Cursus

C

door

Geert Vernaeve

voor

Zeus WPI - Werkgroep Informatica (Universiteit Gent)

9 november 2000

Inleiding

Woord vooraf

. . . a double pair of Joo Janta 200 Super-Chromatic Peril Sensitive Sunglasses, which had been specially designed to help people develop a relaxed attitude to danger.
—Douglas Adams, "The Hitch Hiker's Guide to the Galaxy"

Deze cursus probeert niet-programmeurs een beetje wegwijs te maken in de taal C, maar heeft toch (hopelijk) voldoende extra materiaal aan boord om het ook voor wie al wat kan programmeren (niet noodzakelijk in C) interessant te houden. Dat extra materiaal zal ik telkens aanduiden met het

symbool. Paragrafen die beginnen met zo'n "**gevaarlijke bocht**"-bord kunnen gerust overgeslagen worden door beginners, want dikwijls verwijs ik daarin naar verdere hoofdstukken. Niets belet avontuurlijke lezers al bij de eerste keer een glimp op te vangen van wat komen gaat . . . Als je de cursus een tweede maal doorwerkt, kan je al eens wat meer op de "gevaarlijke bocht"-paragrafen letten om je kennis uit te diepen.

Hier en daar zul je een ▸**OEFENING** tegenkomen. Je zult sneller bijleren (en langer onthouden wat je geleerd hebt) als je de opgaven zelf probeert uit te werken. Daarna kan je achteraan controleren of je antwoord juist was (of kijken wat je had moeten vinden, als je weinig inspiratie had).

Je kan je vingers sparen en de programma's uit deze cursus downloaden (zie §D.2 voor de juiste adressen). Telkens als er zo'n stukje programma komt, heb ik dat met het symbool aangeduid, samen met de `bestandsnaam` waarin je het kan vinden.

Doelpubliek

Deze tekst zou geschikt moeten zijn voor niet-programmeurs als cursus onder begeleiding van een meer ervaren programmeur, en ook als zelfstudiemateriaal voor zij die al enige basiskennis van programmeren (niet noodzakelijk in C) hebben.

In principe kan je na de eerste twee hoofdstukken (waarin de allernoodzakelijkste basis van C wordt uitgelegd) meteen verder springen naar een ander hoofdstuk (bijvoorbeeld hoofdstuk 5 als je snel met bestanden wil werken of hoofdstuk 10 als je een nukkig programma wil debuggen)—en dan natuurlijk doen alsof je neus bloedt als het woord "pointer" of "array" valt.

In hoofdstuk 3 lees je alles over arrays, pointers, strings en hun onderlinge verbanden, die cruciaal zijn in C. Samen met hoofdstuk 2 vormt dit een stap-voor-stap uitleg van de taal C.

Hoofdstuk 4 gaat al meer over in naslag-stijl en behandelt de volledige syntax van C.
Alle daarna komende hoofdstukken kunnen in willekeurige volgorde afgehandeld worden.

Met dank aan ...

'Very deep,' said Arthur, 'you should send that in to the Reader's Digest*. They've got a page for people like you.'*
—Douglas Adams, "The Hitch Hiker's Guide to the Galaxy"

Deze cursus zou nooit afgeraakt zijn (laat staan begonnen) zonder het aanhoudende aandring van Roeland Mertens, waarvoor ik hem dan ook zeer erkentelijk ben, en zou veel minder goed geweest zijn zonder de opbouwende commentaar van Olivier Biot (die ook voor het nodige LaTeX-truukwerk zorgde), Ivo De Decker, Rudy Gevaert, Ilse Lanszweert, Annick Lootens, Filip Rooms, Nick Sabbe, Sven Van den Steene, John Van der Veen, Gregory Van Vooren, Dion Verbeke, Hans Vernaeve en Wim Woittiez. Tenslotte nog dank aan de Zeus Werkgroep Informatica aan de Universiteit Gent voor de motivatie en het voeren van reclamecampagnes voor de cursus.

De ASCII-tabel in Bijlage A is van de hand van Victor Eijkhout, die zijn vriendelijke toestemming gaf voor het overnemen ervan.

Legalia

(...) makes any claim, warranty, or representation, express or implied, that the products described in this manual are designed, intended, or authorized for use as components in systems intended for surgical implant in the body (...)
—MOTOROLA/IBM, "PowerPC™ Microprocessor Family: The Progammer's Reference Guide"

Eerste uitgave: 15 december 1997 (174 pp.)
Tweede uitgave: 2 november 1998 (215 pp.)
Derde uitgave: 8 november 2000 (270 pp.)

Hoofdstuk 1

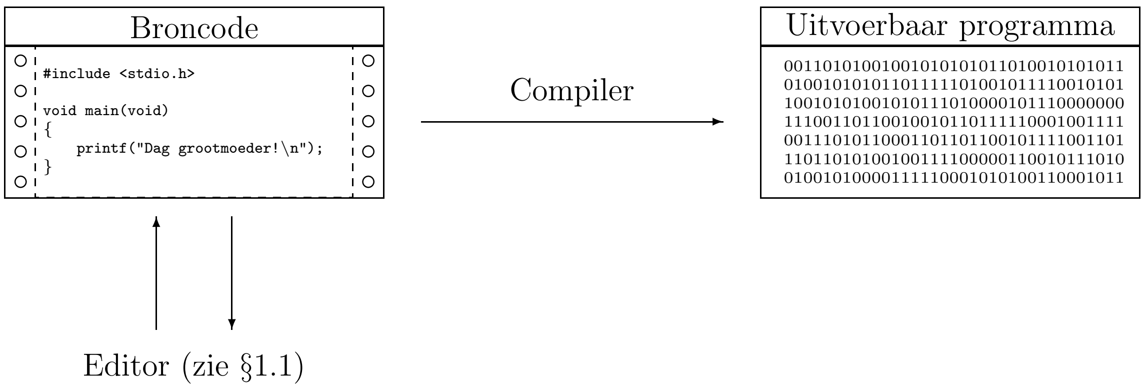
De ontwikkelomgeving

Onder **ontwikkelomgeving** verstaan we de programma's die nodig zijn om een C **listing** "aan de praat" te krijgen.

Een C programma wordt namelijk als een gewoon tekstbestand gemaakt (dus net zoals je een brief aan je grootmoeder zou schrijven). In die vorm kan het programma niet gedraaid worden op een computer; probeer maar eens de brief aan je oma op te starten. Computers begrijpen immers alleen maar nulletjes-en-eentjes-salade: **machinetaal**. Het tekstbestand waar het C programma in zit, wordt ook wel de **listing**, **broncode**, **source code** of kortweg **code** of **source** genoemd. Vandaar worden programmeurs ook wel *coders* genoemd.

Om het eenvoudig uit te drukken: 00011010 zou kunnen betekenen: "tel twee getallen bij elkaar op", niet direkt een aangename manier van programmeren. Tot overmaat van ramp gebruiken niet alle processors dezelfde codes, dus 00011010 zou op een andere processor wel eens kunnen betekenen: "stop het programma". Naast onhandig is machinetaal dus ook niet **overdraagbaar** (**portabel**): een programma dat draait op een systeem draait niet noodzakelijk op een ander.

Gelukkig hoef je de taal van de microprocessor niet te begrijpen. Het tekstbestand dat je geschreven hebt, kan immers volautomatisch naar nulletjes en eentjes vertaald worden door een programma: de **compiler**. Het uiteindelijke resultaat van een compilatie wordt een **uitvoerbaar programma** (**executable** of **binary**) genoemd.



In deze context kom je af en toe de term **IDE** (**Integrated Development Environment**) tegen, wat aanduidt dat compiler en een aantal hulpprogramma's (editor, debugger, linker en wat weet ik niet nog allemaal) sterk met elkaar vervlochten (geïntegreerd) zijn; de compiler is bijvoorbeeld aan te roepen in een menu van de editor. In deze cursus ga ik echter van een eenvoudiger omgeving uit waarin editor en compiler eerder los van elkaar staan.

1.1 De editor

In principe zou je de listing met een tekstverwerker kunnen maken, maar dat is in feite schieten met een kanon op een mug. Je hebt als programmeur al die toeters en bellen (spellingscontrole, lettertypes, ...) niet echt nodig. De C compiler lust trouwens alleen maar "platte tekst", d.i. tekst zonder lettertype-aanduidingen en dergelijke. Er zijn programma's die specifiek gericht zijn op het bewerken van tekstbestanden (en die zijn dan ook een stuk kleiner dan een tekstverwerker): **editor**s.

Als editor zal ik in de voorbeelden `pico` gebruiken; de meeste andere editors werken op een gelijkaardige manier.

Pico is op te starten door in de **shell** (het ding dat iets als `$_` of `C:\>_` zegt en waar je al je hartewensen in de vorm van commando's aan kwijt kunt) het commando `pico` *bestandsnaam* te tikken. Pico heeft echter de vervelende eigenschap regels die breder zijn dan het scherm in tweeën te splitsen; dat kan je verhinderen door `pico -w` *bestandsnaam* te tikken. Eventueel kan je je shell zo instellen (door wat met `alias` te prutsen) dat de `-w` automatisch toegevoegd wordt. (`vi`-gebruikers hebben dáár althans geen last van :-))

1.2 De compiler

Op de meeste UNIX systemen heet de standaard C compiler `cc`; in navolging hiervan noemen de verschillende compilerbakkers hun compilers bijvoorbeeld `gcc` (de GNU C compiler), `bcc` (Borland), enz. ...

Ik zal in de voorbeelden altijd de GNU C compiler gebruiken, omdat die gratis verkrijgbaar is voor bijna alle soorten computers en besturingssystemen en bovendien bijzonder efficiënte code genereert. In §D.2 staat meer over waar je deze compiler kan vinden.

Meer over compileren vind je in §2.1.4.

1.3 De shell

Het is altijd handig om een paar commando's te kennen om bestanden en dergelijke te manipuleren. Omdat dit geen cursus UNIX of MS-DOS is, een heel beknopt overzichtje:

1. `ls` toont alle bestanden in de huidige directory (is te vergelijken met de `dir` van MS-DOS). Meer info krijg je met `ls -l`. Je kan ook één of enkele bestanden bekijken; zo geeft `ls *.c` alle bestanden waarvan de naam op `.c` eindigt.

2. `cd` *dirnaam* "gaat in" een bepaalde directory. Om terug te keren, kan je `cd ..` gebruiken.

3. `cp` *origineel kopie* maakt een nieuw bestand dat *kopie* heet en schrijft daarin dezelfde inhoud als het bestand *origineel*; het resultaat is twee identieke bestanden. (In MS-DOS is dat `copy`.)

4. `rm` *bestandsnaam* verwijdert een bestand. Meerdere bestanden ineens verwijderen kan met bv. `rm *.o` (verwijdert alle bestanden die op `.o` eindigen). Zeer gevaarlijk is natuurlijk `rm *` (in het bijzonder omdat de meeste UNIX-smaken niet vragen of je wel echt zeker bent, in tegenstelling tot bv. het MS-DOS commando `del` ...)

5. `mv` *oudenaam nieuwenaam* geeft een bestand of een directory een nieuwe naam. (Voor de DOSsers: `ren`.) Je kan ook een bestand naar een andere directory verplaatsen met `mv`.

6. `mkdir` *dirnaam* maakt een nieuwe directory aan in de huidige directory.

7. `man` *commando* geeft uitleg (de zogenaamde **man page**) over een bepaald shell- of C-commando. Als je info over een C-commando wilt dat toevallig ook de naam van een shell commando is (bijvoorbeeld `printf`) kan je met een bepaalde optie aangeven welke sectie van de man pages je wilt doorzoeken; onder Solaris gaat dat met de `-s` optie. De C commando's zitten in sectie 3, dus: `man -s3 printf` (onder Linux: `man 3 printf`). De UNIX man pages zijn in de volgende secties onderverdeeld:

Section	Naam	Omschrijving
1	User Commands	Shell commando's (bv. `ls`)
2	System Calls	UNIX specifieke C functies (bv. `fork`)
3	C Library Functions/Subroutines	Standaard C functies (bv. `printf`)

Op de meeste systemen zijn er nog meer secties, maar die zijn niet steeds op dezelfde manier ingedeeld en ze zijn voor ons ook niet erg belangrijk. (Hoewel ... de spelletjes zitten in sectie 6.)

8. `apropos` *commando* geeft een lijst van alle man pages waarin het *commando* vernoemd wordt. Dus als je niet precies weet in welke sectie `printf` weer te vinden is, geef je gewoon `apropos printf`.

1.4 Hulpprogramma's

Er bestaan een aantal handige hulpprogramma's (**tool**s) om het programmeren wat te vergemakkelijken (zoals debuggers, `make`, ...). Die zal ik in aparte hoofdstukken bespreken. In het begin zal je die trouwens nog niet echt nodig hebben, maar eens je wat grotere programma's begint te schrijven, beginnen die hulpjes echt interessant te worden.

Hoofdstuk 2

Eenvoudige programmaatjes

In dit hoofdstuk zie je net genoeg van C om aan pointers te kunnen beginnen. De volledige syntax van C wordt pas in hoofdstuk 4 uit de doeken gedaan. Hierdoor zijn sommige voorbeelden niet zo elegant en beknopt als ze zouden kunnen, waardoor je aanvankelijk een ietwat eenzijdige indruk van de taal C zou kunnen krijgen. De bedoeling van dit hoofdstuk is dat je zo snel mogelijk aan de slag kunt met C; de finesses komen later wel.

2.1 Het eerste programma

Om vertrouwd te geraken met de ontwikkelomgeving, zullen we een eenvoudig programmaatje schrijven en compileren. Het eerste programma doet niets meer of minder dan een tekstje op het scherm afdrukken en stoppen. In C gaat dat ongeveer zo:

```
#include <stdio.h>

void main(void)
{
        printf("Dag grootmoeder!\n");
}
```

Dat heeft wel enige uitleg nodig.

2.1.1 Functies

"Begin at the beginning," the King said, very gravely,
"and go on till you come to the end: then stop."
—Lewis Carroll, "Alice in Wonderland"

De regel `void main(void)` geeft aan dat we een **functie** maken met als naam `main`. Deze regel heet de **functiehoofding**. Een functie is niets anders dan een hoop **opdracht**en (**command**s, **statement**s) achter elkaar. De opdrachten staan tussen accolades en eindigen op een kommapunt (later hierover meer, in §2.4.1).

Even een dom voorbeeldje tussendoor:

```
void snuitNeus(void)
{
        zoekZakdoek();
```

```
        houZakdoekVoorNeus();
        ademUitLangsNeus();
}
```

Deze functie (die ik dus `snuitNeus` genoemd heb) bevat drie opdrachten (het zijn zelf weer functieaanroepen). Wanneer de processor ergens in het programma de opdracht `snuitNeus();` tegenkomt, voert hij de drie opdrachten in `snuitNeus()` één na één uit. Daarna gaat het programma weer verder met de opdracht na `snuitNeus();`. Dat wordt het **aanroepen van een functie** genoemd, en een opdracht als `snuitNeus();` heet dan ook een **functieaanroep** (**function call**).

De drie opdrachten in de **definitie** van `snuitNeus()` zijn zelf weer functieaanroepen. Natuurlijk moeten we dan eerst ergens anders in het programma die drie functies programmeren, en wel ergens vóór de definitie van `snuitNeus()` zelf. Je kan een functie immers niet gebruiken vooraleer ze gedefinieerd is. Dat komt omdat de C compiler nooit "vooruit kijkt" in de broncode; al wat je op een bepaalde plek gebruikt, moet je daarvoor gedefinieerd hebben. Een C programma bestaat uit niets anders dan een hoop functiedefinities achter elkaar.

Zoals je hierboven al kon merken, eindigen functieaanroepen in C altijd op een kommapunt. Dit is een algemene regel in C: alle opdrachten eindigen op een kommapunt.

Ben je wel zeker dat je deze paragraaf moet lezen? Zoals gezegd bevatten deze "gevaarlijke bocht"-paragrafen stukjes die je beter eens leest als je al wat meer ervaring hebt. (Maar af en toe eens gluren kan natuurlijk geen kwaad ...)

In C is het `;` een terminator: het markeert het einde van een opdracht. (Dit in tegenstelling tot bv. Pascal, waar het `;` een separator is, d.i. het dient om twee opdrachten uit elkaar te houden. Het gevolg hiervan is dat de laatste opdracht van een blok in C wél en in Pascal geen `;` vereist.)

Ik zei dat je een functie niet kan gebruiken vóór je ze definieert—later zullen we zien dat het wel kan als je zgn. **prototype**s (zie §2.8.3.2) gebruikt.

De functie `main()` die ik daarnet gemaakt heb, doet dus niets anders dan de functie `printf()` aanroepen. Aan sommige functies kan je gegevens (meestal spreekt men van **argumenten**) doorsturen. De `main()` van daarnet ontvangt geen gegevens, vandaar de `(void)` erachter. Je kan ook functies maken die een resultaat teruggeven[1] (`main()` doet dat ook niet, vandaar de `void` vóór `main`). Schematisch ziet een **functiedefinitie** er als volgt uit:

$\overbrace{\texttt{void}}^{\texttt{uit}}$ *functienaam*($\overbrace{\texttt{void}}^{\texttt{in}}$)

```
{
    opdracht
        ⋮
    opdracht
}
```

Een reeks opdrachten tussen accolades heet een **blok**. Blokken zullen we verderop nog vaak tegenkomen.

Opgelet: na de afsluitende accolade van de functiedefinitie is geen kommapunt nodig.

[1] Voor de Pascal-fans: in C spreken we niet van "**procedure**s", wel van functies die `void` teruggeven.

▸ **OEFENING 2.1**
Als je toch een kommapunt schrijft, compileert het programma netjes. Waarom?

◇

De functie `printf()` heeft als argument een **string** (hierover dadelijk meer in §2.1.2) nodig, hier `"Dag grootmoeder!\n"`. `printf()` doet niets anders dan de string die je hem geeft op het scherm afdrukken.

In feite is het veiliger `main()` een `int` te laten teruggeven, dus had ik `int main(void) { ... }` moeten schrijven. Ik zal deze fout opzettelijk blijven maken tot we (in §2.7) gezien hebben hoe een functie een resultaat teruggeeft.

Je mag de naam van een functie vrij kiezen (bijvoorbeeld `snuitNeus`). We hadden dus net zo goed `mfraqll` of zo kunnen gebruiken; voor de compiler maakt het allemaal niet uit. Maar als je de functie `main` noemt, is er meer aan de hand. Het bijzondere van de functie `main()` is dat elk C programma er precies één moet hebben. Wanneer het programma start, begint de uitvoering bij deze functie. Een functie stopt wanneer de processor de laatste opdracht ervan heeft uitgevoerd (en "van de rand" van de functie "valt"); wanneer de `main()` functie stopt, is het programma daadwerkelijk ten einde. Meestal zal `main()` andere functies van het programma oproepen, die zelf weer andere functies kunnen gebruiken, enz.

Voor het overige mag je de namen van functies kiezen zoals je wilt, hoewel: er mogen geen spaties in voorkomen; cijfers mogen wel, maar niet aan het begin van de naam. Verder toegestaan zijn letters en de underscore (`_`). C maakt onderscheid tussen hoofd- en kleine letters, dus `mijnfunctie()` is iets anders dan `mijnFunctie()` of `MijnFunctie()`. Merk ook op dat het de gewoonte is om na de naam van een functie `()` te schrijven, gewoon om duidelijk te maken dat we het over een functie hebben. Ook belangrijk is dat je geen twee functies met dezelfde naam mag maken. Tenslotte mag je als naam geen van de volgende **gereserveerde woorden** (ook **sleutelwoorden** genoemd) kiezen:

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

Tabel 2.1: Gereserveerde woorden van C

In het `snuitNeus()` voorbeeldje van hierboven gebruikt geen enkele functie argumenten, dus de definities zouden er uit zien als

```
void zoekZakdoek(void)
{
        opdracht
        opdracht
}
```

De enige soort opdrachten die we tot nu toe gezien hebben, zijn functieaanroepen, maar dat zal natuurlijk snel genoeg veranderen ...

Zoals je ook kon zien in het voorbeeldje, moet je altijd haakjes tikken om een functie aan te roepen, ook al geef je geen argumenten door. Nochtans zal de compiler netjes een functie als deze compileren:

```
void snuitNeus(void)
{
        zoekZakdoek;
        houZakdoekVoorNeus;
        ademUitLangsNeus;
}
```

Maar zo'n opdracht doet helemaal niets zonder de haakjes! Vooral voor Pascal gebruikers is het dus even opletten. Een goede C compiler zal wel een waarschuwing geven in de aard van "wat je schrijft is wel correct C, maar waarschijnlijk niet wat je bedoelt". (De GNU-C compiler kan hierop letten als je dat wilt en zegt dan `warning: statement with no effect`—zie bijlage E.)

▸ **OEFENING 2.2**
Verklaar waarom een opdracht als `zoekZakdoek;` (dus zonder haakjes erachter) goed C is en wat er precies gebeurt bij de uitvoering ervan.

◇

2.1.2 Strings

Een string is één van de **datatype**s (zie §2.3) waarmee C kan werken. Een (weinig gebruikte) Nederlandse vertaling is "tekenrij", wat precies aangeeft wat een string is: een rijtje tekens achter elkaar. Onder "teken" verstaan we hier letters, cijfers, leestekens, spaties, ... Met die strings kan je een aantal bewerkingen uitvoeren (bijvoorbeeld aan elkaar plakken, een string in een andere zoeken, ...). Hoe dat precies moet in C zal ik in hoofdstuk 3 uitleggen.

Om de compiler duidelijk te maken dat er een string volgt, zet je hem gewoon tussen dubbele aanhalingstekens, zoals in het voorbeeldprogrammaatje. Vanzelfsprekend rijst het probleem hoe je het teken `"` zelf in een string kan opnemen. Dat gaat door er een backslash (\) vóór te schrijven, bijvoorbeeld:

```
printf("Ze zei: \"Dag grootmoeder!\"");
```

waardoor de tekst `Ze zei: "Dag grootmoeder!"` afgedrukt wordt.

Het lijkt erop dat we gewoon het probleem verplaatst hebben—want hoe stop je een \ in een string? Door er een backslash vóór te zetten natuurlijk:

```
printf("Het teken \\ wordt \"backslash\" uitgesproken.");
```

Ook `\n` heeft een speciale betekenis: het wordt vervangen door het **"nieuwe regel"-teken** (ook wel **newline**-teken genoemd), wat overeenkomt met het indrukken van de Enter- of Return-toets. Zonder `\n` zou in het voorbeeldprogramma de cursor blijven staan vlak na `grootmoeder!`, wat natuurlijk geen zicht is. In het voorbeeld had

```
printf("Dag gr");
printf("ootmoeder!\n");
```

dus precies hetzelfde effect gehad.

Merk op dat speciale tekens als `\\`, `\"` en `\n` in de broncode twee letters innemen, maar door de compiler als één teken worden beschouwd. De string `"Dag grootmoeder!\n"` uit het eerste voorbeeld is dus 17 tekens lang.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
D	a	g		g	r	o	o	t	m	o	e	d	e	r	!	\n

De tekens in de string `"Dag grootmoeder!\n"` (in C beginnen we vanaf 0 te tellen)

In C kunnen strings willekeurig lang zijn. Een string kan ook leeg zijn (de lege string wordt natuurlijk als `""` geschreven).

2.1.3 Meer over `printf()`

De functie `printf()` kan je ook meer dan één argument meegeven. Het eerste argument moet altijd een string zijn:

$$\texttt{printf(}\underbrace{\texttt{"Het gemiddelde van }\overbrace{\texttt{\%d}}\texttt{ en }\overbrace{\texttt{\%d}}\texttt{ is }\overbrace{\texttt{\%d}}\texttt{."}}_{\text{eerste argument}}\texttt{, }\underbrace{\texttt{5}}_{\text{2e}}\texttt{, }\underbrace{\texttt{6}}_{\text{3e}}\texttt{, }\underbrace{\texttt{(5+6)/2}}_{\text{4e arg}}\texttt{);}$$

Telkens als `printf()` een `%`-teken in het eerste argument tegenkomt, krijgt het teken er vlak na een bijzondere betekenis. Hierboven heb ik een paar keer `%d` gebruikt, wat voor `printf()` een aanwijzing is dat het volgende argument hier als decimaal getal moet afgedrukt worden. (Je zou het bijvoorbeeld ook hexadecimaal kunnen afdrukken met `%x`, en er zijn nog veel meer mogelijkheden—zie §5.3.1.)

Het eerste argument van `printf()` bepaalt in feite de manier waarop alle andere argumenten verwerkt worden. Het wordt wel eens een **formaatstring** genoemd; vandaar de naam `printf()`: print formatted.

In tegenstelling tot wat je misschien denkt, is `printf()` niet vast in de taal C "ingebakken", maar een doodgewone functie, die je dus evengoed zelf zou kunnen maken.

Nochtans staat in ons programma nergens een definitie van `printf()` (zoals we wél een `main()` volledig uitgeschreven hebben). De oplossing van het raadsel is de regel met `#include <stdio.h>` bovenaan het programma. In `stdio.h` (wat staat voor *standard input/output header file*) zitten een aantal interessante functies, waaronder `printf()`. Verderop zullen we nog andere **header file**s tegenkomen, zoals `string.h`, waar—zoals de naam zegt—stringfuncties in zitten. Regels die met `#` beginnen heten **preprocessor-aanwijzing**en.

Nog een opmerkingetje over `%d` tegenover `\n`: deze twee tekencombinaties binnen strings worden op een totaal andere manier behandeld door de C compiler. `\n` en soortgelijken worden door de compiler zelf in één karakter omgezet in om het even welke string ze voorkomen. De compiler doet echter niets speciaals met `%d`; de functie `printf()` is volledig zelf verantwoordelijk voor de verwerking.

Regels die met `#` beginnen worden in feite niet door de eigenlijke C compiler gezien. Vlak vóór het eigenlijke compileren wordt het C programma door de **preprocessor** gegooid, die alle regels die met `#` beginnen interpreteert, commentaren verwijdert, macro's expandeert ... Meer hierover in hoofdstuk 6.

▸ **OEFENING 2.3**
Veel compilers zullen het programmaatje van daarnet probleemloos compileren als je de regel met `#include <stdio.h>` weglaat. Verklaar waarom, en probeer redenen te vinden waardoor een bepaalde compiler toch `#include <stdio.h>` nodig zou kunnen hebben om het programma te compileren.

♢

▸ **OEFENING 2.4**
Ik zei dat `printf()` een gewone functie was. Maar hoe slaagt die er dan in om uitvoer naar het scherm te sturen?

♢

2.1.4 Editeren en compileren

Na die bespreking zal je wel willen weten hoe je nu zo'n C listing draaiend krijgt op de computer. Ik ga ervan uit dat je op een UNIX-achtig systeem zit en ingelogd bent. Normaal gezien zou er een shell moeten draaien die iets als

```
/usr/gvernaev$ _
```

tegen je zegt (`_` stelt hier de cursor voor)—als je een MS-DOS systeem gebruikt, zal de shell eerder `C:\usr\gvernaev>_` zeggen.

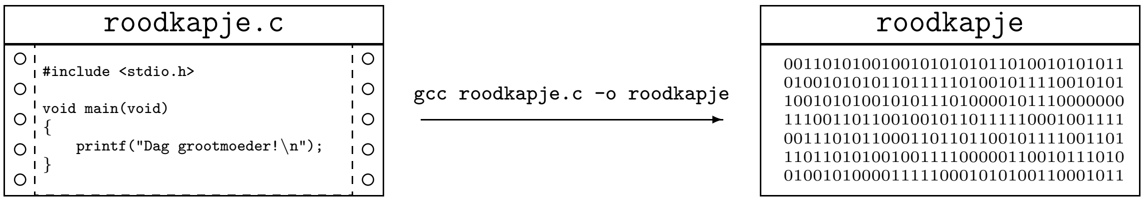
Tik bijvoorbeeld

```
pico roodkapje.c
```

om een bestand met als naam `roodkapje.c` te maken. (Het is een gewoonte om C broncodebestanden op `.c` te doen eindigen, zodat iedereen, en in het bijzonder je C compiler, kan zien dat het een C programma is.) Je kan nu het programma intikken en bewaren (in pico met Ctrl-O[2]). Als je nu uit de editor gaat (Ctrl-X) kom je weer in de shell terecht. (Tik eens `ls` en zie: er is wel degelijk een bestand `roodkapje.c` verschenen in je directory.) De laatste stap is compileren:

```
gcc roodkapje.c -o roodkapje
```

De compiler `gcc` (wat staat voor GNU C Compiler) werkt zich door je programma heen, en als alles goed is (doe weer `ls`) staat het **uitvoerbaar programma** (dat is een versie van je programma dat uit een hoop binaire rommel bestaat waar de processor goeie vriendjes mee is) in het bestand `roodkapje`. Je kan het programma starten door in de shell `roodkapje` te tikken; zoals je ziet gedraagt ons programma zich als elk ander UNIX commando. In feite zou je kunnen zeggen dat we een nieuw commando `roodkapje` geprogrammeerd hebben. Bijna alle shell commando's zijn in feite gecompileerde C programma's (`ls` is bijvoorbeeld afkomstig van `ls.c` …).



[2]Ctrl-O betekent: houd de Ctrl toets ingedrukt, tik een O, en laat de Ctrl toets weer los.

▸ **OEFENING 2.5**
Wat gaat er mis als je het programma `test.c` zou noemen en compileren met `gcc test.c -o test` en hoe kan je het oplossen? (Deze oefening is alléén voor harde UNIX-freaks . . .)

◇

2.1.4.1 Witte ruimte in C

Voor C is een opeenvolging van een aantal returns, spaties of tabs (samen **witte ruimte** genoemd) hetzelfde als één enkele spatie. Je mag ook tussen woorden en operatoren extra witte ruimte tussenvoegen: dus `void main(void)` is even goed als `void main ( void )` maar de spatie tussen `void` en `main` mag niet weg omdat de compiler anders ineens op het onbekende woord `voidmain` zou botsen. De enige uitzondering hierop zijn **preprocessor-aanwijzing**en (regels die met `#` beginnen): je mag ze niet uitsmeren over meerdere regels (spaties of tabs erbij gooien mag in principe wél, maar dus geen returns).

Je kan het programma van daarnet dus opeenproppen tot

```
#include <stdio.h>
void main(void){printf("Dag grootmoeder!\n");}
```

of lekker lang uitsmeren en een beetje artistiek doen met de insprongen:

```
#include <stdio.h>

void
                   main
(
  void
             )
    {
                         printf
    (
"Dag grootmoeder!\n"
  )
 ;
}
```

Merk op dat een string als één eenheid beschouwd wordt, en je die dus niet mag uiteentrekken over meerdere regels. Wil je dat toch, dan mag je ook bijvoorbeeld `"Dag " "grootm" "oeder!\n"` schrijven. De C compiler plakt twee strings die achter elkaar komen zelf aaneen tot één lange string.

Je hebt dus vrij veel vrijheid in de layout van de broncode. De stijl die ik hier gebruik is de zgn. **K&R indentatie-stijl**, zo genoemd omdat die gebruikt werd door Kernighan en Ritchie, de makers van C, in hun klassieke boek *The C programming language*. De kenmerken van deze stijl zijn o.a. een tabdiepte van acht spaties, en het `{` teken staat nooit op een aparte regel, behalve bij een functiedefinitie. Welke stijl je precies gebruikt heeft geen belang, zo lang je maar consistent blijft. (Zie §2.4.1.)

2.2 Functies met argumenten

De functies die we tot nog toe gemaakt hebben, waren van de vorm `void f(void)`, m.a.w. de functie krijgt geen argumenten binnen en geeft ook geen resultaat terug.

Hoogste tijd om daar wat verandering in te brengen:

gemiddelde.c

```
#include <stdio.h>

void gemiddelde(int x,int y)
{
        printf("Het gemiddelde van %d en %d is %d.\n",x,y,(x+y)/2);
}

void main(void)
{
        gemiddelde(5,6);
}
```

Ik heb dus een functie `gemiddelde()` gemaakt die twee argumenten (`x` en `y`) nodig heeft, beide gehele getallen (vandaar `int`—zie verder). De functie geeft geen resultaat terug (aangezien we pas in §2.7 zullen zien hoe dat moet), dus staat er `void` voor.

Misschien lijkt dit nog niet zo erg spectaculair, maar je kan toch zien dat functies handig zijn om veelgebruikte routines in "in te kapselen". (Ik zal dan ook af en toe van **routine** of **subroutine** spreken in plaats van functie.) Telkens als je nu ergens een gemiddelde moet afdrukken, kan je je de moeite besparen de hele `printf()` rimram in te tikken, en volstaan met `gemiddelde()`. In dit voorbeeld is het natuurlijk een beetje overdreven om een functie te maken die uit maar één opdracht bestaat, maar ja ... Meestal zal `main()` andere functies uit het programma aanroepen, die zelf weer andere functies kunnen gebruiken, enz. Op die manier kan je het programma in handelbare brokken opdelen. Een programma mag in principe uit één reusachtige `main()` functie bestaan, maar dat is meestal niet erg duidelijk leesbaar, tenzij het natuurlijk een heel kort programmaatje is.

Tussen haakjes: de vier **hoofdbewerkingen** worden in C genoteerd met de operatoren `+ - * /`. Vermenigvuldigingen en delingen worden eerst uitgevoerd, tenzij je haakjes gebruikt. Je mag altijd extra haakjes toevoegen in geval van twijfel. Zie §4.3.11 voor meer hierover.

Een probleempje bij het programma van daarnet is dat de deling van twee gehele getallen in C ook weer een geheel getal oplevert. De uitdrukking `(5+6)/2` wordt dus als `11/2` uitgerekend, wat echter uiteindelijk `5` oplevert (er wordt afgekapt bij het delen). Een mogelijke oplossing voor dit probleem volgt in §2.3.2.

Over de precieze rol van de `x` en `y` blijf ik hier met opzet een beetje vaag; in feite gedragen ze zich als lokale variabelen die bij het binnenkomen van de functie gevuld worden met de waarden die uitgerekend worden bij de functieaanroep (zie §2.8.2). Dus een opdracht

```
gemiddelde(x*y + z, 2*x);
```

heeft als effect dat de waarde `x*y+z` wordt uitgerekend en in een *nieuwe* variabele `x` wordt gestopt, en `2*x` komt in een nieuwe variabele `y`. De functie `gemiddelde()` zelf weet helemaal niets over het bestaan van de variabelen `x`, `y` en `z` in de functie die `gemiddelde()` aanroept.

In het bijzonder kan een functie geen variabelen van de aanroeper wijzigen:

```
#include <stdio.h>

void test(int x)
{
        x = 8;
        printf("%d",x);
}

void main(void)
{
        int y;

        y = 7;
        test(y);
        printf(" %d\n",y);
}
```

zal als resultaat

```
8 7
```

geven. De functie `test()` krijgt immers alleen maar een *kopie* van de waarde van de uitdrukking "y" door, en de opdracht `x = 8` verandert alleen maar de waarde van die lokale kopie. In programmeursjargon heet de manier waarop argumenten in C worden doorgegeven **by value**: de *waarde* van de uitdrukking wordt doorgegeven, en niet de variabele die toevallig die waarde bevat (dat systeem heet **by reference** en kan in C worden nagebootst met pointers). Meer hierover in §2.8.2.

De functies die ik hier gemaakt heb, gebruiken een vooraf vast bepaald aantal argumenten. Er zijn nochtans functies waarbij het aantal argumenten kan verschillen van aanroep tot aanroep, zoals bij `printf()` het geval is. In §4.6 zal ik uitleggen hoe je zelf zulke functies kunt maken.

2.3 Variabelen

Een interessant concept waar bijna elke programmeertaal op drijft, is **variabele**. Een variabele is een stukje geheugen met een naam (de **variabelnaam**) waarin je een **waarde** kunt stoppen en die er later weer uithalen. Wat voor soort waarden er in passen wordt bepaald door het **type** (ook **datatype** genoemd) ervan. C kent onder andere de datatypes `int` (in een `int` variabele passen gehele getallen), `double` (kommagetallen), en `char` (een letter).

Variabelnamen mogen net zoals functienamen (zie §2.1.1) bestaan uit letters, de underscore, en cijfers (ook hier mag de naam niet met een cijfer beginnen). En ook hier mag je geen twee variabelen met dezelfde naam maken. Zoals steeds heeft de naam die je aan een variabele geeft geen invloed op het verloop van het programma; als je dus in het voorbeeld hieronder alle '`letter`' door '`gfrfw`' zou vervangen, maakt het voor de compiler geen enkel verschil. Tenslotte mag je ook geen twee variabelen met dezelfde naam maken.

