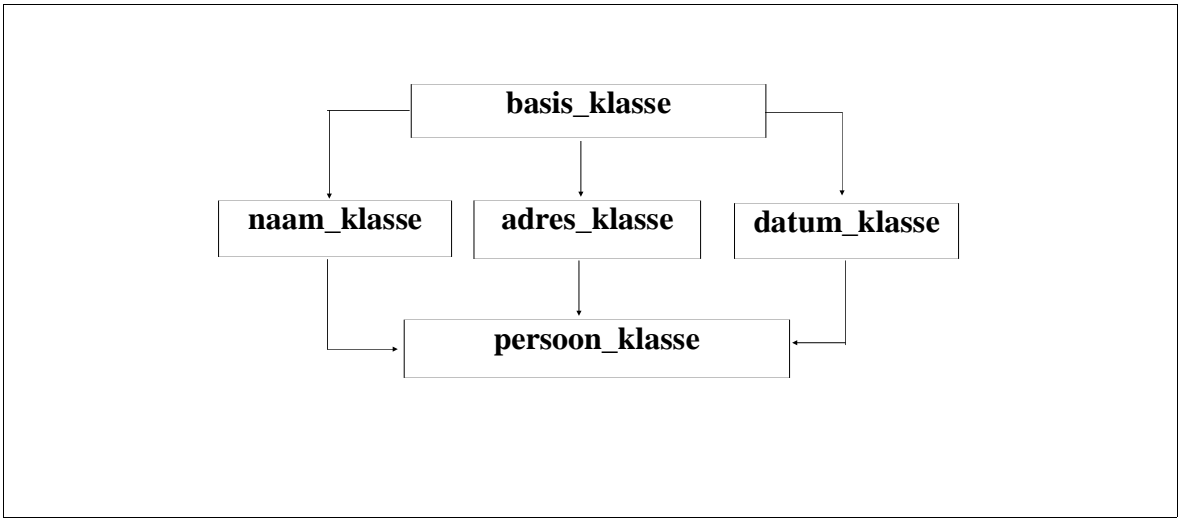


Fig. 11.3 : Database klassen

De klasse *persoon_klasse* erft dus alle eigenschappen van de voorgaande *klassen*. De *basis_klasse* hanteren we hier om een stukje polymorfie te illustreren.

Iedere *klasse* bestaat uit *data* en *funkties.* De funkties betreffen steeds een invoer- en een uitvoerfunktie. De in- en uitvoerfunktie van de *persoon_klasse* vormt uiteindelijk het hart van het programma. In deze funkties wordt gebruik gemaakt van de *overerfde* funkties en dan pas wordt duidelijk dat bij deze opzet programma-wijzigingen *lokaal* aangebracht kunnen worden en geen invloed hebben op de verdere code. Na bestudering van de code zal dit duidelijk worden.

De code van het voorbeeld volgt op de volgende pagina's. Opgemerkt wordt dat het hier niet om de inhoud of waarde van het programma-produkt gaat maar om de programmastruktuur en de verduidelijking van enkele ***C++*** eigenschappen.

```
/* vb11_3.cpp : C++ Database programma                              */

#include <iostream.h>
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>
#include <string.h>
#include <math.h>

/*-------------------------------------------------------------*/
/*   CLASS DEFINITIONS                                         */
/*-------------------------------------------------------------*/
class basis_klasse                  /* geef klasse naam weer  */
{ // MEMBER FUNCTIONS
  public:
    virtual char *klassenaam(void) /* indien virtual wordt   */
    { return("basis_klasse"); }    /* weggelaten zal altijd  */
    void toon_klasse_naam(void);   /* basis_klasse worden    */
};                                 /* getoond                */

class datum_klasse:virtual public basis_klasse
{ // DATA MEMBERS
  protected:
    int   jaar;
    int   maand;
    int   dag;
  // CONSTRUCTOR - DESTRUCTOR
          datum_klasse();
         ~datum_klasse();
  // MEMBER FUNCTIONS
  public:
    char  *klassenaam(void) { return("datum_klasse"); }
    void  datum_invoer();
    void  datum_label(char *string);
};

class naam_klasse:virtual public basis_klasse
{ // DATA MEMBERS
  protected:
    char  voornaam[20];
    char  achternaam[40];
    char  voorvoegsel[10];
    char  voorletters[10];
  // CONSTRUCTOR - DESTRUCTOR
  public:
          naam_klasse();
         ~naam_klasse();
  // MEMBER FUNCTIONS
    char  *klassenaam(void) { return("naam_klasse"); }
    void  naam_invoer();
    void  naam_label(char *string);
};
```

C++ Voorbeeld 11.3 *(blad 1)*

```
class adres_klasse:virtual public basis_klasse
{ // DATA MEMBERS
  protected:
    char  straatnaam[80];
    int   huisnummer;
    char  postcode[10];
    char  plaatsnaam[20];
  // CONSTRUCTOR - DESTRUCTOR
  public:
          adres_klasse();
         ~adres_klasse();
  // MEMBER FUNCTIONS
    char  *klassenaam(void) { return("adress_klasse"); }
    void  adres_invoer();
    void  adres_label(char *string);
};

class persoon_klasse: virtual public naam_klasse,
  virtual public adres_klasse, virtual public datum_klasse
{ // DATA MEMBERS
  protected:
    int kerstkaartje;
    int vakantiekaartje;
  // CONSTRUCTOR - DESTRUCTOR
  public:
         persoon_klasse();
        ~persoon_klasse();
    char *klassenaam(void) { return("persoon_klasse"); }
    void persoon_invoer();
    void persoon_label(char *string);
};

/*-------------------------------------------------------------*/
/*   CLASS MEMBERFUNCTIONS                                     */
/*-------------------------------------------------------------*/
/*   basis klasse
*/
/*+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++*/
void basis_klasse::toon_klasse_naam(void)
{ cout << "naam van de class : " << klassenaam() << "\n";
}

/*+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++*/
/*   datum klasse                                              */
/*+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++*/
datum_klasse::datum_klasse()
{ jaar = 1900; maand = 1; dag = 1;
}

datum_klasse::~datum_klasse()
{ // lege destructor, mag ook weggelaten worden
}
```