Even een voorbeeldje:

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

void main(void)
{
        /* Maak variabelen */
        int geheel;
        double reeel;
        char letter;
        char str[20];

        geheel = 10;        /* De eerste opdracht van het programma */
        reeel = 20.0;
        geheel = geheel*2 + reeel;
        letter = 'c';
        letter = letter + 1;
        strcpy(str,"Dag grootmoeder!");
        str[5] = letter;
        /* Druk een int (%d), een double (%f), een char (%d)
         * en een string (%s) af */
        printf("%d %f %c %s\n",geheel,reeel,letter,str);
}
```

Als eerste nieuwigheidje is er `/* tekst */`, waarmee je **commentaar** aan het programma kunt toevoegen. Op die manier kan je het programma voor anderen (en voor jezelf, een paar maand later) duidelijker maken. De C compiler doet hier niets mee: het hele `/* ... */` stuk wordt intern vervangen door een spatie, m.a.w. een commentaar mag staan overal waar een spatie mag komen.

De eerste vier regels geven telkens aan dat we een variabele van een bepaald type (bijvoorbeeld `int`) met een bepaalde naam (`geheel`) willen maken. Algemeen kan je een variabele met een bepaalde *naam* van een of ander *type* maken met

type naam `;`

Zoals je ziet, moeten variabelen altijd aan het begin van een functie gemaakt (in het vakjargon: gedeclareerd) worden, nog vóór de eerste opdracht. Variabelen**declaratie**s worden *niet* als opdrachten beschouwd, ook al zien ze er wel een beetje zo uit.

Variabelen worden gemaakt op het ogenblik dat de functie aangeroepen wordt, nog net vóór de eerste opdracht ervan uitgevoerd wordt, en ze verdwijnen weer bij het verlaten van de functie. (Dit heet de **levensduur** van een variabele).

Nu we wat variabelen hebben gemaakt, kunnen we er naar hartelust waarden instoppen, wat bij de meeste soorten variabelen gaat met een **toekenningsopdracht** van de vorm

variabele `=` *waarde* `;`

waarbij *waarde* een gecompliceerde **uitdrukking** mag zijn, bijvoorbeeld `geheel*2 + reeel`. Hoe gecompliceerd zulke uitdrukkingen precies kunnen worden, leg ik uit in §4.3. De oorspronkelijke waarde van *variabele* gaat verloren. Op de meeste plaatsen wordt de naam van een variabele vervangen door haar waarde (zoals in `geheel*2 + reeel`), maar natuurlijk niet links van de `=` operator (anders zou `geheel = geheel*2 + reeel` als `10 = 10*2 + 20.0` worden uitgerekend, wat natuurlijk nergens op slaat).

Merk ook op dat een opdracht als `x = x + 1;` wel degelijk in orde is—ook al is het wiskundig gezien grote onzin—omdat `=` betekent: bereken wat rechts staat van `=` en stop dat in de variabele waarvan de naam links van `=` staat. Wat er dus zal gebeuren bij deze opdracht is dat eerst `x + 1` wordt uitgerekend, en dat getal wordt dan gestopt in de variabele die `x` heet. Met andere woorden, na deze opdracht is de waarde van de variabele `x` met één verhoogd. Hiermee zullen we later tellers kunnen bijhouden.

Het heeft dus ook geen zin iets dat geen variabele is links van `=` te plaatsen (`10 = x + 2` werkt dus niet, evenals `x + y = z + 2` . . .).

In het vervolg zal ik niet zo vaak meer spreken van "de variabele met naam `x`" maar kort "de variabele `x`" of zelfs "`x`" zeggen.

Tenslotte: een opdracht als `x = y;` legt geen koppeling tussen `x` en de waarden die `y` verderop in het programma zal bevatten. Als `y` later een andere waarde krijgt, dan heeft `x` daar niets mee te maken.

▸ **OEFENING 2.6**
Hoe verwissel je de inhoud van twee variabelen, bijvoorbeeld de twee `int`s `x` en `y`?

◇

2.3.1 Initialisatie

Aan het begin van de functie worden zoals gezegd de variabelen gemaakt, d.i. er wordt geheugen voor vrijgemaakt. De inhoud van die variabelen is echter **onbepaald**; je weet dus niet welke waarde bijvoorbeeld in `geheel` zit (het is bijna zeker *niet* nul). Daarom heb ik als eerste opdracht van de functie een waarde in `geheel` gestopt (`geheel = 10`).

Men zegt dat de variabelen aan het begin van een functie **ongeïnitialiseerd** zijn, en het stoppen van een waarde in zo'n variabele heet **initialiseren** van een variabele.

2.3.2 `int` en `double`

Uit het voorbeeld blijkt ook dat C `double`s automatisch omzet in `int`s (het omgekeerde is trouwens ook waar). Bij het omzetten van een `double` naar een `int` wordt afgekapt (dus 3.823 wordt 3; −3.823 wordt −3). Het is ook mogelijk dat het getal niet meer in een `int` past (het is bijvoorbeeld te groot); het programma loopt in zo'n geval gewoon verder en de waarde van de `int` is dan onbepaald. Met **onbepaald** wordt bedoeld dat het resultaat afhankelijk is van de gebruikte computer en compiler, en vaak zelfs van het moment waarop het programma draait.

Het bereik van `int` (d.i. de grootste en kleinste waarde die in een `int`-variabele past) verschilt van computer tot computer en van compiler tot compiler. Op de meeste moderne 32-bits systemen kan een `int` waarden van -2.147.483.648 tot en met 2.147.483.647 bevatten; op de meeste 16-bits systemen (zoals veel MS-DOS C compilers) is het bereik van -32768 tot en met 32767.

De verklaring hiervoor is dat `int` wordt beschouwd als het "natuurlijke geheel getal datatype" van het systeem waarop het programma draait, wat op bijvoorbeeld een 32-bits systeem een groter bereik geeft dan op een 16-bits systeem.

Uit het voorbeeld blijkt dat je `int`s en `float`s kunt afdrukken met `printf()` door respectievelijk `%d` en `%f` in het formaatargument te schrijven. (Als je een kommagetal met `%d` probeert af te drukken, of een geheel getal met `%f`, kunnen er rare dingen gebeuren.)

We kunnen nu ook een betere versie van `gemiddelde()` uit §2.2 maken:

```
void gemiddelde(int x,int y)
{
        printf("Het gemiddelde van %d en %d is %f.\n",x,y,(x+y)/2.0);
}
```

Bij het uitvoeren van de functieaanroep `gemiddelde(5,6)` wordt nu het gehele getal `11` door het kommagetal `2.0` gedeeld. Als minstens één argument van `/` een kommagetal is, is het resultaat dat ook (dit is overigens ook zo bij de andere operatoren).

2.3.3 `char` en strings

In het voorbeeld kan je zien dat `char`-waarden tussen enkele aanhalingstekens worden gezet. Het probleem hoe een enkel aanhalingsteken in een `char` constante te zetten wordt op dezelfde manier als bij strings opgelost: `letter = '\''`.

Interessant is de opdracht `letter = letter + 1`. Bij een `char` kan je dus een getal optellen(!). Dat komt omdat `char`s in C in feite kleine `int`s zijn (met een bereik dat meestal -128 t/m 127 of 0 t/m 255 is), en een `char` constante, zeg `'c'`, wordt door de compiler vertaald in een getal (op de meeste systemen de ASCII code van die letter, hier dus de ASCII code van `c`, 99). Met andere woorden: of je nu ergens `99` of `'c'` schrijft, maakt niets uit—tenzij je op een systeem werkt dat een andere lettercodering gebruikt (EBCDIC om maar iets te zeggen ...). In bijlage A vind je de een tabel van de ASCII codes.

Chars en strings kan je afdrukken door `%c` respectievelijk `%s` in het formaatargument van `printf()` te schrijven.

Strings in C zijn ook een verhaal apart; om ze volledig te begrijpen moet je in feite al iets weten over arrays en pointers. Een aantal dingen zullen dus voorlopig een beetje "ad hoc" overkomen. Om maar meteen te beginnen: merk op dat ik niet

```
str = "Dag grootmoeder!";
```

maar wel de functie `strcpy` gebruikt heb (die leeft in `string.h`).

Wie de pointers in C al machtig is weet dat `str = "Dag grootmoeder!";` wel degelijk goed C is, op voorwaarde dat `str` gedeclareerd is als `char *`; een stringconstante (zoals `"Dag grootmoeder!"`) is immers niets anders dan een pointer naar de eerste letter ervan (een `char *` dus).

Voorlopig zal je dus moeten onthouden dat strings `strcpy()` nodig hebben. Er zullen trouwens nog meer "eigenaardigheden" in verband met strings volgen; trek je er niet te veel van aan en bijt door tot §3.3 waar je alles over pointers te weten komt ...

Terug naar de listing. Zoals je ziet, maakt `char str[20]` een stringvariabele genaamd `str` waar 20 `char` variabelen in zitten, namelijk `str[0]` (waar de eerste letter van de string in zit) t/m `str[19]`. Net als alle andere variabelen zijn die alle twintig nog niet geïnitialiseerd (aangeduid in de figuur door "?"). Iets om voor op te letten is dat `str[20]` dus niet bestaat. De C compiler waarschuwt overigens *niet* als je een index gebruikt die te groot of te klein is (bijvoorbeeld bij `str[-5]`).

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?

Een C-string wordt altijd afgesloten met een **nulbyte**. (We spreken van een **null-terminated** string.) Dat is niet het cijfer `'0'` (wat voor de C compiler gelijk is aan de ASCII waarde van 0, nl. 48) maar dus echt `0`. De string `"Dag grootmoeder!"` wordt dus als volgt opgeslagen:

’D’	’a’	’g’	’ ’	’g’	’r’	’o’	’o’	’t’	’m’	’o’	’e’	’d’	’e’	’r’	’!’	0	?	?	?

In plaats van die `strcpy()` had ik dus even goed dit kunnen schrijven:

```
        str[0] = 'D';
        str[1] = 'a';
        /* enzovoort */
        str[14] = 'r';
        str[15] = '!';
        str[16] = 0;
```

Eigenlijk is dit de manier waarop `strcpy()` van binnen werkt.

De langste string die in `str` past heeft dus 19 letters (geen 20!): `str[0]` t/m `str[18]` bevatten dan de letters van de string en `str[19]` de nulbyte.

▸ **OEFENING 2.7**
Ga na wat er met `str[0]` t/m `str[19]` gebeurt tijdens de uitvoering van deze opdrachten:

```
{
        char str[20];

        strcpy(str,"Dag grootmoeder!\n");
        str[12] = str[15];
        strcpy(str,"Aan");
        str[3] = ' ';
        str[13] = 0;
}
```

Maak hiertoe een hoop tekeningetjes die de inhoud van `str` op verschillende tijdstippen voorstellen. Het is overigens in het algemeen een goed idee om veel tekeningetjes tijdens het programmeren te maken . . .

◊

▸ **OEFENING 2.8**
In oefening 2.6 moest je de inhoud van twee `int` variabelen omwisselen. Hoe wissel je de inhoud om van twee stringvariabelen, zeg `zus` en `zo`?

◊

2.4 Lussen

. . . and he started to count to ten.
He was desperately worried that one day sentient life forms would forget how to do this.
Only by counting could humans demonstrate their independence of computers.
—Douglas Adams, “The Hitch Hiker’s Guide to the Galaxy”

Variabelen zijn hét gedroomde middel om tellers bij te houden. Eerst enkele simpele voorbeeldjes om het duidelijk te maken: we gaan van 1 tot 100 tellen, en daarna in een tweede voorbeeld ook nog een rij sterretjes afdrukken:

```
#include <stdio.h>

void main(void)
{
        int teller;

        teller = 1;
        while (teller <= 100) {
                printf("%d ",teller);
                teller = teller + 1;
        }
        printf("\n");
}
```

Nieuw hier is `while`. In tegenstelling tot wat je aanvankelijk zou kunnen denken, is `while` *geen* functie, maar een ingebakken **sleutelwoord** van C. `while` wordt als volgt gebruikt:

```
while ( voorwaarde ) {
        opdracht
        ⋮
        opdracht
}
```

waarbij de reeks opdrachten tussen accolades weer een **blok** (dadelijk meer daarover in §2.4.1) vormt.

Als het programma op een `while` opdracht botst, gebeurt het volgende:

1. Controleer of de *voorwaarde* waar is. Als de voorwaarde niet vervuld is, gaat het programma verder met de opdracht na het blok (in het voorbeeld is dat `printf("\n")`); de uitvoering van de `while` opdracht is hiermee afgelopen. Is de voorwaarde wel vervuld, ga dan naar stap 2:

2. Voer alle opdrachten van het *blok* één voor één uit (net zoals bij een functie).

3. Herbegin bij de eerste stap.

Met andere woorden, *zolang* de voorwaarde vervuld is, wordt het blok telkens opnieuw uitgevoerd. In het blok staat natuurlijk meestal één of andere opdracht die ervoor zorgt dat de voorwaarde na verloop van tijd onwaar wordt, want anders krijgen we een **oneindige lus** (**infinite loop**)—hier zorgt `teller = teller + 1` ervoor dat na honderd doorlopen de **lus** afbreekt.

`while` is ook een opdracht (deze opdracht eindigt *niet* op een kommapunt—maar zie §2.4.2) en kan dus zelf in het blok van een andere `while` staan, die weer in het blok van nog een andere `while` staat, enz. Dat wordt het **nesten** van `while`s genoemd. (Zie ook §2.4.2.)

Laten we eens kijken wat er precies gebeurt in het voorbeeldprogramma. Eerst wordt `teller = 1` gemaakt (zodat het eerste getal dat verderop afgedrukt zal worden een 1 is), en dan storten we ons in de `while` lus. De voorwaarde wordt `1 <= 100`, wat natuurlijk waar is, en dus worden de twee opdrachten in het blok uitgevoerd, m.a.w. er wordt een 1 afgedrukt en `teller` bevat nu de waarde 2.

De voorwaarde wordt weer gecontroleerd: `2 <= 100`, ...
Na achtennegentig lusdoorlopen bevat `teller` uiteindelijk de waarde 99; `99 <= 100` is waar, dus wordt het blok uitgevoerd (druk 99 af, `teller` wordt 100). `100 <= 100` is nog net waar, dus er wordt 100 afgedrukt, `teller` wordt opgehoogd tot 101 en de voorwaarde wordt een laatste maal gecontroleerd: `101 <= 100`, wat vals is, dus de `while` lus is gedaan en het programma gaat verder bij de opdracht na het blok (de `printf("\n")` aanroep).

Handige operatoren in de voorwaarde van een `while` zijn:

`>=`	groter dan of gelijk aan	`<=`	kleiner dan of gelijk aan
`>`	groter dan	`<`	kleiner dan
`==`	gelijk aan	`!=`	niet gelijk aan

Let vooral op bij de `==` operator: als je `while (x = 10)` schrijft, zal de compiler netjes voortcompileren (want het is goed C; wat het precies betekent zal in §4.3.5.1 duidelijk worden), maar het programma zal niet echt lopen zoals je dat had bedoeld ... De GNU C compiler kan als je dat wil voor dit soort gevaarlijke toestanden waarschuwen (zie §4.3.5.1).

▸ **OEFENING 2.9**
Laat het programma tellen van 1 tot 100 in stappen van 3 (dus `1 4 7 ... 97 100`).

♢

▸ **OEFENING 2.10**

'Three minutes and forty seconds.'
'Will you stop counting!' snarled Zaphod.
'Yes,' said Ford Prefect, 'in three minutes and thirty-five seconds.'
—Douglas Adams, "The Hitch Hiker's Guide to the Galaxy"

Herschrijf het programma zodat het van 100 tot 1 aftelt.

♢

Sterretjes afdrukken is nog eenvoudiger:

```
#include <stdio.h>

void main(void)
{
        int teller;

        teller = 1;
        while (teller < 80) {
                printf("*");
                teller = teller + 1;
        }
        printf("\n");
}
```

Je zult nu wel ongeveer doorhebben wat er hier aan de hand is.

▸ **OEFENING 2.11**
Hoeveel sterretjes worden er hier precies afgedrukt?

♢

▸ **OEFENING 2.12**
Maak een programmaatje dat een tabel op het scherm zet met in de eerste kolom de getallen van 1 tot 10, in de tweede kolom het kwadraat van het getal uit de eerste kolom, en in de

derde kolom het omgekeerde ervan. (Gebruik `\t`, het tab-karakter, dat de cursor op het begin van een volgende tab-positie zet om makkelijk tabellen te maken.)

♢

2.4.1 Blokken

In My Egotistical Opinion,
most people's C programs should be indented
six feet downward and covered with dirt.
—Blair P. Houghton

Blokken worden veel gebruikt in C; in feite is de hele definitie van een functie ook een blok. Een blok ziet er uit als

{
    variabelendeclaraties

    opdrachten
}

Zoals al eerder bleek, eindigen alle variabelendeclaraties en opdrachten (`while` uitgezonderd) op een kommapunt. De gedeclareerde variabelen worden gemaakt vlak vóór de eerste opdracht van het blok uitgevoerd wordt, en verdwijnen weer als het programma het blok verlaat.

Bovendien mag je overal waar een gewone opdracht staat een blok schrijven, dus ook plots midden in een hoop opdrachten:

```
{
        printf("Tien sterrekes:\n");
        {       /* Hier begint het "middelste" blok */
                int teller;

                teller = 0;
                while (teller < 10) {
                        printf("*");
                        teller = teller + 1;
                }
        }
        printf("\n");
}
```

Het voordeel hiervan boven

```
{
        int teller;

        printf("Tien sterrekes:\n");
        teller = 0;
        while (teller < 10) {
                printf("*");
                teller = teller + 1;
        }
        printf("\n");
}
```

is dat de variabele `teller` alleen blijft bestaan waar ze echt nodig is. Dat heeft wel degelijk voordelen:

- De compiler kan *efficiëntere* code genereren door het geheugen dat `teller` inneemt verder in de functie te hergebruiken om andere variabelen (in dit simpele voorbeeldje zijn er toevallig geen) in op te slaan (en voor de assemblerfreaks: de compiler zal makkelijker zien dat `teller` in een register kan bijgehouden worden).

- Deze aanpak is *veiliger*: de compiler zal een foutmelding geven als je `teller` probeert te gebruiken op een "verkeerde" plaats (i.e. buiten het middelste blok). De variabele is **onzichtbaar** buiten het blok.

- Het programma wordt *duidelijker* om lezen: het is onmiddellijk te zien dat `teller` alleen gebruikt wordt om sterretjes te tellen (in grotere functies zou je bijvoorbeeld kunnen twijfelen of er nu verderop nog iets met `teller` gedaan wordt of niet).

We zeggen dat `teller` een **lokale variabele** is van het blok.

Je mag omgekeerd ook een blok dat maar één opdracht bevat vervangen door alleen die opdracht, dus

```
        x = 0;
        while (x < 10)
                x = x + 1;
                printf("%d\n",x);
```

is hetzelfde als

```
        x = 0;
        while (x < 10) {
                x = x + 1;
        }
                printf("%d\n",x);
```

Merk op dat de `printf()` dus maar één keer wordt uitgevoerd (en het getal `10` afdrukt)! Met andere woorden, de indentatie (het inspringen als er een blok begint) in dit voorbeeld is (opzettelijk) verkeerd.

De enige uitzondering hierop is de definitie van een functie:

```
void gemiddelde(x,y)
        printf("Het gemiddelde van \%d en \%d is \%d.\n",x,y,(x+y)/2);
```

mag *niet*; de C compiler moet wel degelijk accolades zien aan het begin van een functie. Als je twijfelt: zie §2.2 voor de juiste versie.

Het spreekt vanzelf dat een goed geïndenteerd programma makkelijker leest. Er zijn een aantal populaire stijlen in omloop (zie ook §2.1.4.1); zo kan je bijvoorbeeld indenteren met tabs of met een aantal spaties, of ook een iets ruimere stijl kiezen:

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

        teller = 1;
        while (teller <= 100)
```

```
        {
                printf("%d ",teller);
                teller = teller + 1;
        }
        printf("\n");
}
```

Dit alles is natuurlijk vooral een kwestie van persoonlijke voorkeur en smaak. Er bestaat trouwens een programma `indent` dat C programma's op alle mogelijke manieren kan indenteren.

2.4.2 Geneste blokken en nog eens `while`

Omdat je overal waar een gewone opdracht mag staan ook een blok mag schrijven, kan je blokken in blokken maken (en daarin weer nieuwe blokken ...). We noemen dit het **nesten** van blokken. Belangrijk om op te merken is dat je in C geen functiedefinities kan nesten.

De eigenlijke syntax van `while` is dan ook:
`while (` *voorwaarde* `)` *opdracht*
hetgeen verklaart waarom er geen kommapunt na een while met een blok kwam: de *opdracht* kan ofwel een enkelvoudige opdracht zijn (die netjes op een kommapunt eindigt) of een blok (dat geen kommapunt achteraan heeft).

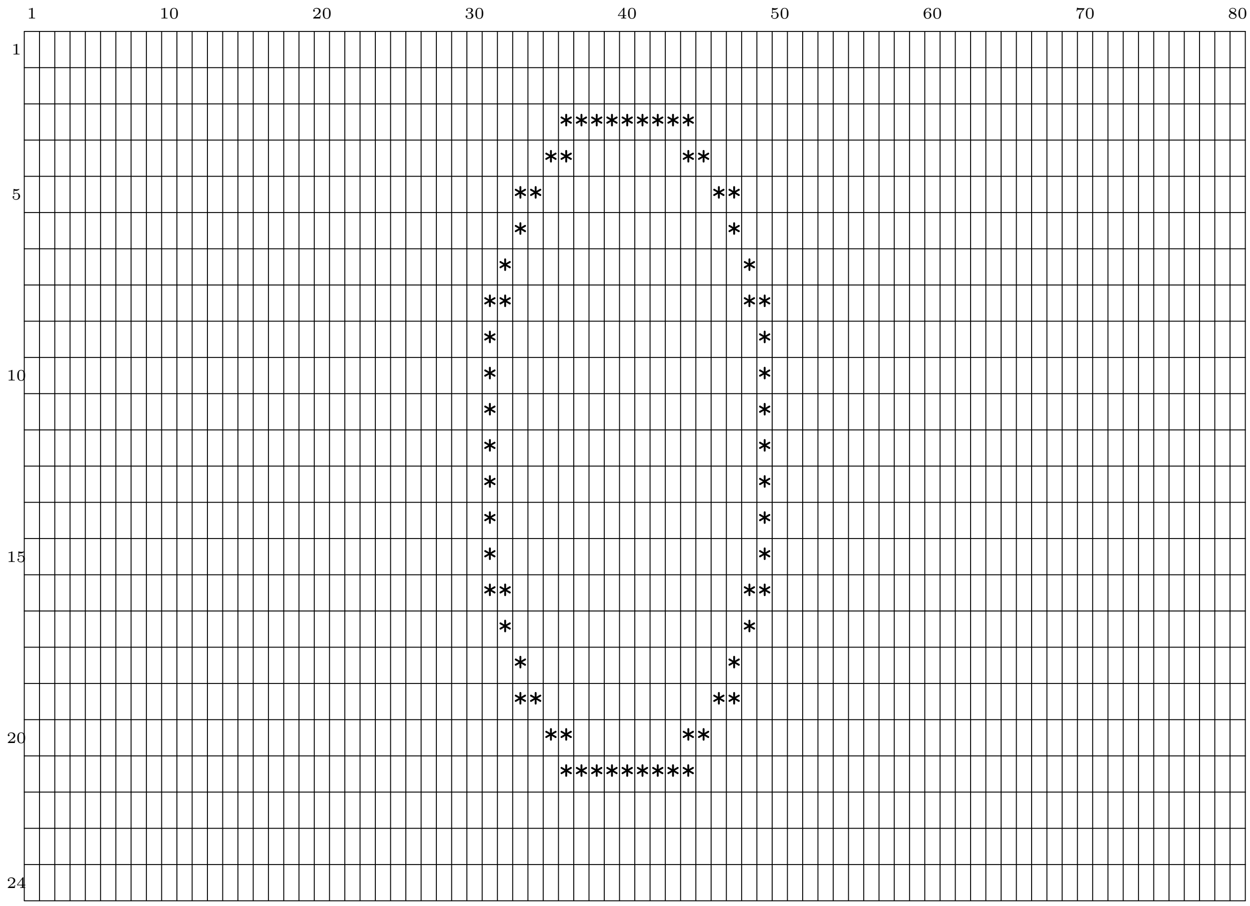
Variabelen blijven zichtbaar in "diepere" blokken:

```
{
        int x;

        x = 0;
        {
                int y;
                /* in dit blok blijft x zichtbaar */

                y = x;
        }
        /* hier is de variabele y alweer verdwenen */
        {
                int z;

                z = x;
        }
        /* hier kunnen we y en z niet meer gebruiken */
}
```

Meer over deze problematiek in §2.8.3.

2.4.3 Escape codes

Om eens een wat interessanter voorbeeld te kunnen geven (namelijk een soortement cirkel op het scherm tekenen) leg ik even uit hoe je de cursor op een bepaalde plaats op het scherm kunt zetten: met **escape sequenties** (escape sequences).



De "cirkel" die we willen op het scherm krijgen

De meeste terminals[3] reageren namelijk nogal speciaal als ze een *escape*-byte ontvangen. Escape is net als `\n` een **controlekarakter**; het wordt `\033` genoteerd in een C string[4]. Wat ze precies doen, hangt af van hetgeen volgt na die escape byte; ons interesseert de `\033[H` escape-sequentie. Als je namelijk

`\033[11;42H`

afdrukt, zal de terminal dat alles niet letterlijk op het scherm tonen (vanwege die 'escape `[`'), maar de cursor op de 42e positie van de 11de regel plaatsen. Op die manier kan je de cursor sturen naar waar je maar wil. Net zoals vroeger blijft de cursor op die plek staan, klaar voor een volgende `printf()`.

Escape sequenties zijn nogal veelzijdig; zo kan je er vette, cursieve, knipperende of gekleurde letters mee maken, het scherm wissen, . . . naargelang de mogelijkheden van de terminal die je gebruikt.

Om een cirkel te kunnen maken moeten we nu alleen nog maar iets weten over

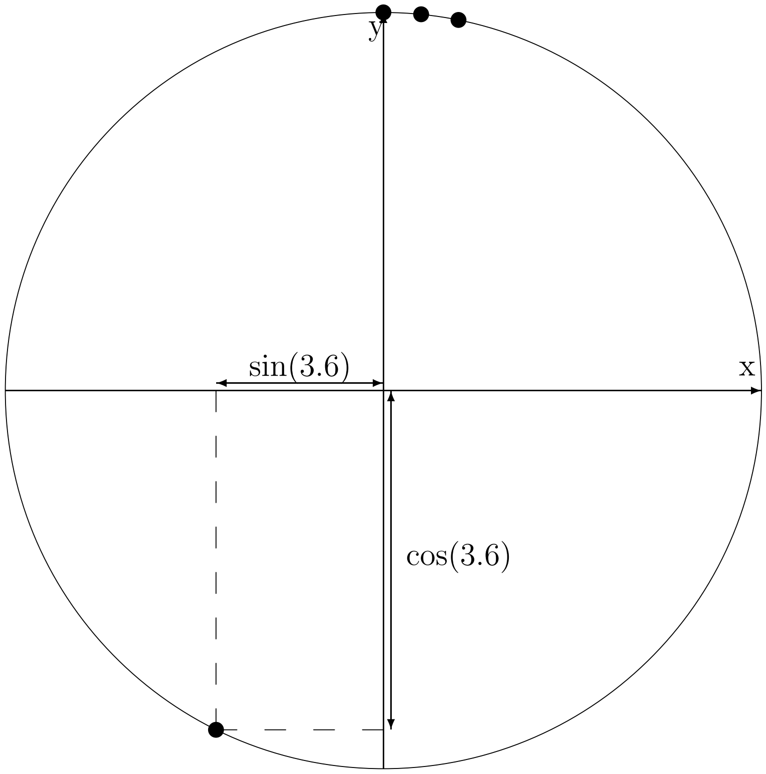
[3]Een terminal is een raar soort teeveetje met een raar soort schrijfmachineklavier eraan. Soms is er zelfs een heel raar soort muis bij, maar dat is optioneel. Af en toe worden deze terminals gebruikt door rare soorten van mensen, zoals *coders* of *users*.

[4]De reden hiervoor is dat de ASCII-code van escape 27 is, en in het achttallige stelsel is dat 33. Zie ook bijlage A.

2.4.4 `math.h`, `sin()` en `cos()`

Net zoals `printf()` in `stdio.h` te vinden is, bevat `math.h` een aantal wiskundige functies, waaronder `sin()` en `cos()`. Beide hebben ze een `double` argument nodig en geven ze ook een `double` terug, namelijk de sinus of cosinus van het argument (in radialen).

Het enige dat ons interesseert aan `sin()` en `cos()` is dat ze de coördinaten van een punt van een cirkel met straal 1 en middelpunt (0,0) opleveren wanneer je in beide hetzelfde getal stopt. Met andere woorden, de punten $(\sin(0), \cos(0)), (\sin(0.1), \cos(0.1)), \ldots$ liggen allemaal op die cirkel, en we zijn rond bij $(\sin(2\pi), \cos(2\pi))$. (Ik tel hier in stapjes van 0.1, en $2\pi \approx 6.283$, dus levert die methode 62 punten van de cirkel op. Als dat er te weinig zouden zijn, tel je natuurlijk met kleinere stapjes.) In plaats van in graden te rekenen (van 0 tot 360) werken `sin()` en `cos()` met zgn. **radialen** (van 0 tot 2π).

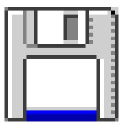


Op de figuur: de punten $(\sin(0), \cos(0))$, $(\sin(0,1), \cos(0,1))$, $(\sin(0,2), \cos(0,2))$ en $(\sin(3,6), \cos(3,6))$.

Het scherm heeft niet (0,0) als middelpunt maar (ik ga er even van uit dat je een klassiek 80×24 scherm gebruikt) (40,12), en een cirkeltje met straal 1 zou ook niet veel te zien geven, dus we maken daar straal 10 van. Het blijkt dat punten op de cirkel met middelpunt (40,12) en straal 10 gegeven worden door $(40 + 10 \times \sin(0), 12 + 10 \times \cos(0)), (40 + 10 \times \sin(0.1), 12 + 10 \times \cos(0.1)), \ldots$ (Als je wil weten waarom: vraag dat maar eens aan een vriendje dat wiskunde doet :-))

2.4.5 Een cirkel op het scherm

Na al dat gedoe kunnen we eindelijk nog eens een programma schrijven:



cirkel1.c

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

```
void main(void)
{
        double teller;

        teller = 0.0;
        while (teller <= 6.3) {
                /* Cursor naar de juiste plek en een sterreke afdrukken */
                printf("\033[%d;%dH*", 12 + (int)(10*sin(teller)),
                        40 + (int)(10*cos(teller)) );
                teller = teller + 0.01;
        }
}
```

Experimenteer een beetje met de stapgrootte (ik heb 0.01 genomen, wat 628 sterretjes oplevert, waarvan er een aantal elkaar overlappen, dus probeer bijvoorbeeld eens 0.05, 0.1 of 0.2 en kijk wat er gebeurt).

In dit programma staat nog iets nieuws: `(int)(10*sin(teller))`. Als ik gewoon `10*sin(teller)` had geschreven, was het resultaat van het type `double`. Een `int` (`10`) vermenigvuldigd met een `double` (het resultaat van `sin()`) geeft immers een `double`, omdat een `int` altijd in een `double` past maar niet andersom; m.a.w. er gaat zo weinig mogelijk informatie verloren. En in het formaatargument van `printf` staat `%d` (als we iets als `\033[12.432;23.3H` zouden sturen naar de terminal, zou die niet de cursor op de 12,432e regel en de 23,3e kolom zetten, maar waarschijnlijk flink tilt slaan ...), dus moet ik van die `double` op een of andere manier een `int` maken. Dat gebeurt zoals je ziet door er `(int)` voor te schrijven. De compiler gooit het gedeelte na de komma weg (er wordt dus niet afgerond).

Algemeen kan je een *uitdrukking* naar een bepaald *type* (hier dus `int`) omzetten met

(*type*) *uitdrukking*

wat een **type cast** (kortweg **cast**) genoemd wordt. Een string naar een `int` casten doet niet wat je zou verwachten; het resultaat is onbepaald. (Waarom dit zo is, heeft ook weer te maken met het feit dat een string in C eigenlijk een soort pointer is—zie hoofdstuk 3.)

Om een string naar een `int` om te zetten kan je de functie `atoi()` gebruiken (die in `stdlib.h` zit); zo geeft `atoi("25")` als resultaat de `int` `25`. Om een string naar een `double` om te zetten, kan je `atof()` gebruiken.

En om een string naar een `long` om te zetten, is er nog `atol()`.

De omgekeerde omzetting is ook mogelijk met `sprintf(str,"%d",getal)`. Het hoe en waarom zal je pas in §5.3.1 duidelijk worden.

Bij veel compilers zitten de functies met kommagetallen (zoals `sin()` en `cos()`) in een aparte **functiebibliotheek** (**library**). Bij de GNU C compiler moet je bij het compileren dan ook expliciet aangeven dat je van de math library gebruik maakt met de optie `-lm`:

```
gcc cirkel.c -lm -o cirkel
```

2.5 Macro's: nadere kennismaking met de preprocessor

The other kids at school nicknamed him Ix,

which in the language of Betelgeuse Five translates as
'boy who is not able satisfactorily to explain what a Hrung is,
nor why it should choose to collapse on Betelgeuse Seven'.
—Douglas Adams, "The Hitch Hiker's Guide to the Galaxy"

Het programma uit de vorige paragraaf is in feite nogal lelijk geschreven: alle constanten (breedte en hoogte van het scherm, straal van de cirkel) zitten verborgen in het programma zelf. Wanneer je bijvoorbeeld de straal van de cirkel wilt veranderen in 8, moeten een aantal 10's in 8 veranderd worden. Je moet dus het hele programma uitpluizen voor zo'n onnozele verandering; in grote programma's met duizenden regels is deze aanpak natuurlijk fataal. Daar heeft C iets voor: de **preprocessor**.

Nog vóór de C listing aan de eigenlijke C compiler doorgegeven wordt, werkt de preprocessor zich er door. Elke regel die met `#` begint, bevat een aanwijzing voor de preprocessor. Die regels slikt de preprocessor dan ook in (i.e. de C compiler zal ze nooit te zien krijgen). Een handige en veel gebruikte **preprocessor-aanwijzing** is `#define`. De regel

```
#define MACRONAAM vervangtekst
```

heeft als effect dat de preprocessor in al wat volgt (behalve in regels die met `#` beginnen) het woord `MACRONAAM` zal vervangen door `vervangtekst`. Er wordt ook wel eens gezegd dat `MACRONAAM` een **macro** is. De `vervangtekst` mag uit meer dan één woord bestaan (m.a.w. er mogen spaties in voorkomen).
De preprocessor is met andere woorden in staat een "zoek en vervang" operatie door te voeren zoals die ook voorkomt in tekstverwerkers. Macronamen voldoen weer aan de typische regels: alleen letters, cijfers en underscores zijn toegelaten, en de naam mag niet met een cijfer beginnen.

Een voorbeeldje voor Pascal-freaks:

```
#include <stdio.h>

#define WHILE while (
#define DO )
#define BEGIN {
#define END ; }
#define PROCEDURE void
#define INTEGER int
#define write printf

PROCEDURE main(void)
BEGIN
        INTEGER teller;

        WHILE teller < 10 DO
        BEGIN
                write("Hello");
                teller = teller + 1
        END
END
```

Het is de gewoonte de macronamen in hoofdletters te schrijven, zodat je duidelijk ziet dat het om een macro gaat.

Merk op dat ik niet

```
#define := =
```

kon doen (om dan natuurlijk `teller := teller + 1` te kunnen schrijven) omdat `:=` geen geldige macronaam is.

▸ **OEFENING 2.13**
Speel zelf voor preprocessor: schrijf uit wat de preprocessor van het bovenstaande Pascal-achtige programma maakt.

◊

▸ **OEFENING 2.14**
Deze versie van `write` werkt niet zoals bedoeld met alle strings. Schrijf een versie van de `write`-macro die geen problemen geeft.

◊

Zoals je ziet, trekt de preprocessor zich helemaal niets van de syntax van C aan; in het bijzonder hoeft het aantal geopende haakjes of accolades in de vervangtekst niet gelijk te zijn aan het aantal gesloten haakjes of accolades. We kunnen dus rustig `BEGIN` laten vervangen door een enkele accolade.

Het vervangen gebeurt niet in strings (dus `writeln("BEGIN")` drukt wel degelijk `BEGIN` en niet `{` af) of in commentaren. Commentaren in `#define`s hebben ook geen effect. (In feite is het allereerste dat de preprocessor doet, de commentaren vervangen door een spatie.) In tegenstelling tot tekstverwerkers vervangt de preprocessor alleen hele woorden; als het woord `DONKER` in het programma zou voorkomen, blijft de preprocessor er af (en maakt er niet `)NKER` van).

Het cirkelprogramma kan je dus als volgt opkuisen:

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

#define SCHERM_BREED  80
#define SCHERM_HOOG   24
#define STRAAL        10
#define STAP          0.01
#define PI            3.141592

void main(void)
{
        double teller;

        teller = 0.0;
        while (teller <= 2*PI) {
                /* Cursor naar de juiste plek en een sterreke afdrukken */
                printf("\033[%d;%dH*",
                        SCHERM_HOOG/2 + (int)(STRAAL*sin(teller)),
                        SCHERM_BREED/2 + (int)(STRAAL*cos(teller)) );
                teller = teller + STAP;
        }
}
```

Merk op dat ik niet `(int)(SCHERM_HOOG/2)` schrijf: de deling van twee `int`s geeft weer een `int` terug, waarbij het deel na de komma afgekapt wordt. Als je toch het deel na de komma wilt hebben, m.a.w. je wilt een `double` resultaat, kan je bijvoorbeeld `SCHERM_HOOG/2.0` of `((double)SCHERM_HOOG)/2` schrijven. Als immers één van de twee argumenten van `/` een `double` is, is het resultaat ook weer een `double` (net zoals dat bij vermenigvuldigingen ook het geval is—zie §2.4.5 en §4.3.1).

▸ **OEFENING 2.15**
Wat is het verschil met `(double)(SCHERM_HOOG/2)`?

◊

Het is misschien interessant even op te merken dat het programma niet minder efficiënt wordt door `2*PI` in plaats van `6.283184` te schrijven. De compiler rekent dat zelf uit; wanneer het programma draait, moet dus niet telkens een vermenigvuldiging gemaakt worden. De berekening gebeurt dus **at compile time** (tijdens het compileren) en niet **at runtime** (tijdens het draaien van het programma). Over `PI` gesproken: in `math.h` is de macro `M_PI` gedefinieerd, dus ik had die even goed kunnen gebruiken.

▸ **OEFENING 2.16**
Een fout die bijna iedereen wel eens gemaakt heeft is

```
#define PI 3.141592;
```

(dus met een kommapunt achteraan). Leg uit wat voor effect dat heeft in het programma van daarnet, en meer in het bijzonder: leg uit waarom het programma niet gecompileerd zal worden.

◊

2.6 Meer over in- en uitvoer

C heeft een aantal functies voor uitvoer (d.i. tekst op het scherm zetten) en invoer (iets van het toetsenbord inlezen). Ik heb al een aantal uitvoerfuncties behandeld, die ik hier even zal herhalen, en ik zal ook laten zien hoe de invoer geregeld wordt in C. In- en uitvoer wordt ook wel eens **I/O** (input/output) genoemd.

In- en uitvoer wordt volledig uitgediept in hoofdstuk 5.

2.6.1 Uitvoer

In paragraaf 2.3 heb ik al getoond dat je met `printf()` `int`s (zet `%d` in de formaatstring), `double`s (`%f`), `char`s (`%c`) en strings (`%s`) kan afdrukken. Verder is er nog `puts()`, dat een string afdrukt gevolgd door een regelovergang; m.a.w. `puts(str)` doet net hetzelfde als `printf("%s\n",str)`. (In tegenstelling tot `printf()` kan `puts()` maar één argument hebben). Eén enkele letter uitvoeren gaat met `putchar()`; zo doet `putchar('\n');` net hetzelfde als `printf("\n");` (en een beetje sneller).

▸ **OEFENING 2.17**
Waarom werkt `printf(str)` niet altijd goed (`str` is natuurlijk een variabele die een string bevat) en hoe kan je dan wel een string afdrukken?

◊

▸ **OEFENING 2.18**
Schrijf zelf een `putchar()` functie, die gebruik maakt van `printf()`.

◊

▸ **OEFENING 2.19**
Waarom is `putchar('\n')` sneller dan `printf("\n")`?

◇

2.6.2 Invoer

Het "tegengestelde" van `printf()` in C is `scanf()`. De functie `scanf()` leest invoer van het toetsenbord. Ook hier is het eerste argument een formaatstring, die bepaalt wat er met alle andere argumenten zal gebeuren. Bijvoorbeeld:

```
/* eerst wat variabelen maken */
char str[100];
char letter;
float reeel;
int geheel;

scanf("%s",str);
```

leest een string in van het toetsenbord en plaatst die in de stringvariabele `str`, die hopelijk groot genoeg is om alle invoer te kunnen bevatten—zoniet probeert `scanf` de bytes voorbij het einde van `str` te wijzigen en is de kans groot dat het programma klemloopt (**crash**t). De functie `scanf()` slaat witte ruimte vóór de ingelezen string over. Dus als de gebruiker twee spaties tikt en dan een woord, komen die spaties niet voor in `str`. `scanf()` blijft net zolang lezen tot er witte ruimte volgt. Als de gebruiker dus "`dit is een test`" tikt, zal de `scanf()` opdracht de string `"dit"` in `str` plaatsen. Van rest van de invoer (m.a.w. "` is een test`") is `scanf()` af gebleven (merk op dat `scanf()` de spatie na `dit` *niet* heeft ingelezen); die kan je met een volgende `scanf()` opdracht verwerken.

Analoog heb je

```
scanf("%c",&letter);
scanf("%f",&reeel);
scanf("%d",&geheel);
```

Of toch niet helemáál analoog, want hier is een extra `&` ("ampersand") opgedoken. De precieze betekenis zal je pas te weten komen in het hoofdstuk over pointers (§3.3), en zoals je ziet nemen de strings ook hier weer een bijzondere plaats in doordat hier geen `&` nodig is. Voorlopig zal je er dus moeten aan denken altijd `&` te schrijven bij `scanf()`, behalve bij string-argumenten.

OPGELET: als je een `&` te veel of te weinig schrijft, zal het programma waarschijnlijk op dat punt klemlopen. Je bent gewaarschuwd ... Gelukkig kunnen slimme C compilers het gebruik van ampersands in `scanf()` controleren.

Alle `scanf()`-varianten slaan witte ruimte aan het begin van de invoer over, behalve de `%c` vorm. Bij `%f` en `%d` verwerkt `scanf()` zo veel mogelijk invoer; als de invoer niet met een getal begint, stopt `scanf()`. Je kan net zoals bij `printf()` meerdere `%`-tekens in het formaatargument gebruiken. Onderstel dat de gebruiker bijvoorbeeld "`1234.5 .6`" intikt. Een `scanf("%f%d%c",&floot,&dubbel,&kar");` zal dan eerst `1234.5` in `floot` stoppen. Daarna blijft er nog "` .6`" te verwerken. Eerst slaat `scanf()` de spatie over, maar met de rest van de invoer valt geen geheel getal mee te maken, dus stopt `scanf()` (en blijft van de inhoud van `dubbel` en `kar` af), en de invoer "`.6`" is niet verwerkt.

Het "tegengestelde" van `putchar()` is `getchar()`:

```
        char letter;

        letter = getchar();
```

leest één `char` van het toetsenbord.

De tegenhanger van `puts()` is de functie `gets()`, die een regel van het toetsenbord (d.i. tot er op return gedrukt wordt) inleest:

```
        char str[80];

        gets(str);
```

De functie `gets()` leest alle invoer tot en met het eerstvolgende nieuwe-regel teken. Het nieuwe-regel teken wordt echter niet in de string geplaatst.

Merk op dat ik in §2.2 zei dat een functie geen variabelen van de aanroeper kan wijzigen. Nochtans is dat precies wat `gets()` en `scanf()` hier schijnen te doen. De oplossing van dit raadsel zal in het hoofdstuk over pointers onthuld worden.

▸ **OEFENING 2.20**

Het volgende programma werkt wel, maar het is nogal "gevaarlijk" geprogrammeerd:

gevaarlijktotaal.c

```
#include <stdio.h>

void main(void)
{
        float getal;
        float totaal;

        getal = 1;
        while (getal != 0.0) {
                puts("Tik een getal in (0 om te stoppen)");
                scanf("%f",&getal);
                totaal = totaal + getal;
        }
        printf("Totaal: %f\n",totaal);
}
```

De opdracht is voor de hand liggend: bedenk wat er hier mis kan gaan …

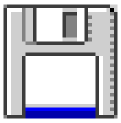
… en verbeter.

◇

2.6.3 Voorbeeld

Most of the others secretly believe that the decision-making process is handled by a computer. They couldn't be more wrong.
—Douglas Adams, "The Hitch Hiker's Guide to the Galaxy"

Laten we om maar eens iets te doen een programma maken dat een tekst inleest, en diezelfde tekst afdrukt waarbij alle `g`'s een `h` worden en omgekeerd. Het programma stopt als je een `&` intikt.


bruhhe.c

```
#include <stdio.h>

void main(void)
{
        while (1 == 1) {
                char c;

                c = getchar();        /* scanf ("%c",&c); zou ook gaan */
                if (c == 'g') {
                        putchar('h');
                } else if (c == 'h') {
                        putchar('g');
                } else if (c == '&') {
                        break;
                } else {
                        putchar(c);
                }
        }
        /* break springt hierheen */
}
```

Hier komen we eerst de `if` en `else` sleutelwoorden tegen. Net als `while` zijn het ingebakken woorden van C, en volgt na `if` een uitdrukking tussen haakjes:
`if (` *voorwaarde* `)` *opdracht*

Wanneer de processor in de loop van het programma een `if`-opdracht tegenkomt, wordt gecontroleerd of de *voorwaarde* waar is. *Als* dat het geval is, wordt de *opdracht* uitgevoerd. Is de voorwaarde vals, dan gebeurt er niets. In beide gevallen gaat het programma verder met de volgende opdracht.

Samen met `if` kan ook optioneel een `else` voorkomen:
`if (` *voorwaarde* `)` *waar-opdracht*
`else` *vals-opdracht*

Als de *voorwaarde* waar is, wordt de *waar-opdracht* uitgevoerd; *anders* de *vals-opdracht*. Het programma gaat dan weer verder op de opdracht na de *vals-opdracht*. Zoals altijd in C mag je de *waar-* of *vals-opdracht* vervangen door een blok (zie §2.4.1), dat eventueel weer `if`, `while`, . . . kan bevatten.

Merk op dat als ik alle blokken voluit schrijf, het bovenstaande er zo komt uit te zien:

```
                if (c == 'g') {
                        putchar('h');
                } else {
                        if (c == 'h') {
                                putchar('g');
                        } else {
                                if (c == '&') {
                                        break;
                                } else {
                                        putchar(c);
                                }
```

```
        }
}
```

Dit is werkelijk géén zicht, en het is nog behoorlijk onduidelijk ook. (Bovendien zou het blad al snel te smal worden ...) Daarom worden `else if`s zelden zo extreem doorgeïndenteerd als ik daarnet deed. Iets duidelijker wordt het daarentegen als je de extra accolades weglaat:

```
if (c == 'g')
        putchar('h');
else if (c == 'h')
        putchar('g');
else if (c == '&')
        break;
else
        putchar(c);
```

want zoals al uitgelegd in §2.4.1 mag je bij een blok met maar één opdracht de accolades weglaten. Wil je minder gezigzag in de linkermarge, dan kan je zelfs dit schrijven:

```
if (c == 'g') putchar('h');
else if (c == 'h') putchar ('g');
else if (c == '&') break;
else putchar c;
```

Er bestaat dus niet iets als een `elseif` sleutelwoord.

Ook nieuw is `break`. Het effect van `break` is dat het programma de lus verlaat waar het in zit. In het voorbeeld wordt dus gesprongen naar de opdracht na het `while` blok, als die er zou geweest zijn; in dit geval dus naar het einde van `main()`.

Als de `break` in een lus binnen een andere lus staat, heeft `break` altijd betrekking op de binnenste lus:

```
while (x < y) {
        if (z > 10) break;              /* springt naar B */
        while (y < z) {
                if (x < y + z) break;   /* springt naar A */
                z = z + x-y;
        }
        /* A */
        if (x > y+2) break;             /* springt naar B */
}
/* B */
```

Een `break` wordt veel gebruikt om uit een oneindige lus te springen: ik heb in het eerste voorbeeld een voorwaarde die altijd waar is, `1 == 1`, gebruikt, zodat de lus nooit verlaten wordt doordat de voorwaarde bij `while` vals wordt, wat zonder `break` inderdaad een oneindige lus zou opleveren. Je ziet hier ook nog eens (zie §2.4) het verschil tussen `=` (stop een waarde in een variabele) en `==` (vergelijk twee getallen met elkaar). Het is van belang die twee niet door elkaar te haspelen—maar je zult het bijna gegarandeerd wel eens doen; de eerste paar keren dat zoiets gebeurt zal je je vrijwel zeker suf zoeken waar die fout nu weer in geslopen is ...

Als je dit programma draait, zul je zien dat het niet genoeg is op `&` te drukken om het programma te stoppen; je moet ook nog eens een return geven. Alle invoer wordt immers

regel gebufferd, wat betekent dat het C programma de invoer regel per regel ziet. Merk ook op dat de ingetikte invoer vanzelf op het scherm verschijnt (wat **echo** genoemd wordt). Hoe je dat gedrag kan veranderen (en de invoer direkt teken per teken kan verwerken), wordt uitgelegd in §5.7.

Het zou geen slecht idee zijn nu oefening 2.8 eens te bekijken.

2.7 Functies die een resultaat teruggeven

Intussen zijn we al enkele functies tegengekomen die een **resultaat** (ook wel eens **terugkeerwaarde** of **return value** genoemd) teruggeven: `sin()` en `cos()` geven een `double` terug, en `getchar()` een `char`. Het wordt stilaan tijd om zelf eens zo'n functie in elkaar te steken. Ik zal een variant op `sin()` in elkaar knutselen waarbij je als argument niet de hoek in radialen (van 0 tot 2π) maar in graden (van 0 tot 360) moet opgeven. Wat ik in feite doe is het argument `graden` delen door 180 en vermenigvuldigen met π, zodat 360 graden effectief tot $\frac{360}{180}\pi = 2\pi$ radialen wordt omgerekend:

```
#include <stdio.h>

#define PI 3.14159265

double Sin(double graden)
{
        double radialen;

        radialen = graden / 180.0 * PI;
        return sin(radialen);
}
```

(verderop in het programma moet natuurlijk nog een `main()` volgen).

Ik heb hierboven een functie `Sin()` gemaakt die een `double` teruggeeft. De laatste opdracht is `return`, ook een ingebakken C sleutelwoord, dat als effect heeft dat de functie beëindigd wordt. Al onze vorige functies eindigden gewoon door "van de rand te vallen"; hier hebben we een middel nodig om aan te geven wat het resultaat van de functie precies is. De syntax is

`return` *uitdrukking*;

Bemerk dat de *uitdrukking* niet tussen haakjes moet staan (het mag natuurlijk wel—want je mag altijd extra haakjes schrijven—en het wordt vaak gedaan, zodat de return-opdracht er als een functieaanroep uit komt te zien). Het effect van de `return`-opdracht is: de *uitdrukking* wordt berekend, waarna de functie verlaten wordt, met de waarde van de *uitdrukking* als functieresultaat.

▸ **OEFENING 2.21**
Voor wie hoofdstuk 6 gelezen heeft: schrijf een macro-versie van `Sin()`.

◇

In functies die niets teruggeven (met enig spraakmisbruik wordt wel eens gezegd: functies die `void` teruggeven) kan je ook `return` gebruiken, maar dan zonder uitdrukking erachter. Op die manier kan je de functie op een willekeurige plaats afbreken:

```
void deel(double a,double b)
{
        if (b == 0.0) return;    /* delen door nul gaat niet */
        printf("%f\n", a/b);     /* wordt niet uitgevoerd als
                                  * b de waarde 0.0 bevatte */
}
```

Je ziet dat als `b` niet nul is, de functie `deel()` beëindigd wordt door "van de rand te vallen". Je mag de functie ook zo schrijven:

```
void deel(double a,double b)
{
        if (b == 0.0) return;
        printf("%f\n", a/b);
        return;
}
```

om expliciet aan te geven dat de functie daar eindigt.

Bij functies die iets anders teruggeven dan `void` is een `return` aan het eind verplicht; als de functie toch van de rand zou vallen, is het resultaat van die functie een onbepaalde waarde. GNU C kan hiervoor een `control reaches end of non-void function` waarschuwing geven (zie bijlage E.2).

Merk op dat een functie maar één resultaat kan teruggeven. Dat zou nochtans handig kunnen zijn in sommige gevallen—bijvoorbeeld als je een functie wil maken die tegelijkertijd het aantal medeklinkers en het aantal klinkers van een string telt (zie §3.3.3). Deze beperking kan, zoals we later zullen zien, met pointers omzeild worden.

2.7.1 `main()` opnieuw bekeken

Tot nog toe heb ik alle `main()` functies geen resultaat laten teruggeven. Maar eigenlijk wordt `main()` verondersteld een `int` terug te geven die een **foutcode** bevat: nul betekent geen fout, een andere waarde betekent dat er ergens een fout is opgetreden. Die foutcode wordt gebruikt door de shell van waaruit je programma werd opgestart.

Vanaf nu zal ik dus netjes alle programma's schrijven zoals het hoort:

```
int main(void)
{
        /* het eigenlijke programma komt hier */

        if (er_ging_iets_fout) return 10;                 /* foutcode */
        else if (er_ging_iets_anders_fout) return 20;     /* foutcode */
        else return 0;     /* alles OK */
}
```

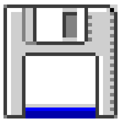
In feite schrijft de ANSI C-standaard geen bepaald returntype van `main()` voor, maar beveelt enkel aan dat `int` zou werken. Veel compilers ondersteunen naast `int main(void)` ook de `void main(void)` variant.

2.8 Globale en lokale variabelen; zichtbaarheid

2.8.1 Globale en lokale variabelen

Alle variabelen die we tot nog toe gemaakt hebben, waren **lokaal** voor een bepaalde functie (of een bepaald blok in een functie): ze werden gemaakt vlak vóór het begin van de functie (resp. het blok) en verdwenen bij het beëindigen ervan.

In tegenstelling daarmee kan je ook **globale variabelen** maken. Deze variabelen worden gemaakt vlak vóór het begin van het programma en blijven bestaan zolang het programma loopt. Bijvoorbeeld:



lokaalglobaal.c

```
#include <stdio.h>

int globaal;

void drukAf(void)
{
        int lokaal;

        lokaal = 15;
        printf("lokaal=%d, globaal=%d\n",lokaal,globaal);
        globaal = lokaal;
}

int main(void)
{
        int lokaal;

        lokaal = 10;
        globaal = 20;
        drukAf();
        printf("lokaal=%d, globaal=%d\n",lokaal,globaal);
        return 0;
}
```

Zoals je ziet, worden globale variabelen gedeclareerd *buiten* het blok van een functie. In het voorbeeld zijn twee verschillende variabelen gemaakt die `lokaal` heten: eentje in `main()` en eentje in `DrukAf()`. Beide functies kunnen mekaars lokale variabelen niet veranderen, maar wel de globale variabelen. Het resultaat van het programmaatje zou dus zijn:

```
lokaal=15, globaal=20
lokaal=10, globaal=15
```

In tegenstelling tot lokale variabelen worden globale variabelen wél vanzelf geïnitialiseerd: `int`s en `char`s krijgen de waarde `0`, en `double`s de waarde `0.0`. Strings worden opgevuld met nulbytes.

Het is niet zo'n goede stijl globale variabelen te gebruiken. Het nadeel van globale variabelen is immers dat je zeker bij grote programma's op den duur niet goed meer kunt weten welke functie welke globale variabelen verandert of gebruikt. Het is veel properder om alles wat een functie nodig heeft via de argumenten ervan door te geven.

2.8.2 Functie-argumenten

Functie-argumenten gedragen zich als lokale variabelen die bij het begin van de functie met de juiste waarden gevuld worden (zie §2.2 voor een voorbeeld). String-argumenten worden ook hier bijzonder behandeld: de strings worden **by reference** (zie ook §2.2) doorgegeven, hetgeen betekent dat de functie *niet* op een eigen kopie van de string werkt. Het gevolg hiervan is dat wanneer een functie de inhoud van de string wijzigt, de aanroeper van die functie de wijzigingen ook ziet. (De precieze reden hiervoor zal in het hoofdstuk over pointers duidelijk worden.)

Als binnen een functie een string gewijzigd wordt, blijven de wijzigingen aan de string dus voor de aanroepende functie zichtbaar:

```
void wisString(char str[])
{
        str[0] = 0;
}

void test(void)
{
        char mijnString[20];

        strcpy(mijnString,"de boze wolf");
        /* zal afdrukken:  mijnString bevat: de boze wolf.
        printf("mijnString bevat: %s.\n",mijnString);
        wisString();
        /* zal afdrukken:  mijnString bevat: .
         * dus wisString() heeft de inhoud van mijnString
         * daadwerkelijk gewijzigd! */
        printf("mijnString bevat: %s.\n",mijnString);
}
```

Je kan ook in dit voorbeeld zien hoe je een functie met een string-argument kunt maken. In §3.2.3 zal ik daar wat meer uitleg over geven.

2.8.3 Zichtbaarheid

2.8.3.1 Zichtbaarheid van variabelen

Met **zichtbaarheid** van een variabele bedoelen we het gebied in de programmatekst waarbinnen een bepaalde variabele gebruikt mag worden. Een lokale variabele is bijvoorbeeld zichtbaar in het blok waarin ze gedeclareerd is:

```
void test(void)
{
        x = 0;  /* fout: x is hier niet zichtbaar */
}

void nogeen(void)
{
        int x;
```

```
        x = 5;  /* OK: x is hier zichtbaar */
}
/* vanaf hier is x weer onzichtbaar */
```

Globale variabelen zijn zichtbaar vanaf de plaats waar ze gedeclareerd zijn tot aan het eind van het programma:

```
void test(void)
{
        glob = 0;  /* fout: glob is hier nog niet zichtbaar */
}

int glob;          /* vanaf hier is glob zichtbaar */

void Test(void)
{
        glob = 20; /* OK */
}
```

Wat gebeurt er nu in situaties als

```
int x;     /* de "eerste" x */

void test(void)
{
        int x;     /* de "tweede" x */

        x = 1;
        {       /* dit is het "middelste" blok */
                int x;     /* de "derde" x */

                x = 2;
        }
        x = 3;
}
```

We hebben hier drie *verschillende* variabelen gemaakt, die wel allemaal de naam `x` hebben gekregen. Toen ik vroeger zei dat je geen twee variabelen met dezelfde naam mocht maken, heb ik dus een klein beetje gelogen: je mag geen twee variabelen met dezelfde naam maken *in hetzelfde blok*.

De regel in dit soort situaties is dat de variabele gebruikt wordt die het “kortst geleden” gedeclareerd is. Bij de opdracht `x = 1;` wordt dus de tweede `x` gebruikt, en bij `x = 2;` de derde `x`. De `x = 3` gebruikt de tweede `x` (de derde `x` is wel korter bij gedeclareerd, maar is niet meer zichtbaar: ze wordt immers vernietigd bij het afsluiten van het middelste blok). Binnenin `test()` is de eerste `x` dus niet bruikbaar, want de declaratie van de tweede `x` staat “in de weg”. Het spreekt vanzelf dat je in de praktijk zulke situaties maar beter vermijdt . . .

Er is dus een verschil tussen **levensduur** en **zichtbaarheid** van een variabele. Vaak stemmen ze met elkaar overeen: de derde `x` blijft leven zolang het programma in het middelste blok draait, wat precies de plaats is waarin `x` zichtbaar is. Maar de globale `x` leeft zolang het programma draait en is toch niet zichtbaar binnenin de functie `test()`.

2.8.3.2 Zichtbaarheid van functies

Net zoals variabelen mag je een functie maar gebruiken vanaf de plaats waar ze gedefinieerd is. In de praktijk kijkt de C compiler een beetje door de vingers: als je een nog niet eerder gedefinieerde functie gebruikt, neemt de compiler aan dat het returntype `int` is en dat alle argumenten het goede type hebben. Bijvoorbeeld:

```
#include <stdio.h>

int main(void)
{
        /* De compiler kan niet vooruitkijken ... en ziet de
         * definitie van drukaf() na de main() dus nog niet */
        drukaf(25);
}

void drukaf(double getal,double getal2)
{
        printf("%f\t%f",getal,getal2);
}
```

In de definitie van `main()` botst de compiler op de onbekende functie `drukaf()`, en veronderstelt dus maar dat die een `int` (het type van `25`) nodig heeft en ook een `int` teruggeeft. De GNU C compiler kan een waarschuwing geven bij de `drukaf(25);` opdracht als je de `-Wimplicit` optie van `gcc` gebruikt: `implicit declaration of function`.

Sterker nog, zelfs bij

```
int main(void)
{
        test(10);
        test("roodkapje",12.3);
}
```

veronderstelt de compiler dat je weet wat je doet en compileert rustig verder.

Dit gedrag van de compiler verklaart ook waarom je `printf()` kan gebruiken zonder `stdio.h` te includen: de compiler gaat ervan uit dat alles OK is, met alle risico's vandien.

Je kan het probleem in het vorige programmaatje oplossen door de volgorde van de functiedefinities om te wisselen, of door een **prototype** te gebruiken:

```
#include <stdio.h>

/* het prototype voor drukaf() */
void drukaf(double getal,double getal2);

int main(void)
{
        drukaf(25);
}

void drukaf(double getal,double getal2)
{
```

```
        printf("%f\t%f",getal,getal2);
}
```

Hierdoor weet de compiler dat er "ergens verderop" een functie `drukaf()` zal gedefinieerd worden die twee `doubles` nodig heeft. De compiler zal dan ook flink sputteren bij de `drukaf(25);` opdracht. Omdat de precieze variabelnamen in een prototype geen belang hebben, mag je ze weglaten, zodat het prototype van daarnet er zo komt uit te zien:

```
void drukaf(double,double);
```

Header files bevatten meestal een hele hoop zulke prototypes.
Je kan dus op twee manieren je functies ordenen:

- De C-stijl, met prototypes:

```
int ftie1(char);
int ftie2(char);

int main(void) { ... ftie1(letter); ... ftie2(iets); ... }

int ftie1(char c) { ... }

int ftie2(char c) { ... }
```

- De Pascal-stijl, waarbij een functie eerst volledig uitgeschreven wordt vooraleer ze ergens anders gebruikt wordt. In deze stijl moet je het programma "van onder naar boven" lezen:

```
int ftie1(char c) { ... }

int ftie2(char c) { ... }

int main(void) { ... ftie1(letter); ... ftie2(iets); ... }
```

Zoals zo vaak is het een kwestie van smaak welke stijl je gebruikt.

2.9 Willekeurige getallen

Als toepassing van het voorgaande zal ik de bibliotheekfuncties laten zien die nodig zijn om willekeurige getallen te genereren. Wie weinig tijd heeft, kan deze paragraaf overslaan.

Wat bedoel ik eigenlijk met een **willekeurig getal**? Ongeveer hetgeen er gebeurt bij het gooien van een dobbelsteen: ik zou een functie `dobbelsteen()` willen die een `int` tussen 1 en 6 teruggeeft, waarbij ik op voorhand niet kan voorspellen welk getal het zal zijn, en waarbij geen enkel getal opvallend meer of minder voorkomt dan de rest. Nu kan een computer niks anders dan braafjes de ene opdracht na de andere opdracht uitvoeren. Als ik dus het programma ken, dan kan ik alle instructies zelf nadoen, en dus kan ik altijd op voorhand weten welk getal de dobbelsteen-functie zal verzinnen! We moeten ons dus een beetje behelpen. Computer-gegenereerde willekeurige getallen zijn nooit echt willekeurig, maar **pseudo-willekeurig**. We zullen zien dat pseudo-willekeurig in de praktijk toch behoorlijk goed kan zijn. Zo'n dobbelsteen-functie wordt ook wel een **random-generator** genoemd.

Een eerste poging zou er zo kunnen uitzien:

dobbelsteen.c

```
int laatste; /* globaal; wordt dus op 0 gezet
              * bij de start van het programma */

int dobbelsteen(void) {
        laatste = laatste + 1;
        if (laatste == 7) laatste = 1;
        return laatste;
}
```

Deze functie geeft achtereenvolgens als resultaat 1, 2, 3, 4, 5, 6, 1, 2, 3, 4, 5, 6, ... Er is dus geen enkel getal dat bevoordeeld wordt, maar deze dobbelsteen is toch nogal erg voorspelbaar. In `stdlib.h` zit een iets gesofistikeerder random-generator `rand()` die een `int` tussen 0 en `RAND_MAX` teruggeeft. (Met *tussen* bedoel ik dat `RAND_MAX` een resultaat van `rand()` kan zijn, maar `RAND_MAX+1.0` niet meer.) `RAND_MAX` is een `#define` uit `stdlib.h`; de precieze waarde hiervan hangt af van systeem tot systeem (op de Linux machine waarop ik deze tekst schrijf meer dan twee miljard, ruimschoots genoeg om een dobbelsteen na te bootsen). Als we dat getal vermenigvuldigen met 6 en delen door `RAND_MAX+1.0`, dan krijgen we dus een willekeurig getal tussen 0 en $\frac{\texttt{RAND\_MAX}}{\texttt{RAND\_MAX+1.0}} \approx 5,99999$ (wat bij het omzetten naar `int` een getal tussen 0 en 5 oplevert). Hierbij 1 optellen geeft ons ook een bruikbare dobbelsteen-variant:

dobbelsteen2.c

```
#include <stdlib.h>

int dobbelsteen2(void) {
        return (int)(rand() * 6.0 / (RAND_MAX+1.0) )  +  1;
}
```

Ik maak hier een `double` omdat het niet zeker is dat `RAND_MAX*6` nog in een `int` past. Meestal is `RAND_MAX` ongeveer zo groot als de grootste waarde die nog net in een `int` past, dus vermenigvuldigen met 6 geeft al zeker een te groot resultaat. Daarom tel ik dan ook niet `1` maar `1.0` op bij `RAND_MAX`: het zou kunnen dat `RAND_MAX` werkelijk het grootste getal is dat in een `int` past.

▸ **OEFENING 2.22**
Waarom deel ik niet door `RAND_MAX` in plaats van `RAND_MAX+1.0`; m.a.w. waarom schrijf ik niet

```
int dobbelsteen3(void) {
        return (int)(rand()*5/RAND_MAX) + 1;
}
```

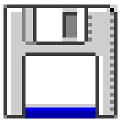
◇

We hebben nog altijd een probleem met `dobbelsteen()` en `dobbelsteen2()`: ze geven altijd *dezelfde* reeks waarden terug. Op mijn systeem geeft `dobbelsteen2()` steeds de waarden 1, 4, 3, 4, 2, 6, 5, 2, 4, 1, 5, 5, 6, 3, 3, 4, 1, 5, 3, ... Als ik het programma nog eens draai, krijg ik opnieuw precies dezelfde waarden. Bij `dobbelsteen()` gebruikte ik de variabele `laatste` om in bij te houden wat het laatste "gegooide" getal was. De functie `rand()` heeft intern ook zo'n variabele. Telkens als je `rand()` aanroept, wordt een nieuwe waarde berekend uit de oude. Het belangrijkste verschil met `dobbelsteen()` is dat de formule om de nieuwe waarde uit de oude te berekenen ingewikkelder en dus wat onvoorspelbaarder is.

Met `srand()` (ook uit `stdlib.h`) kan je de interne variabele van `rand()` instellen. Het enige probleem is dus een min of meer willekeurige startwaarde te vinden. Hiervoor kunnen we de functie `time()` uit `time.h` gebruiken: `time(NULL)` geeft een `int` die de tijd aangeeft in seconden sinds 1 januari 1970 middernacht. (De `NULL` heeft iets met pointers te maken en zal dus in een verder hoofdstuk duidelijk moeten worden. Voorlopig gewoon niks van aantrekken ...)

De magische truuk is dus om aan het begin van het programma de opdracht `srand(time(NULL))` te geven.

Als voorbeeld een raad-het-getal spelletje:



raadgetal.c

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

#define GROOTSTE 100

int main(void) {
        int hetgetal;

        printf("Raad het getal tussen 1 en %d.\n",GROOTSTE);

        /* Kies het te raden getal */
        srand(time(NULL));
        hetgetal = rand() * (float)GROOTSTE / (RAND_MAX+1.0) + 1;

        while (1 == 1) {
                int gok;

                scanf("%d",&gok);
                if (gok == hetgetal) {
                        puts("Proficiat!");
                        break;    /* einde programma */
                } else if (gok < hetgetal) {
                        puts("Te laag!");
                } else {
                        puts("Te hoog!");
                }
        }
        return 0;
}
```

Zie oefening 2.20 in verband met het gebruik van `scanf()`.

▸ **OEFENING 2.23**

Schrijf een programma om de hoofdbewerkingen (+, −, / en ∗) te oefenen. Het programma vraagt de gebruiker bijvoorbeeld `4 + 3 =`, leest de oplossing in en zegt of die goed of fout was.

◊

▸ **OEFENING 2.24**
Schrijf een functie die een array vult met willekeurige getallen. Hiermee zou je de sorteerroutines uit §3.1 kunnen testen door zo'n array dan te laten sorteren.

◊

Hoofdstuk 3

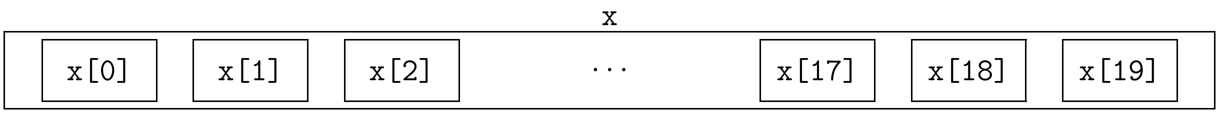
Arrays, pointers en structures

3.1 Arrays

Een **array** is niets meer of minder dan het computerees voor een tabel. Je bent in feite al arrays van `char`s tegengekomen; ik heb ze toen **string**s genoemd. Op dezelfde manier kan je arrays maken van om het even welk datatype (wel, behalve `void` natuurlijk):

```
int main(void)
{
        int x[20];
        int matrix[10][5];
        char namen[10][40];

        return 0;
}
```

maakt een tabel van twintig `int`s, een tabel van tien tabellen van elk vijf `int`s, en een tabel van tien strings van lengte 40 (wat hetzelfde is als een tabel van tien tabellen van 40 `char`s).



De eendimensionale array `x`

Je kan zo trouwens doorgaan en bijvoorbeeld `int y[4][10][3][18][2];` of zo schrijven. Bedenk wel dat `y` dan in totaal $4 \times 10 \times 3 \times 18 \times 2 = 4320$ `int`jes bevat: een **meerdimensionale array** kan al snel heel wat geheugen vreten . . .

Terug met onze voetjes op de grond: eens kijken wat we met `x` zoal kunnen doen. Net als bij strings kan je de variabelen `x[0]` tot en met `x[19]` gebruiken. Een stukje C dat de hele tabel met nullen vult ziet er bijvoorbeeld zo uit:

```
        int teller;

        teller = 0;
        while (teller < 20) {
                x[teller] = 0;
                teller = teller + 1;
        }
```

Denk erom dat arrays, net als gewone variabelen, niet geïnitialiseerd zijn bij de declaratie! De tabel van volgorde omkeren gaat zo:

```
int teller;

teller = 0;
while (teller < 10) {
        int wissel;

        /* verwissel x[teller] en x[19-teller] */
        wissel = x[teller];
        x[teller] = x[19-teller];
        x[19-teller] = wissel;

        teller = teller + 1;
}
```

`matrix` is een tweedimensionale array, die je je kan voorstellen als een tabel van 10 bij 5 gehele getallen.

`matrix`

`matrix[0]`	`matrix[1]`		`matrix[9]`
`matrix[0][0]`	`matrix[1][0]`		`matrix[9][0]`
`matrix[0][1]`	`matrix[1][1]`		`matrix[9][1]`
`matrix[0][2]`	`matrix[1][2]`	...	`matrix[9][2]`
`matrix[0][3]`	`matrix[1][3]`		`matrix[9][3]`
`matrix[0][4]`	`matrix[1][4]`		`matrix[9][4]`

De tweedimensionale array `matrix`

Vullen met nullen kan je zo doen:

```
int teller1;

teller1 = 0;
while (teller1 < 10) {
        int teller2;

        teller2 = 0;
        while (teller2 < 5) {
                matrix[teller1][teller2] = 0;
                teller2 = teller2 + 1;
        }
        teller1 = teller1 + 1;
}
```

en met de array van strings `namen` kan je bijvoorbeeld `strcpy(namen[8],"geert"));` of `strcpy(namen[5],namen[8]);` doen.

▸ **OEFENING 3.1**
Schrijf een programma dat de tabel `namen` met lege strings vult.

♢

▸ **OEFENING 3.2**
Schrijf een programma dat het gemiddelde van een array `int`s berekent. (Voor wie er écht niet genoeg van kan krijgen: standaardafwijking, kurtosis, ...)

♢

▸ **OEFENING 3.3**
Schrijf een functie die het grootste getal uit een array zoekt.

♢

In tegenstelling tot Pascal kan je geen arrays aan een andere array toekennen:

```
int x[20];
int y[20];
int matrix[10][5];

x = y;    /* gaat niet! */
matrix[2] = matrix[3];    /* gaat ook niet! */
```

Je moet in zulke gevallen zelf een lus schrijven die de array element per element kopieert.

▸ **OEFENING 3.4**
Verklaar waarom C geen array-toekenning toestaat.

♢

3.1.1 Arrays en strings

Stel weer dat `x` een array van twintig `int`s is. Als je deze array als string gebruikt, kan je `x[19]` alleen maar gebruiken om de nulbyte van een 19-letterige string in te bewaren. Niets belet je natuurlijk er een andere waarde in te stoppen, want je bent niet verplicht elke array van `char`s ook als string te gebruiken. Alleen moet je er aan denken dat als je de array als string gebruikt, er daadwerkelijk een nulbyte aan het eind moet staan:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
’D’	’a’	’g’	’ ’	’g’	’r’	’o’	’o’	’t’	’m’	’o’	’e’	’d’	’e’	’r’	’!’	’\n’	’E’	’e’	’n’

is OK als array van `char`s, maar als je op deze array een stringfunctie zou loslaten (zoals `strcpy()` of `puts()`), dan zou alles flink in het honderd lopen. De stringfuncties blijven immers doorzoeken tot ze een nulbyte vinden ...

Aan de andere kant kijken de stringfuncties niet verder dan de eerste nulbyte. Dus als string beschouwd is dit een lege string:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
0	’j’	’j’	0	’n’	’n’	0	0	’j’	0	0	’j’	’n’	’j’	0	’n’	’n’	0	0	’j’

terwijl dit als array van `char`s perfect gebruikt kan worden om de antwoorden op een twintigvragige enquête op te slaan (door ’j’, ’n’ en 0 te interpreteren als ’ja’, ’nee’ resp. ’geen mening’).

3.2 Toepassing: sorteren

Om een beetje te leren werken met arrays zal ik twee eenvoudige sorteermethoden behandelen. Bedoeling ervan is een tabel met bijvoorbeeld `int`s erin te sorteren, dus `tabel[0]` moet op het einde van het proces de kleinste waarde bevatten, `tabel[1]` de op één na kleinste, enz...

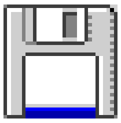
3.2.1 Bubble sort

De strategie van bubble sort is eenvoudig: bekijk het eerste en het tweede getal van de tabel. Als het eerste getal groter is dan het tweede, verwissel ze dan. Doe hetzelfde met het tweede en het derde getal, het derde en het vierde, enz. Hierdoor "bubbelen" de kleinere getallen naar boven (i.e. naar het begin van de tabel) en de grotere naar onder.

teller=0		teller=1		teller=2	
1	OK	1		1	
8		8	OK	8	
9		9		0	wissel
0		0		9	

De eerste doorloop van het sorteerproces

Dit bubbelproces wordt herhaald todat er tijdens een hele doorloop niets meer verwisseld werd, want dat betekent dat de tabel gesorteerd is.


bubble.c

```
#define LENGTE 20

int tabel[LENGTE];

void bubbleSort(void)
{
        while (1 == 1) {
                int teller;
                int omgewisseld;

                omgewisseld = 0;
                teller = 0;
                while (teller < LENGTE-1) {
                        if (tabel[teller] > tabel[teller+1]) {
                                /* omwisselen */
                                int wissel;

                                wissel = tabel[teller];
                                tabel[teller] = tabel[teller+1];
                                tabel[teller+1] = wissel;
                                omgewisseld = 1;
                        }
                        teller = teller + 1;
                }
                if (omgewisseld == 0) break;
```

```
        }
}
```

Het programma van hierboven werkt wel, maar het is niet erg efficiënt: het controleert telkens de *hele* tabel, terwijl we weten dat de tweede doorloop het laatste element al goed is, de derde doorloop staan de twee laatste al in orde, ... Het is dus overbodig telkens te controleren of die laatste elementen verwisseld moeten worden.