C++ Voorbeeld 11.3 *(blad 2)*

```
void datum_klasse::datum_invoer()
{ int ok;
  do
  { ok=0;
    cout << "geef jaartal           : "; cin >> jaar;
    if (jaar > 0) ok=1;
  } while (!ok);
  do
  { ok=0;
    cout << "geef maand     [1..12] : "; cin >> maand;
    if( (maand > 0) && (maand <13)) ok=1;
  } while (!ok);
  do
  { ok=0;
    switch (maand)
    { case 1 : case 3 : case 5 : case 7 : case 8 : case 10:
      case 12: cout << "geef dag       [1..31] : "; cin >> dag;
               if ((dag > 0) && (dag < 32))   ok=1; break;
      case 4 : case 6 : case 9 :
      case 11: cout << "geef dag       [1..30] : "; cin >> dag;
               if ((dag > 0) && (dag < 31))   ok=1; break;
      case 2 : if (jaar%4==0)
               { cout << "geef dag       [1..29] : ";
                 cin >> dag;
                 if ((dag > 0) && (dag < 30)) ok=1;
               }
               else
               { cout << "geef dag       [1..28] : ";
                 cin >> dag;
                 if ((dag > 0) && (dag < 29)) ok=1;
               } break;
      default: break;
    }
  } while (!ok);
}

void datum_klasse::datum_label(char *string)
{ toon_klasse_naam();
  cout << string << " : " << dag << "-" << maand ;
  cout << "-" << jaar << "\n";
}

/*+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++*/
/*   naam klasse                                                 */
/*+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++*/
naam_klasse::naam_klasse()
{ strcpy(achternaam ,"geen");
  strcpy(voornaam   ,"geen");
  strcpy(voorvoegsel,"geen");
  strcpy(voorletters,"geen");
}

naam_klasse::~naam_klasse()
{
}
```

C++ Voorbeeld 11.3 *(blad 3)*

```
void naam_klasse::naam_invoer()
{ cout << "geef achternaam  : "; gets(achternaam);
  cout << "geef voornaam    : "; gets(voornaam);
  cout << "geef voorvoegsel : "; gets(voorvoegsel);
  if (!strlen(voorvoegsel)) strcpy(voorvoegsel,"---");
  cout << "geef voorletters : "; gets(voorletters);
}

void naam_klasse::naam_label(char *string)
{ toon_klasse_naam();
  printf("%s : ",string);
  if (strcmp(voorvoegsel,"---")==0)
    printf("%s %s\n",voorletters,achternaam);
  else
    printf("%s %s %s\n",voorletters,voorvoegsel,achternaam);
}

/*+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++*/
/*   adres klasse                                                */
/*+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++*/
adres_klasse::adres_klasse()
{ strcpy(straatnaam,"geen"); huisnummer=0;
  strcpy(postcode,"geen");   strcpy(plaatsnaam,"geen");
}

adres_klasse::~adres_klasse()
{
}

void adres_klasse::adres_invoer()
{ char tmp[80];
  printf("geef straatnaam  : "); gets(straatnaam);
  printf("geef huisnummer  : "); gets(tmp);
  huisnummer=atoi(tmp);
  printf("geef postcode    : "); gets(postcode);
  printf("geef plaatsnaam  : "); gets(plaatsnaam);
}

void adres_klasse::adres_label(char *string)
{ toon_klasse_naam();
  printf("%s : ",string);
  printf("%s %d %s %s\n",
         straatnaam,huisnummer,postcode,plaatsnaam);
}
/*+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++*/
/*   persoon klasse                                              */
/*+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++*/
persoon_klasse::persoon_klasse()
{ kerstkaartje = 0; vakantiekaartje = 0;
}

persoon_klasse::~persoon_klasse()
{
}
```

C++ Voorbeeld 11.3 *(blad 4)*

```
void persoon_klasse::persoon_invoer()
{ naam_invoer();
  adres_invoer();
  datum_invoer();
  printf("kerstkaartje    [0/1]   : "); cin >> kerstkaartje;
  printf("vakantiekaartje [0/1]   : "); cin >> vakantiekaartje;
}

void persoon_klasse::persoon_label(char *string)
{ toon_klasse_naam();
  printf("%s gegevens :\n",string);
  naam_label ("naam  ");
  adres_label("adres ");
  datum_label("datum ");
  cout << "kerst  : " << kerstkaartje << " vakantie : ";
  cout << vakantiekaartje;
  cout << "\n";
}

/*-------------------------------------------------------------*/
/*   MAIN PROGRAM                                              */
/*-------------------------------------------------------------*/
int main()
{ int                  i,n=1;
  class persoon_klasse persoon[100];          /* variabelen */
  clrscr();
  for (i=0; i < n; i++)                       /* invoer     */
  { clrscr();
    persoon[i].persoon_invoer();
  }
  clrscr();
  for (i=0; i < n; i++)                       /* uitvoer    */
  { printf("\n"); persoon[i].persoon_label("persoons");
  }
  return (0);
}
```

C++ Voorbeeld 11.3 *(laatste blad)*

De gebruikte klassen hebben allen een data-member klassenaam. Hoe krijg je nu de juiste naam te zien ? Door met het weergeven programma te spelen zal aan den lijve moeten worden ondervonden hoe deze polymorfie werkt.

Meer over ***C++*** kan gevonden worden in referentie 3 en vele andere publikaties. Met deze kleine introduktie is hopelijk duidelijk geworden dat ***C++*** meer mogelijkheden biedt tot een nog betere strukturering dan ***C*** al in zich heeft en dat daardoor ***C++*** in de toekomst één van de belangrijkste talen zal worden.