▸ **OEFENING 3.5**
Waarom is het niet zo dat na de eerste doorloop het kleinste getal vooraan staat?

◊

▸ **OEFENING 3.6**
Pas het programma aan zodat die overbodige controles niet meer gebeuren.

◊

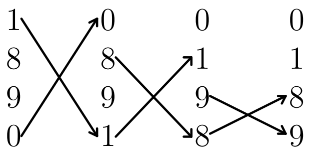
teller=0		teller=1		teller=0	
1	}OK	1		0	}wissel
8		0	}wissel	1	
0		8		8	}staan goed
9	staat goed	9	staat goed	9	

De tweede en derde doorloop van het sorteerproces

Bubble sort is heel eenvoudig te programmeren, maar ook een van de traagste methodes om een array te sorteren. De getallen "bubbelen" langzaam naar hun juiste plaats (vandaar de naam); vooral naar boven bubbelen gaat traag. Daarom zullen we een iets betere methode beschouwen:

3.2.2 Selection sort

Selection sort is een stuk efficiënter dan bubble sort. De gebruikte methode is: zoek het kleinste element uit de lijst, en verwissel dat met het eerste element. Het eerste element staat dan op zijn plaats. Zoek nu in de rest van de lijst (d.i. alles behalve het eerste element) het kleinste element, en verwissel dat met het tweede element van de array. Nu staan de eerste twee elementen goed, ... tot de hele lijst gesorteerd is.



Voorbeeld van een selection sort

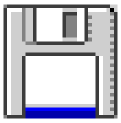
selectionsort.c

```
#define LENGTE 20

int tabel[LENGTE];

void selectionSort(void)
{
```

```
        int doorloop = 0;

        while (doorloop < LENGTE-1) {
                int kleinste;     /* Plaats van het kleinste element */

                {
                        int teller;

                        kleinste = doorloop;
                        teller = doorloop;
                        while (teller < LENGTE) {
                                if (tabel[teller] < tabel[kleinste])
                                        kleinste = teller;
                                teller = teller + 1;
                        }
                }
                /* Omwisselen */
                {
                        int wissel;

                        wissel = tabel[kleinste];
                        tabel[kleinste] = tabel[doorloop];
                        tabel[doorloop] = wissel;
                }
                doorloop = doorloop + 1;
        }
}
```

3.2.3 Array als datatype

Je kan ook een functie maken die als argument een array heeft. Het zou bijvoorbeeld handig zijn om een functie te maken die het kleinste getal uit de een tabel zoekt. Dat kan met de volgende functiedefinitie:



kleinste.c

```
int kleinste(int tab[])
{
        int teller;
        int resultaat;

        resultaat = tab[0];    /* Voorlopig resultaat */
        teller = 1;
        while (teller < LENGTE) {
                if (tab[teller] < resultaat) resultaat = tab[teller];
                teller = teller + 1;
        }
        return resultaat;
}
```

Zoals je ziet, is het niet nodig in de functiehoofding op te geven hoe groot de array precies is. (Het mag wel, maar de compiler zal er toch geen rekening mee houden.) Net zoals strings worden ook array-argumenten **by reference** (zie §2.8.2) overgedragen. Een nettere versie van de sorteerroutine van daarnet zou er zo kunnen uitzien:

selsort.c

```
/* Zoek de plaats van het kleinste element uit het deel van de tabel
 * tussen tabel[start] t/m tabel[eind]
 */
int zoekKleinste(int tabel[],int start,int eind)
{
        int teller;
        int kleinste;   /* Positie van het tot nu toe kleinste element */

        kleinste = start;
        teller = start + 1;
        while (teller <= eind) {
                if (tabel[teller] < tabel[kleinste])
                        kleinste = teller;
                teller = teller + 1;
        }
        return kleinste;
}

void selectionSort(int tabel[],int lengte)
{
        int doorloop = 0;

        while (doorloop < lengte-1) {
                int kleinste;     /* Plaats van het kleinste element */
                int wissel;

                kleinste = zoekKleinste(tabel,doorloop,lengte-1);

                /* Omwisselen */
                wissel = tabel[kleinste];
                tabel[kleinste] = tabel[doorloop];
                tabel[doorloop] = wissel;

                doorloop = doorloop + 1;
        }
}
```

Merk op dat deze versie iets minder efficiënt is: tijdens het zoeken van het kleinste element zijn er nu twee variabelen met als naam `kleinste` in gebruik. Dit bezwaar wordt ruimschoots vergoed door de grotere duidelijkheid van het programma. Als we ergens anders in het programma (of in een ander programma) ook het kleinste element uit een array willen vinden, kunnen we de `zoekKleinste()` functie hergebruiken. Het is in het algemeen goede programmeerstijl één functie voor één taak te programmeren. Wat ook een verbetering is ten opzichte van de vorige versie, is dat we er niet meer zomaar van uitgaan dat de tabel een vaste grootte heeft: we maken nergens gebruik van de define `LENGTE`, maar krijgen de lengte door als argument. Je kan dus met dezelfde functie lijsten van verschillende grootte sorteren. De functie is dus veel algemener geworden, en de kans dat we ze ergens anders kunnen hergebruiken is daardoor ook veel vergroot. Door **herbruikbare code** (**reusable code**) te schrijven kan je makkelijker een **bibliotheek** van veelgebruikte, algemene en goed uitgeteste functies op te bouwen. De kans op fouten in programma's neemt daardoor ook drastisch af.

Merk op dat een C functie geen enkele manier heeft om te bepalen hoe groot een doorgegeven array is. Zelfs met de `sizeof()` operator, die wat verder aan bod komt, lukt het niet (zie §3.5.1).

3.3 Pointers

In feite valt C niet te begrijpen zonder het begrip **pointer**. Typisch aan C is de hechte manier waarop arrays en pointers samenhangen, dit in tegenstelling tot veel andere programmeertalen.

Een pointer is net als een array een **afgeleid datatype** (**derived datatype**). Hiermee bedoel ik dat er niet zoiets als een variabele van het type "pointer" bestaat, net zoals er ook geen variabelen van het type "array" zijn, maar wel bijvoorbeeld van het type "array van `int`s". Zoals de naam al aangeeft "wijst" een pointer ergens naar; je krijgt dan ook types als "pointer naar `int`". Zo'n variabele wordt als volgt gemaakt:

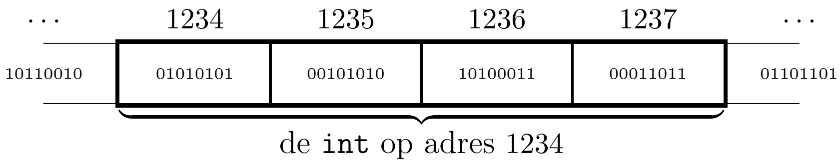
```
int *MijnPointer;
```

(Spaties vóór of na het `*` zijn niet belangrijk.)
Wat kan je nu met zo'n pointervariabele doen, of meer bepaald: welke waarden passen in `MijnPointer`? Om dat uit te leggen moet je eerst iets weten over

3.3.1 Geheugen en adressen

Het geheugen van een computer is georganiseerd als een hoop bytes achtereen. In elke byte kan je een waarde (van 0 tot 255, of acht nulletjes en eentjes) stoppen (waarbij de oude waarde onherroepelijk verloren gaat), en je kan van elke byte opvragen welke waarde erin zit. Met andere woorden, het geheugen is net een reusachtige array van `char`s. Variabelen die groter zijn dan één byte worden bewaard in opeenvolgende bytes. Een `int` neemt bijvoorbeeld, afhankelijk van het computersysteem, vaak twee of vier bytes in.

Zoals elk element van een array te bereiken is via het volgnummer ervan (zoals al gezegd, is bijvoorbeeld `tabel[2]` het derde element van de array `tabel`), heeft elke byte van het geheugen ook een volgnummer dat **adres** genoemd wordt. Het enige dat een geheugen in feite kan doen is "stop waarde x in de byte met adres y" en "vertel eens welke waarde er in de byte met adres z zit" (wat respectievelijk **schrijven** en **lezen** genoemd wordt). Als adres van een variabele die meer dan één byte in beslag neemt, wordt meestal het laagste adres van de bytes van die variabele gebruikt. Als een `int` dus vier bytes inneemt, wordt de `int` met adres 1234 in de bytes 1234, 1235, 1236 en 1237 bewaard. De bytes van het geheugen worden ook wel eens **geheugenplaats**en genoemd, dus kan je ook zeggen dat die `int` van daarnet op geheugenplaatsen 1234 tot en met 1237 wordt bewaard.



De manier waarop een `int` precies over de vier bytes verdeeld wordt hangt af van het gebruikte systeem.

3.3.2 Pointers

Nu we zover zijn is het moeilijkste in feite achter de rug: “**pointer**” is immers niets meer dan een duur woord voor “adres”! De variabele `MijnPointer` die ik daarnet gemaakt heb, bevat dus een “pointer naar een `int`”, wat dus niets meer of minder is dan het adres van een `int` variabele. Laten we voor het gemak nog vlug een paar variabelen bijmaken om wat mee te kunnen spelen:

```
int Getal;
char Letter;
char *LetterWijzer;
```

Het adres van een variabele bekom je door de `&`-operator erop los te laten: `&Getal` is dus het adres van de `int` variabele `Getal`, m.a.w. een *pointer naar* `Getal`. Die waarde kan je dus in `MijnPointer` stoppen:

```
MijnPointer = &Getal;
```

Nu bevat `MijnPointer` het adres van de variabele `Getal`. Hetzelfde kan je doen voor `char`s met

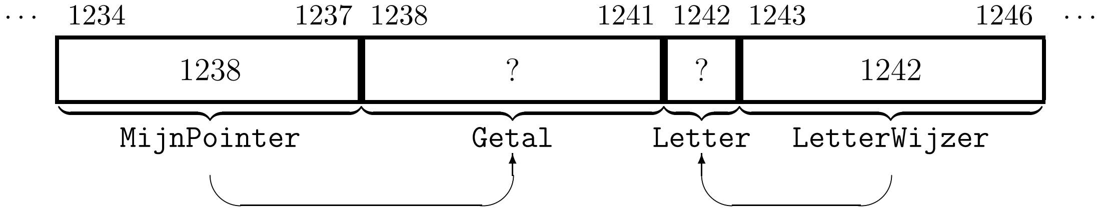
```
LetterWijzer = &Letter;
```

De twee mengen is niet erg gezond: `LetterWijzer = &Getal` zal er wel voor zorgen dat `LetterWijzer` het adres van `Getal` bevat (adressen zijn tenslotte adressen, of het nu van een `int` of een `char` is), maar de meeste C compilers zullen een waarschuwing geven, al zullen ze het programma wel goed compileren. Dit soort hokuspokus is alleen aan te raden als je heel goed weet wat je aan het doen bent. Om duidelijk te tonen dat je niet per ongeluk twee soorten pointers mengt, cast je best `&Getal` (dat een `int *` is) naar `char *`:

```
LetterWijzer = (char *) &Getal;
```

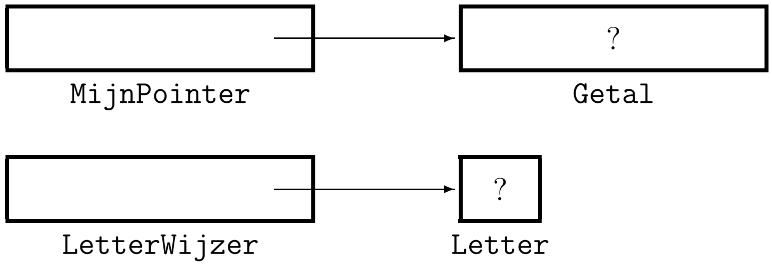
In sommige gevallen kan dit soort constructies nuttig zijn, maar als je niet goed oplet is het een geweldige manier om onvindbare bugs in je programma te verbergen ...

Schematisch kunnen we de situatie als volgt voorstellen (ongeïnitialiseerde variabelen zijn met een “?” aangeduid):



(De manier waarop een adres als “1238” over de vier bytes van een pointervariabele verdeeld wordt, hangt af van de gebruikte processor en is ook niet erg belangrijk voor ons. Het is trouwens niet eens zeker dat een pointer vier bytes inneemt—ook dat hangt af van het gebruikte systeem.)

Omdat het ons eigenlijk niet zo interesseert op welke geheugenplaats alle variabelen precies terechtkomen, of hoeveel bytes ze innemen, zal ik dat alles gewoon weglaten en een pointer met een pijltje aanduiden:



Tijd om eens iets nuttigers te doen dan alleen maar wat adressen over en weer gooien: als je vóór een pointervariabele een `*` schrijft, krijg je een nieuwe variabele die in feite “de variabele is waar de pointer naar wijst”. De pointer `MijnPointer` wijst naar `Getal`, dus `*MijnPointer` is “de `int` op het adres dat in `MijnPointer` zit”, m.a.w. niets anders dan `Getal`. Dus

```
*MijnPointer = 42;
```

(waarbij je `*MijnPointer` zou kunnen lezen als “het ding waar `MijnPointer` naar wijst”) heeft net hetzelfde effect als `Getal = 42`. Wat dus ook werkt is

```
*LetterWijzer = *MijnPointer;
```

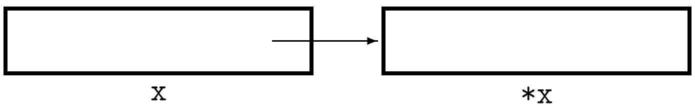
Even uiteenrafelen:

- De uitdrukking rechts van `=` wordt berekend, hier `*MijnPointer`. We moeten dus de inhoud van de variabele `*MijnPointer` nemen. Aangezien `MijnPointer` wijst naar `Getal`, wordt dat dus de inhoud van `Getal`, wat 42 oplevert.
- De waarde 42 moet in de variabele `*LetterWijzer` gestopt worden. We hebben `LetterWijzer` naar `Letter` doen wijzen, dus `Letter` zal de waarde 42 bevatten (de ASCII code van `'*'`).

Nog even voor de duidelijkheid de datatypes van alle argumenten:

$$\underbrace{*\underbrace{\texttt{LetterWijzer}}_{\texttt{char *}}}_{\texttt{char}} \;\texttt{=}\; \underbrace{*\underbrace{\texttt{MijnPointer}}_{\texttt{int *}}}_{\texttt{int}}$$

De `*` operator wordt ook wel eens de **indirectie-operator** genoemd: als `x` een pointer is naar een ding, is `*x` het ding zelf; m.a.w. `*` is een soort “ontpointer” operator.



3.3.3 Functies met pointer-argumenten

Ik heb al eerder gezegd dat functies hun argumenten niet kunnen wijzigen, en dat ze maar één resultaat kunnen teruggeven, tenzij je met pointers werkt. Tijd om dat even uit de doeken te doen. Ik zal als voorbeeld een functie `wissel()` maken die twee `int` variabelen met elkaar verwisselt. Wat natuurlijk niet werkt is iets als

```
void wissel(int a,int b)
{
        int hulp;

        hulp = a;
        a = b;
        b = hulp;
}
```

omdat bij een aanroep als `wissel(x,y)` de variabelen `a` en `b` een *kopie* van de inhoud van `x` en `y` als waarde krijgen. Het gevolg hiervan is dat `wissel()` alleen die lokale kopie kan wijzigen, en niet de inhoud van de oorspronkelijke variabelen `x` en `y`. De pointer-versie werkt wel:

```
void wissel(int *a,int *b)
{
        int hulp;

        hulp = *a;
        *a = *b;
        *b = hulp;
}
```

Als je nu `wissel(&x,&y)` gebruikt, zal alles werken zoals het hoort. Enige nadeel is dat je al die ampersands moet schrijven. Met de preprocessor kan je hier eventueel iets aan doen. (Zie oefening 6.2.)

Op dezelfde manier kan je functies maken die meer dan één resultaat teruggeven. In zekere zin zou je kunnen zeggen dat `wissel()` twee `int`s teruggeeft. Een duidelijker voorbeeld is deze functie, die zowel het aantal kleine als hoofdletters telt in een string:

```
void telhoofdklein(char string[],int *hoofd,int *kleine) {
        *hoofd = 0;
        *kleine = 0;

        int tel;
        while (1 == 1) {
                /* de letter waar we nu naar kijken */
                char c = string[i];
                if (c == 0) break;
                if (c >= 'a') {
                        if (c <= 'z') {
                                /* de letter ligt tussen 'a' en 'z' */
                                *klein = *klein + 1;
                        }
```

```
                }
                if (c >= 'A') {
                        if (c <= 'Z') {
                                *hoofd = *hoofd + 1;
                        }
                }
                i = i + 1;
        }
}
```

Deze functie kan je bijvoorbeeld zo gebruiken:

```
void testje(void) {
        char tekst[80];
        int klein,hoofd;

        strcpy(tekst,"Lang geleden heel ver hier vandaan ...");
        /* De twee &'s op de volgende lijn niet vergeten! */
        telhoofdklein(tekst,&hoofd,&klein);
        printf("De tekst '%s' bevat %d kleine en %d hoofdletters.\n",
                tekst,klein,hoofd);
}
```

▸ **OEFENING 3.7**
Herschrijf deze functie zodat ze zo klein mogelijk wordt. Variabelnamen kleiner maken telt niet; alle `hoofd` door `h` vervangen beschouw ik dus niet als een verkleining. Kortom: zorg ervoor dat deze functie zo weinig mogelijk "woorden" bevat (waarbij we "`+`" en "`<=`" als "woord" opvatten).

◊

3.4 Pointers en arrays

In C zijn, zoals ik al zei, pointers en arrays sterk met elkaar verweven. Stel dat ik de volgende array maak:

```
        int getallen[20];
```

dan kan je een pointer naar het vierde getal krijgen met `&getallen[3]`. Het bijzondere in C is nu dat je een pointer naar het eerste getal (d.i. naar het begin van de tabel) niet alleen krijgt met `&getallen[0]` maar ook met `getallen`! Met andere woorden: *de naam van een array is niets anders dan een (handige afkorting voor een) pointer naar het eerste element ervan.*

Dit is nog maar het begin:

```
{
        int getallen[20];
        int *wijzer;

        getallen[3] = 4;
```

```
        wijzer = getallen;       /* dit kan! (zoals uitgelegd) */
        wijzer[7] = 2;           /* (!) [meer uitleg volgt] */
}
```

Ik heb hier dus de `[]` operator gebruikt op een *pointer*. Hieruit blijkt al hoe hecht pointers en arrays samenhangen. Let wel: `getallen` is *geen* variabele, maar een constante, net als bijvoorbeeld `"Hello"` of `3.1415` (`getallen[0]` t/m `getallen[19]` zijn natuurlijk wél variabelen). Je kan dus niet zoiets doen als `getallen = wijzer;` net zoals `5 = x+y` ook niet gaat. Dus,

```
int getallen[20];
```

maakt een groepje van twintig `int`s en zorgt ervoor dat we `getallen` als handige afkorting van het adres van de eerste `int` kunnen gebruiken, terwijl

```
int *wijzer;
```

alleen een pointer maakt.

3.4.1 Pointers, arrays en strings

We hebben in §2.8.2 de functie `wisString()` gemaakt:

```
void wisString(char str[])
{
        str[0] = 0;
}

void test(void)
{
        char mijnString[20];

        wisString(mijnString);
}
```

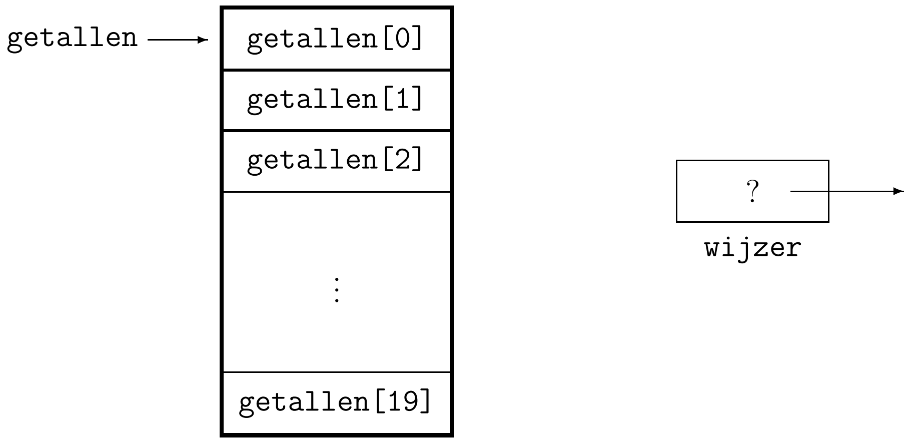
Nu weten we dat het argument `mijnString` van `wisString()` eigenlijk een pointer is naar de eerste letter van de string. In C worden de argumenten altijd **by value** doorgegeven. De functie `wisString()` krijgt dus een kopie van deze pointer in `str`. Er wordt geen kopie van de string zelf gemaakt; alleen de pointer ernaar wordt gekopieerd. Het is dus alsof stringargumenten **by reference** doorgegeven worden.

Merk ook op dat stringfuncties dus eigenlijk werken op pointers naar strings. We hadden de functiehoofding van `wisString()` net zo goed als

```
void wisString(char *str)
```

kunnen schrijven. In functiehoofdingen is `[]` een synoniem van `*`.

3.4.2 Pointer arithmetic



Om beter te begrijpen waarom `wijzer[7]` werkt, moet je weten dat alle elementen van een array achter elkaar in het geheugen worden bewaard. Met "achter elkaar" bedoel ik: op opeenvolgende adressen. Stel dat een `int` weer vier bytes inneemt en dat de array `getallen` toevallig begint op adres 1000. Dan wordt `getallen[0]` op geheugenplaatsen 1000 tot en met 1003 bewaard, `getallen[1]` op 1004 t/m 1007, ..., en `getallen[19]` op adressen 1076 t/m 1079.

In het voorbeeld wijst `wijzer` dus naar de `int` op adres 1000, en omdat de C compiler weet dat een `int` vier bytes inneemt, kan hij uitrekenen welke `int` je bedoelt met `wijzer[7]` (namelijk die op adres $1000 + 4 \times 7$). Merk op dat zoiets alleen kan omdat de `int`s van de array netjes achter elkaar (en niet zomaar willekeurig verspreid over het geheugen) bewaard worden. Als je dus zomaar een pointer pakt en er de `[]` operator op loslaat, onderstelt de C compiler dat die pointer wijst naar een array.

Dat is één van de redenen waarom pointers soms lastig zijn: de C compiler kan fouten als in het volgende programma bijna onmogelijk vinden (zeker als er tussen die instructies een paar duizend regels code verstopt zitten):

```
int main(void)
{
        int Tabel1[100];
        int NogEenTabel[10];
        int *daarheen;

        daarheen = Tabel1;
        daarheen[20] = 3;        /* OK */
        daarheen = NogEenTabel;
        daarheen[20] = 3;        /* fout! */
        return 0;
}
```

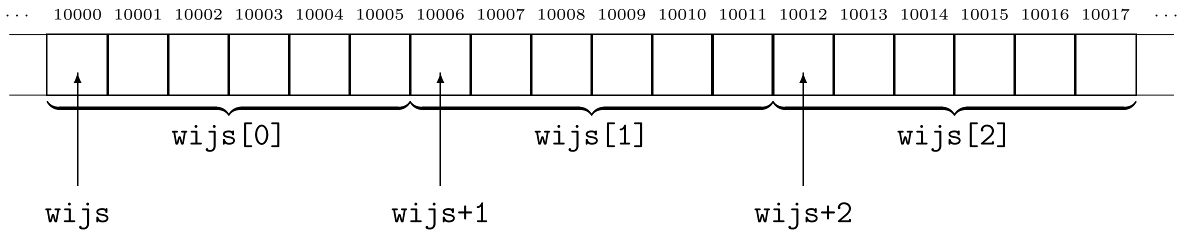
▸ **OEFENING 3.8**
Wat denk je dat er gebeurt als ik

```
        daarheen = &Tabel1[3];
        daarheen[18] = 4;
```

zou laten draaien?

◇

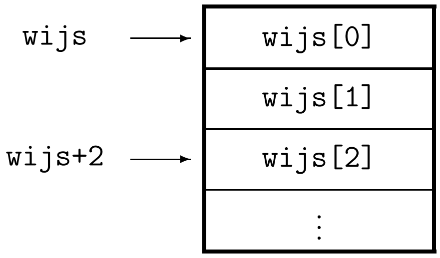
Laten we eens een heel expliciet voorbeeldje in elkaar zetten: stel dat `wijs` een variabele van het type `double *` is (dus een pointer naar een `double`) die voor het ogenblik wijst naar het adres 10000 (dus in de variabele `wijs` zit effectief iets als "10000"). Wat is dan het effect van `*wijs` (bijvoorbeeld in `printf("%f",*wijs)`)? Zoals al eerder uitgelegd: niets anders dan "neem de waarde van de `double` die op adres 10000 zit". Laten we nog wat explicieter worden en veronderstellen dat een `double` zes bytes inneemt. Wat is nu `wijs[2]`? De `double` waar `wijs` naar wijst, is die op adressen 10000 tot en met 10005; de `double` vlak daarna is die op 10006 t/m 10011 (en is te bereiken met `wijs[1]`), dus is `wijs[2]` niets anders dan "de waarde van de `double` die op adressen 10012 t/m 10017 bewaard wordt".



Algemeen: onderstel dat `daarheen` een pointer is naar een bepaald type `DataType` en dat een `DataType` variabele n bytes inneemt. Stel verder dat `daarheen` wijst naar adres *adres*. Dan bedoelen we met `daarheen[index]` niets anders dan "de variabele van het type `DataType` die bewaard wordt op de adressen $adres + \texttt{index} \times n$ en volgende".

Ik ga nog een stapje verder met het vorige voorbeeld: wat is `&wijs[2]`? Letterlijk invullen in wat we weten over de `&` operator geeft "het adres van de variabele `wijs[2]`". Maar aangezien we daarnet beredeneerden dat die variabele adres 10012 heeft, is het antwoord dus, jawel, 10012. We hebben dus in feite niets anders bekomen dan het adres dat in `wijs` zit plus 6×2, m.a.w. `&wijs[2]` is gewoon een verkapte optelling! Het optellen en aftrekken van pointers met getallen wordt "**pointer arithmetic**" genoemd en is één van die dingen waar C berucht/beroemd voor is.

In C kan dat handiger: `wijs+2` doet net hetzelfde. Bij pointers kan je dus een getal optellen(!). Dat komt omdat pointers in feite een getal bevatten (dat getal stelt toevallig het adres voor van een variabele van een bepaald type, maar dat laat ons even koud). Alleen worden getallen, wanneer ze bij een pointer opgeteld worden, vermenigvuldigd met het aantal bytes dat "`*pointer`" inneemt (6 in het voorbeeld van die `double`s).



Even kijken of je alles goed doorhebt:

▸ **OEFENING 3.9**
Wat is

```
*(wijs+2)
```

(waarbij `wijs` nog steeds een `double *` is die naar adres 10000 wijst)?

◇

Dan is er nog een laatste finesse: twee pointers kan je ook van elkaar aftrekken. Laten we zeggen dat `wijs` nog steeds naar die `double` op adres 10000 wijst, en dat we een `double *wijs2` maken die we laten wijzen naar de `double` op adres 10012 (`wijs2 = wijs+2`). Wat zou je dan verwachten dat `wijs2 - wijs` oplevert? Verrassing, verrassing: `2`, wat in feite niets anders is dan "(het adres waar `wijs2` naar wijst − het adres waar `wijs` naar wijst) gedeeld door 6" m.a.w. $(10012 - 10000)/6$. Vanzelfsprekend gaat dat alleen maar goed als `wijs` en `wijs2` naar dezelfde array van `double`s wijzen (als ze respectievelijk naar adressen 10000 en 10010 zouden wijzen, krijgen we enige probleempjes ...)

3.4.3 de `NULL` pointer

Er is een manier om aan te geven dat een pointer naar "niets" wijst: `pointer = 0` (`0` is het enige *getal* dat je in een *pointer*variabele mag stoppen). Netter is echter de macro `NULL` te gebruiken die in `stdio.h` gedefinieerd wordt. Deze pointer wordt de **nulpointer** (**null pointer**) genoemd.

Het spreekt vanzelf dat het uit den boze is om dingen als `*pointer` of `pointer[2]` te zeggen wanneer `pointer` de waarde `NULL` bevat ...

`0` is ook het enige getal waarmee je een pointer mag vergelijken. In het kort komt het er dus op neer dat het getal `0` ook als een bijzondere pointer gebruikt mag worden, de nulpointer.

Het bitpatroon dat met een `NULL` pointer overeenkomt, bestaat niet noodzakelijk uit allemaal nullen. Normaal zou je hier niet veel last van mogen hebben omdat de C compiler zelf voor de omzetting tussen 0 als `int` en als pointer zorgt, maar in verband met functies als `calloc()` (zie §7.4.2) kan het wel degelijk van belang zijn.

De meeste C compilers laten toe dat je andere getallen dan 0 in een pointer stopt. Het resultaat is dan meestal dat die pointer naar het aangeduide adres wijst. De enige gerechtvaardigde toepassing hiervan is in programma's die rechtstreeks met bepaalde hardware moeten communiceren.

3.5 Geheugenbeheer met `malloc()` en `free()`

3.5.1 de `sizeof()` operator

Zo dadelijk zullen we moeten weten hoeveel bytes een bepaalde variabele inneemt. C heeft hiervoor een operator: `sizeof(variabele)` of `sizeof(type)` geven als resultaat het aantal bytes dat die `variabele`, of respectievelijk een variabele van een bepaald `type` inneemt. Let wel: hoe zeer `sizeof()` zich ook mag uitgeven voor een functie, het is wel degelijk een operator (die dus vast in de C compiler ingebakken zit).

Deze operator is nodig omdat je er in C bijvoorbeeld niet zomaar van uit mag gaan dat op elk systeem een `int` evenveel bytes inneemt (op 32-bits systemen meestal vier, op MS-DOS en andere 16-bits systeempjes vaak twee).

▸ **OEFENING 3.10**
We hebben al eens af en toe iets dergelijks geschreven:

```
#define LENGTE 20
```

```
int tabel[LENGTE];

/* ... programma ... */
        while (teller < LENGTE) {
                /* ... */
        }
/* ... */
```

Vind een manier om die `#define` te vermijden, zodat het programma er zo uit ziet:

```
int tabel[20];

/* ... */
        while (teller < ??? ) {  /* Die ??? moet je zelf */
                                 /* invullen natuurlijk... */

        }
```

♢

3.5.2 Geheugenbeheer

Met de functie `malloc()` kan je een stuk **geheugen reserveren**. Dat houdt in dat je aan het besturingssysteem vraagt of je een blok van `zoveel` bytes groot mag hebben. Als dat er is, geeft `malloc(zoveel)` je een pointer naar het begin van zo'n blok geheugen terug. Is er niet genoeg geheugen vrij, dan krijg je een null-pointer terug. Het geheugen dat `malloc()` levert is niet geïnitialiseerd.

Als je het geheugen niet meer nodig hebt, kan je het teruggeven aan het systeem met `free(pointer naar het begin van het blok)`. Het is niet nodig op te geven hoe groot het blok was. Aan het einde van het programma wordt alle geheugen automatisch ge`free()`d, maar het is netter om het expliciet zelf te doen.

Een voorbeeldje (`malloc()` en `free()` leven in `stdlib.h`):

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

int main(void)
{
        int grootte;
        int *tabel;
        int teller;

        puts("Hoeveel getallen: ");
        scanf("%d",&grootte);

        /* maak een array van 'grootte' ints */
        tabel = malloc(sizeof(int) * grootte);
        if (tabel == NULL) {
                puts("Niet genoeg geheugen!");
```

```
                return 10;
        }

        teller = 0;
        while (teller < grootte) {
                printf("getal %d:\n",teller + 1);
                scanf("%d",tabel+teller);
                teller = teller + 1;
        }

        /* doe iets met de tabel */

        free(tabel);
        return 0;
}
```

Heel gevaarlijk (en vaak moeilijk op te sporen) is natuurlijk het verder gebruiken van `tabel` na de `free(tabel)` opdracht.

Een ander risico is **memory leak**: als we de tabel niet meer nodig hebben, en bijvoorbeeld `tabel = NULL` doen, maar `free(tabel)` vergeten, dan is het geheugen dat we gereserveerd hadden voor goed onbereikbaar, want we hebben geen enkele pointer meer die ernaar wijst. In het bijzonder kunnen we het geheugen niet meer vrijgeven (we hebben immers geen pointer meer om aan `free()` te geven!). Denk erom dat geheugen dat je in een bepaalde functie reserveert, *niet* automatisch wordt vrijgegeven bij het verlaten ervan, zoals wel gebeurt met lokale variabelen.

▸ **OEFENING 3.11**
Wat gaat er fout in de volgende functie om een tabel op te vullen?

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

void vullen(int tabel[])
{
        int aantal,tel;

        puts("Aantal getallen?");
        scanf("%d",&aantal);
        tabel = malloc(aantal * sizeof(int));
        /* tabel opvullen */
        for (tel = 0; tel < aantal; tel = tel + 1) {
                int getal;

                printf("Getal %d?\n",tel+1);
                scanf("%d",&getal);
                tabel[tel] = getal;
                /* Kortere maar onduidelijkere versie:
                   scanf("%d",&(tabel[tel])); */
        }
}
```

```
int main(void)
{
        int *getallen;
        vullen(getallen);
        /* hier volgt dan een stuk programma dat
         * de elementen van ‘getallen’ gebruikt */
        return 0;
}
```

Schrijf een correcte versie.

◇

3.5.3 Het geheugenmodel van C

Voor C is het geheugen van de computer een groot, ondoorzichtig blok bytes. Functies als `malloc()` en `free()` beschouwen het geheugen in feite als een lange array van bytes. Je kan dus nooit van een willekeurig stuk geheugen weten wat het nu eigenlijk voorstelt; een byte kan bijvoorbeeld een stukje van een `float` bevatten of een letter van een string voorstellen. Een functie als

```
int kleinste(int tab[]) {
        ...
}
```

heeft geen enkele mogelijkheid om de lengte van de array `tab` te bepalen. `tab` is immers niets meer of minder dan een pointer naar het eerste element van de array. Meer dan dat kan het programma niet weten. Bijvoorbeeld:

```
void voorbeeld(void) {
        int *eenArray;
        int eenTabel[20];
        eenArray = malloc(20 * sizeof(int));

        /* hier zou normaal code komen om de arrays te initialiseren */

        /* op de volgende regel komen arrays van
         * resp 20, 20 en 15 ints voor */
        printf("%d %d %d",
                kleinste(eenTabel),kleinste(eenArray),kleinste(eenArray+5));

        eenArray = &(eenTabel[5]);
        /* op de volgende regel komen arrays van
         * resp 20, 15 en 10 ints voor */
        printf("%d %d %d",
                kleinste(eenTabel),kleinste(eenArray),kleinste(eenArray+5));
}
```

Na de eerste `printf()` laat ik `eenArray` wijzen naar het vijfde element van `eenTabel`. Nu is `eenArray` een pointer geworden naar een array van 15 `ints` (de 5 `ints` ervoor tellen immers niet meer mee). In het voorbeeld komt dus precies dezelfde `printf()` opdracht twee keer voor, maar de grootte van de arrays is verschillend. De compiler kan onmogelijk bijhouden hoe groot de arrays zijn, zeker als er tussen beide `printf()`s niet één lijntje code zou staan maar een groot programma.

3.6 Air on the C String

Aan het begin van dit hoofdstuk zei ik al dat strings eigenlijk arrays van `char`s zijn. Uit het voorafgaande blijkt dat je ook een pointer naar een `char` zich als een string gedraagt (de pointer wijst dan naar de eerste letter van die string).

Als je in een programma een stringconstante tikt, zeg `"Hallo"`, dan zorgt de C compiler er voor dat ergens in het geheugen die string staat en levert een pointer ernaar. Belangrijk is dat je de inhoud van zo'n string niet mag wijzigen (daarom heet het ook een string*constante*). Dus een programma als

stringconst.c

```
#include <stdio.h>
int main(void)
{
        char *s = "strinch";

        s[5] = 'g';
        s[6] = 0;
        puts(s);
        return 0;
}
```

zal crashen op de opdracht `s[5] = 'g';`. Op een UNIX machine krijg je onmiddellijk een `Segmentation fault (core dumped)`. Sommige systemen (in het bijzonder MS-DOS) zijn minder beveiligd tegen dit soort dingen, en daar draait het programma gewoon verder.

Als voorbeeld van stringbewerkingen zal ik een functie schrijven die de lengte van een string bepaalt:

```
/* int strlen(char string[])    had ook gekund natuurlijk, maar
 *                              meestal wordt de * vorm gebruikt
 */
int strlen(char *string) {
        int lengte;

        lengte = 0;
        while (string[lengte] != 0)
                lengte = lengte + 1;
        return lengte;
}
```

Zoals je weet uit §2.3.3 wordt het einde van een string aangegeven door een nulbyte, en `strlen()` zoekt gewoon de string af tot het die nulbyte vindt. Deze functie zit overigens standaard al in `string.h`.

Iets waarvoor je moet opletten, is dat alle lokale variabelen die in een functie gemaakt zijn, verdwijnen na het beëindigen van die functie. Dat is al lang (§2.3) bekend, maar in combinatie met strings kan je verraderlijke situaties krijgen zoals in deze functie (waarin ik gebruik maak van de `%` operator die de rest na een deling oplevert; zo geeft `getal%10` de rest die overblijft wanneer `getal` door 10 gedeeld wordt—zie §4.3.1):

maakString.c

```
/* maak een string die de inhoud van het getal bevat;
 * bv. maakString(42) geeft de string "42" */
char *maakString(int getal)
{
        char resultaat[20];
        int lengte;

        /* Bijzonder geval. Als we deze regel zouden weglaten,
         * zou het resultaat van maakString(0) een lege string zijn
         */
        if (getal == 0) return "0";

        lengte = 0;

        /* Eerst de string achterstevoren maken
         * (getal%10 geeft het laatste cijfer)
         */
        while (getal != 0) {
                /* We veronderstellen dat de code van '0'
                 * 1 minder is dan '1', 2 minder dan '2' enz.
                 * (dit is het geval in de ASCII codering)
                 */
                resultaat[lengte] = getal%10 + '0';
                getal = getal / 10;
                lengte = lengte + 1;
        }

        /* De string netjes afsluiten met een nulbyte */
        resultaat[lengte] = 0;

        {       /* Resultaat omkeren */
                int tel, telaf;

                tel = 0;
                telaf = lengte-1;
                while (tel < telaf) {
                        /* verwissel resultaat[tel] en resultaat[telaf] */
                        char wissel;

                        wissel = resultaat[tel];
                        resultaat[tel] = resultaat[telaf];
                        resultaat[telaf] = wissel;

                        tel = tel + 1;
                        telaf = telaf - 1;
```

```
                }
        }

        return resultaat;
}
```

Maar de string `resultaat` is verdwenen na de `return`! Het functieresultaat is een pointer die wijst naar de plaats waar de string `resultaat` heeft gestaan. Als je niet te veel andere functies aanroept, zal dat stukje geheugen misschien toevallig niet verstoord worden, en *lijkt* het programma verder goed te lopen, totdat dat geheugen toch gebruikt wordt en er op onverwachte plaatsen in het programma heel rare dingen gebeuren. De oplossing is `malloc()` gebruiken:

```
/* maak een string die de inhoud van het getal bevat;
 * bv. maakString(42) geeft de string "42" */
char *maakString(long getal)
{
        char *resultaat;
        int lengte;

        resultaat = malloc(20);
        if (resultaat == NULL) return NULL;

        /* Bijzonder geval. Als we deze regel zouden weglaten,
         * zou het resultaat van maakString(0) een lege string zijn
         */
        if (getal == 0) return "0";

        lengte = 0;

        /*** enzovoort --- zie vorige versie ***/

        return resultaat;
}
```

Op het eerste gezicht hebben we hier nog steeds hetzelfde probleem: `resultaat` is verdwenen na de `return`. Maar het blokje geheugen waar `resultaat` naar wijst, blijft bestaan: ge`malloc()`t geheugen wordt *alleen* vrijgegeven als je het expliciet vrijgeeft met `free()`. De variabele `resultaat` verdwijnt dus, maar niet de string waar ze naar wijst.

De GNU C compiler waarschuwt overigens met `warning: function returns address of local variable` voor dit soort fouten.

▸ **OEFENING 3.12**
Deze versie van `maakString()` reserveert altijd 20 bytes geheugen, terwijl het resultaat zeker nooit zo lang is. Schrijf een zuiniger versie, die precies genoeg bytes reserveert.

◇

In `string.h` zitten een hoop handige stringfuncties, die ik nu zal overlopen.

3.6.1 `strcmp()`, `strncmp()`, `strcasecmp()`, `strncasecmp()`

De prototypes van deze functies zijn:

```
int strcmp(const char *s1, const char *s2);
int strncmp(const char *s1, const char *s2, size_t n);
int strcasecmp(const char *s1, const char *s2);
int strncasecmp(const char *s1, const char *s2, size_t n);
```

`size_t` is o.a. in `string.h` gedefinieerd als een geheel datatype (dus iets soortgelijks als `int`) en `const` betekent dat de functie de string niet wijzigt (zie §4.1.6).

`strcmp()` vergelijkt de strings met elkaar.

- Als de eerste string alfabetisch vóór de tweede komt, is het resultaat van `strcmp()` kleiner dan nul.
- Zijn de strings gelijk, dan is het resultaat nul.
- Komt de eerste string alfabetisch na de tweede, dan is het resultaat groter dan nul.

`strncmp()` doet hetzelfde, maar vergelijkt hoogstens de eerste `n` letters van beide strings. Als één van beide strings minder dan `n` letters heeft, doet `strncmp()` natuurlijk precies hetzelfde als `strcmp()`.

Het begrip "alfabetisch" moet je wel met een korreltje zout nemen. Er wordt vergeleken volgens de lettercodering van het gebruikte systeem. In ASCII komen de hoofdletters bijvoorbeeld vóór de kleine letters (zie bijlage A), dus `strcmp("AB","aa")` is kleiner dan nul. Dit wordt wel eens "ASCIIbetische volgorde" genoemd.

De functies `strcasecmp()` en `strncasecmp()` behandelen kleine letters en hoofdletters als gelijk, dus `strcasecmp("AB","aa")` is groter dan nul. Deze twee functies zijn niet beschikbaar bij alle C compilers (ze behoren enkel tot de BSD UNIX standaard).

3.6.2 strcpy(), strncpy()

Deze functies hebben de volgende prototypes:

```
char *strcpy(char *doel, char *bron);
char *strncpy(char *doel, char *bron, size_t n);
```

De string `bron` wordt gekopieerd in de string `doel`, inclusief de nulbyte aan het eind. Als die string niet in `bron` past, kunnen er rare dingen gebeuren (omdat `strcpy()` gewoon blijft door kopiëren totdat `bron` ten einde is, en dus het geheugen na `doel` overschrijft). Het is daarom veiliger `strncpy()` te gebruiken. Als de `bron`string minder dan `n` tekens lang is, wordt de rest van `doel` opgevuld met nulbytes. Maar als de `bron`string `n` of meer tekens bevat, zal `strncpy()` in het resultaat geen nulbyte aanbrengen. Om zeker te zijn dat het resultaat netjes een nulbyte heeft, is het dus nodig om expliciet de laatste byte zelf op nul te zetten:

```
void kopieer(char *string)
{
        char buffer[16];

        /* Kopieer hoogstens 15 tekens.
         * Als de string 14 tekens of minder bevat, brengt strncpy()
         * zelf een nulbyte aan.
         */
        strncpy(buffer,string,15);
```

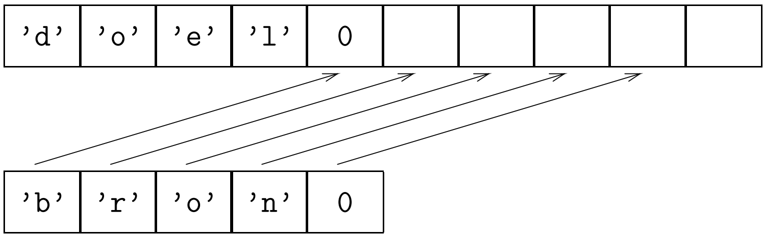
```
        buffer[15] = 0; /* voor wanneer de string 15 tekens of langer is */

        /* de rest van de functie */
}
```

Beide functies geven als resultaat het `doel` argument terug. In het voorbeeld heb ik dat resultaat niet gebruikt (zoals we in §4.3.5.1 zullen zien mag je in C altijd het resultaat van een functie weggooien).

3.6.3 strcat, strncat

```
char *strcat(char *doel, const char *bron);
char *strncat(char *doel, const char *bron, size_t n);
```

Deze functies plakken twee strings aan elkaar. Dat gebeurt door `bron` achter `doel` te kopiëren. Alle letters van `doel` blijven dus gewoon staan, en de `bron`string wordt over de nulbyte van `doel` en de bytes die erop volgen heen geschreven. Beide strings mogen elkaar niet overlappen.



De werking van `strcat()`

De `strncat()` functie doet net hetzelfde, maar kopieert hoogstens `n` tekens uit `bron`. Er wordt achteraan altijd een nulbyte toegevoegd (die niet meegeteld wordt; `strncat()` wijzigt dus `n`+1 bytes als de `bron`string uit `n` of meer letters bestaat).

Het resultaat van beide functies is `doel`.

3.7 Arrays en pointers: een voorbeeld

Omdat in C arrays en pointers nogal goed bevriend zijn, is het ook makkelijk de draad kwijt te raken. Vandaar is een expliciet voorbeeldje wel op zijn plaats. Ik zal laten zien op welke manieren je zoal een lijstje strings kan bijhouden.

3.7.1 Een tweedimensionale array van chars

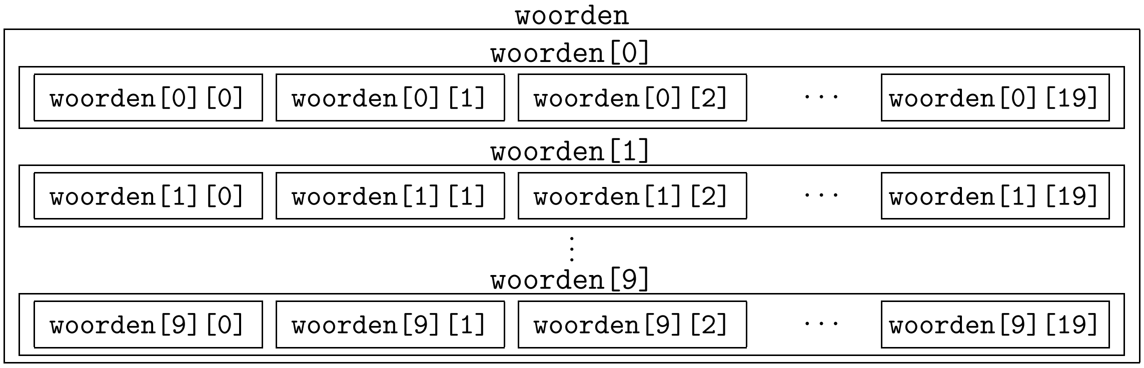
Met

```
char woorden[10][20];
```

maak ik zo'n array. Deze declaratie moet je op een soort binnenste-buiten manier lezen:

- `woorden[10][20]` is een `char`.
- `woorden[10]` is een array van 20 `chars`.
- `woorden` is een array van tien arrays van 20 `chars`.

Dat laatste is alleen begrijpelijk met een tekeningetje:



Deze techniek is de eenvoudigste: met één regeltje heb je genoeg om de hele tabel aan te maken; het enige wat nog moet gebeuren is de tabel met lege strings vullen.

3.7.2 Een array van pointers naar `chars`

We willen dat `woorden` een array van tien `char *`'s moet worden, dus

```
char *woorden[10];
```

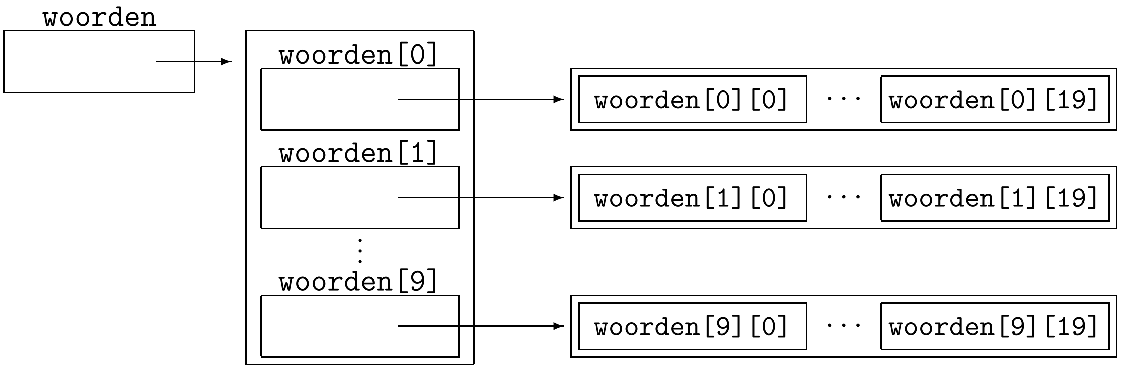
is wat we moeten schrijven. Helaas is de kous daarmee niet af, want dat maakt enkel een tabel van tien pointers, die zelfs niet eens geïnitialiseerd zijn. We hebben dus enkel nog maar het linkse stukje uit de figuur. Er is enig werk nodig om alles op orde te krijgen:

```
{
        int tel;

        tel = 0;
        while (tel < 10) {
                woorden[tel] = malloc(20);
                woorden[tel][0] = 0;
                tel = tel + 1;
        }
}
```

Voor het gemak heb ik de array ook al gevuld met lege strings.

Deze aanpak is wat lastiger dan de vorige, maar heeft als voordeel dat we niet verplicht zijn voor alle strings 20 bytes te reserveren—als een string korter is, kunnen we geheugen sparen, en als we een langere string willen bewaren, kan dat.

3.7.3 Een pointer naar een array van pointers naar chars



Nu moeten we

```
char **woorden;
```

schrijven. Ondanks het feit dat `woorden` naar een array wijst, komt er hier geen `[]` aan te pas. Dat komt omdat `woorden` zelf maar een pointer is; we moeten de tabel van tien pointers dus ook nog eens zelf aanmaken:

```
{
        int tel;

        woorden = malloc(10 * sizeof(char *));
        tel = 0;
        while (tel < 10) {
                woorden[tel] = malloc(20);
                woorden[tel][0] = 0;
                tel = tel + 1;
        }
}
```

Het voordeel van deze manier van werken boven de vorige is dat we nu ook makkelijk het aantal opgeslagen woorden kunnen laten variëren tijdens de loop van het programma door bijvoorbeeld

```
woorden = malloc(aantal_gewenste_woorden * sizeof(char *));
```

te schrijven.

Merk op dat we met alle drie de methoden de uiteindelijke letters van de woorden kunnen aanspreken met

```
woorden[woordnr][letternr]
```

3.8 structures

Stel dat we een programma willen maken om namen en adressen bij te houden. We zouden dan zo te werk kunnen gaan:

```
#define MAXAANTAL 100
int main(void)
{
        char naam[MAXAANTAL][40];
        char voornaam[MAXAANTAL][20];
        char straat[MAXAANTAL][30];
        int huisnr[MAXAANTAL];
        /* enzovoort ... */
}
```

en we zouden dan een aantal hulpfuncties kunnen schrijven, zoals

```
void drukAdres(char *naam, char *voornaam, char *straat, int huisnr)
{
        printf("    Naam: %s\n",naam);
        printf("Voornaam: %s\n",voornaam);
        printf("   Adres: %s %d\n",straat,huisnr);
}
```

Om dan het `zoveel`ste adres uit de lijst af te drukken moet je dan

```
drukAdres(naam[zoveel],voornaam[zoveel],straat[zoveel],huisnr[zoveel]);
```

schrijven. Wat erger is: telkens je beslist nog een extra array aan te maken (`int postcode[MAXAANTAL]` of zo) moet je overal in het programma aanpassingen maken. Dat is niet zo leuk en daarom heeft C de mogelijkheid om nieuwe datatypes te maken: de **typedeclaratie**

```
struct persoon {
        char naam[40];
        char voornaam[20];
        char straat[30];
        int huisnr;
        /* enzovoort */
};      /* Let op het kommapunt hier! */
```

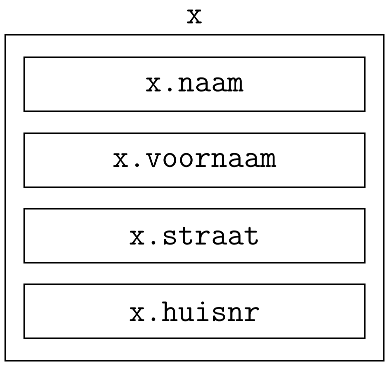
maakt een nieuw datatype aan dat `struct persoon` heet. Let wel: we hebben hier nog *geen* variabele gemaakt, alleen maar een *beschrijving* van hoe variabelen ons nieuwe type er zullen uit zien. Typedeclaraties kunnen overal in het programma staan waar variabelendefinities kunnen staan. Net zoals je een variabele van het type `int` kan maken met

```
int x;
```

kan je een variabele van ons nieuwe type maken met

```
struct persoon x;
```

Wat kunnen we nu allemaal stoppen in een `struct persoon` variabele? In feite bestaat zo'n variabele uit (in dit geval) vier stukken (de zgn. **leden** (**members**) van een struct), die we met de **punt-operator** kunnen aanspreken: `x.naam`, `x.voornaam`, `x.straat` en `x.huisnr`, die zich netjes als `char[]` en `int` gedragen. Een `struct` is dus niets meer dan een soort verpakking rond een aantal bijeenhorende variabelen.

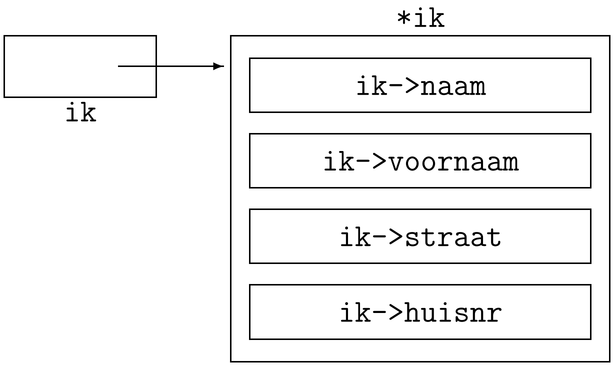


Structures zijn echter vooral heel handig in combinatie met pointers, zoals uit de volgende toepassingen zal blijken.

Zoals gewoonlijk blijft een typedeclaratie zichtbaar tot het einde van het blok. Je kan typedeclaraties ook globaal maken door ze buiten het blok van een functie te schrijven.

3.8.1 de `->` operator

Stel dat `ik` een variabele van het type `struct persoon *` is (dus een pointer naar een `struct persoon`). Als je dan de elementen van die variabele wilt gebruiken, moet je iets als `(*ik).naam` schrijven. Omdat zulke constructies veel voorkomen, en dat gepruts met sterretjes, punten en haakjes niet echt elegant is, kan je dat in C afkorten tot `ik->naam`.



▸ **OEFENING 3.13**
Waarom schreef ik niet `*ik.naam`?

◇

3.8.2 Gelinkte lijsten

Het adreslijstprogramma van daarnet heeft een nadeel: we moeten op voorhand de grootte van de arrays opgeven (met de define `MAXAANTAL`). Als je meer personen wil verwerken, zit er niets anders op dan `MAXAANTAL` aan te passen en het programma te hercompileren. Gebruik je daarentegen veel minder personen dan het maximum, dan is het natuurlijk zonde van al dat gereserveerde geheugen ...

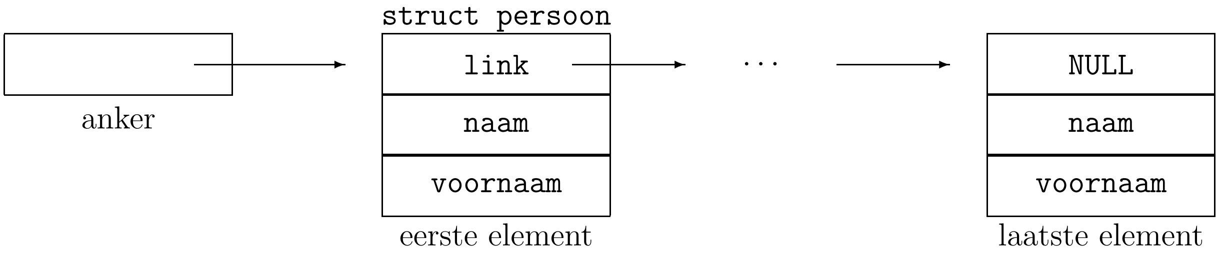
Dat alles kan opgelost worden met gelinkte lijsten, die kunnen groeien en inkrimpen tijdens de loop van het programma.

Het idee van een gelinkte lijst is het volgende: elk element van de lijst bevat een pointer naar het volgende element ervan (bij het laatste element van de lijst is die pointer `NULL`).

De lijst bestaat dus uit een aantal `structs` (in het voorbeeld heb ik ze `struct persoon` genoemd) die aan elkaar gelinkt zijn door pointers (in het voorbeeld het `link` lid):

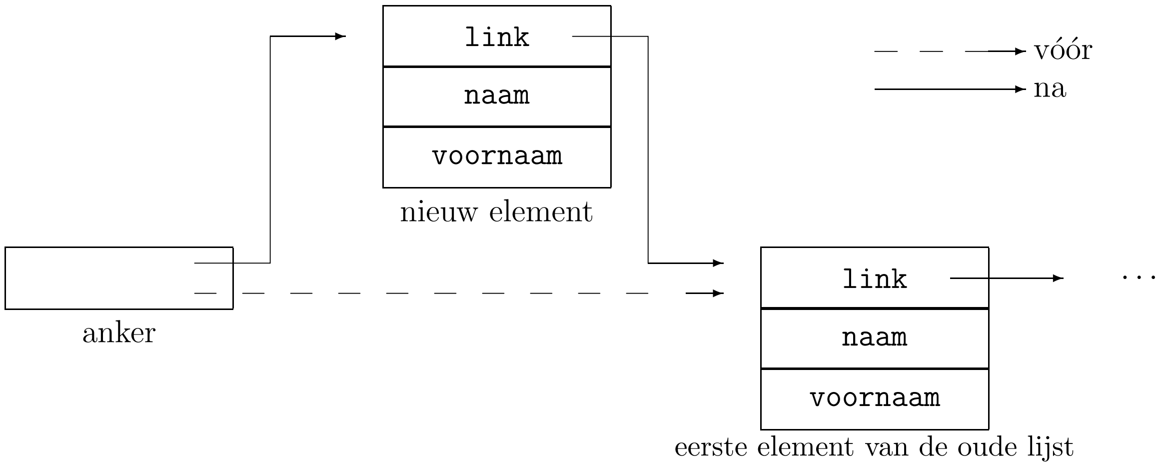
```
struct persoon {
        struct persoon *link;
        char naam[40];
        char voornaam[20];
};
```

Meestal wordt zo'n gelinkte lijst bijgehouden door een pointer naar het eerste element ervan te onthouden. Zo'n pointer heet een **anker**. Als de lijst leeg is, bevat het anker `NULL`.



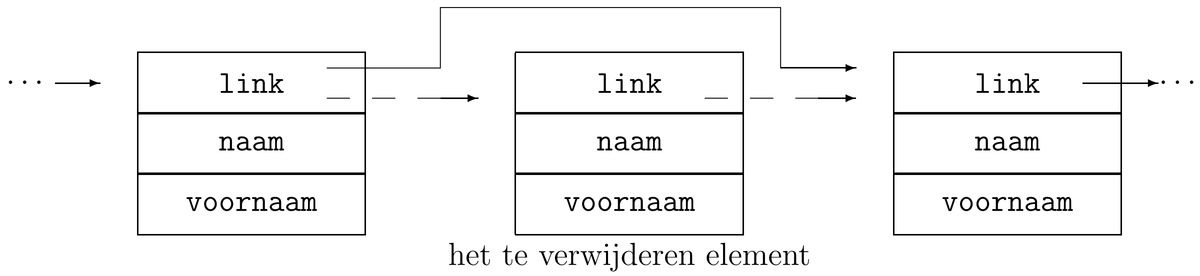
Een gelinkte lijst

Willen we een element bijvoorbeeld vooraan de lijst toevoegen, dan volstaat het om de pointer in dat nieuwe element naar het begin van de lijst te doen wijzen.



Een element vooraan een gelinkte lijst toevoegen

Elementen verwijderen gaat ook makkelijk: laat de pointer in het element vóór het te verwijderen element wijzen naar het element erna.



Een element verwijderen uit een gelinkte lijst

Merk op dat we in de definitie van `struct persoon` al een pointer naar een `struct persoon` kunnen definiëren. Dit is geen recursieve definitie omdat de grootte van een pointer altijd gekend is, onafhankelijk van de grootte van hetgeen naar waar die pointer wijst. (Wat natuurlijk niet mag is het opnemen van een `struct persoon` zelf in de definitie van `struct persoon`.)

Een variant hierop is een **dubbel gelinkte lijst** (**doubly linked list**): elk element bevat niet alleen een pointer naar het volgende element van de lijst, maar ook naar zijn voorganger.

Afhankelijk van hetgeen waarnaar het laatste element in een lijst wijst (bij dubbel gelinkte lijsten: waarnaar de opvolger van het laatste element wijst en waarnaar de voorganger van het eerste element wijst) spreken we van **lineaire lijst**en als dat `NULL` is (bij dubbel gelinkte lijsten is de opvolger van het laatste element `NULL` en de voorganger van het eerste ook) of **circulaire lijst**en als dat het eerste element is (bij dubbel gelinkte lijsten: de voorganger van het eerste element is het laatste element; de opvolger van het laatste element is het eerste).

Een voorbeeldje van een programma dat met een lineaire lijst werkt:

linlijst.c

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

struct persoon {
        struct persoon *link;
        char naam[40];
        char voornaam[20];
};

/* in: lijst  anker van de lijst
 *     nieuw  in te voegen element
 * uit: anker van de nieuwe lijst
 */
struct persoon *voegVooraanToe(struct persoon *lijst,struct persoon *nieuw)
{
        nieuw->link = lijst;
        return nieuw;
}

/* in: anker van de af te drukken lijst */
void drukLijst(struct persoon *lijst)
{
        struct persoon *volg;

        volg = lijst;
        while (volg != NULL) {
                printf("%s %s\n",volg->naam,volg->voornaam);
                volg = volg->link;
        }
}

void freeLijst(struct persoon *lijst)
{
        struct persoon *volg;
```

```
        volg = lijst;
        while (volg != NULL) {
                struct persoon *volgende;

                volgende = volg->link;
                free(volg);
                /* op dit punt is volg->link ongeldig!
                 * Daarom heb ik die pointer eerder bewaard in 'volgende'. */
                volg = volgende;
        }
}

int main(void)
{
        struct persoon *lijst;   /* anker */
        struct persoon *ik;

        lijst = NULL;
        ik = malloc(sizeof(struct persoon));
        strcpy(ik->naam,"Vernaeve");
        strcpy(ik->voornaam,"Geert");
        /* merk op dat ik->link niet geinitialiseerd is */
        lijst = voegVooraanToe(lijst,ik);

        ik = malloc(sizeof(struct persoon));
        strcpy(ik->naam,"Ritchie");
        strcpy(ik->voornaam,"Dennis");
        lijst = voegVooraanToe(lijst,ik);

        drukLijst(lijst);

        freeLijst(lijst);

        return 0;
}
```

Je kan het programma uitbreiden met functies die andere bewerkingen op lijsten doen; de volgende oefeningen geven een paar ideetjes. In de oplossingen vind je de versie voor lineaire lijsten; je kan natuurlijk hetzelfde doen met circulaire lijsten, en met de dubbel gelinkte varianten ervan.

▸ **OEFENING 3.14**
Schrijf een functie `struct Persoon *zoekPersoon (struct Persoon *lijst, char *naam, char *voornaam)` die in de lijst een bepaalde persoon zoekt. Het functieresultaat is een pointer naar de gevonden persoon, of `NULL` als de persoon niet in de lijst gevonden werd.

◇

▸ **OEFENING 3.15**
Schrijf een functie `struct Persoon *verwijderPersoon (struct Persoon *lijst, char *naam, char *voornaam)` die een persoon uit de lijst verwijdert. Het functieresultaat

is het anker van de gewijzigde lijst.

◊

▸ **OEFENING 3.16**
Schrijf een functie `struct Persoon *voegAchteraanToe (struct Persoon *lijst, struct Persoon *nieuw)` die een nieuwe persoon achteraan de lijst toevoegt. Het functieresultaat is ook hier een anker van de gewijzigde lijst. Veronderstel dat het `link` element van `nieuw` nog niet correct ingevuld is (de functie moet dat dus zelf doen).

◊

▸ **OEFENING 3.17**
Schrijf een functie `struct Persoon *plakLijst (struct Persoon *lijst1, struct Persoon *lijst2)` die achteraan de eerste lijst de tweede hangt. Het functieresultaat is een anker van de nieuwe lijst.

◊

▸ **OEFENING 3.18**
Schrijf een functie die een lijst sorteert. (Niet zo makkelijk.)

◊

Je kan de `for` opdracht handig gebruiken om gelinkte lijsten te doorlopen:

```
void drukLijst(struct persoon *lijst)
{
        struct persoon *volg;

        for (volg = lijst; volg != NULL; volg = volg->link) {
                printf("%s %s\n",volg->naam,volg->voornaam);
        }
}
```

3.8.3 Toepassing: mungwall

Problemen met `malloc()` en `free()` zijn soms lastig op te sporen. Daarom zullen we een programmaatje ontwikkelen dat deze functies ‘omleidt’ naar zelfgemaakte functies `mung_malloc()`, `mung_free()`, ...

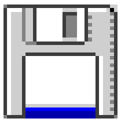
3.8.3.1 Mungwall gebruiken

Het gebruik van mungwall is eenvoudig: schrijf in elk `.c` bestand van je programma een `#include "mungwall.h"`. Je kan mungwall volledig uitschakelen door `NDEBUG` te definiëren, dus door `#define NDEBUG` in het `.c` bestand op te nemen. Op die manier kan je tijdens het ontwikkelen van het programma de mungwall functies gebruiken (waardoor het programma wat trager zal lopen), en eens alles stabiel draait makkelijk een productie-versie maken die op volle snelheid draait zonder mungwall. Als je aan `gcc` de optie `-Dsymbool` meegeeft bij het compileren, doet de compiler net of er een extra regel `#define symbool` in het C bestand staat. Je kan dus door `-DNDEBUG` als optie te geven mungwall uitschakelen zonder dat je de broncode moet wijzigen.

Het is de bedoeling dat je `mungwall.c` samen met je eigen `.c` bestand compileert. De eenvoudigste manier hiervoor is bij `gcc` meerdere `.c` bestanden te vermelden:

```
gcc mijnprog.c mungwall.c -o mijnprog
```

volstaat meestal wel. Wie geavanceerder wil werken, leest §9.1. In `mijnprog.c` moet je de regel `#include "mungwall.h"` opnemen. Een voorbeeld:


mungtest.c

```
#include <stdio.h>
#include <malloc.h>
#include "mungwall.h"

void main(void) {
        char *buf;
        int tel;

        buf = malloc(100);

        /* vul buf op met nullen */
        tel = 0;
        while (tel <= 100) {
               buf[tel] = 0;
               tel = tel + 1;
        }

        free(buf);
}
```

Als we dit programma compileren met `gcc mungtest.c mungwall.c -o mungwall` en doen draaien, dan krijgen we dit te zien:

```
checkmem(): memory 0x8049cf8 bad in back wall, 0th byte
checkmem() bailing out at mungtest.c:18
Aborted
```

Er is een fout in het geheugenblok op adres 0x8049cf8 gevonden (het precieze adres kan verschillen van uitvoering tot uitvoering). De 0x geeft aan dat dit getal niet in het tientallige, maar in het 16-tallige stelsel staat, wat niet echt veel uitmaakt omdat zoals gezegd het precieze getal weinig uitmaakt—meer hierover verder.

Wat is er nu precies gebeurd? Mungwalls versie van `alloc()` reserveert 80 bytes meer geheugen dan het programma vraagt: 40 bytes aan het begin ervan en 40 bytes aan het eind. Dit vormen twee 'muren' (walls) rond het gevraagde geheugenblok. Mungwall vult de muren op met bepaalde waarden, zodat het kan zien of de muren intact zijn gebleven als je het geheugen vrijgeeft. Nu blijkt dat de nulde byte van de achterste muur beschadigd is. Als je het programma nog eens goed bekijkt, kan je zien waarom: we vragen 100 bytes, maar zetten er 101 op nul! Als je de `while` regel corrigeert tot

```
while (tel < 100)
```

en hercompileert, loopt alles goed.

Een methode om nog meer informatie te krijgen, is het symbool `VERBOSE` te definiëren. Compileer dus met `gcc -DVERBOSE mungtest.c mungwall.c -o mungwall` en draai het programma. Je ziet iets als

```
mungtest.c:9 malloc(100)=0x8049e18
mungtest.c:18 free(0x8049e18)
```

gevolgd door de drie lijnen die mungwall vroeger ook al gaf. We zien dat op regel 9 een stuk geheugen van 100 bytes groot gereserveerd is vanaf adres 0x8049e18. De precieze waarde heeft zoals eerder gezegd niet veel belang (wie goed kijkt, heeft gemerkt dat ze verschilt van toen we de eerste keer het programma draaiden), maar we kunnen wel zien dat exact dezelfde pointer teruggegeven wordt aan `free()` op regel 18 van het programma. Bij grotere programma's kan je op deze manier nagaan op welke plaats in het programma een stuk geheugen gealloceerd geweest is dat verderop rare kuren begint te vertonen.

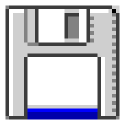
Als laatste extraatje kan je ook nog een functie `checkmem()` in je programma gebruiken. Deze functie heeft geen argumenten en geeft ook niets terug. Ze controleert of alle eerder gereserveerde geheugenblokken nog in orde zijn. Door op strategische plaatsen `checkmem()` aanroepen in je programma te strooien, kan je te weten komen vanaf welke regel het fout begint te gaan.

3.8.3.2 Implementatie

There are two approaches to software design:
one is to make it so simple that there are obviously no deficiencies;
the other is to make it so complicated that there are no obvious deficiencies.

In deze paragraaf bespreek ik hoe we mungwall zelf kunnen programmeren. Een aantal technieken hebben we op dit punt van de cursus nog niet gezien; beginnende lezers kunnen dit stuk met een gerust hart overslaan.

Het omleiden van de functies `alloc()`, `free()` en dergelijke doen we door enkele `#define`s: we schrijven in `mungwall.h`



mungwall.h

```
#ifndef NDEBUG

#include <malloc.h>
void mung_checkmem(int,char *,int);
void *mung_malloc(size_t,int,char *);
void *mung_realloc(void *,size_t,int,char *);
void *mung_calloc(size_t,size_t,int,char *);
void mung_free(void *,int,char *);
#ifndef MUNGPRIVATE
# define malloc(s) mung_malloc(s,__LINE__,__FILE__)
# define realloc(m,s) mung_realloc(m,s,__LINE__,__FILE__)
# define calloc(s,t) mung_calloc(s,t,__LINE__,__FILE__)
# define free(m) mung_free(m,__LINE__,__FILE__)
# define checkmem() mung_checkmem(__LINE__,__FILE__,1)
#endif

#else
#define checkmem()
#endif
```

We hebben nu alle aanroepen van `malloc()` en aanverwanten naar `mung_malloc()` omgeleid. In `mungwall.c` maken we die functies. De bedoeling is dat `mung_malloc()` uiteindelijk de echte `malloc()` aanroept, maar ook wat extra werk achter de schermen doet:

- Vlak vóór en na het gevraagde stuk geheugen reserveert `mung_malloc()` nog 40 extra bytes. De muur aan het begin van het blok wordt opgevuld met 0, 1, . . . , 39 en die aan het eind met 39, 38, . . . , 1, 0. Op die manier kunnen we later in het programma kijken of het programma niet per ongeluk bytes vóór of na het gevraagde stuk geheugen heeft overschreven; het is een soort 'muur' rond het echte geheugenblok heen. De reden dat ik de achterste muur met 39, 38 enz. vul en niet met 0, 1, enz. zoals de voorste muur is dat anders de vaak voorkomende fout van het voorbeeldprogramma niet gedetecteerd kan worden: in het voorbeeld wordt een nulbyte in de eerste byte van de achterste muur gestopt. Als die byte normaal zou moeten bevatten, kan `checkmem()` natuurlijk niets verdachts opmerken.

- De eerste paar bytes (hoeveel precies hangt af van het aantal bytes dat een `long` op je systeem inneemt) van het extra blok (die dus 0, 1, 2, enz. bevatten) gebruiken we echter om de grootte van het gevraagde blok in te bewaren. Op die manier kan `mung_free()` weten hoe groot het blok was. `mung_free()` heeft die grootte nodig om te weten waar de 'muur' van 40 bytes na het geheugenblok ergens staat.