12 EXECUTABLE MAKEN

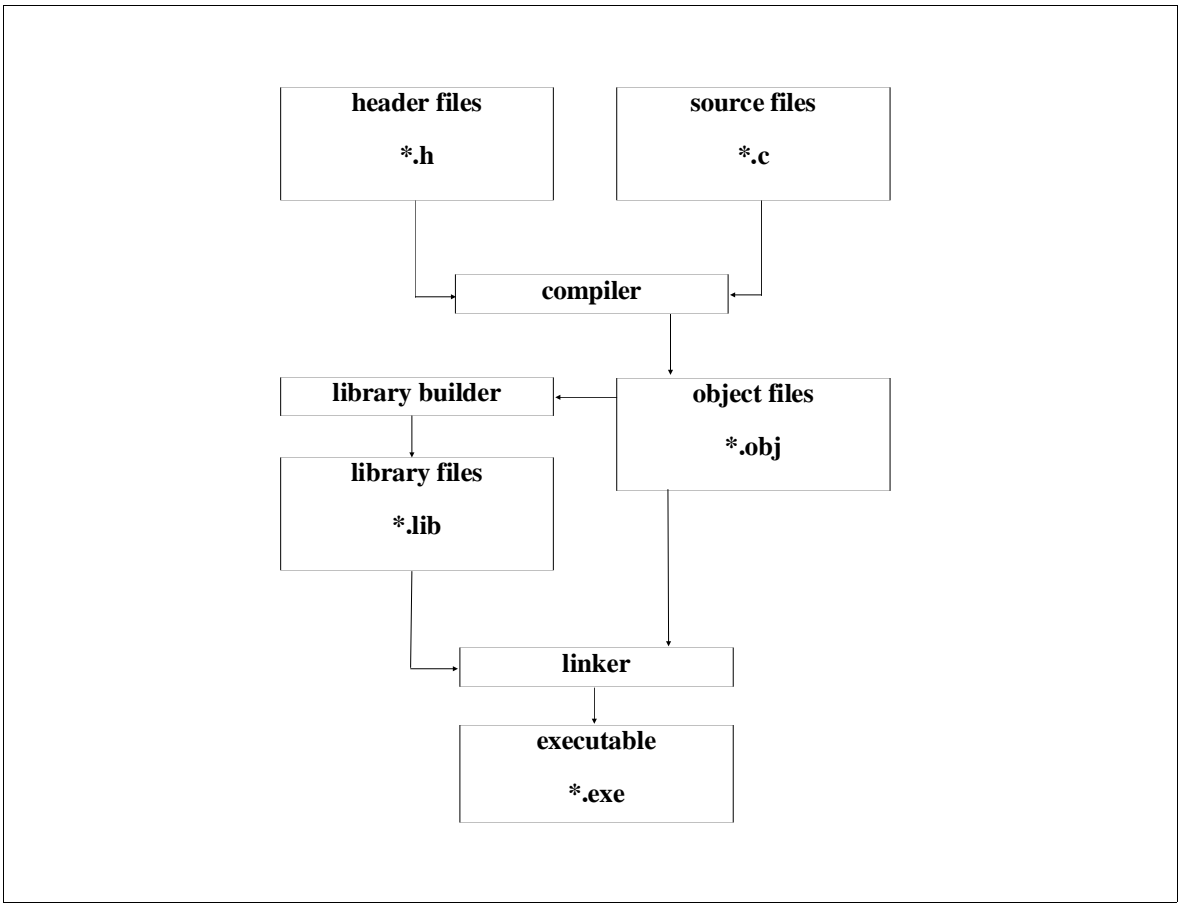
Grote programma's zullen bijna altijd bestaan uit meerdere files. Het is zelfs aan te raden dit ook voor kleine programma's te laten gelden want het komt de overzichtelijkheid ten goede. Uiteindelijk zal het programma gestart worden door de aanroep van één *executable file*. Hoe een dergelijke *executable* tot stand komt zal in dit hoofdstuk worden uitgelegd.

12.1 Splitsen van de bron code in diverse files

Uit het *file-overzicht* van in de hieronder weergegeven figuur blijkt dat er nog al wat verschillende files zijn die samen het uiteindelijke programma vormen :

***.h**	*header files*
***.c**	*source files*
***.obj**	*object files*
***.lib**	*library files*
***.exe**	*executable file*

In de volgende paragrafen zullen deze files aan de orde komen. Daarna zal uiteengezet worden hoe de *executable* uiteindelijk tot stand komt



figuur 12.1 : Executable onderdelen

12.1.1 Header files

De betekenis van *header-files* is al eerder aan de orde geweest. Naast de standaard *header-files* van ***C*** zelf kunnen er ook eigen *header-files* opgenomen worden in de broncode. De *header-files* worden met behulp van een *#include statement* opgenomen in de source-files. Dit statement kan er op twee manieren uitzien :

#include <header-file.h>
#include "header-file.h"

De eerste methode zoekt naar de *header-files* in de aan de compiler bekend gemaakte *include directories* (standaard directories). De tweede methode zoekt eerst naar het header-bestand in de *huidige directory* en zal daarna in de *standaard directories* gaan zoeken. Hieronder wordt een opsomming gegeven van standaard ***C*** header-files. Met de helpoptie van **TURBO-C** kan gekeken worden welke funkties in de onderstaande *header-files* zijn opgenomen.

tabel 12.1 : TURBO-C header files

header-file	omschrijving
ALLOC.H	dynamische geheugen allocatie funkties
ASSERT.H	macro voor de *assert()* funktie (error handling)
BIOS.H	ROM-BIOS funkties
CONIO.H	scherm funkties
CTYPE.H	karakter funkties
DIR.H	directory funkties
DOS.H	DOS interface funkties
ERRNO.H	error codes
FCNTL.H	constanten voor open() funktie
FLOAT.H	floating point implementaties
GRAPHICS.H	grafische funkties
IO.H	UNIX achtige I/O funkties
LIMITS.H	grenzen van implementaties
MATH.H	mathematische funkties
MEM.H	memory funkties
PROCESS.H	*exec()* funktie
SETJMP.H	nonlocal jumps
SHARE.H	file sharing
SIGNAL.H	signaal waarden
STDARG.H	variabele lengte van argumenten-lijst
STDDEF.H	veel gebruikte constanten
STDIO.H	stream IO-funkties
STDLIB.H	hulp declaraties
STRING.H	string funkties
TIME.H	tijd funkties
VALUES.H	machine afhankelijke constanten

12.1.2 Object files

Zoals uit figuur 12.1 blijkt zijn *object-files* het resultaat van *compilatie* van *header-files* en *source-files*. Nadat we de *compiler* opdracht hebben gegeven om ons ***C***-programma te compileren onstaan de *object-files*. De extensie van deze files is ***.obj**. Met deze object-files kunnen we verder nog niets.

12.1.3 Libraries

Naast onze eigen *source-files* bestaan er ook *standaard libraries* in ***C*** die we ook kunnen meelinken. In deze *libraries* staan allerlei handige funkties die we zouden kunnen gebruiken in ons programma. Aangezien deze funkties in een *library* staan zullen alleen die funkties in onze *executable* komen die we ook daadwerkelijk gebruiken. In principe kan er dus eindeloos gelinked worden, de uiteindelijke code wordt bepaald door wat we gebruiken. Het is dus verstandig om zoveel mogelijk (eigen) standaard funkties in een *libray* onder te brengen die door verschillende programma's kan worden gebruikt. Om een *library* samen te stellen uit *object-files* is een *library-builder* nodig. In **TURBO-C** is dat **TLIB** onder **LINUX** is dat de archiver **ar r**. (met optie r).

12.1.4 Hoofdprogramma

Om tot een werkzaam geheel te komen moeten we *object-files* en *libraries* linken tot *executables*. Linken houdt niets meer in dan dat de verschillende funkties en data in de source-files met elkaar *"kennis maken"*. De geheugenadressen van de data wordt met elkaar in verbinding gebracht door een apart programma : de *linker*. In **TURBO-C** is dit **TLINK**. Pas hierna kan het programma uitgevoerd worden.

12.2 Compilatie en Linken van objectcode

De compiler welke gebruikt wordt kan vaak niet alleen *object-code* genereren maar ook *linken*. Een dergelijke compiler is op UNIX vaak te vinden onder de naam **cc**. **TURBO-C** levert de **TCC** *commandline compiler* waarmee *source-files* en *libraries* tot *executable-files* gecompileerd kunnen worden (het gebruik van **TLINK** is dan overbodig).

Als alleen de compiler-naam wordt aangeroepen verschijnt bij de meeste compilers een overzicht van de opties die meegegeven kunnen worden. Deze opties worden ook wel *compiler-vlaggen* genoemd. Zo kan het *geheugenmodel*, *default directories voor libraries en include files* en *code optimalisaties* worden opgegeven. Om een simpel programma **test.c** te compileren met de **TCC** compiler moet de volgende handeling worden verricht :

TCC test.c (voor LINUX is dit : **gcc** test.c)

Nadat de *compiler* en *linker* hun werk hebben gedaan zal de file **test.exe** de *executable* bevatten. Als het programma uit meerdere onderdelen bestaat en er ook eigen *libraries* moeten worden aangemaakt onstaat een complex geheel van handelingen die beter met behulp van zogenaamde *makefiles* kunnen worden uitgevoerd.

12.3 Makefiles

Een *makefile* is een invoerfile voor het programma **MAKE**. Met **MAKE** kunnen automatisch alle compilaties verzorgd worden waarbij tevens gelet wordt op de afhankelijkheidsrelaties die tussen files bestaan. Indien een file in de keten veranderd wordt, zal **MAKE** ervoor zor-

gen dat alle van deze file afhankelijke *objects* of *libraries* opnieuw worden gecompileerd. In een *makefile* kunnen tevens netjes alle *compiler-vlaggen* worden meegegeven. In het volgende voorbeeld wordt een *makefile* getoond waarmee een *library* wordt gecreëerd en een *executable* wordt gemaakt. Dit systeem werkt ook zo onder LINUX (UNIX).

Stel we hebben de volgende *source-files* waarmee we een *library* **mylib.lib** willen maken :

lees.c
schrijf.c
plot.c
reken.c

In deze files bevinden zich allemaal *basisfunkties* die we ook bij andere programma's willen gebruiken, ze vormen een persoonlijke aanvulling op de ***C***-funktie bibliotheek.

Het hoofdprogramma en een eigen header bestand zijn bijvoorbeeld :

prog1.c
prog.h

De *makefile* voor dit probleem zou er als volgt uit kunnen zien :

```
# Makefile voor TURBOC
#
# (c) : J.W. Welleman
#-----------------------------------------directories---------
TURBODIR = C:\TC
EXEDEST  = C:\BIN
PROGDIR  = .
LIB      = $(TURBODIR)\LIB;$(TURBODIR);
INCL     = $(TURBODIR)\INCLUDE;$(PROGDIR)
#-----------------------------------------compiler and lib----
CC       = $(TURBOEXE)\TCC
TLIB     = $(TURBOEXE)\TLIB
CFLAG    = -A -f -ml -DTURBOC -I$(INCL) -L$(LIB)
OFLAG    = -A -c -f -ml -DTURBOC -I$(INCL) -L$(LIB)
#-----------------------------------------make exe------------
prog1: mylib.lib
  TAB   $(CC) $(CFLAG) -eprog1.exe prog1.c mylib.lib
<------>move prog1.exe $(EXEDEST)

mylib.lib: lees.obj schrijf.obj plot.obj reken.obj
       $(TLIB) mylib +lees +schrijf +plot +reken

lees.obj: lees.c
       $(CC) $(OFLAG) -olees.obj lees.c
schrijf.obj: schrijf.c
       $(CC) $(OFLAG) -oschrijf.obj schrijf.c
plot.obj: plot.c
       $(CC) $(OFLAG) -oplot.obj plot.c
reken.obj: reken.c
       $(CC) $(OFLAG) -oreken.obj reken.c
```

voorbeeld 12.2 : makefile

MAKE zal default als invoer zoeken naar de file *makefile* dus het vorige voorbeeld kan gestart worden door :

```
MAKE
```

Van belang is dat de *afhankelijkheden* tussen de files op een juiste manier worden aangegeven. PROG1.EXE hangt af van MYLIB.LIB welke afhangt van de diverse .OBJ files die ieder op hun beurt weer afhangen van de *.C *source-files*. Door onderscheid te maken tussen *compiler/linker* -vlag (CFLAG) en *compiler/object*-vlag (OFLAG) kan het com- pileren met TCC worden uitgevoerd en hoeft de losse LINKER niet gebruikt te worden.

Van de opties die hier zijn aangegeven zijn de volgende belangrijk :

-Dnaam	een externe *define* die wordt meegegeven aan de source
-Inaam	de directory voor de *header-files*
-Lnaam	de directory voor de *library-files*
-c	alleen compileren dus niet linken
-ml	large geheugen model
-f	floating point emulatie indien geen coprocessor aanwezig
-A	**ANSI-C**

De overige vlaggen kunnen worden nagelezen uit het overzicht van opties welke verkregen kan worden door TCC aan te roepen. Voor LINUX zijn de volgende opties van belang:

-Dnaam	een externe *define* die wordt meegegeven aan de source (vaak b.v. **-D**GNU)
-Inaam	de directory voor de *header-files*
-Lnaam	de directory voor de *library-files*
-c	alleen compileren dus niet linken
-pedantic	strikte ISO-C controle
-ansi	**ANSI-C**

Van belang is ook dat de *uitvoerende commando's* ($(CC), move, $(TLIB) ed.) voorafgegaan worden door slechts één **TAB** en niet door spaties. Dit is een kenmerkende eigenschap die de meeste make-programma's bezitten.