```
void *mung_malloc(size_t size,int line,char *file)
{
        char *mem = malloc(size + 2*WALLSIZE);
        int i;

        for (i=0; i<WALLSIZE; i++) {
                mem[i] = i;
                mem[i+size+WALLSIZE] = i;
        }
        *((long *)mem) = size;
        addchunk(mem,size);
        mung_checkmem(line,file,0);
#ifdef VERBOSE
        fprintf(stderr,"%s:%d malloc(%d)=%p\n",file,line,size,mem+WALLSIZE);
#endif
        return mem + WALLSIZE;
}
```

Zoals het nette programma's past, heb ik de grootte van de muur (40 bytes) niet rechtstreeks in het programma gezet, maar een `#define` gemaakt met naam `WALLSIZE`. Je kan ook bij het compileren van mungwall het symbool `VERBOSE` definiëren; in dat geval genereert mungwall bij elke geheugenoperatie (geheugen reserveren of vrijgeven) een regel met informatie op de uitvoer (wat dat `fprintf()` gedoe precies betekent, kan je in een verder hoofdstuk uitpluizen).

De `addchunk()` aanroep zorgt ervoor dat de informatie over het geheugenblok (beginadres en grootte in bytes) in een gelinkte lijst wordt bijgeschreven. De functie `mung_checkmem()` gaat die hele lijst af en controleert blok voor blok of de muur nog volledig in orde is. Daarvoor maakt deze functie gebruik van de functie `checkchunk()`, die één blok controleert. Het blok wordt in de volgende datastructuur beschreven:

```
struct memchunk {
        struct memchunk *next;
        void *memory;   /* size+2*WALLSIZE bytes from here, or NULL */
```

```
        long size;
};
```

en checkchunk() ziet er zo uit:

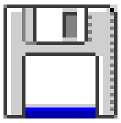
```
/* Check if the wall is still untouched
 * return value: 0 if chunk is OK, 1 if it is damaged */
static int checkchunk(struct memchunk *chunk) {
        int i;
        int bad = 0;

        for (i = sizeof(long); i < WALLSIZE; i++) {
                if (((char *)chunk->memory)[i] != i) {
                        fprintf(stderr,"checkmem(): memory %p bad in"
                                " front wall, %dth byte\n",
                                (char *)chunk->memory+WALLSIZE,i);
                        bad = 1;
                }
        }
        for (i = 0; i < WALLSIZE; i++) {
                if (((char *)chunk->memory)[i+chunk->size+WALLSIZE]
                                != WALLSIZE-i-1) {
                        fprintf(stderr,"checkmem(): memory %p bad in"
                                " back wall, %dth byte\n",
                                (char *)chunk->memory+WALLSIZE,i);
                        bad = 1;
                }
        }
        return bad;
}
```

De free() aanroepen worden doorgesast naar mung_free():

```
void mung_free(void *ptr,int line,char *file)
{
        struct memchunk *chunk;

#ifdef VERBOSE
        fprintf(stderr,"%s:%d free(%p)\n",file,line,ptr);
#endif
        chunk = findchunk((char *)ptr - WALLSIZE);
        if (!chunk) {
                fprintf(stderr,"free(%p) at %s:%d mem not in memory list\n",
                        ptr,file,line);
                abort();
        }
        mung_checkmem(line,file,0);
        chunk->memory = NULL;
        free((char *)ptr - WALLSIZE);
}
```

De functie findchunk() doorzoekt de lijst met geheugenblokken (eerder aangemaakt door mung_malloc(). Merk op dat ik de chunk niet vrijgeef: ze blijft in de lijst staan, maar de pointer naar het geheugenblok (`chunk->memory`) wordt op `NULL` gezet. Het idee hierachter is dat een extra `free()` het programma nog meer zou vertragen dan nu al het geval is. Of dat ook echt zo is op je systeem is een interessante vraag ... :-)

De rest van de broncode kan je op de website vinden.



mungwall.c

Merk tenslotte nog op dat de implementatie verre van optimaal is: we bewaren alle informatie over de gereserveerde gebeugenblokken in een lineaire lijst. Dat betekent dat er relatief veel tijd nodig is om een bepaald blok in de lijst te vinden, zeker als het programma in kwestie veel geheugenblokken aanmaakt. Er zijn slimmere manieren om zo'n lijst te bewaren, maar die zouden dan weer de code van mungwall ingewikkelder maken, en bij dit soort programma's is het veel belangrijker dat je zeker bent dat ze correct draaien, dan dat ze snel draaien. Als we er niet vrij overtuigd van zijn dat mungwall zelf geen fouten bevat, is het een eerder nutteloze operatie geweest zo'n programma in te proberen zetten om fouten uit andere programma's proberen te halen! Dit soort redenering wordt wel vaker gevolgd en dan aangeduid als het KISS-principe (Keep It Simple Stupid): liever een programma dat wat trager werkt, maar robuust is, dan een programma dat heel snel een foutief resultaat berekent.

Hoofdstuk 4

C syntax: de laatste finesses

Tot hier toe heb ik net genoeg van C uitgelegd om "aan de slag te kunnen". In dit hoofdstuk zal ik je wat meer de echte smaak van C laten proeven. Je zal nooit meer zoiets vreselijks als `teller = teller + 1` hoeven te schrijven (met al mijn excuses aan de C puristen die zich al die tijd al staan te ergeren hebben :-))

4.1 Datatypes

4.1.1 Gehele getallen

We hebben al de datatypes `char` en `int` gezien, waar telkens gehele getallen in pasten. Hier volgt het volledige arsenaal van gehele datatypes van C:

type	grootte	bereik (unsigned)	bereik (signed)
`char`	8 bits	0 t/m 255	-128 t/m 127
`short`	16 bits	0 t/m 65.535	-32.768 t/m 32.767
`int`	16/32 bits		
`long`	32 bits	0 t/m 4.294.967.295	-2.147.483.648 t/m 2.147.483.647
`long long`	64 bits	0 t/m 18.446.744.073.709.551.615	-9.223.372.036.854.775.808 t/m 9.223.372.036.854.775.807

De groottes in bits en de bereiken van de datatypes liggen niet vast, maar op de meeste gangbare systemen zijn ze zoals aangegeven. Het enige dat je zeker weet is dat een `short` minstens zo groot is als een `char`, een `int` minstens zo groot als een `short`, enz. Een `short` is gegarandeerd minstens 16 bits groot. Het type `long long` is niet standaard, maar wordt door o.a. de GNU C en de Sun Solaris compilers ondersteund.

In plaats van `long` mag je ook `long int` schrijven, en in plaats van `short` ook `short int`.

Verder hebben al deze datatypes een `signed` en een `unsigned` variant, waar je respectievelijk positieve en negatieve of alleen maar positieve getallen in kwijt kunt. De C compiler kiest standaard voor `signed`, behalve bij `char`, waar afhankelijk van de compiler `signed` of `unsigned` gebruikt wordt. Kiezen voor één van de twee varianten doe je door het woord `signed` of `unsigned` vóór het type te zetten:

```
unsigned long x;     /* 0..4 miljard */
long int y;          /* -2 miljard .. +2 miljard */
unsigned char a;     /* 0..255 */
char b;              /* 0..255 of -128..127 maar in elk geval goed
                      * om letters ('a', 'b', ...) in te bewaren */
```

```
signed char c;      /* -128..127 */
```

Gehele constanten kan je zoals gezien noteren als `123` (het type is `int` of, als de constante te groot is, `long`) of `'c'` (het type is ook `int`—en niet `char`). Voorts kan je een getal in hexadecimale notatie schrijven door er `0x` vóór te schrijven (bv. `0x1A` is hetzelfde als `26`) of in octale notatie door er een `0` vóór te schrijven (bv. `032` is hetzelfde als `26`). Je kan een getal expliciet `long` maken door er een `l` of `L` achter te schrijven (de hoofdlettervorm is aan te raden, omdat een `l` vaak verdacht goed op een `1` lijkt), en/of unsigned door er een `u` of `U` achter te schrijven. Je mag de laatste twee combineren: `23LU` is een `unsigned long`.

4.1.2 Kommagetallen (floating point datatypes)

Naast het datatype `double` bestaat er ook nog `float`, dat een kleiner bereik heeft en minder precisie dan `double`, maar ook minder plaats inneemt (en op sommige systemen iets sneller rekent). Verder is er ook `long double`, dat minstens zo nauwkeurig werkt als `double`. De floating point types kennen geen `signed` of `unsigned`, er passen altijd ook negatieve getallen in.

Floating point constanten worden in **wetenschappelijke notatie** geschreven, bv.

$$\underbrace{\texttt{12.347777}}_{\text{mantisse}}\underbrace{\texttt{E+5}}_{\text{exponent}}$$

heeft als waarde 12.347777×10^5 (of dus `1234777.7`). Niet alle delen van een float constante zijn verplicht; de enige twee regels zijn dat je niet tegelijkertijd het deel vóór en na de komma van de mantisse mag weglaten (dus `.E+45` of zoiets), en dat er minstens een punt óf een `E` in het getal moet voorkomen (anders denkt de compiler dat het een geheel getal is). Je mag ook een kleine letter `e` gebruiken voor de exponent. Geldige floating point constanten zijn bijvoorbeeld: `23.45`, `.45`, `23.`, `23.45E4`, `23.E-24`, `.23E+4`, `23e+4` of `23.E4`. Het type van een floating point constante is `double`, tenzij je er een `f` of `F` achter schrijft (dan is het type `float`), of een `l` of `L` (om een `long double` te maken).

Om je een idee te geven van het bereik en de precisie van kommagetal-datatypes een tabelletje met de waarden van de populaire IEEE 754 standaard (in de kolom "nauwkeurigheid" staat het getal dat nog net van 0 kan onderscheiden worden):

type	grootte	bereik	nauwkeurigheid
`float`	32 bits	±3.4028235E+38	±1.4012985E-45
			(7,22 cijfers na de komma)
`double`	64 bits	±1.7976931348623157E+308	±4.9406564584124654E-324
			(15,95 cijfers na de komma)
`long double`	96 bits	±1.18973149535723176E+4932	±3.64519953188247460E-4951
			(19,27 cijfers na de komma)

Bewerkingen met floating point datatypes zijn van huis uit een beetje onnauwkeurig, omdat er maar een beperkt aantal beduidende cijfers onthouden kunnen worden en afrondingen noodzakelijk zijn. Als je bijvoorbeeld `1 + 1E-20` wil uitrekenen, is het resultaat niet `1.00000000000000000001` maar gewoon `1` omdat er minder dan 20 cijfers na de komma bijgehouden worden. Het is daarom ook raadzaam niet met `if (getal == 0.0)` te testen of een getal nul is, maar met iets in de trant van `if (abs(getal) < 1E-5)`.

4.1.3 Samengestelde datatypes

4.1.3.1 Pointers

Je kan, zoals uitgelegd in §3.3, pointers maken naar om het even welk type (bv. `unsigned long long *` is een pointer naar een, jawel, `unsigned long long`). Een bijzondere soort pointer is het type `void *`, die je zou kunnen opvatten als een "pointer naar een `void`", ware het niet dat er natuurlijk geen variabelen van het type `void` bestaan. Met `void *` bedoelen we gewoon "een" pointer naar "om het even wat". Zo'n variabele wordt automatisch omgezet in een pointer naar een *bepaald* datatype als dat nodig is (je hoeft dus niet expliciet te casten). (C++ programmeurs opgelet: in C++ is de cast wel verplicht.) Bijvoorbeeld:

```
{
        void *geheugen;
        char *tekst;

        geheugen = malloc(1000);
        tekst = geheugen;   /* geheugen wordt gecast naar (char *) */
}
```

Ik zou dus net zo goed `tekst = (char *) geheugen;` kunnen geschreven hebben, wat sommige mensen duidelijker vinden. Het resultaat van `malloc()` is inderdaad van het type `void *`, een geknipte toepassing voor dit datatype.

Ook de omgekeerde cast gebeurt automatisch: een pointer naar een willekeurig datatype wordt gecast naar een `void *`. De functie `free()` heeft bijvoorbeeld als argument een `void *`, en dus kan je zonder meer `free(tekst);` zeggen.

4.1.3.2 Andere samengestelde types

Ik heb al een en ander gezegd over `structs` (§3.8) en arrays (§3.1). Verderop volgt uitleg over `union` (§4.5) en `enum` (§4.7.1).

4.1.3.3 `structs` en padding

De compiler mag extra bytes invoegen tussen de velden van een `struct`. Op sommige systemen *moet* een `int` bijvoorbeeld op een *even* adres staan en in een `struct` als

```
struct Padding {
        char letter;
        int getal;
};
```

moet de compiler een "gat" van één byte tussen `letter` en `getal` inlassen om de `int` op een even adres te krijgen (de compiler verwacht natuurlijk wel dat `struct`ures beginnen op een even adres). Deze extra bytes worden **padding** bytes genoemd.

Om deze gaten zo veel mogelijk te vermijden, is het verstandig variabelen die een "rond" aantal bytes innemen, eerst op te nemen in `structs`, aangezien de compiler de volgorde waarin de variabelen binnen een `struct` bewaard worden niet mag wijzigen. Met "rond aantal" bedoel ik rond in de binaire notatie, wat erop neerkomt dat een variabele die een achtvoud ($2 \times 2 \times 2$) bytes groot is, "ronder" is dan een variabele die een tweevoud bytes groot is. Variabelen die een oneven aantal bytes innemen (bijvoorbeeld `char`s of arrays van `char`s met een oneven lengte) hoop je dus het best zoveel mogelijk achteraan de `struct` op.

In `stddef.h` kan je de macro `offsetof()` vinden. Deze macro geeft de positie aan van een `veld` in een `structuur`, meer bepaald: `offsetof(struct structuur, veld)` geeft het aantal bytes aan die tussen het begin van de `structuur` en het `veld` zitten (dit aantal wordt de **offset** van het veld **vanaf het begin van de structuur** genoemd). Op die manier kan je programma's die moeten weten op welke offset een bepaald veld bewaard wordt, onafhankelijk maken van de gebruikte compiler.

Moraal van het verhaal: een `struct` kan soms meer bytes innemen dan de som van het aantal bytes van de leden afzonderlijk.

4.1.3.4 Declaraties van samengestelde types

We hebben voortdurend eenvoudige declaraties in de trant van `int x;` of `int *y;` gebruikt. Algemeen ziet een declaratie er zo uit:

type declarator;

waarbij *type* een enkelvoudig type is (dat is één van de gehele of kommagetal datatypes, een `struct` *naam*, `union` *naam* of een typedef-naam (zie §4.1.3.5)). De *declarator* is in het eenvoudigste geval de variabelnaam zelf. Declaratoren mogen ook combinaties bevatten van

- de `*` operator, om pointers te maken.
- de `()` operator, om functies te maken. Tussen de haakjes moeten de types van de argumenten staan. De namen van de argumenten hebben geen belang en je mag ze dan ook weglaten; zo maakt `char (*f)(int,int)` of `char (*f)(int x,int y)` een pointer naar een functie die een `char` teruggeeft en twee `int`s nodig heeft. De haakjes rond `*f` zijn wel degelijk nodig, omdat `char *f(int x,int y)` zou betekenen dat `f` een functie is die twee `int`s als argumenten nodig heeft en een `char *` teruggeeft. De `()` operator heeft immers een hogere prioriteit dan de `*` operator, en dus wordt `char *f(int,int)` als `char *(f(int,int))` geïnterpreteerd (de `()` "plakt harder" dan de `*`). In tabel 4.3 kan je de prioriteit van alle operatoren vinden (let wel dat je niet alle operatoren mag gebruiken in een declaratie).

  Een variabele kan nooit een functie zelf bevatten (wel een pointer ernaar), dus `char *f(int,int);` is geen geldige variabelendefinitie (het is een prototype— zie §2.8.3.2). `char *f(int,int);` betekent immers: `f(int,int)` geeft een `char *` terug, m.a.w. `f is een functie met twee ints als argumenten en een char * als resultaat. Een correcte variabelendeclaratie zou zijn: char (*f)(int,int), wat betekent: *f is een functie met twee int argumenten die een char teruggeeft, of dus: f is een pointer naar zo'n functie.`

  `Meer over het gebruik van pointers naar functies vind je in §4.3.6.`

- `de [] operator, om arrays te maken. De grootte van de array moet tussen de haakjes staan; het moet een constante uitdrukking zijn. Wat dus niet mag is`

```
void functie(int grootte)
{
```

```
        int tabel[grootte];    /* grootte  is geen constante! */
    }
```

Door deze manier van werken moet je C declaraties als het ware “binnenste buiten” lezen.

Enkele voorbeelden zijn wel welkom: de functiedefinitie

```
char *f(int x[10],int *y[10])  { ... }
```

maakt een functie `f` die een `char *` teruggeeft en twee argumenten `x` en `y` nodig heeft. `x` is een array van tien `int`s, maar wat `y` precies bevat, is minder duidelijk: is het een pointer naar een array van tien `int`s, of een array van tien pointers naar `int`? De prioriteit van `[]` is hoger dan die van `*`, dus is `int *y[10]` hetzelfde als `int *(y[10])`, wat betekent dat `*(y[10])` een `int` is, dus `y[10]` is een pointer naar een `int`, dus `y` is een array van tien pointers naar `int`. Een pointer naar een array van tien `int`s zou je als `int (*y)[10]` moeten schrijven.

De types van `f`, `x` en `y` zijn respectievelijk `char *(int [10],int *[10])` (een functie die een pointer naar een `char` teruggeeft en twee argumenten van het aangeduide type nodig heeft), `int [10]` (een array van tien `int`s) en `int *[10]` (een array van tien pointers naar `int`). Typenamen krijg je dus eenvoudigweg door de variabelnamen uit de declaraties te schrappen.

▸ **OEFENING 4.1**
Wat wordt de declaratie als we toch liever hebben dat `y` een pointer naar een array van tien `int`s is? Wat is het type van `y` dan?

◇

Als de declaraties te ingewikkeld worden, is het aangeraden gebruik te maken van `typedef`.

4.1.3.5 typedef

Met `typedef` kan je nieuwe type-namen bij maken:

```
typedef int geheel;
```

zorgt ervoor dat `geheel` een synoniem wordt van `int`.

Bij ingewikkelde typedeclaraties kan `typedef` erg verduidelijkend werken. Zo zorgt

```
typedef void (*SignalHandler)(int);
        /* een SignalHandler is dus een pointer naar een functie die
         * geen resultaat teruggeeft en een int als argument heeft */
SignalHandler signal(int signum,SignalHandler handler);

void (*zignal(int signum,void (*handler)(int)))(int);
```

ervoor dat `signal` en `zignal` hetzelfde type hebben.

Omdat het sleutelwoord `typedef` door de C syntax net als een opslagklasse-aanduiding wordt behandeld, zou je theoretisch ook

```
void typedef (*SignalHandler)(int);
```

kunnen schrijven, hoewel het niet echt aangeraden is.

4.1.3.6 Typenamen

Zoals eerder uitgelegd kan je een variabele casten naar een bepaald type door er `(` *type* `)` vóór te schrijven. Namen van complexe types kan je schrijven door gewoonweg de declaratie ervan over te nemen, maar de variabelnaam weg te laten.

We zagen zo dat `char * (*f)(int x[10],int *y[10])` een variabele `f` maakt die een pointer is naar een functie die een `char *` teruggeeft en twee argumenten van de aangegeven types nodig heeft. De naam van het type "pointer naar een functie die een `char *` teruggeeft en als argumenten een array van tien `int`s en een pointer naar een array van tien `int`s nodig heeft" krijgen we dus door die `f` eruit weg te laten: `char * (*)(int x[10],int *y[10])` of korter (aangezien de namen van de argumenten geen belang hebben) `char * (*)(int [10],int *[10])`.

Het spreekt voor zich dat je maar beter een `typedef` gebruikt als je ongezond veel sterretjes begint te zien.

▸ **OEFENING 4.2**
Wat betekenen de types

a) `char *`

b) `char *[42]`

c) `char ***`

d) `char (*)[42]`

e) `char *(void)`

f) `char (*)(void)`

g) `char * (*)[42](void)` en

h) `char * (*)(void)[42]` ?

◇

4.1.4 Initializers

Omdat een constructie als

```
int variabele;

variabele = waarde;
```

veel voorkomt, kan je die in C afkorten tot

```
int variabele = waarde;
```

De `waarde` wordt een **initializer** genoemd.

Welke vorm je gebruikt is een kwestie van smaak; een expliciete toekenning is langer om te schrijven maar soms iets duidelijker (initializers vallen soms weinig op tussen een grote hoop declaraties).

De initializer wordt telkens uitgevoerd als het programma het blok binnenkomt waarin die initializer staat. Het programma uit §2.6.3 kan dus ietsje verkort worden tot

```
#include <stdio.h>

void main(void)
{
        while (1 == 1) {
                char c = getchar();

                if (c == 'g') putchar('h');
                else if (c == 'h') putchar ('g');
                else if (c == '&') break;
                else putchar('c');
        }
}
```

4.1.4.1 Initializers van samengestelde datatypes

Arrays kan je initialiseren door als initializer de elementen tusssen accolades op te sommen:

```
        int x[5] = { 10, 20, 20, 10, 2 };
```

Als je te weinig elementen opgeeft, dan vult de compiler de rest op met nullen.

Je kan ook de compiler de grootte van de array zelf laten bepalen door geen dimensies in te vullen:

```
        int x[] = { 10, 20, 20, 10,2 };
```

Meerdimensionale arrays vullen gaat met hetzelfde systeem:

```
        int matrix[10][5]
        = { { 1, 2, 3, 2, 1 },      /* initializer van matrix[0] */
            { 3, 2, -23, 4, 2 } ,   /* matrix[1] */
            { } ,                   /* matrix[2] met nullen vullen */
            { 2, 3 }       /* de rest met nullen vullen */
           };
```

Als je precies genoeg elementen schrijft, mag je de binnenste accolades van de initializer weglaten (zodat er dus nog maar één paar accolades overblijft waar alle elementen tussen staan). Over het algemeen is het beter de extra accolades te schrijven omwille van de duidelijkheid.

Omdat het nogal lastig is een array van `char`s te initialiseren met

```
        char boodschap[80] = { 'D', 'a', 'g', ' ', 'g', 'r', 'o', 'o', 't',
                'm', 'o', 'e', 'd', 'e', 'r', '!' };
```

is het toegestaan de afkorting

```
        char boodschap[80] = "Dag grootmoeder!";
```

te schrijven. Let wel: dit is gewoon een kortere schrijfwijze; er is hier nergens sprake van pointers in dit voorbeeld (in alle andere gevallen levert een stringconstante immers een pointer naar het begin van de string op). Merk ook op dat ik geen extra nullbyte op het eind van de initializer heb geschreven. De compiler vult de rest immers op met nullen.

Initializers van structs en arrays ervan volgen hetzelfde stramien:

```
struct Produkt {
        int hoeveelheid;
        char naam[80];
        float eenheidsprijs;
};
struct Produkt kassaticket[3] = {
        { 3, "koekje", 20.0 },
        { 1, "appel", 15.0 },
        { 7, "cursus C", 199.95 };
};
```

▸ **OEFENING 4.3**
Beschrijf in woorden wat voor soort ding `namen` is. (Iets in de trant van: een struct van pointers naar structs van arrays van ints die elk weer wijzen naar tien andere . . . enzovoort . . .)

```
char *namen[] = {
        "roodkapje",
        "wolf"
        "houthakker",
        "heks"
};
```

♢

4.1.5 Meervoudige variabelendeclaraties

Je kan meerdere variabelen ineens declareren:

```
        int x,y,z;
```

doet net hetzelfde als

```
        int x;
        int y;
        int z;
```

maar spaart een hoop schrijfwerk.

Meervoudige declaraties van samengestelde types gaat ook, maar ligt iets subtieler:

```
        int *x,y[10],z;
```

doet hetzelfde als

```
        int *x;
        int y[10];
        int z;
```

Zeker bij declaraties als `int *x,y,z;` is het gevaar voor verwarring met `int *x,*y,*z;` groot; in dat geval trek je het best de declaratie uiteen in `int *x;` en `int y,z;`.

Meervoudige declaraties kunnen eventueel ook initializers krijgen:

```
        int x = 20, *y, z[10] = {1,2,3,4,5,6,7,8,9,0}, a;
```

Let op: bij

```
        int x, y, z = 10;
```

wordt alleen `z` geïnitialiseerd!

4.1.6 const en volatile

Een functie als

```
void drukString(char *str)
{
        /* doe iets met de inhoud van *str */
}
```

gebruikt hetgeen waar een pointerargument (hier `str`) naar wijst, maar wijzigt dat niet. In dat geval kan je het `const` sleutelwoord gebruiken:

```
void drukString(const char *str)
{
        /* doe iets met de inhoud van *str, maar wijzig hem niet */
}
```

Dit geeft aan dat `str` een pointer is naar `const char`. Als je zo'n constante toch probeert te wijzigen, bijvoorbeeld door in `drukString()` ergens `*str = 0;` te schrijven, zal de compiler een waarschuwing geven.

`const` kan overal in een typenaam opduiken:

- `char *x` is een niet-constante pointer naar een niet-constante (array van) `char`s
- `const char *x` is een niet-constante pointer naar een constante (array van) `char`s
- `char *const x` is een constante pointer naar een niet-constante (array van) `char`s
- `const char *const x` is een constante pointer naar een constante (array van) `char`s

In de meeste gevallen heeft de compiler je door als je toch een `const` variabele probeert te wijzigen, maar waterdicht is de garantie niet (we zagen al eerder dat je redelijk sterke toeren met pointers kan uithalen die de compiler bijna onmogelijk kan door hebben). Je moet dus zelf ook een beetje discipline hebben.

Op dezelfde manier als `const` kan je ook `volatile` in een type gebruiken (ze kunnen ook samen voorkomen). Het `volatile` sleutelwoord geeft aan dat een variabele ook nog door een ander programma kan gewijzigd worden. De compiler zal geen slimme truuks op die variabele proberen toepassen, zoals het tijdelijk bewaren van de variabele in een register (wanneer de originele variabele zou gewijzigd worden, zou het programma alleen de kopie in het register gebruiken en die wijziging dus niet zien—zie ook §4.4.1). `volatile` wordt vooral gebruikt bij multitasking programma's of om te communiceren met bepaalde hardware.

4.2 Controlestructuren

4.2.1 Voorwaardelijke structuren

In C bestaat naast `if ... then ... else` (zie §2.6.3) ook nog `switch ... case` als voorwaardelijke opdracht. De opbouw is als volgt:

`switch (`*uitdrukking*`) {`
    `case` *const1*: *opdrachten1*
    `case` *const2*: *opdrachten2*
    `case` *const3*: *opdrachten3*

```
    ⋮
    default:   default-opdrachten
}
```

Een `switch`-opdracht wordt als volgt verwerkt. De *uitdrukking* wordt uitgerekend. Het moet een waarde van een geheel type zijn (één van de types uit §4.1.1 dus). De *opdrachten* worden uitgevoerd vanaf het overeenkomstige `case` label. Als *uitdrukking* bijvoorbeeld gelijk blijkt te zijn aan *const2*, dan worden de *opdrachten1* niet uitgevoerd, maar alle andere opdrachten wel (dus vanaf *opdrachten2* tot en met *default-opdrachten*). Is *uitdrukking* geen van de *const* waarden, dan worden de opdrachten uitgevoerd vanaf het `default:` label. Merk op dat een `case` of `default` niets meer is dan een label, een aanduiding van een plaats in de reeks opdrachten. Zo'n label heeft dus geen enkel effect als het programma er langs komt; het programma **valt door** de labels (**fall-through**). Als je de `switch` wilt verlaten, moet je expliciet een `break` invoegen:

```
double bewerk(char bewerking,double arg1,double arg2)
{
        double resultaat;

        switch (bewerking) {
                case '+': resultaat = arg1 + arg2;
                        break;
                case '-': resultaat = arg1 - arg2;
                        break;
                case 'x':
                case '*': resultaat = arg1 * arg2;
                        break;
                default: printf("Ongeldige bewerking %c\n",bewerking);
                        break;
        }
        return resultaat;
}
```

Als ik bijvoorbeeld geen `break` zou geschreven hebben na de `'+'` case, dan zou eerst

```
resultaat = arg1 + arg2;
```

uitgevoerd worden en direkt daarna

```
resultaat = arg1 - arg2;
```

De `break` zorgt ervoor dat het programma de `switch` verlaat en verder gaat met de eerstvolgende opdracht (hier de `return`). Hierdoor is het ook makkelijk om in twee of meer `case`s dezelfde opdrachten te laten uitvoeren, zoals in het voorbeeld de `'x'` en `'*'` cases. Ik had natuurlijk net zo goed

```
return arg1 + arg2;
```

kunnen schrijven zodat een extra `break` niet nodig was, maar ja, dit is tenslotte een voorbeeld ...

In gevallen waarbij de fall-through minder opvallend is, bijvoorbeeld als er na `case 'x':` nog een hoop opdrachten zouden staan, kan het geen kwaad ter verduidelijking een `/* FALL THROUGH */` te schrijven.

Merk ook op dat de volgorde van de `case`s en de `default` niet vast is voorgeschreven. In het bijzonder mag de `default` dus ook ergens in het midden van het `switch` blok komen te staan.

De laatste `break` in het voorbeeld is eigenlijk overbodig, maar het wordt als goede stijl beschouwd om die `break` toch te schrijven. Als je later immers beslist achteraan een extra `case` toe te voegen, ben je vóór je het weet vergeten de vorige `case` met een `break` af te sluiten.

Strikt genomen is de `switch` opdracht natuurlijk overbodig, omdat je net zo goed

```
if (uitdrukking == const1) {
    opdrachten1
} else if (uitdrukking == const2) {
    opdrachten2
    ⋮
} else {
    default-opdrachten
}
```

had kunnen schrijven (of iets soortgelijks bij fall-through—eventueel met behulp van de `||` operator, die we verderop zullen tegenkomen). Een `case` is overzichtelijker en wordt vaak efficiënter gecompileerd; daar tegenover staat dat je niet `case <0:` of zoiets kunt schrijven om `if (uitdrukking<0)` na te bootsen.

4.2.2 Lussen

In §2.4 zijn we de `while` lusopdracht al tegengekomen.

Een vergelijkbare lusstructuur kan je maken met `for`:

```
for (initialisatie ; voorwaarde ; uitdrukking) {
    opdracht
    ⋮
    opdracht
}
```

waarbij *initialisatie, voorwaarde* en *uitdrukking* drie uitdrukkingen zijn. De reeks opdrachten vormt zoals steeds weer een blok. Een `for`-lus wordt als volgt verwerkt:

1. Evalueer de *initialisatie*-uitdrukking (hierin worden vaak tellers geïnitialiseerd.) Eventueel mag je de *initialisatie* leeg laten.

2. Controleer of de *voorwaarde* waar is. Als de voorwaarde niet vervuld is, ga dan verder met de opdracht na het blok; de lus is gedaan. Is de voorwaarde wel vervuld, ga dan naar stap 3.

3. Voer alle opdrachten van het blok één voor één uit.

4. Evalueer de derde *uitdrukking* van de `for` opdracht (hierin worden vaak tellers opgehoogd). De *uitdrukking* mag ook weggelaten worden; in dat geval gebeurt er gewoon niets.

5. Ga terug naar stap 2.

Een voorbeeldje:

```
#include <stdio.h>

int main(void)
{
        int teller;

        for (teller = 1; teller <= 100; teller = teller + 1)
                printf("%d ",teller);
        printf("\n");
}
```

doet net hetzelfde als het voorbeeld uit §2.4. (Merk op dat `teller = 1` wel degelijk een uitdrukking is; `teller = 1;` is een opdracht—zie ook §4.3.5.1.) Algemeen kan je een `for` met een `while` nabootsen:

initialisatie;
`while (`*voorwaarde*`) {`
    opdracht
    ⋮
    opdracht
    uitdrukking;
`}`

doet net hetzelfde als de `for` opdracht. En `for (;` *uitdrukking* `;)` doet hetzelfde als `while(`*uitdrukking*`)`. De drie uitdrukkingen in `for` mogen dus ook weggelaten worden (er moeten natuurlijk wel altijd twee kommapunten staan). Als de *voorwaarde* ontbreekt, wordt ze beschouwd als altijd waar. Met `for(;;)` kan je dus een **oneindige lus** maken. Net zoals bij een `while`-lus kan je een `for` verlaten met `break`. Het programmaatje uit §2.6.3 kan dus als volgt herschreven worden:

```
#include <stdio.h>

int main(void)
{
        for (;;) {
                char c;

                c = getchar();
                switch (c) {
                        case 'g': putchar('h'); break;
                        case 'h': putchar('g'); break;
                        case '&': break;
                        default: putchar(c); break;
                }
                if (c == '&') break;
        }
        return 0;
}
```

Merk op dat `break` alleen één structuur (`case`, `for` of `while`) kan verlaten, zoals we al ondervonden in §2.6.3. De `break`s binnen de `switch` kunnen de `for`-lus dus niet doorbreken. Daarom moet ik expliciet buiten de `switch` testen op `'&'` (de laatste `break` verlaat wel degelijk de `for`-lus). De `break` opdracht heeft immers geen betrekking op andere constructies dan lussen en `switch`, en trekt zich meer in het bijzonder niets aan van de `if` waar het toevallig in staat.

Soms kan je de `for`-lus zo compact schrijven dat al wat er gedaan moet worden al in de `for`-opdracht staat, en de het opdrachtenblok dus leeg blijft:

```
/*** tel van 1 tot 10 ***/
void tellen(void) {
        int tel;

        for (tel = 1; tel <= 10; printf("%d ",tel++)) {
        }
        printf("\n");
}
```

In zulke gevallen wordt vaak gebruik gemaakt van de **lege opdracht**:

```
        for (tel = 1; tel <= 10; printf("%d ",tel++)) ;
```

Zoals altijd mag je immers een blok vervangen door één enkele opdracht. Omdat je die kommapunt nogal vlug over het hoofd kan zien, is het veiliger die op een aparte regel te schrijven:

```
void tellen(void) {
        int tel;

        for (tel = 1; tel <= 10; printf("%d ",tel++))
                ;
        printf("\n");
}
```

zodat je duidelijk kan zien dat de `for`-lus door de lege opdracht gevolgd wordt.

Tenslotte is er nog een derde lusstructuur in C, die echter veel minder gebruikt wordt. Hier wordt de lusvoorwaarde pas op het *einde* van de lus gecontroleerd:

`do {`
    opdracht
    ⋮
    opdracht
`} while (`*voorwaarde*`);`

Een `do ... while` wordt als volgt verwerkt:

1. Voer alle opdrachten (die zoals gewoonlijk weer een blok vormen) één voor één uit.

2. Controleer of de *voorwaarde* waar is. Zo ja, ga terug naar stap 1. Anders is de lus gedaan en gaat het programma verder met de opdracht na `while`.

Met `do ... while` kan het programma uit oefening 2.20 herwerkt worden tot

```
#include <stdio.h>

int main(void)
{
        float totaal;
        float getal;

        totaal = 0;
        do {
                puts("Tik een getal in (0 om te stoppen)");
                scanf("%f",getal);
                totaal = totaal + getal;
        } while (getal != 0.0);
        printf("Totaal: %f\n",totaal);
}
```

Omdat de controle of `getal` al dan niet nul is nu pas op het einde van de lus gebeurt, is het niet meer nodig aan het begin van het programma een opdracht als `getal = 1;` te schrijven.

Tenslotte is er nog `goto`, waarmee je naar een willekeurige plek binnen dezelfde functie kunt springen. De plek naar waar je wilt springen, moet je met een **label** aanduiden. Het label bestaat uit een willekeurige naam, gevolgd door een dubbelpunt. De syntax van `goto` is eenvoudig:

`goto` *label*;

Het voorbeeld van daarnet kunnen we dus zo herschrijven:

```
#include <stdio.h>

int main(void)
{
        for (;;) {
                char c;

                c = getchar();
                switch (c) {
                        case 'g': putchar('h'); break;
                        case 'h': putchar('g'); break;
                        case '&': goto einde;
                        default: putchar(c); break;
                }
        }
        einde:
        return 0;
}
```

Over het algemeen wordt het gebruik van `goto` als **heel slechte programmeerstijl** beschouwd. De enige toepassing die nog min of meer door de beugel kan, is het verlaten van geneste constructies (zoals hier een `switch` binnen een `break`). Je ziet ook dat de `goto` opdracht mag verwijzen naar een label dat pas verderop in het programma komt, één van de zeldzame gevallen in C waar je kan verwijzen naar iets dat nog niet gedefinieerd is.

Een nettere, `goto`-loze versie van het programma van daarnet is makkelijk te maken:

```
#include <stdio.h>

int main(void)
{
        for (;;) {
                char c;

                c = getchar();
                switch (c) {
                        case 'g': putchar('h'); break;
                        case 'h': putchar('g'); break;
                        case '&': return 0;
                        default: putchar(c); break;
                }
        }
}
```

In feite is `goto` enkel in de C taal opgenomen om automatische **programmageneratoren** het leven aangenaam te maken (dat zijn programma's die als uitvoer een C programma hebben). Het gebruik van `goto` is niet alleen afgrijselijke programmeerstijl (die heel vlug tot de beruchte "**spaghetti-code**" leidt); de compiler kan functies die `goto`'s bevatten minder goed optimaliseren omdat de normale loop van het programma doorbroken wordt.

4.2.3 `continue`

In alle drie de lusstructuren (maar niet in `switch`) kan je de `continue` opdracht gebruiken. Die zorgt ervoor dat de rest van het lusblok overgeslagen wordt en het programma een volgende lusdoorloop begint. Bij `for`-lussen wordt wel nog de derde *uitdrukking* geëvalueerd (waarin vaak tellers opgehoogd worden). Bijvoorbeeld:

```
/* zoek het keinste element uit een tabel ints */
int kleinste(int *tabel,int grootte) {
        int resultaat = tabel[0];
        int teller;

        for (teller = 0; teller < grootte; teller = teller + 1) {
                if (tabel[teller] >= resultaat) continue;
                /* tabel[teller] is kleiner dan alle vorige */
                resultaat = tabel[teller];
        }
        return resultaat;
}
```

Dit voorbeeldje is een beetje kunstmatig, omdat ik net zo goed de twee opdrachten binnen de `for`-lus had kunnen vervangen door

```
                if (tabel[teller] < resultaat) resultaat = tabel[teller];
```

De vorm met `continue` is vooral handig als er in plaats van alleen `resultaat = tabel[teller];` een hoop opdrachten zou gestaan hebben; in dat geval is de `if`-binnen-een-`for`

```
for (...) {
        if (...) {
                /* een hele hoop opdrachten */
        }
}
```

een stuk onduidelijker dan de `continue` variant:

```
for (...) {
        if (!...) continue;
        /* een hele hoop opdrachten */
}
```

Ik gebruik hier de `!` operator ("niet" uitgesproken) die waar oplevert als het argument vals is en omgekeerd (zie §4.3.3).

Merk op dat in alle gevallen de teller opgehoogd wordt: ofwel doordat het programma de `continue` bereikt, ofwel doordat het van het eind van het `for`-blok valt. Een `while` versie is daardoor wat lastiger:

```
int resultaat = tabel[0];
int teller = 0;

while (teller < grootte) {
        if (tabel[teller] >= resultaat) {
                teller = teller + 1;
                continue;
        }
        /* tabel[teller] is kleiner dan alle vorige */
        resultaat = tabel[teller];
        teller = teller + 1;
}
return resultaat;
```

Je moet immers op twee plaatsen in de lus `teller = teller + 1;` schrijven.

▸ **OEFENING 4.4**
Bij de uitleg over `for` zei ik dat je `for`-lussen met `while` kon nabootsen en omgekeerd. Er is toch een klein verschil tussen beide vormen. Welk?

♢

▸ **OEFENING 4.5**
Herschrijf het programma uit oefening 2.20 met een oneindige lus, `continue` en `break`.

♢

4.2.3.1 Duff's Device

Whenever possible, steal code.
—TOM DUFF

Deze hele paragraaf is "gevaarlijke bocht" omdat ik hier de operatoren `++` en `--` en gebruik, die pas verderop in dit hoofdstuk uitgelegd worden, en ook omdat je niet zo vaak van Duff's Device gebruik zal maken (zeker niet in het begin).

Duff's Device is genoemd naar de uitvinder ervan, Tom Duff (die toen bij Lucasfilm werkte). Het idee erachter is dat een lus als

```
/* kopieer een aantal bytes van een bepaalde plek naar ergens anders */
void kopie(char *naar, char *van, int aantal) {
        while (aantal-- > 0)
                *naar++ = *van++;
}
```

vrij onefficiënt is, omdat er relatief veel tijd verloren wordt met het bijhouden van `aantal` en de controles of `aantal` al nul heeft bereikt. Veel efficiënter is bijvoorbeeld

```
void kopie(char *naar, char *van, int aantal) {
        /* Kopieer in groepjes van 3 bytes */
        while (aantal > 3) {
                aantal = aantal - 3;
                *naar++ = *van++;
                *naar++ = *van++;
                *naar++ = *van++;
        }
        /* Nu moeten er hoogstens nog twee bytes gekopieerd worden */
        if (aantal == 2) {
                *naar++ = *van++;
                *naar++ = *van++;
        } else if (aantal == 1) {
                *naar++ = *van++;
        }
}
```

Deze methode heet **loop unrolling**; het programma wordt groter, maar draait sneller.

Op 9 november 1983 vond Tom Duff een wel heel compacte manier om dit te schrijven (dit programma kopieert in groepjes van acht bytes in plaats van drie):

```
void kopie(char *naar, char *van, int aantal) {
        int n;

        n = (aantal + 7) / 8;
        /* n = aantal doorlopen (de eerste onvolledige doorloop
         *       wordt ook als een doorloop geteld)
         * aantal%8 = aantal te kopieren bytes tijdens
         *       de eerste doorloop
         */
        switch (aantal%8) {
        case 0: do {     *naar++ = *van++;
        case 7:          *naar++ = *van++;
        case 6:          *naar++ = *van++;
        case 5:          *naar++ = *van++;
        case 4:          *naar++ = *van++;
        case 3:          *naar++ = *van++;
        case 2:          *naar++ = *van++;
        case 1:          *naar++ = *van++;
            } while (--n > 0);
```

```
        }
}
```

Opmerkelijk is de door elkaar gestrengelde `switch` en `while`. (Na grondige inspectie van de ANSI C standaard bleek dit programma volledig correct C(!))

Hieruit blijkt ook duidelijk dat een `switch` voor de compiler in feite niets anders is dan een soort `goto` met de `case`s als labels, dus iets als

```
if (aantal%8 == 0) goto case_0;
else if (aantal%8 == 1) goto case_1;
/* enzovoort */
else if (aantal%8 == 7) goto case_7;
else goto doe_niks;

case_0: do {    *naar++ = *van++;
case_7:         *naar++ = *van++;
/* enzovoort */
case_1:         *naar++ = *van++;
} while (--n > 0);
doe_niks:
```

Moeten we dit nu lelijk en gekunsteld of elegant en beknopt vinden? Tom Duff zelf moet ook ongeveer zo'n gevoel gehad hebben—hij omschreef het als "a mixture of pride and revulsion" ...

4.3 Operatoren en uitdrukkingen

Af en toe heb ik al eens het woord **uitdrukking** in de mond genomen, zonder er al te veel over te zeggen. Een algemene regel in C is dat overal waar een waarde van een bepaald type staat (zeg `int`), je ook een uitdrukking van dat type mag schrijven (bijvoorbeeld `5+3/2`). Met "uitdrukking" bedoelen we dus een samenraapsel van constanten (bv. `3` of `"Hello"`), variabelen (bv. `teller`) en operatoren (`+`, `->`, ...).

Het spreekt vanzelf dat niet alle operatoren met alle datatypes willen samenwerken; zo zal je moeilijk de `+` operator kunnen gebruiken op twee `struct persoon`s. Elke operator geeft ook weer een resultaat van een bepaald type (het resultaat van `int + double` is bijvoorbeeld van het type `double`).

Wanneer het programma tegen een uitdrukking aanbotst, wordt die uitgerekend (`5+3/2` wordt `5+1` wordt `6`), hetgeen ook wel eens het **evalueren** van een uitdrukking genoemd wordt.

Een operator gebruikt een aantal **operand**en, d.i. de "dingen" die je een operator te eten geeft. Bijvoorbeeld: in de uitdrukking `5+3` zijn `5` en `3` de twee operanden, en het resultaat van de `+` operator zal `8` zijn. Afhankelijk van het aantal operanden spreken we van unaire (één operand), binaire (twee operanden, bv. `+`) of ternaire operatoren (drie operanden).

De **evaluatievolgorde** is in C over het algemeen niet vast bepaald. Dat betekent dat je niet weet of de compiler eerst het linker- of het rechterargument van een operator uitrekent. De enige uitzondering hierop zijn de operatoren `&&` en `||`, die we zo dadelijk zullen ontmoeten.

Een voorbeeldje:

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

```
int een(void) {
        printf("1 ");
        return 1;
}

int twee(void) {
        printf("2 ");
        return 2;
}

int main(void) {
        printf("%d\n",een() + twee());
        return 0;
}
```

geeft als resultaat ofwel `1 2 3`, ofwel `2 1 3`, afhankelijk van de gebruikte compiler. De compiler mag immers kiezen of hij het linkerargument van `+` eerst uitrekent (`een()`) of het rechterargument (`twee()`).

Zelfs als je haakjes gebruikt, kan de compiler nog de volgorde van evaluatie veranderen. De compiler kan een uitdrukking als `x*(6/5)` dus uitrekenen door eerst de waarde van `x` met 6 te vermenigvuldigen (waardoor het tussenresultaat misschien te groot kan worden voor dat datatype), en dat resultaat dan door vijf te delen, of eerst de waarde van `x` door 5 te delen (waardoor er misschien precisie zou kunnen verloren gaan als het type van `x` een geheel datatype is) en daarna met 6 te vermenigvuldigen.

Als de evaluatievolgorde van belang is, moet je de uitdrukking expliciet uiteenrafelen, door bijvoorbeeld

```
        y = x * 6;
        y = y / 5;
```

of

```
        y = x / 5;
        y = y * 6;
```

te schrijven.

4.3.1 Rekenkundige (arithmetische) operatoren

De vier **hoofdbewerkingen** worden geleverd door de al gekende operatoren `+`, `-`, `*` en `/`. Het type van het resultaat van deze operatoren hangt af van de types van de operanden. Het resultaat is van het “nauwkeurigste” type van beide operanden, waarbij de types van nauwkeurigst tot minst nauwkeurigst als volgt geordend zijn: `double`, `float`, `long`, `int`, `short`, `char`. Bijvoorbeeld: als je een `float` en een `int` bij elkaar optelt, is het resultaat `float` (want `float` is nauwkeuriger dan `long`). De verklaring hiervoor is dat elke `long` in een `float` past, maar niet omgekeerd (er gaat dus zo weinig mogelijk precisie verloren).

Als je `signed` en `unsigned` types met elkaar mengt, ligt de zaak iets subtieler. Het resultaat is dan `signed` of `unsigned` al naargelang het nauwkeurigste type dat is, tenzij het resultaat te groot zou kunnen zijn. In dat geval is het resultaat altijd `unsigned`.

In de tijd vóór ANSI C was de regel eenvoudigweg: als één van beide uitdrukkingen `unsigned` is, is het resultaat dat ook.

Als het resultaat te groot of te klein is, gaat het programma gewoon verder met een verkeerd resultaat. Je krijgt dus geen foutmelding of zo in het volgende stukje code (ik veronderstel dat een `unsigned short` getallen van 0 t/m 65535 kan bevatten):

```
{
        unsigned short a;
        unsigned short b;

        a = 30000;
        b = 40000;

        a = a + b;
}
```

Het resultaat is niet 70000 (dat past immers niet in een `unsigned short`) maar een of andere onbepaalde waarde. Dit soort fout heet **overflow**. Ook als je `20000` wilt aftrekken van de `signed short -30000`, krijg je overflow (-50000 past immers niet in een `signed short`). Bij de types `float` en `double` kan ook **underflow** optreden. Stel dat het kleinste nog van nul verschillende getal dat in een `float` past 10^{-38} (in C genoteerd als `1E-38`) is. Als je dan `1E-20 / 1E19` probeert te berekenen, krijg je niet `1E-39` maar `0`.

Als de `/` operator op twee gehele datatypes werkt, is het resultaat ook weer (zoals hierboven uitgelegd) een geheel type. Eventueel wordt het gedeelte na de komma afgekapt (dus niet afgerond). `5/3` geeft dus `1` (en niet `2`).

Als laatste rekenkundige operator is er nog `%`. Deze operator werkt alleen met gehele datatypes. Net als `/` worden de twee operanden door elkaar gedeeld, maar het resulaat van `%` is de rest van de deling. Dus `23%3` geeft 2 (omdat 23 een drievoud +2 is). Op deze manier kun je testen of een getal `a` eindigt op 99:

```
if (a%100 == 99) printf("%d eindigt op 99\n",a);
```

of kijken of een getal even is:

```
/* geeft 1 terug als x even is, anders 0 */
int iseven(int x)
{
        if (x%2 == 0) return 1; /* x is een tweevoud */
        else return 0;
}
```

▸ **OEFENING 4.6**

Het spel "ding-bottel" gaat als volgt. De eerste speler zegt `1`, de tweede `2`, de eerste `3`, de tweede `4`, en zo verder. Als er in het getal echter een 5 voorkomt, mag je dat getal zelf niet zeggen, maar moet je "ding" zeggen. Hetzelfde geldt als het getal een vijfvoud is. Zo wordt 25 "ding-ding", aangezien er een vijf in voorkomt en het een vijfvoud is. Analoog moet bij getallen waar een zeven in voorkomt en bij zevenvouden "bottel" gezegd worden. Dus in plaats van 35 moet je "ding-ding-bottel" zeggen (vijfvoud, er komt een vijf in voor, en zevenvoud). Wie zich vergist is natuurlijk verloren.

Schrijf een programma waarbij een speler dit spel tegen de computer kan spelen. Om het gemakkelijk te maken moet de speler alleen maar een `d` of `b` tikken in plaats van voluit `ding` of `bottel`.

◇

Dit zijn de enige rekenkundige bewerkingen die C "van huis uit" ondersteunt. Voor alle andere bewerkingen met getallen moet je in principe zelf functies maken. Gelukkig zitten er een groot aantal al in de standaardbibliotheek `math.h` (zie §7.6), zoals bijvoorbeeld `pow()` voor de machtsverheffing.

4.3.2 Vergelijkingsoperatoren

De operatoren `==`, `!=`, `>`, `>=`, `<` en `<=` (die ik al getoond heb in §2.4) kan je gebruiken om getallen of pointers met elkaar te vergelijken (pointers met getallen vergelijken is om problemen vragen, tenzij dat getal natuurlijk `0` is ...)

Het resultaat van een vergelijkingsoperator is verrassend genoeg een `int`: `1` als de vergelijking waar is, of `0` als de vergelijking vals is. Dat werkt omdat voorwaarden in opdrachten als `if` en `while` als *waar* beschouwd worden wanneer de uitdrukking verschillend is van nul, en als *vals* wanneer de uitdrukking nul is. Dus de

```
while (1 == 1)
```

opdracht die we al eens gebruikt hebben om een oneindige lus te maken, hadden we korter als

```
while (1)
```

kunnen schrijven. (Pascal fans zullen misschien `#define TRUE 1` en `#define FALSE 0` of zo schrijven.) En `if (pointer != NULL)` kan korter als `if (pointer)` geschreven worden.

Je kan deze eigenschap van vergelijkingsoperatoren ook gebruiken door een variabele te maken die "waar" of "vals" bevat, zoals de variabele `negatief` in dit programma (dat een aanpassing van `maakString()` uit §3.6 is, die ook voor negatieve getallen werkt):

```
char *maakString(long getal)
{
        char resultaat[20];
        int lengte;
        int negatief;

        if (getal < 0) {
                getal = -getal;
                negatief = 1;
        } else negatief = 0;

        lengte = 0;

        /* Eerst de string achterstevoren maken
         * (getal%10 geeft het laatste cijfer)
         */
        do {
                resultaat[lengte] = getal%10 + '0';
                getal = getal / 10;
```

```
                lengte = lengte + 1;
        } while (getal != 0);

        /* De string netjes afsluiten met een nulbyte */
        resultaat[lengte] = 0;
        /* nu bevat resultaat het getal als string, maar zonder teken
         * en achterstevoren, dus bv. "1234" als getal 4321 of -4321 was */

        {       /* Resultaat omkeren;
                 * als negatief==1 dan ook nog vooraan
                 * plaats voor een - invoegen */
                int tel, telaf;

                tel = 0;
                if (!negatief) telaf = lengte-1;
                else telaf = lengte;

                while (tel < telaf) {
                        /* verwissel resultaat[tel] en resultaat[telaf] */
                        char wissel;

                        wissel = resultaat[tel];
                        resultaat[tel] = resultaat[telaf];
                        resultaat[telaf] = wissel;

                        tel = tel + 1;
                        telaf = telaf - 1;
                }
        }

        if (negatief) {
                /* door het omkeren is de nulbyte
                 * in resultaat[0] komen te staan. */
                resultaat[0] = '-';
                resultaat[lengte+1] = 0;     /* nieuwe nulbyte */
        }
        return resultaat;
}
```

Merk terloops ook op dat we het bijzondere geval kwijt zijn waarbij `getal` nul is, door het gebruik van een `do ... while`-lus in plaats van de `while` uit de originele versie.

En de constructie

```
        if (getal < 0) {
                getal = -getal;
                negatief = 1;
        } else negatief = 0;
```

had ik korter (maar iets onduidelijker) als

```
negatief = getal<0;
if (negatief) getal = -getal;
```

kunnen schrijven, want zoals eerder uitgelegd wordt `getal < 0` als `0` of `1` geëvalueerd.

4.3.3 Logische operatoren

Als je wil testen of twee voorwaarden tegelijkertijd waar zijn, bijvoorbeeld `teller>=0` en `teller<=20` (om te zien of `teller` tussen de 0 en de 20 ligt), kan je de `&&` operator gebruiken (uitgesproken "en"):

```
if (teller>=0 && teller<=20) { ... }
```

Merk op dat je wel `if (0<=teller<=20)` zou kunnen schrijven, maar dat de C compiler dat interpreteert als `if ( (0<=teller) <= 20)`. Deze voorwaarde is altijd waar, want `(0<=teller)` geeft 0 of 1, al naargelang `teller` groter is dan nul of niet, en dat is altijd kleiner dan 20. (De GNU C compiler heeft hier niets in de gaten en geeft zelfs geen waarschuwing—opletten geblazen dus.)

Het resultaat van een logische operator is natuurlijk ook weer `1` (als beide voorwaarden waar waren, m.a.w. beide operanden waren verschillend van nul) of `0` (in het andere geval), net als bij de vergelijkingsoperatoren.

`l`	`r`	`l && r`	`l || r`
vals	vals	`0`	`0`
vals	waar	`0`	`1`
waar	vals	`0`	`1`
waar	waar	`1`	`1`

Tabel 4.1: Waarheidstabellen van de `&&` en `||` operatoren
waarbij "vals" `0` betekent, en "waar" `!= 0`

Het interessante van deze operator is dat *eerst* de linkervoorwaarde wordt getest (hier dus `teller>=0`), en *alleen als die waar was*, wordt de rechtervoorwaarde uitgerekend (`teller<=20` in het voorbeeld). Als de linkervoorwaarde immers vals blijkt, is het resultaat van `&&` toch altijd vals. Dat is van belang als de voorwaarden bijvoorbeeld functieresultaten zijn:

```
if (binnenBereik() && statusOk()) { ... }
```

Als `binnenBereik()` een nul zou teruggeven, wordt `statusOk()` dus niet eens uitgevoerd.

Verder is er nog de verwante `||` ("of")-operator, die `1` oplevert als één van beide operanden waar is of als ze allebei waar zijn, en `0` als geen van de twee waar is. Ook bij deze operator wordt de rechtse operand alleen geëvalueerd als dat nodig is, m.a.w. alleen als de linkse operand "vals" oplevert (was het resultaat van de linkse operand "waar', dan zou het resultaat van `||` toch "waar" zijn, onafhankelijk de waarde van de rechtse operand). Het `|`-teken wordt overigens op veel toetsenborden als ¦ afgebeeld.

Ook hier moet je opletten niet iets als `if (teller==3 || 5)` te schrijven, maar

```
if (teller==3 || teller==5)
```

a	!a
waar	0
vals	1

Tabel 4.2: Waarheidstabel van de `!` operator

Een laatste logische operator is `!` ("niet"). Dit is een **unaire operator**, d.i. er is maar één operand. `!uitdrukking` heeft als resultaat vals (meer bepaald `0`) als de `uitdrukking` waar (dus verschillend van nul) was, en omgekeerd.

OEFENING 4.7
Wat is het verschil tussen `!!x` en gewoon `x` (waarbij `x` een `int` variabele is)?

◇

Met de `!` operator kan je vlot leesbare uitdrukkingen schrijven als

```
int omgewisseld;
char *geheugen;

if (!omgewisseld) return;
geheugen = malloc(1000);
if (!geheugen) return;
```

In deze gevallen is de conventie "nul betekent vals" verhelderend: de `if` opdrachten lees je vanzelf als "als er niet omgewisseld werd" en "als er geen geheugen was".

4.3.4 Voorbeeld: recursief een lijn tekenen

4.3.4.1 Recursie

Een functie kan zichzelf aanroepen in C. Natuurlijk geeft

```
void recursief(void) {
        puts("Ik roep mezelf nu aan!");
        recursief();
        puts("Ik heb mezelf nu aangeroepen!");
}
```

als resultaat een eindeloze lus die voortdurend `Ik roep mezelf nu aan!` afdrukt. De laatste `puts()` opdracht van de functie wordt nooit uitgevoerd, simpelweg omdat de aanroep `recursief()` nooit beëindigd wordt.

Bovendien zal het geheugen langzaam beginnen vollopen. Het ziet er op het eerste gezicht niet naar uit dat we ergens geheugen voor nodig hebben. Bij elke functieaanroep moet de computer echter bijhouden naar waar hij moet terugspringen (het **terugkeeradres**) als de functie gedaan is. Wat er dus gebeurt is ongeveer het volgende:

- Druk `Ik roep mezelf nu aan!` af.
- Onthoud de plaats waar het programma nu is en roep de functie `recursief()` aan.
- De tweede aanroep van `recursief()` drukt weer de tekst af.
- Onthoud de plaats waar het programma nu is en roep de functie `recursief()` aan.

- ...

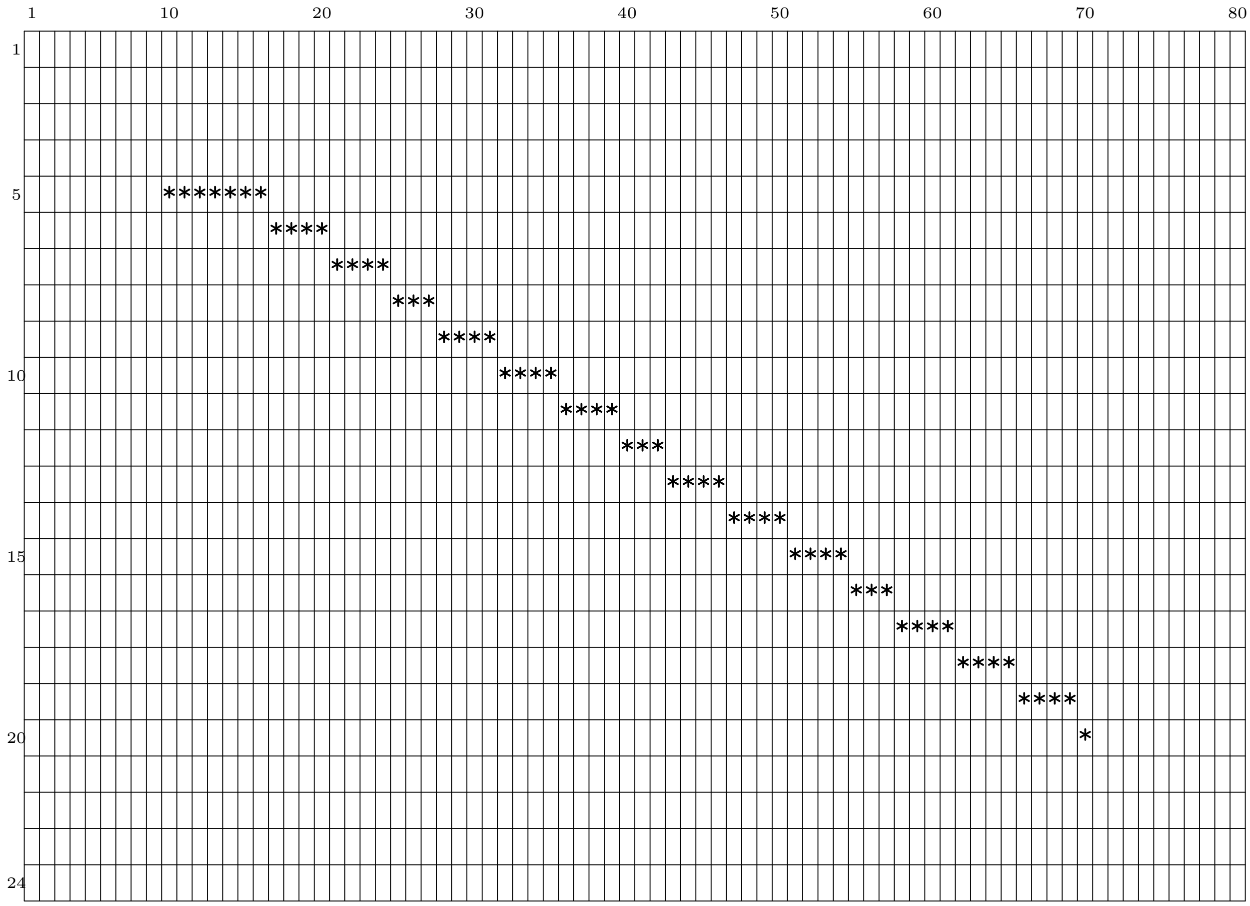
Dit is een algemeen nadeel van recursieve programma's: het onthouden van de terugkeeradressen kan soms veel geheugenruimte vragen. Het is dus dikwijls interessant een niet-recursieve variant van een algoritme te vinden. In dit vergezochte voorbeeld ligt deze variant voor de hand:

```
void nietrecursief(void) {
        while (1 == 1) {
                puts("Ik roep mezelf aan!");    /* Dit programma liegt! */
        }
}
```

Recursie is nogal populair in de computerwereld; zo betekent GNU offcieel "GNU's not Unix", een recursieve afkorting dus, en GNU is zeker niet de enige ...

4.3.4.2 Lijnen tekenen

Een interessanter voorbeeld is het trekken van een lijn op het scherm. Ik wil dus een functie `lijn()` maken die tussen twee punten (x_1, y_1) en (x_2, y_2) een lijn van sterretjes trekt, bijvoorbeeld tussen $(10, 5)$ en $(70, 20)$. De vraag is natuurlijk hoe we kunnen uitrekenen op welke plaatsen we een sterretje moeten zetten.



De "lijn" die we willen op het scherm krijgen

Een eenvoudige methode gaat als volgt: verdeel de lijn in twee helften. Het middelpunt van de lijn heeft als x- en y-coördinaten het gemiddelde van de x- en y-coördinaten van de eindpunten, dus in ons geval $\frac{10+70}{2}, \frac{5+20}{2}$ of $(40, 12.5)$. Omdat de terminal niet met halve posities werkt, kappen we af tot $(40, 12)$. Nu moeten we enkel nog een lijn tussen $(10, 5)$—$(40, 12)$ en een lijn tussen $(40, 12)$—$(70, 20)$ zien te trekken.

Het ziet er niet naar uit dat het probleem zo ooit opgelost zal raken: we moeten alsmaar meer alsmaar kortere lijntjes trekken. Maar vanaf een zeker ogenblik zal het lijntje zo kort geworden zijn, dat het maar één sterretje op het scherm zal innemen. Het heeft geen zin om zulke lijntjes nog te blijven in twee hakken. Op die manier eindigt de recursie steeds.

Als voorbeeld ga ik na wat er na het in twee knippen telkens met het eerste lijntje gebeurt. De eerste helft van $(10, 5)$—$(40, 12)$ wordt $(10, 5)$—$(25, 8)$; de eerste helft daarvan $(10, 5)$—$(17, 6)$; daarna $(10, 5)$—$(13, 5)$, $(10, 5)$—$(11, 5)$ en tenslotte $(10, 5)$—$(10, 5)$. Als we dus een lijntje moeten trekken met hetzelfde begin- en eindpunt, dan hebben we geen verdere recursie meer nodig, maar volstaat het een sterretje te tekenen. We hebben dus als algoritme om een lijn tussen (x_1, y_1) en (x_2, y_2) te trekken:

- als begin- en eindpunt gelijk zijn, zet dan een sterretje op die positie.
- als begin- en eindpunt niet gelijk zijn, trek dan een lijn tussen het eerste punt en het middelpunt, en vervolgens een lijn tussen het middelpunt en het tweede punt. Hier zit de recursie, want om uit te leggen hoe je een lijn moet trekken gebruik ik al “trek een lijn”.

Als je dit algoritme zou programmeren, dan zou je merken dat er nog één klein probleempje onder de oppervlakte schuilt. Wat gebeurt er namelijk als het programma een lijn tussen $(10, 5)$—$(11, 5)$ wil trekken? Het middelpunt van de lijn is (10.5,5), wat na afkappen $(10, 5)$ geeft. Dus gaat het programma een lijn van het eerste punt naar het middelpunt trekken, dus $(10, 5)$—$(10, 5)$ (hier is er nog niets aan de hand), en een lijn van het middelpunt naar het eindpunt, dus $(10, 5)$—$(11, 5)$. Deze twee punten zijn niet gelijk; we zitten dus in het tweede geval en ... de hele draaimolen begint van voor af aan.

We moeten dus nog een extra geval inbouwen:

- Als de twee punten vlak naast elkaar liggen, plaats dan twee sterretjes.

Met “vlak naast elkaar” bedoel ik hier natuurlijk dat de x- en y- coördinaten hoogstens 1 van elkaar mogen verschillen:

```
***
*x*
***
```

Al de sterretjes liggen “vlak naast” de `x`

In C ziet dit alles er zo uit:

lijn.c

```
#include <stdio.h>

#define ster(x,y) printf("\033[%d;%dH*",y,x);

/* hulpfunctie: geeft 1 als (x1,y1) en (x2,y2) naast elkaar liggen */
int naastelkaar(int x1,int y1,int x2,int y2)
```

```
{
        if (x1<x2-1 || x1>x2+1) return 0;
        if (y1<y2-1 || y1>y2+1) return 0;
        return 1;
}

void lijn(int x1,int y1,int x2,int y2)
{
        if (x1==y1 && x2==y2) {
                ster(x1,y1);
                return;
        } else if (naastelkaar(x1,y1,x2,y2)) {
                ster(x1,y1);
                ster(x2,y2);
                return;
        } else {
                int middenx;
                int middeny;

                middenx = (x1+x2)/2;
                middeny = (y1+y2)/2;
                lijn(x1,y1,middenx,middeny);
                lijn(middenx,middeny,x2,y2);
        }
}

int main(void)
{
        lijn(10,5,70,20);
        return 0;
}
```

4.3.5 Toekenningsoperatoren

4.3.5.1 De = toekenningsoperator

De enige toekenningsoperator die we tot nog toe ontmoet hebben is `=`. Ook deze operator geeft een resultaat terug, namelijk hetgeen rechts van `=` staat. Dus als de uitdrukking `teller = 10` wordt geëvalueerd, is het resultaat ervan 10.

Met andere woorden:

```
teller = 10;
```

schrijven is net zoiets als

```
10;
```

schrijven. En beide zijn correct C! In C is elke willekeurige uitdrukking gevolgd door een kommapunt een geldige opdracht.

Over het algemeen zijn opdrachten als `10;` natuurlijk niet bijster interessant. En we zijn ook niet echt geïnteresseerd in het feit dat het resultaat van `teller = 10` 10 is: C gooit dat resultaat gewoon weg. Dus ook als we een functie hebben die gedeclareerd is als

```
struct Persoon *leesPersoon(void)
{
        ...
}
```

mag je de opdracht `leesPersoon()` geven. Het resultaat ervan wordt gewoon weggegooid (of dat nu je bedoeling was of niet . . .)

Samengevat hebben we dus twee interessante eigenschappen van C:

- elke uitdrukking gevolgd door een kommapunt is een geldige opdracht
- het resultaat van een uitdrukking wordt weggegooid.

Voorts hebben we nog ondervonden dat het evalueren van een uitdrukking tot gevolg kan hebben dat de inhoud van variabelen verandert. Dat noemt men een **neveneffect** (**side effect**) van een uitdrukking. (In feite is het uitvoeren van `leesPersoon()` ook een neveneffect van het evalueren van de uitdrukking `leesPersoon()`.)

Zoals al uitgelegd in §2.4 is het van cruciaal belang de operatoren `=` en `==` niet te verwarren. De C compiler zal

```
void istien(int x)
{
        if (x = 10) printf("tien");
}
```

netjes compileren. Wat er echter gebeurt is dat bij het evalueren van de uitdrukking `x = 10` de variabele `x` altijd de inhoud 10 krijgt. Het resultaat van de uitdrukking is 10, wat niet gelijk is aan nul, en dus wordt altijd `printf("tien")` uitgevoerd.

In sommige gevallen is een toekenningsoperator in een `if` of `while` wel degelijk nuttig:

```
{
        char *geheugen;

        if (geheugen = malloc(1000)) {
                /* geheugen is niet 0 (de nul pointer), dus
                 * malloc() is gelukt */

                /* ...              (doe vanalles met geheugen) */

                free(geheugen);
        } else {
                puts("Te weinig geheugen!");
        }
}
```

De GNU C compiler kan waarschuwingen geven als er een `=` op een verdachte plaats staat als je de `-Wparentheses` optie gebruikt. Je krijgt dan de waarschuwing `suggest parentheses around assignment used as truth value`. Om aan te geven dat een `=` wel degelijk is wat je wil, kan je er een extra paar haakjes rond plaatsen:

```
        if( (geheugen = malloc(1000)) )
```

In dat geval krijg je geen waarschuwing meer.

Een ander voorbeeld is

```
if (negatief = getal<0) getal = -getal;
```

wat hetzelfde doet als

```
negatief = getal<0;
if (negatief) getal = -getal;
```

Het is soms verleidelijk om je te laten gaan en erg compacte, maar ook totaal onleesbare code te beginnen schrijven…

4.3.5.2 Overige toekenningsoperatoren

In tegenstelling tot vele andere progammeertalen kent C nog een aantal andere toekenningsoperatoren:

```
*= /= %= += -= <<= >>= &= ^= |=
```

De uitdrukking `x *= y` doet net hetzelfde als `x = x * y`, behalve dan dat `x` maar één keer wordt geëvalueerd (wat van belang kan zijn als `x` een uitdrukking is die neveneffecten oplevert). De andere toekenningsoperatoren werken op dezelfde manier.

▸ **OEFENING 4.8**
Bedenk een voorbeeld waar het wel degelijk van belang is dat de linker operand maar één keer wordt geëvalueerd.

◇

Een paar voorbeeldjes: er is geen enkele rechtgeaarde C programmeur die `x = x + 10;` over zijn lippen (toetsen) krijgt, want `x += 10;` is natuurlijk veel eleganter en korter. En

```
tabel[aantal - index*2 + 1] /= 3;
```

is veel duidelijker dan

```
tabel[aantal - index*2 + 1] = tabel[aantal - index*2 + 1] / 3;
```

Bovendien zal de compiler meestal efficiëntere code genereren.

4.3.5.3 Resultaat van de toekenningsoperatoren

Het resultaat van een toekenningsoperator is de nieuwe waarde van de variabele links van `=`. Zo is het resultaat van `tabel[i] += j` niets anders dan de waarde `tabel[i]` na de toekenningsopdracht (dus de som van `tabel[i]` vóór de toekenning en `j`).

Dat resultaat kan je verder gebruiken, bijvoorbeeld:

```
x = ( tabel[i]+=j );
```

Deze opdracht heeft hetzelfde effect als

```
tabel[i] = tabel[i] + j;
x = tabel[i];
```

Even controleren of je het doorhebt:

▸ **OEFENING 4.9**
Schrijf uit wat de opdracht `x *= ( tabel[i+=x] += j );` doet.

◇

4.3.6 Functies, pointers naar functies

De `()` in een functieaanroep is ook een operator. Tussen de haakjes komen natuurlijk argumenten, gescheiden door komma's, maar hetgeen ervoor komt moet een *pointer naar een functie* of een *functie* zijn. Je kan geen variabelen maken van het type "functie", maar er zijn wel constanten van dat type, namelijk al de namen van de functies die je gemaakt hebt.

Uitdrukkingen van het type "functie" worden automatisch gecast naar "pointer naar functie" als het nodig is:

```
int mijnfunctie(long x)
{
        /* ... */
}

int main(void)
{
        int (* fwijzer)(long);

        fwijzer = mijnfunctie;
        return (*fwijzer) (10);
}
```

De variabele `fwijzer` is van het type "pointer naar een functie die een `int` teruggeeft met als parameter een `long`" en `mijnfunctie` heeft als type "functie die een `int` teruggeeft met als parameter een `long`", dus kan de automatische cast plaatsvinden.

Je mag op pointers naar functies geen pointer arithmetic toepassen (vergelijken met de `NULL` pointer mag natuurlijk wel).

Aangezien de `()` operator naar believen op pointers naar functies of op functies werkt, kan de `return` ingekort worden tot

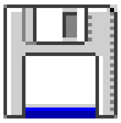
```
        return fwijzer(10);
```

Wat er precies zit in het geheugen waar een pointer naar een functie naar wijst is niet belangrijk (het kan trouwens van systeem tot systeem verschillen) maar over het algemeen wijst die pointer naar de eerste byte van de machinetaalroutine die de compiler van die functie gemaakt heeft.

Meer over declaraties van pointer-naar-functie variabelen staat in §4.1.3.4.

4.3.6.1 Toepassing: functietabel

Soms kan het handig zijn een hoop pointers naar functies in een array te stoppen. In het volgende voorbeeld maak ik zo'n array, `functietabel` genoemd. Om alles wat overzichtelijk te houden, maak ik een `typedef` voor "pointer naar een functie die twee `int`s als parameters heeft en een `int` teruggeeft". Als ik dan een pointer naar functie nummer `bewerking` uit de lijst wil hebben, volstaat het `functietabel[bewerking]` te schrijven. Door erachter `(getal1,getal2)` te schrijven maak ik er een functieaanroep van, want een pointer naar een functie gevolgd door een argumentenlijst tussen haakjes wordt altijd als functieaanroep beschouwd. De extra haakjes rond `functietabel[bewerking]` zijn eigenlijk overbodig en dienen enkel om alles wat duidelijker te maken.



functab.c