12.4 TURBOC programmeer-omgeving

Naast de hierboven weergegeven *commandline* methode voor het compileren van files heeft **TURBOC** ook een mooie *programmeer-omgeving* gemaakt met een eigen editor en een menu-gestuurde *compiler/linker/library-builder*. Deze kan gestart worden met het commando **TC**. Het gebruik van deze omgeving wijst zichzelf. De ingebouwde helpfunktie [F1] lost de meeste vragen op. De meeste gebruikers zullen daarom deze omgeving prefereren boven de handmatige aanpak zoals eerder behandeld. Toch vond ik het nodig om deze andere aanpak te laten zien omdat daardoor meer inzicht wordt verkregen over hoe een *executable* tot stand komt. Tevens zal op andere systemen vaak niet de fraaie werk-omgeving van **TURBO-C** te vinden zijn en dan is het makkelijk te weten hoe er omgegaan kan worden met *makefiles*.

De toekomst zal weliswaar steeds meer gericht zijn op het gebruik van fraaie programmeer-omgevingen zoals bijvoorbeeld **TURBOC for WINDOWS**. Wellicht gaat het met **UNIX** dezelfde kant op als **WINDOWS-NT** goed van de grond komt. De nieuwe richting zal dan ook gericht zijn op ***C++*** in combinatie met **WINDOWS**.

13 VOORBEELD VAN EEN PROGRAMMA

In dit laatste hoofdstuk zal een voorbeeld worden behandeld waarin zoveel mogelijk elementen uit de cursus aan de orde komen. Het programma dat uiteindelijk moet worden geschreven moet een strakke opbouw hebben en een heldere datastruktuur bezitten. In dit hoofdstuk zal eerst het op te lossen probleem worden geintroduceerd waarna in het kort iets gezegd zal worden over de oplostechniek. Daarna zal de datastruktuur van het programma en de opbouw van het programma aan de orde komen. Tot slot wordt een complete *source-listing* aan het einde van het hoofdstuk gegeven. Bij de beschrijving van de onderdelen kan zodoende meegekeken worden hoe de daadwerkelijke implementatie van het een en ander eruit ziet.

13.1 Probleem omschrijving

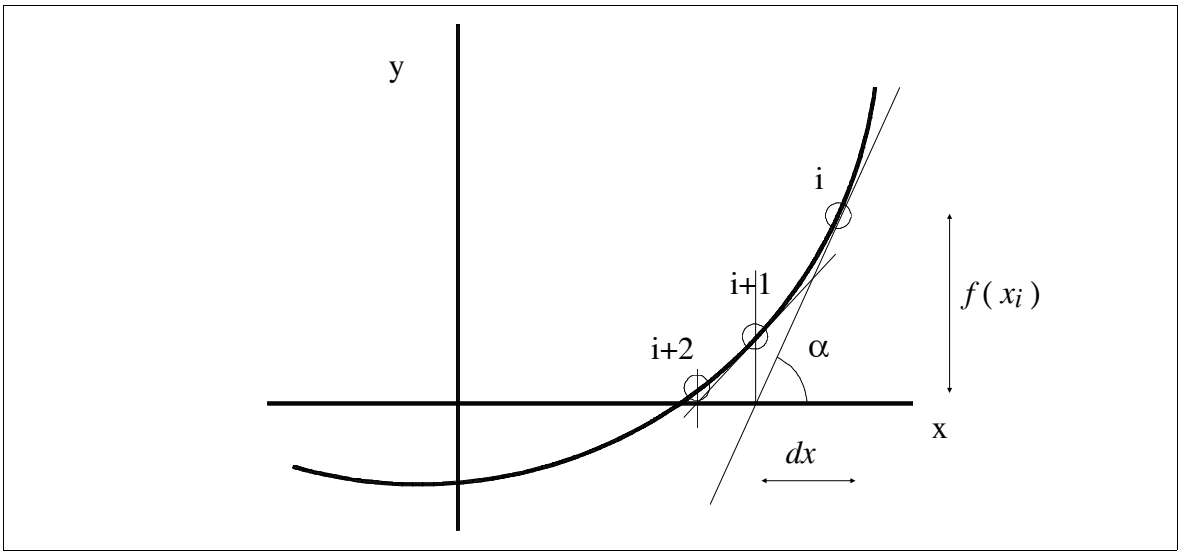
Het op te lossen probleem in dit voorbeeld is om van een gegeven polynoom de nulpunten te bepalen. De op te lossen vergelijking luidt als volgt :

$$a_n x^n + a_{n-1} x^{n-1} + \ldots + a_1 x + a_0 = 0$$

Het bepalen van de nulpunten zal met een numerieke oplossingstrategie worden uitgevoerd. De methode daarvoor staat bekent als de Newton-Raphson methode. De invoer van het polynoom zal met behulp van een invoer-file plaatsvinden.

13.2 Oplostechniek

Het zoeken naar de nulpunten van een bepaalde funktie kan met een iteratieve methode worden uitgevoerd. In de onderstaande figuur is een funktie getekend met een startpunt [*x, f(x)*] van waaruit het zoeken naar het nulpunt wordt begonnen. De raaklijn aan de funktie in het startpunt heeft een helling waarvan de richtingscoëfficient gelijk is aan de afgeleide van de funktie in het raakpunt.



Figuur 13.1 : Newton - Raphson methode

Geometrisch geldt voor de afstand *dx* (zie figuur) :

$$\tan\alpha = f'(x_i) = \frac{dx}{f(x_i)} \quad \text{dus} \quad dx = \frac{f(x_i)}{f'(x_i)}$$

Het punt waarin de raaklijn aan de funktie de x-as snijdt nemen we nu als nieuw startpunt :

$$x_j = x_i - \frac{f(x_i)}{f'(x_i)}$$

Daarmee is het algoritme voor het vinden van de nulpunten bepaald.

13.3 Benodigde datastruktuur

De globale data welke nodig is in het programma zal bestaan uit data voor het opbergen van de polynoom-data en de benodigde filepointers voor het communiceren met files.

De data voor het polynoom kan per macht worden opgeborgen. Hierdoor ontstaat een rij van data (macht *n* en constante a_n) waardoor de polynoom data opgeborgen kan worden in een *array van struct* :

```
struct polynoom
{ int n;
  double a;
};
```

Figuur 13.2 : Polynoom struct

De lengte van het uiteindelijke array is gelijk aan de graad van het polynoom + 1. Hiervoor kan een globale integer variabele *ngraad* worden gedeclareerd. Het array dat we zodoende nodig hebben kan statisch of dynamisch worden gealloceerd. Als we het array statisch willen alloceren met bijvoorbeeld als maximum graad 99 dan declareren we een array data van srtruct polynoom :

```
struct polynoom data[100];
```

De overige globale data blijft beperkt tot een filepointer voor de invoerfile.

13.4 Programmastruktuur

De programmastruktuur zal heel eenvoudig worden gehouden. Feitelijk bestaat het programma uit zes afzonderlijke handelingen welke verricht moeten worden :

1 invoer-filenaam doorgeven aan het programma
2 lees de invoer vanuit de aangegeven file
3 geef het gelezen polynoom weer op het beeldscherm
4 vraag om een interval waarbinnen de nulpunten moeten worden gezocht
5 zoek naar de nulpunten binnen dit interval
6 geef een tabel weer met de gevonden nulpunten

Deze zes onderdelen zullen zoveel mogelijk in aparte funkties worden ondergebracht. In de hierna volgende paragrafen worden deze zes punten nader belicht. De feitelijke uitwerking ervan kan worden gevonden in de *source-listing* welke aan het einde van dit hoofdstuk is opgenomen.

13.4.1 Invoer van de filenaam

Het bekendmaken van de naam van een invoerfile aan een ***C***-programma kan eenvoudig uitgevoerd worden door op de *commandline* bij het starten van de *executable* een *argument* (filenaam) mee te geven. We willen bijvoorbeeld het programma starten met de in- voerfile **invoer.dat**. Dit kan dan door op de *commandline* het volgende commando te geven :

```
C:> nulpunt invoer.dat <ENTER>
```

De eerste handeling in het programma zal nu moeten zijn het controleren van het aantal argumenten (2 stuks) en de controle of de aangegeven file bestaat en kan worden geopend om uit te lezen. Zoals aangegeven in hoofdstuk 6 paragraaf 5 kunnen *command-line argumenten* worden doorgegeven aan de hoofdfunktie door middel van de *formele data* van *main()*. Het hoofdprogramma zal er dan dus als volgt uitzien :

```
/* programma nulpunt.c
*
*  Een programma voor het bepalen van de nulpunten van
*  een polynoom van de graad n
*
*  (c) 1993 : J.W. Welleman
*
*************************************************************/
#include stdio.h
#include math.h
#include string.h

struct polynoom
{ int n;
  double a;
};

FILE finp;

main(argc, argv) int arg; char *argv[];
{ if (argc != 2) { printf("root <filenaam> verwacht>\n");
                   exit(1);}
  if ((finp = fopen(argv[1],"r"))==NULL)
  { printf("fout bij het lezen van %s\n",argv[1]);
    exit(2);
  }
}
```

Voorbeeld 13.3 : Basisprogramma NULPUNT

In de uiteindelijke versie is dit blok van handelingen ondergebracht in een funktie *OpenFile()*. Aan deze funktie worden de formele parameters van *main()* doorgegeven.

13.4.2 Lezen van de invoerfile

In de invoerfile staat de data waarmee het polynoom-voorschrift kan worden opgesteld. Het betreft hier de coëfficienten en de graad van het polynoom. Door een bepaalde *syntax* aan te houden in de invoerfile kan het programma controleren of de invoer correct is. Als syntax wordt de volgende invoer aangehouden :

```
$ regels beginnend met een dollar zijn commentaar
# zo ook voor regels met een #
; of een ;
POLYNOOM [graad]
-+- [constante a] -+
 +--------<--------+
    graad+1 maal
```

Figuur 13.4 : Invoer syntax

Op deze manier kan getest worden of het sleutelwoord POLYNOOM voorkomt in de invoerfile. Op de plaats tussen de rechte haken dient een getal te worden geplaatst (De haken uiteraard niet opnemen in de invoerfile).

Het lezen van de invoerfile is ondergebracht in de funktie *LeesFile()*. Deze funktie maakt gebruik van de reeds eerder geopende file en leest vervolgens *strings* uit deze file met behulp van de zelf ontworpen leesroutine *StringFromFile()*. Door gebruik te maken van één leesroutine kan ook het overslaan van commentaar geregeld worden. Zo worden *integers* en *doubles* ook eerst gelezen als *string* en daarna geconverteerd naar *int* en *double*. Bekijk daartoe de funkties *read_in()* en *read_do()*. Let vooral op het doorgeven van de data bij deze funkties. Aangezien de ingelezen waarden veranderen wordt hier gebruik gemaakt van een *call by reference*.

Een aardigheidje bij het inlezen van de polynoom-data is dat deze data zowel statisch als dynamisch kan worden opgeborgen. In dit programma is de dynamische methode gevolgd en is de statische optie tussen commentaar gezet.

13.4.3 Weergeven van het gelezen polynoom

De funktie *PrintPolynoom()* geeft het ingelezen polynoom weer. De data welke doorgegeven wordt aan deze funktie is de *filepointer*, de *polynoomdata* en de *graad* van het polynoom. Aangezien de data van het polynoom als een array is opgeborgen wordt hier gewerkt met een pointer. Alleen het *begin-adres* van het array hoeft te worden doorgegeven.

13.4.4 Interval

De volgende stap in het hoofdprogramma is het interactief vragen om de intervalgrenzen waarbinnen de nulpunten moeten worden gezocht. Hier wordt gebruik gemaakt van de in eerdere hoofdstukken gebruikte *gets()* funktie voor het lezen van een *string* vanaf het toetsenbord. De *string* wordt vervolgens geconverteerd naar het *type double* (xmin en xmax).

13.4.5 Nulpunten bepaling

De belangrijkste funktie van het gehele programma is uiteraard de funktie *Newton_Raphson()* (afgekort NR funktie) waarmee de nulpunten worden bepaald. Aangezien dit een iteratieve methode is, is er gebruik gemaakt van een *do ~ while* konstruktie.

Een probleem van de NR-methode is dat steeds hetzelfde nulpunt kan worden gevonden. Dit kan opgelost worden door te zoeken naar een *sub-interval* waar tekenwisseling in de funktie optreedt. Op dit sub-interval wordt dan vervolgens het nulpunt bepaald waarna dit wordt opgeborgen in de *polynoom struct data*. Hierna wordt gezocht naar een volgend sub-interval met een tekenwisseling. Deze gang van zaken is terug te vinden in de NR-funktie.

Ook de NR-funktie krijgt het *begin-adres* van het *polynoom-array* mee. Als *retunrwaarde* geeft de NR-funktie het aantal gevonden nulpunten terug zodat bij het afdrukken van de nulpunten het aantal gevonden nulpunten bekend is.

Het algoritme dat het nulpunt zoekt heeft de funktiewaarde en de waarde van de afgeleide nodig die behoort bij de x-waarde van het steunpunt. Hiervoor zijn twee funkties geschreven die deze waarden bepalen :

double val(x,p,n);	funktie waarde
double deriv(x,p,n);	waarde van de afgeleide

De funkties geven de waarden terug als *doubles*. Het polynoom wordt weer doorgegeven door het *begin-adres* mee te geven aan *val()* en *deriv()*. Ook de graad van het polynoom wordt als *formele data* aan de beide funkties meegegeven.

13.4.6 Weergeven van de gevonden nulpunten

De laatste handeling in het programma is het weergeven van een staatje met de nulpunten en de funktie-waarden in deze nulpunten, hopelijk leveren deze allen 0.0 op. Dit stukje code is als een *sub-programma* (ook een funktie in ***C***) ondergebracht in *PrintNulpunten()*. Deze funktie krijgt de *filepointer*, het aantal nulpunten, het beginadres van het polynoom-array en de graad van het polynoom als formele data mee. Aangezien het aantal gevonden nulpunten het resultaat is van de funktie *Newton_Raphson()* zijn deze twee funkties in elkaar geschoven zoals te zien is in het hoofdprogramma.

Als *filepointer* is hier gekozen voor het *standaard output device stdout* (*beeldscherm*). Hier had ook een *"echte"* filepointer mogen staan of een *stream* naar de printer (*stdprn*). De funktie *PrintNulpunten()* is op deze manier dus zeer flexibel qua uitvoer-bestemming.

13.5 Source - Listing

Hieronder is de complete *source* weergegeven van het programma NULPUNT. De struktuur van het programma en de beschrijving van de funkties zijn in de voorgaande paragrafen terug te vinden.

```
/*
* programma nulpunt.c
*
* (c) 1993 : J.W. Welleman
*
*************************************************************/
/*  I N L U D E S,  D E F I N E S  en  G L O B A L E  D A T A  */
/*************************************************************/
#include  stdio.h
#include   math.h
#include string.h
#include  ctype.h

#define MAXCHAR 300                 /* max char-array lengte    */
#define NGRAAD   99                 /* indien statische arrays  */
#define NEAR_ZERO 1.0e-10           /* nauwkeurigheids conditie */
#define ALLOC(y,x,n) \
          if ((y)==NULL) \
          { (y)=(x *)calloc((unsigned int)(n),sizeof(x));\
            if ((y) == NULL) error("ALLOC error",99);\
          } \
          else error("ALLOC : pointer != NULL",100)

typedef struct polynoom             /* struc voor polynoom      */
{ int    n;                         /* graad                    */
  double a,root;                    /* constante                */
};

FILE            *finp;
int            ngraad;
struct polynoom *data;
/*+++++++ INDIEN dynamische array allocatie ++++++++++++++++++++*/
struct polynoom *data;
/*+++++++ INDIEN statisch   array ++++++++++++++++++++++++++++++*/
/* struct polynoom data[NGRAAD+1];                               */
/*+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++*/
```

Figuur 13.5 : Broncode

```
/*************************************************************************/
/*                            F U N K T I E S                            */
/*************************************************************************/
void error(mes,errno) char *mes; int errno;
{ printf("%s\n",mes); exit(errno);
}

/*-----------------------------------------------------------------------*/
/*   ALGEMENE LEES FUNKTIES                                              */
/*-----------------------------------------------------------------------*/
void strtolower(string)             /* stringconversie naar kleine letters */
  char *string;
{ int i;
  for (i=0; i strlen(string); i++)
    if ((isalpha(string[i])) && (isupper(string[i])))
      string[i] = (char) (((int) string[i]) + 32) ;
}

void strtoupper(string)             /* stringconversie naar hoofdletters  */
  char *string;
{ int i;
  for (i=0; i strlen(string); i++)
    if ((isalpha(string[i])) && (islower(string[i])))
      string[i] = (char) (((int) string[i]) - 32) ;
}

void StringFromFile(fp,string) FILE *fp; char *string;  /* lees een string */
{ int sw=0; char s[MAXCHAR];
  if (fp)
  { while ((sw==0) && (!feof(fp)))
    { if ((fscanf(fp,"%s",s)!=0) && (!feof(fp)))
      { if ((strchr(s,'$') != NULL) ||                  /* regels die   */
            (strchr(s,'#') != NULL) ||                  /* beginnen met */
            (strchr(s,';') != NULL) ||                  /* $ # ; of "   */
            (strchr(s,'"') != NULL) ) fgets(s,MAXCHAR,fp); /* overslaan    */
        else sw= 1;
      }
      else sw= -1;
    }
    if (sw == 1)
    { strcpy(string,s);
    }
  }
}

read_in(fp,var) FILE *fp; int *var;                   /* lees een integer */
{ char string[MAXCHAR];
  StringFromFile(fp,string); *var = atoi(string);
}

read_do(fp,var) FILE *fp; double *var;                /* lees een double  */
{ char string[MAXCHAR];
  StringFromFile(fp,string); *var = atof(string);
}
```

```
/*-------------------------------------------------------------------------*/
/*   PROGRAMMA FUNKTIES                                                    */
/*-------------------------------------------------------------------------*/
void OpenFile(argc, argv) int argc; char *argv[];
{ if (argc!=2)                          error("fout : root filenaam" ,1);
  if ((finp=fopen(argv[1],"r"))==NULL) error("file kan niet geopend woren",2);
}

double intpow(x,n) double x; int n;        /* x^n waarbij n een integer is */
{ int i; double answer = x;
  if (n == 0) answer = 1.0 ;
  for (i=0; i n-1; i++) answer = answer * x;
  return (answer);
}

void LeesFile()
/*-------------------------------------------------------------------------*/
/*   lees de invoer file met daarin de volgende SYNTAX :                   */
/*                                                                         */
/*   POLYNOOM [graad]                                                      */
/*                                                                         */
/*   -+- [constante a] -+-                                                 */
/*    +--------<--------+-                                                 */
/*        graad+1 maal                                                     */
/*-------------------------------------------------------------------------*/
{ int i=0; char string[100];
  StringFromFile(finp,string); strtoupper(string);
  if (strcmp(string,"POLYNOOM")!=0) error("syntax fout POLYNOOM [graad]",4);
  else
  { read_in(finp,&ngraad);                              /* globale data */
    /*+++++++ INDIEN dynamische array allocatie ++++++++++++++++++++++++++*/
      data = NULL; ALLOC(data,struct polynoom,ngraad+1);   /* globale data */
    /*+++++++ INDIEN statisch   array ++++++++++++++++++++++++++++++++++++*/
    /* if (ngraad  NGRAAD) error("graad van polynoom te groot");           */
    /*+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++*/
    do
    { data[i].a = 0.0; data[i].n = ngraad - i; data[i].root = 0.0;
      read_do(finp,&data[i].a);
      i++;
    }
    while ((i  ngraad+1) && (!feof(finp)));
  }
}
```

```
void PrintPolynoom(fp,p,n) FILE *fp; struct polynoom *p; int n;
/*-------------------------------------------------------------------------*/
/*  aanroep van funktie : geef beginadres van array door = pointers !!!    */
/*  flexibel gebruik van fprintf() kan dus met fp=stdout naar beeldscherm  */
/*-------------------------------------------------------------------------*/
{ int i; struct polynoom *pp;                                 /* lokale data */
  if (fp)
  { fprintf(fp,"polynoom van de graad : %d\n\n",n-1);
    for (i=0; i ; i++)
    { pp = &p[i];
      if (i)
      { if (pp->a < 0.0) fprintf(fp,"-%lf",fabs(pp->a));
        else             fprintf(fp,"+%lf",fabs(pp->a));
      }
      else fprintf(fp,"%lf",pp->a);
      if (pp->n)
      { fprintf(fp,"*x");
        if (pp->n != 1) fprintf(fp,"^%d",pp->n);
      }
      if (i==n-1) fprintf(fp," = 0.0\n\n");
    }
  }
}

double val(x,p,n) double x; struct polynoom *p; int n;
/*-------------------------------------------------------------------------*/
/* polynoom funktie-waarde in het punt (x)                                 */
/*-------------------------------------------------------------------------*/
{ int i; struct polynoom *pp; double uitkomst=0.0;
  for (i=0; i ; i++)
  { pp = &p[i];
    if (pp->n!=0) uitkomst += pp->a * intpow(x,pp->n);
    else          uitkomst += pp->a;
  }
  return (uitkomst);
}

double deriv(x,p,n) double x; struct polynoom *p; int n;
/*-------------------------------------------------------------------------*/
/* polynoom afgeleide-waarde in het punt (x)                               */
/*-------------------------------------------------------------------------*/
{ int i; struct polynoom *pp; double uitkomst=0.0;
  for (i=0; i ; i++)
  { pp = &p[i];
    if (pp->n != 0)
      uitkomst += pp->a * pp->n * intpow(x,(pp->n - 1));
  }
  return (uitkomst);
}
```

```
int NewtonRaphson(xmin,xmax,p,n) double xmin,xmax; struct polynoom *p; int n;
/*-------------------------------------------------------------------------*/
/* Newton-Raphson procedure : xj = xi - f(x)/f'(x)                         */
/*-------------------------------------------------------------------------*/
{ int    ngevonden=0,iiter,i=0,ok;
  double x=xmin, dx = 0.01 * (xmax - xmin), fx, fdx;
  struct polynoom *pp;
  do
  { fx = val(x,p,n); fdx = val(x+dx,p,n);
    if ( ((fx <= 0.0) && (fdx >= 0.0)) ||     /* indien tekenwisseling dan */
         ((fx >= 0.0) && (fdx <= 0.0))        /* zoeken naar nulpunt       */
       )
    { iiter = 0;
      do
      { if (fabs(deriv(x,p,n)) <= NEAR_ZERO)
          x = x - val(x,p,n)/deriv((1.1*x),p,n);
        else x = x - val(x,p,n)/deriv(x,p,n);
        iiter++;
      }
      while ( (fabs(val(x,p,n)) > NEAR_ZERO) && (iiter  100) );
      if (fabs(val(x,p,n)) <= NEAR_ZERO)      /* opbergen van x in data[i] */
      { pp = &p[ngevonden]; pp->root = x; ngevonden++;
      }
      i++;
    }
    x += dx;
  } while ((i < n) && (x < xmax));
  return (ngevonden);
}

void PrintNulpunten(fp,p,ngevonden,n)
  FILE *fp; struct polynoom *p; int ngevonden,n;
/*-------------------------------------------------------------------------*/
/*  flexibel gebruik van fprintf() kan dus met fp=stdout naar beeldscherm  */
/*-------------------------------------------------------------------------*/
{ int i; struct polynoom *pp;
  if (fp)
  { fprintf(fp,"\nnulpunten van het polynoom :\n\n");
    for (i=0; i gevonden; i++)
    { pp = &p[i];
      fprintf(fp,"nr %4d x-value = %8.3lf f(x) = %8.3lf\n",i+1,
      pp->root,val(pp->root,p,n));
    }
    fprintf(fp,"\n");
 }
}

/*************************************************************************/
/*                            M A I N  P R O G R A M                     */
/*************************************************************************/
main(argc,argv) int argc; char *argv[];
{ int i; double xmin=0.0, xmax=0.0; char s[100];
  OpenFile(argc,argv);
  LeesFile();
  PrintPolynoom(stdout,data,ngraad+1);
  printf("give minimum x : "); gets(s); xmin = atof(s);
  printf("give maximum x : "); gets(s); xmax = atof(s);
  PrintNulpunten(stdout,data,NewtonRaphson(xmin,xmax,data,ngraad+1),ngraad+1);
  printf("klaar\n");
}
```

REFERENTIES

Herbert Schildt, *"TURBO C The Complete Reference"*, Osborne McGraw-Hill, Berkeley, California 1988, ISBN 0-07-881346-8, pp 893

Leen Ameraal, *"TURBO C",* Academic Service, Schoonhoven, The Netherlands 1988, ISBN 90-6233-393-1, pp 217

H.G. Schumann, *"Programmeren in TURBO C++ 3.0"*, Addison - Wesley", Amsterdam, The Netherlands 1992, ISBN 90-6789-351-X, pp 577

Wiliam Roetzheim, *"C++ Programmeren in WINDOWS 3.1"*, A.W. Bruna Informatica, Utrecht, The Netherlands 1992, ISBN 90-229-3807-7, pp 407

Borland International Corparation, *"TURBO C++ for WINDOWS version 3.0"*, Programmers Guide, User's Guide, and Resource Workshop, 1992

INDEX