```
#include <stdio.h>
typedef int (*verwerker)(int,int);

int telop(int a,int b) { return a+b; }
int trekaf(int a,int b) { return a-b; }
int vermenigv(int a,int b) { return a*b; }
int deel(int a,int b) { return a/b; }

int main(void) {
        int bewerking,getal1,getal2,resultaat;
        verwerker functietabel[] = {
                telop,trekaf,vermenigv,deel
        };

        puts("0 optellen");
        puts("1 aftrekken");
        puts("2 vermenigvuldigen");
        puts("3 delen");
        scanf("%d",&bewerking);
        puts("Eerste getal:");
        scanf("%d",&getal1);
        puts("Tweede getal:");
        scanf("%d",&getal2);

        resultaat = (functietabel[bewerking])(getal1,getal2);
        printf("Resultaat: %d\n",resultaat);
        fflush(stdout);
        return 0;
}
```

Het programma loopt natuurlijk in het honderd als je bewerking `7` kiest; het is dan ook enkel bedoeld om het principe van functietabellen duidelijk te maken.

Ik had hier natuurlijk net zo goed een `switch(bewerking)` opdracht kunnen schrijven in plaats van het me moeilijk te maken met zo'n functietabel. Het voordeel van een functietabel is dat tijdens de loop van het programma de inhoud ervan kan gewijzigd worden, wat niet mogelijk is met een `switch`.

4.3.7 Bitsgewijze operatoren

Bitsgewijze operatoren werken op gehele getallen. Elk getal heeft een binaire voorstelling, en deze operatoren werken zich bit per bit door de getallen die je ze te eten geeft.

4.3.7.1 Talstelsels

Computers werken intern met getallen in **binaire** of **tweetallige** voorstelling, in tegenstelling tot het onder mensen meer gangbare **tientallige** talstelsel. Om te zien hoe dat juist werkt, kijken we eerst even hoe het tientallige stelsel ineen zit.

Wat is de waarde van een getal als 1374? Wat een domme vraag: het laatste cijfer heeft **gewicht** 1, het voorlaatste gewicht 10, en zo voort. We moeten gewoon elk cijfer met zijn gewicht vermenigvuldigen en alles optellen: $1 \times 1000 + 3 \times 100 + 7 \times 10 + 4 \times 1$.

Als ik nu eenvoudigweg 10 in de vorige uitleg door 2 vervang, krijgen we het recept voor binaire getallen. Het laatste cijfer van een binair getal heeft dus gewicht 1, het voorlaatste gewicht 2, dat daarvoor 4, ... Als je dus wil weten hoe groot 100010 in binaire notatie is (ook wel 100010_2 genoteerd om het te onderscheiden van de gewone tientallige notatie 100010_{10}), dan moet je weer elk getal met zijn gewicht vermenigvuldigen en alles optellen: $1 \times 32 + 0 \times 16 + 0 \times 8 + 0 \times 4 + 1 \times 2 + 0 \times 1$ of dus 34 (34_{10} om preciezer te zijn). In het binaire talstelsel hebben we alleen de 0 en de 1 als cijfers (vandaar de naam tweetallig).

1	3	7	4
1000 tallen	100 tallen	tien- tallen	een- heden
waarde 1000	waarde 300	waarde 70	waarde 4

Totaal: $1000 + 300 + 70 + 4 = 1374$

1	0	0	0	1	0
32- tallen	16- tallen	acht- tallen	vier- tallen	twee- tallen	een- heden
waarde 32	waarde 0	waarde 0	waarde 0	waarde 2	waarde 0

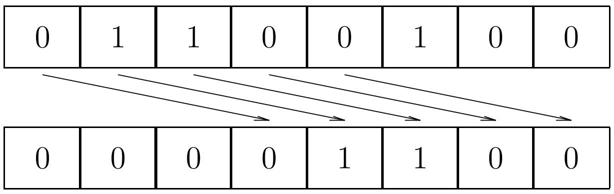
Totaal: $32 + 0 + 0 + 0 + 2 + 0 = 34$

Omdat binaire getallen nogal omslachtig zijn om noteren, wordt vaak in het achttallige (**octale**) of zestientallige (**hexadecimale**) stelsel gewerkt. In het hexadecimale stelsel hebben we zestien cijfers nodig; naast `0` ... `9` worden nog de "cijfers" `A` (waarde 10), `B` (11), `C` (12), `D` (13), `E` (14) en `F` (15) gebruikt. Octale gehele getallen kan je in C schrijven door er een `0` voor te plaatsen en hexadecimale door er `0x` of `0X` voor te schrijven. Dus stellen `18`, `022` en `0x12` hetzelfde getal voor. Er is geen notatie in C voor binaire getallen.

4.3.7.2 Bitsgewijze schuifoperatoren

Als je achter een geheel getal in het tientallige stelsel een nul schrijft of het laatste cijfer ervan schrapt, wordt het respectievelijk met tien vermenigvuldigd of door tien gedeeld. Hetzelfde geldt wanneer je het getal in het tweetallige stelsel noteert; alleen wordt hier met twee vermenigvuldigd of door twee gedeeld.

Een getal `n` plaatsen naar rechts schuiven gaat met `getal>>n`. Dus `100>>3` is 12, want 100 is binair 1100100 en 12 is binair 1100.



De werking van de `>>` operator

Naar links schuiven gaat net op dezelfde manier met `x<<n`. En `100<<3` geeft 800 (800_{10} is binair 1100100000_2).

In het tientallige stelsel kan je een getal delen door 10 door het laatste cijfer ervan te schrappen, en vermenigvuldigen met 10 door er een nul achter te schrijven. Bij de schuifoperatoren gebeurt er iets soortgelijks, alleen wordt er een aantal keer door twee gedeeld of met twee vermenigvuldigd.

4.3.7.3 De unaire poortoperator `~`

De operator `~` keert alle bits van waarde om (1 wordt 0 en omgekeerd). Dus `~100` wordt 156 (bij 8-bits variabelen, zoals vaak `unsigned chars`), of 65435 (bij 16-bits variabelen, typisch `shorts`), enz ...

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

De `~` operator werkend op het 16-bits getal $100_{10} = 0000000001100100_2$
geeft $65535_{10} = 1111111110011011_2$

Merk op dat dit getal te groot is om in een `signed` variabele te passen.

4.3.7.4 Binaire poortoperatoren

Het resultaat van de operator `&` is een getal dat een 1-bit heeft als beide operanden van `&` op die plaats een 1 hadden, en op alle andere plaatsen een 0.

0	0	1	0	1	1	0	1	`a`
0	1	1	1	0	1	0	0	`b`
0	0	1	0	0	1	0	0	`a&b`

De `&` operator in actie

Merk op dat de `&` operator (net als de `*` operator) in twee versies bestaat: als unaire operator ("neem het adres van") en als binaire operator. De compiler maakt zelf uit de context op welke variant je bedoelt.

De `|` en `^` operatoren werken analoog. Alleen heeft bij `|` het resultaat een 1-bit als minstens één van de twee operanden op die plaats een 1-bit heeft, en bij `^` ("exclusieve of") heeft het resultaat een 1-bit op die plaatsen waar *precies één* van beide operanden een 1-bit heeft.

0	0	1	0	1	1	0	1	`a`
0	1	1	1	0	1	0	0	`b`
0	1	1	1	1	1	0	1	`a|b`

0	0	1	0	1	1	0	1	`a`
0	1	1	1	0	1	0	0	`b`
0	1	0	1	1	0	0	1	`a^b`

De `|` en `^` operatoren

Merk tenslotte nog op dat de binaire operator `^` geen logische tegenhanger heeft—C kent dus geen `^^` operator.

4.3.7.5 Toepassing: bitbewerkingen

Je kan de bitsgewijze operatoren gebruiken om de bits van bijvoorbeeld een `int` apart te bewerken.

Bit `n` van een int `bits` aanzetten (d.i. op 1 zetten) gaat met `bits = bits | (1<<n)`. De waarde van `n` geeft aan welke bit; 0 is de meest "rechtse" bit, ook wel de **LSB** (**Least Significant Bit**) genoemd omdat de getalwaarde van die bit het kleinst is (namelijk 1). De meest linkse bit wordt analoog **MSB** (**Most Significant Bit**) genoemd.

Bit `n` uitzetten gaat met `bits = bits & ~(1<<n)`. Testen of de `n`-de bit aanstaat gaat met `if ((bits & (1<<n)) != 0)` of korter met `if (bits & (1<<n))`.

De volgende figuren zouden een en ander moeten verduidelijken; voor het gemak van spreken en schrijven heb ik voor `n` de waarde 3 gekozen:

	7	6	5	4	3	2	1	0	bitnr
	0	0	0	0	1	0	0	0	`1<<3`
Een bit aanzetten:	0	1	1	0	0	1	0	1	`bits`
	0	1	1	0	**1**	1	0	1	`bits | (1<<3)`

	7	6	5	4	3	2	1	0	bitnr
	1	1	1	1	0	1	1	1	`~(1<<3)`
Een bit uitzetten:	0	1	0	1	1	0	1	0	`bits`
	0	1	0	1	**0**	0	1	0	`bits & ~(1<<3)`

	7	6	5	4	3	2	1	0	bitnr
	0	0	0	0	1	0	0	0	`1<<3`
Een bit testen:	0	1	0	1	1	0	1	0	`bits`
	0	0	0	0	1	0	0	0	`bits & (1<<3)`

4.3.8 Incrementeer- en decrementeeroperatoren ++ en --

'You just let the machines get on with the adding up', warned Majikthise,
'and we'll take care of the eternal verities thank you very much.'
—Douglas Adams, "The Hitch Hiker's Guide to the Galaxy"

Deze operatoren vermeerderen of verminderen de inhoud van een variabele met één. Ze kunnen zowel in prefix- (d.i. de operator komt vóór het operand: `++x` en `--x`) als postfixvorm (d.i. `x++` en `x--`) gebruikt worden. Het effect van beide vormen is dat `x` met 1 verhoogd respectievelijk verlaagd wordt, maar in de prefixvorm is het resultaat van de operator de nieuwe waarde van `x`, terwijl in de postfixvorm het resultaat de oude waarde van `x` is. Een voorbeeldje zegt zoals steeds meer dan duizend regels blabla:

```
{
	int x;

	x = 10;
	printf("x++ geeft ons %d\n",x++);
```

```
        printf("x is nu %d\n",x);

        x = 10;
        printf("En ++x geeft %d\n",++x);
        printf("x is nu %d\n",x);
}
```

geeft als resultaat

```
x++ geeft ons 10
x is nu 11
En ++x geeft 11
x is nu 11
```

Je zou `++x` dus kunnen lezen als "tel 1 bij `x` op en neem dan de waarde van `x`" en `x++` als "neem de waarde van `x` en verhoog dan `x` met 1".

4.3.9 De komma-operator

De komma-operator is een binaire operator. Het effect van de uitdrukking `a,b` is dat eerst `a` wordt uitgerekend, waarna het resultaat weggegooid wordt. Daarna wordt `b` uitgerekend, wat het resultaat van de uitdrukking is. Je kan `a,b` dus interpreteren als "doe eerst `a` en dan `b`". De komma-operator is vooral populair in combinatie met `for`:

```
for (i=0, j=DIM-1; i<j; i++,j--) {
        /* verwissel tabel[i] en tabel[j] */
        int wissel;

        wissel = tabel[i];
        tabel[i] = tabel[j];
        tabel[j] = wissel;
}
```

is een stukje C code dat de array `tabel` van volgorde omkeert.

Let wel: de komma die je gebruikt om de argumenten van een functieaanroep van elkaar te scheiden, zijn *geen* komma-operatoren. Wil je in een functieaanroep toch een komma-operator gebruiken, dan moet je extra haakjes gebruiken:

```
f(x,(y,z));
```

roept de functie `f` aan met *twee* argumenten. De eerste komma scheidt de twee argumenten van de functieaanroep; de andere is een echte komma-operator. Het effect hiervan is hetzelfde als

```
y;
f(x,z);
```

(dit gaat natuurlijk niet op als y een complexere uitdrukking zou zijn die de waarde van `x` verandert). Want het eerste argument van `f(x,(y,z))` is natuurlijk de waarde van `x` en het tweede was `y,z`. Dat betekent dat eerst `y` uitgerekend wordt, waarna dat resultaat weggegooid wordt, en vervolgens `z` geëvalueerd wordt. De waarde van `z` is dan de waarde van het tweede argument.

De komma-operator wordt ook dankbaar gebruikt door macro-ontwerpers (zie §6.4.2).

▸ **OEFENING 4.10**
Wat loopt er mis in het volgende programma?

```
int main(void) {
        int tab[5][20][3];
        int x;

        /* Vul de tabel op met nullen */
        for (x=0; x<5; x++) {
                int y;

                for (y=0; y<20; y++) {
                        int z;

                        for (z=0; z<3; z++)
                                tab[x,y,z] = 0;
                }
        }
}
```

♢

4.3.10 De ? : operator

'Either die in the vacuum of space, or …
tell me how good you thought my poem was!'
—Douglas Adams, "The Hitch Hiker's Guide to the Galaxy"

Dit is de enige ternaire operator die C kent. De uitdrukking `a?b:c` wordt als volgt geëvalueerd:

- Bereken de waarde van de uitdrukking `a`.
- Als `a` waar is (d.i. niet nul), is het resultaat van `a?b:c` de waarde van `b`. In dat geval wordt `c` *niet* geëvalueerd.
- Is daarentegen `a` vals (dus gelijk aan nul), dan is het resultaat de waarde van `c`, en `b` wordt niet uitgerekend.

Je kan `a?b:c` dus lezen als "als `a` dan `b`, anders `c`".

Door deze operator kan je `if ... then ... else` opdrachten uitsparen:

```
int abs(int x)
{
        return x<0 ? -x : x;
}
```

doet hetzelfde als

```
        int abs(int x)
        {
                if (x<0) return -x;
                else return x;
        }
```

▸ **OEFENING 4.11**
(voor wie het hoofdstuk over macro's al doorgewerkt heeft:) Er zijn toch enkele subtiele verschillen tussen de macro- en de functie-versie. Welke?

◊

Een manier om een array van strings achter elkaar op één lijn af te drukken is

```
void drukarray(int grootte, char **strings)
{
        int tel;

        for (tel = 0; tel < grootte; tel++) {
                printf("%s",strings[tel]);
                if (tel < grootte-1) putchar(' ');
        }
        putchar('\n');
}
```

wat je korter kan schrijven als

```
void drukarray(int grootte, char **strings)
{
        int tel;

        for (tel = 0; tel < grootte; tel++) {
                printf("%s",strings[tel]);
                putchar ((tel < grootte-1) ? ' ' : '\n');
        }
}
```

De uitdrukking `(tel < grootte) ? ' ' : '\n')` heeft immers een spatie als resultaat wanneer `tel < grootte`, en `'\n'` als resultaat wanneer `tel >= grootte`.

In deze eenvoudige gevallen is het gebruik van de `? :` operator nog te verantwoorden, maar voor je het weet wordt de code ondoorzichtig, dus het is beter deze operator met mate te gebruiken. In het voorbeeld leest de constructie vrij natuurlijk als je er aan gewend bent: "schrijf een spatie na elke string, tenzij het de laatste uit de rij is; schrijf dan een newline". Merk tenslotte nog op dat de haakjes rond `tel < grootte-1` strikt genomen niet nodig zijn, maar daar alleen staan om de uitdrukking duidelijker te maken.

Doordat de `? :` operator rechts-naar-links associatief is, kan je hem als volgt samenstellen:

```
int teken(float getal) {
        return (getal < 0) ? -1
                : (getal > 0) ? 1
                : 0;
}
```

waarbij je de uitdrukking na `return` kan lezen als "als het getal negatief is, dan -1, als het positief is, dan 1, en anders 0".

4.3.11 Prioriteit en associativiteit van operatoren

De uitdrukking `3+5*2` zou je in principe op twee manieren kunnen opvatten: als `3+10` of als `8*2`, naar gelang je eerst `*` of `+` uitwerkt. In C worden vermenigvuldigingen eerst uitgewerkt, dus in feite staat er `3+(5*2)`. We zeggen dat `*` een **hogere prioriteit** heeft dan `+`. Door haakjes in te voegen kan je de prioriteit aanpassen zoals je wil. Het kan nooit kwaad extra haakjes in te lassen als je twijfelt—je programma zal er niet trager of groter door worden. Let wel: deze haakjes hebben niets te maken met de functieaanroep-operator (zie §4.3.6).

Alle operatoren van C staan in tabel 4.3. Ze staan gegroepeerd per prioriteit, van hoog naar laag. Binnen elke groep hebben de operatoren dezelfde prioriteit.

Operatoren met hogere prioriteit worden geëvalueerd "vóór" operatoren met lagere prioriteit. Ik bedoel daarmee dat in een uitdrukking als `3+5*2` eerst de `*` wordt geëvalueerd, zoals hierboven al uitgelegd. Maar wat gebeurt er met operatoren die dezelfde prioriteit hebben, bijvoorbeeld in de uitdrukking `8-3+2`? Bij elke groep operatoren staat in de tabel een associativiteitsrichting vermeld (links naar rechts of rechts naar links). De groep met `+` en `-` heeft links naar rechts associativiteit, wat inhoudt dat de meest linkse operator eerst wordt geëvalueerd; in het voorbeeld wordt dat dus `(8-3)+2`. Een ander voorbeeld zijn de toekenningsoperatoren, waarbij de rechts naar links associativiteit ervoor zorgt dat een uitdrukking als `a=b=c=0` effectief als `a=(b=(c=0))` wordt geïnterpreteerd.

In sommige gevallen werken de prioriteitsregels van C een beetje tegen. Een veel gebruikte constructie is bijvoorbeeld

```
while ( (c=getchar()) != EOF) {...}
```

De haakjes rond `c=getchar()` zijn nodig, omdat zonder de haakjes de C compiler de `!=` eerst evalueert, net alsof er

```
while ( c= (getchar() != EOF)) {...}
```

zou gestaan hebben.

4.4 Opslagklassen, programma's verspreid over meerdere bestanden

Tot nog toe hebben we lokale en globale variabelen gebruikt, en hebben we alle functies van onze programma's in één `.c` bestand geschreven. Variabelen en functies hebben ook een **opslagklasse** (**storage class**), die de zichtbaarheid en levensduur ervan mee bepaalt. De naam van de opslagklasse schrijf je helemaal vooraan de declaratie, bijvoorbeeld `register int len;`.

4.4.1 De `auto` (automatic) en `register` opslagklassen

De variabelen binnen een functie zijn allemaal `auto`, tenzij je expliciet een andere klasse opgeeft. Andere dingen (dus functies of globale variabelen) kunnen niet `auto` of `register` zijn. `auto` variabelen worden automatisch aangemaakt bij het binnengaan van het blok en verwijderd bij het verlaten ervan. `auto` variabelen zonder initializer bevatten een onbepaalde

<table>
<tr><td>()</td><td>functieaanroep</td><td>→</td></tr>
<tr><td>[]</td><td>array</td><td></td></tr>
<tr><td>.</td><td>element van een struct</td><td></td></tr>
<tr><td>-></td><td>element van een pointer naar een struct</td><td></td></tr>
<tr><td>!</td><td>logisch niet</td><td>←</td></tr>
<tr><td>~</td><td>binair niet</td><td></td></tr>
<tr><td>-</td><td>tegengestelde van een getal</td><td></td></tr>
<tr><td>++</td><td>incrementeer</td><td></td></tr>
<tr><td>--</td><td>decrementeer</td><td></td></tr>
<tr><td>&</td><td>adres</td><td></td></tr>
<tr><td>*</td><td>indirectie</td><td></td></tr>
<tr><td>(type)</td><td>cast</td><td></td></tr>
<tr><td>sizeof()</td><td></td><td></td></tr>
<tr><td>*</td><td>vermenigvuldiging</td><td>→</td></tr>
<tr><td>/</td><td>deling</td><td></td></tr>
<tr><td>%</td><td>rest na deling</td><td></td></tr>
<tr><td>+</td><td>som</td><td>→</td></tr>
<tr><td>-</td><td>verschil van twee getallen</td><td></td></tr>
<tr><td><<</td><td>binair naar links schuiven</td><td>→</td></tr>
<tr><td>>></td><td>binair naar rechts schuiven</td><td></td></tr>
<tr><td><</td><td>kleiner dan</td><td>→</td></tr>
<tr><td><=</td><td>kleiner dan of gelijk</td><td></td></tr>
<tr><td>></td><td>groter dan</td><td></td></tr>
<tr><td>>=</td><td>groter dan of gelijk</td><td></td></tr>
<tr><td>==</td><td>gelijk</td><td>→</td></tr>
<tr><td>!=</td><td>ongelijk</td><td></td></tr>
<tr><td>&</td><td>binair en</td><td>→</td></tr>
<tr><td>^</td><td>binair exclusieve of</td><td>→</td></tr>
<tr><td>|</td><td>binair of</td><td>→</td></tr>
<tr><td>&&</td><td>logisch en</td><td>→</td></tr>
<tr><td>||</td><td>logisch of</td><td>→</td></tr>
<tr><td>? :</td><td></td><td>←</td></tr>
<tr><td>=</td><td>toekenning</td><td>←</td></tr>
<tr><td>*=</td><td></td><td></td></tr>
<tr><td>/=</td><td></td><td></td></tr>
<tr><td>%=</td><td></td><td></td></tr>
<tr><td>+=</td><td></td><td></td></tr>
<tr><td>-=</td><td></td><td></td></tr>
<tr><td><<=</td><td></td><td></td></tr>
<tr><td>>>=</td><td></td><td></td></tr>
<tr><td>&=</td><td></td><td></td></tr>
<tr><td>^=</td><td></td><td></td></tr>
<tr><td>|=</td><td></td><td></td></tr>
<tr><td>,</td><td></td><td>→</td></tr>
</table>

Tabel 4.3: Operatoren in C
← betekent: rechts-naar-links evaluatievolgorde
→ betekent: links-naar-rechts evaluatievolgorde

waarde. Functie-argumenten worden ook als `auto` beschouwd (ze horen bij het buitenste blok van de functiedefinitie).

Menselijke programmeurs zullen haast nooit `auto` schrijven; dit sleutelwoord is er vooral voor de **programmageneratoren**.

De `register` klasse is een variant op `auto`. Als je een variabele deze klasse geeft, is dat een hint voor de compiler dat die variabele veel gebruikt zal worden in de functie waar ze in voorkomt. De compiler zal dan proberen die variabele niet in het geheugen te bewaren, maar in de processor zelf. De processor heeft immers een klein stukje geheugen aan boord (meestal in totaal enkele bytes tot enkele tientallen bytes); de variabelen die in dat geheugen bewaard worden heten **registers**. Variabelen van de `register` klasse hebben dan ook geen adres. De compiler is niet verplicht om een `register` variabele daadwerkelijk in een register te bewaren—het kan gebeuren dat er geen vrije registers meer zijn, of dat de variabele te groot is om in een register te passen. Tegenwoordige compilers zijn slim genoeg om zelf uit te maken welke variabelen het best in een register bewaard kunnen worden en behandelen `register` als `auto`. Ook functie-argumenten mogen `register` zijn, wat de efficiëntie van korte functies enorm kan verhogen. Er wordt dan ook veel van gebruik gemaakt in bibliotheekfuncties, die bijna per definitie zo efficiënt mogelijk moeten zijn:

```
int strlen(register char *s)
{
        register int resultaat = 0;

        while (*s++) resultaat++;
        return resultaat;
}
```

4.4.2 De `static` opslagklasse

Alleen lokale variabelen kunnen `static` zijn. De zichtbaarheid ervan is net hetzelfde als bij de `auto` variabelen: ze zijn alleen maar zichtbaar in het blok waarin ze gedeclareerd zijn. Maar in tegenstelling tot `auto` variabelen wordt een `static` variabele maar één keer aangemaakt, namelijk bij het begin van het programma (zelfs nog vóór `main()` begint uitgevoerd te worden). De variabele blijft voor de hele duur van het programma bestaan. De waarde ervan blijft dus behouden, ook bij het verlaten van de functie. Je kan `static` variabelen een beginwaarde geven met een initializer; die waarde krijgen ze op het ogenblik dat ze aangemaakt worden, dus bij het begin van het programma. Als je geen initializer schrijft, wordt de variabele met nul gevuld.

Als voorbeeld zal ik het cirkelprogramma uit §2.5 herschrijven met een functie `ster()` die op een bepaalde plaats een sterretje zet, tenzij je vlak daarvoor al een sterretje op die plaats had gezet:

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

#define SCHERM_BREED 80
#define SCHERM_HOOG 24
#define STRAAL 10
#define STAP 0.01
#define PI 3.141592
```

```
void ster(int x,int y)
{
        static int vorigx = -1;   /* x en y positie van het */
        static int vorigy = -1;   /* laatst gezette sterretje */

        if (x==vorigx && y==vorigy) return;
        /* onthou de positie van deze ster */
        vorigx = x;
        vorigy = y;
        /* Cursor naar de juiste plek en een sterreke afdrukken */
        printf("\033[%d;%dH*",SCHERM_HOOG/2 + x,SCHERM_BREED/2 + y );
}

void main(void)
{
        double teller;

        for (teller = 0.0; teller <= 2*PI; teller += STAP)
                ster(STRAAL*sin(teller),STRAAL*cos(teller));
}
```

Enkele opmerkingen: ik ben de vervelende cast van `STRAAL*sin(teller)` naar `int` kwijt, omdat het prototype van `ster()` duidelijk maakt dat die functie twee `int`s nodig heeft; het casten gaat dus automatisch. Voorts maak ik gebruik van initializers door `vorigx` en `vorigy` bij het begin van het programma op `-1` te zetten, zodat het eerste sterretje altijd getekend wordt.

4.4.3 De `extern` opslagklasse

Globale variabelen en functies zijn `extern`, als je niet zelf een opslagklasse opgeeft. Een externe variabele of functie is ook zichtbaar in andere bestanden. Als je het sleutelwoord `extern` schrijft, zoals dus in

```
        extern int breedte;
        extern void f(void);
```

dan wordt er geen nieuwe variabele gemaakt, maar betekent alleen maar: ergens anders (in dit of een ander `.c` bestand) is er een `extern`e `int` variabele `breedte`, en een `extern`e functie `f`. Dit is de enige vorm van variabelendeclaratie in C die geen geheugenruimte voor de variabele zelf reserveert. Bij de declaratie van `f` is het vermelden van `extern` niet nodig, want als je het weglaat blijft er een prototype over, dat precies hetzelfde effect heeft (namelijk de compiler melden dat er ergens anders in dit bestand, of in een ander bestand een functie `f` bestaat). Ook bij variabelen mag je eerst een `extern` declaratie maken, gevolgd door een echte definitie in hetzelfde bestand, net zoals je in hetzelfde bestand eerst een prototype van een functie mag schrijven en later de functiedefinitie zelf.

Toepassingen hiervan volgen in §9.1.

4.4.4 De `static extern` opslagklasse

Daarnet heb ik al uitgelegd dat lokale variabelen de `static` klasse kunnen hebben. Ook globale variabelen en functies kunnen een `static` krijgen; in dat geval spreken we van de "static extern" opslagklasse (volgens ANSI C eigenlijk "`extern` with internal linkage" genoemd). Static extern variabelen gedragen zich als gewone globale variabelen, op één verschil na: ze zijn alleen maar zichtbaar in dit `.c` programmabestand. Je kan dus in twee `.c` bestanden een static extern variabele of functie maken met dezelfde naam, zonder dat ze elkaar in de weg zitten.

Zie ook weer §9.1 voor toepassingen van deze opslagklasse.

4.5 unions

Met een `struct` kon je een variabele maken waar een aantal kleine deelvariabelen achter elkaar in zaten. Een `union` maak en gebruik je op net dezelfde manier:

```
union hoeveelheid {
        int stuks;
        float liter;
        double kilogram;
};
```

Het belangrijke verschil is dat bij een `union` al deze deelvariabelen 'over' elkaar liggen, en niet 'achter' elkaar. Dus zal de deelvariabele `stuks` dezelfde bytes van het geheugen innemen als `liter` en `kilogram`. Verander je dus `stuks`, dan zal de waarde van `liter` en `kilogram` op een onvoorspelbare manier mee veranderen (hoe precies hangt af van de juiste manier waarop de computer in kwestie een `int`, `float` of `double` opslaat). De grootte van een `union`-variabele is de grootte van het grootste veld ervan (in ons geval dus net zo groot als een `double`).

Een `union` komt uitstekend van pas om iets in te bewaren dat naar gelang de omstandigheden een verschillend type kan hebben:

```
#define PERSTUK      0
#define PERVOLUME    1
#define PERGEWICHT   2

struct artikel {
        long streepjescode;
        char soort;
        union hoeveelheid in_voorraad;
};

struct artikel supermarkt [100];
```

In dit voorbeeld bevat elk element van de array `supermarkt` een `struct artikel`. Elk artikel bevat een `long`, een `char` en een `union hoeveelheid`. Afhankelijk van de soort artikel zullen we de grootte van de voorraad met een `int` (`stuks`), `float` (`liter`), enz. bijhouden. Dat kan aanleiding geven tot functies als

```
void printArtikel(struct artikel *art) {
        printf("%ld ",art->streepjescode);
        switch (art->soort) {
        case PERSTUK: printf("%d stuks",art->in_voorraad.stuks);
                break;
```

```
        case PERVOLUME: printf("%f liter",art->in_voorraad.liter);
                break;
        case PERGEWICHT: printf("%f kg",art->in_voorraad.kilogram);
                break;
        }
}
```

Het is natuurlijk belangrijk dat we nooit in een artikel met als `soort` de waarde `PERSTUK` ineens het `liter` of `kilogram` veld van `in_voorraad` gaan gebruiken, of ons programma draait danig in de soep.

Je kan zien dat de velden van een union net als die van een struct worden aangesproken: met de `.`-operator.

4.6 stdargs

Alle functies die we tot nog toe geschreven hebben, nemen een vast aantal argumenten. Je kan ook functies maken met een niet vooraf bepaald aantal argumenten, zoals bijvoorbeeld bij `printf()`. Het principe is dat er een aantal vaste argumenten zijn (bij `printf()` één, het formaatargument) gevolgd door de niet vast bepaalde argumenten. Laten we een `vdrukf()` functie schrijven die een beetje `printf()`-achtig werkt: alle letters in het formaatargument worden letterlijk afgedrukt, behalve als er `%d` in voorkomt. In dat geval verwacht `vdrukf()` een `int`-argument, dat net als bij `printf()` als getal afgedrukt wordt. Als het getal tussen 0 en 10 ligt wordt het echter voluit afgedrukt. Dus we willen dat

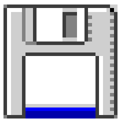
```
vdrukf("Roodkapje had %d pannekoeken en %d flessen wijn in haar mandje",5,2);
```

ervoor zorgt dat

```
    Roodkapje had vijf pannekoeken en twee flessen wijn in haar mandje
```

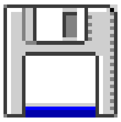
afdrukt.

Om te beginnen moeten we in het prototype van `vdrukf()` aangeven dat de functie een willekeurig aantal argumenten kan hebben. Dat gebeurt door `...` na de vaste argumenten te schrijven:


vdrukf.h

```
void vdrukf(char *,...);
```

De functie heeft dus één vast argument van het type `char *` (de formaatstring), gevolgd door de optionele argumenten. Om aan die extra argumenten te geraken, kan de functie gebruik maken van enkele macro's uit `stdarg.h`. Ze beginnen allemaal met `va_`.


vdrukf.c

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

void vdrukf(char *formaat,...)
{
        va_list arg;
        static char* getallen[] = {
```

```
                "nul","een","twee","drie","vier","vijf",
                "zes","zeven","acht","negen","tien"
        };

        va_start(arg,formaat);
        while(*formaat) {
                if (*formaat == '%') {
                        formaat++;  /* ga voorbij de '%' */
                        if (*formaat == 'd') {
                                int getal = va_arg(arg,int);
                                formaat++;  /* ga voorbij 'd' */
                                if (getal>=0 && getal<=10)
                                        printf("%s",getallen[getal]);
                                else printf("%d",getal);
                        } else if (*formaat == 0) return;
                        else putchar(*formaat++);
                } else putchar(*formaat++);
        }
        va_end(arg);
}
```

`va_list` is een datatype dat gebruikt wordt om door de lijst optionele functie-argumenten te wandelen. De `stdarg` macro's hebben een variabele van dit type (in het voorbeeld `arg`) nodig.

De macro `va_start` initialiseert zo'n variabele. Er zijn twee parameters: de te initialiseren `va_list` variabele en de naam van het laatste vaste argument van de functie. In ons geval is er maar één vast argument, `formaat`, dus moeten we `va_start(arg,formaat);` schrijven. Intern neemt de macro `va_start` het adres van het laatste vaste argument. Daarom mag dat laatse vaste argument geen `register` variabele zijn; functies of arrays zijn ook niet toegelaten.

Eens de `va_list` variabele geïnitialiseerd is, kan je de extra argumenten één voor één langslopen met `va_arg`. Je kan de argumenten alleen van links naar rechts bekijken; eens een argument opgehaald kan je dus niet meer de waarde van de vorige argumenten opvragen (tenzij je weer helemaal van voor af aan begint door weer `va_start` te gebruiken). `va_arg` heeft de `va_list` variabele nodig om bij te houden aan welk functieargument je bezig bent, en als tweede parameter ook het type van het op te halen functieargument. In het voorbeeld werken we enkel met `int` argumenten, vandaar `va_arg(arg,int)`. De macro `va_arg` moet het type onder andere weten om te weten hoeveel bytes er moet 'opgeschoven' worden in de argumentenlijst. Als er geen volgende argument meer is, of als het type ervan niet overeenkomt met het aan `va_arg` opgegeven type, kunnen er alle soorten rare dingen gebeuren (wat precies hangt af van het gebruikte systeem).

Als alle argumenten verwerkt zijn, moet je `va_end` aanroepen. Deze macro zorgt voor de nodige opkuis-acties. De `va_` macro's moeten immers nogal wat achter-de-schermen truukwerk uithalen.

Merk op dat het onmogelijk is een functie te maken die alleen maar variabele argumenten heeft; er is altijd een vast argument nodig.

4.7 Weinig gebruikte syntaxelementen

4.7.1 enum

Met `#define` kan je constanten maken, maar er is ook een alternatief:

```
int main(void) {
        enum weekdagen { MAANDAG, DINSDAG, WOENSDAG, DONDERDAG, VRIJDAG };
        enum weekdagen dag;

        dag = MAANDAG;
        /* je kan zotte dingen doen met een enum! */
        dag++;
        dag = 1033;
        {       int hetmag;
                hetmag = MAANDAG;
        }
        return 0;
}
```

Op de eerste regel maak ik een nieuw type `enum weekdagen` genaamd. Op de tweede regel maak ik een variabele van dat type, `dag`.

Je kan ook zien dat de compiler een `enum` precies als een `int` behandelt. De `MAANDAG`, `DINSDAG`, enz. worden als constanten van het type `int` beschouwd (je mag dus niet `MAANDAG=2;` schrijven, net zo min als `7=3;` mag). Er is, zoals uit het programma blijkt, geen enkele vorm van controle of je wel geldige waarden in een `enum`-variabele stopt: ik mag gerust `1033` in zo'n variabele stoppen. De GNU C-compiler geeft zelfs met geen mogelijkheid een waarschuwing dat ik ook maar iets verkeerd zou doen in het hele programma! Je mag eenzelfde enum-constante ook niet in twee verschillende `enum`s gebruiken; zo mag je niet binnen `main()` nog een enum `dagen` maken die er zo uitziet:

```
enum dagen { MAANDAG, DINSDAG, WOENSDAG, DONDERDAG,
        VRIJDAG, ZATERDAG, ZONDAG };
```

De compiler zal klagen dat `MAANDAG` al van een ander enum-type is.

Standaard begint de compiler de enum-constanten te nummeren vanaf 0 (dus `WOENSDAG` is bijvoorbeeld 2); je kan expliciet een andere keuze opgeven:

```
        enum weekdagen { MAANDAG=1, DINSDAG=7, WOENSDAG, DONDERDAG, VRIJDAG };
```

Als je zelf geen waarde opgeeft, telt de compiler gewoon verder (dus `WOENSDAG` is `8`, `DONDERDAG` is `9` enz). Het is toegestaan dat twee verschillende enum-constanten dezelfde waarde hebben.

4.8 Communicatie met de buitenwereld

Strikt genomen is al wat ik hier ga vertellen geen deel van de syntax van C, maar goed …

4.8.1 Argumenten van `main()`

Naast de `int main(void)` vorm van `main()` is er een tweede vorm toegelaten:

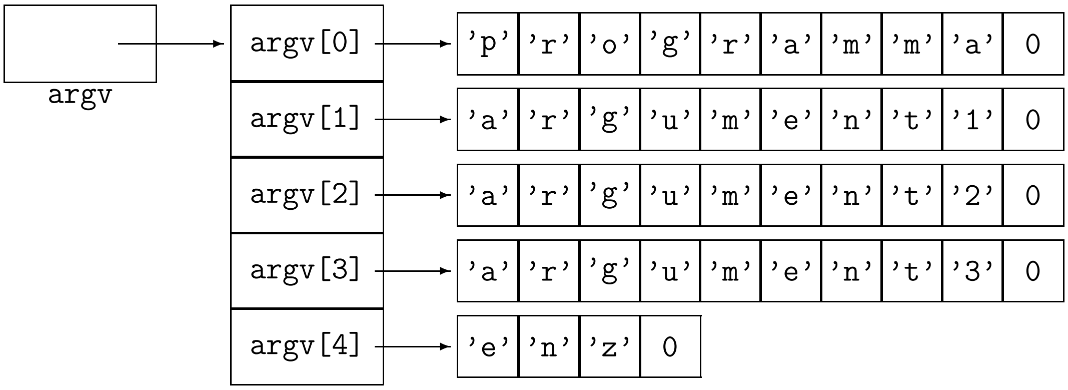
```
int main(int argc,char **argv)
```

De namen `argc` en `argv` staan voor *argument count* en *argument vector* ('vector' is een geleerd woord voor 'array'). Je mag natuurlijk andere namen kiezen als je dat wilt, hoewel het niet vaak gedaan wordt.

Onder de meeste besturingssystemen kan je aan programma's argumenten geven door iets in de shell in te tikken in de trant van

```
programma argument1 argument2 argument3 enz
```

In dat geval zal `argc` gelijk zijn aan 5 (de naam van het programma wordt dus ook als een argument meegeteld). `argv` wijst naar een array van stringpointers: `argv[0]` bevat een pointer naar `programma`, `argv[1]` naar `argument1`, en zo verder tot aan `argv[4]`, dat naar `enz` wijst.



De opbouw van `argv`

Hier is een programmaatje dat zijn argumenten afdrukt (vaak te vinden onder de naam `echo`):

echo.c

```
#include <stdio.h>

int main(int argc, char **argv)
{
        int tel;

        for (tel = 1; tel < argc; tel++) {
                printf("%s",argv[tel]);
                if (tel < argc-1) putchar(' ');
        }
        putchar('\n');
        return 0;
}
```

4.8.2 Environment variabelen

Onder de meeste besturingssystemen kan de gebruiker een aantal instellingen doen door middel van een **environment**. Het environment bestaat uit symbolische variabelen (de zogenaamde **environment-variabelen**) die de gebruiker een waarde kan geven, bijvoorbeeld met (afhankelijk van de gebruikte shell)

```
export EDITOR=pico
```

waardoor de environment variabele `EDITOR` de waarde `pico` zal bevatten.

In C kan je een environment variabele opvragen met de `getenv()` functie die in `stdlib.h` zit:

```
char *getenv(const char *naam);
```

Dus `getenv("EDITOR")` zal `"pico"` opleveren. Als de environment-variabele niet gedefinieerd is, krijg je een `NULL` terug.

Hoofdstuk 5

Bestanden

5.1 Inleiding: bestanden en stromen (streams)

5.1.1 Bestanden

Een belangrijke manier om gegevens permanent (dus ook nadat het programma gestopt is) te bewaren zijn **bestand**en (**file**s). Een bestand is gewoon een sliert bytes achter elkaar die als één geheel bewaard worden, bijvoorbeeld op een diskette of harde schijf. Net zoals bij een array heeft elke byte een volgnummer, en zoals gewoonlijk in de computerwereld beginnen we ook hier te tellen vanaf nul. Het verschil met arrays is dat een array in het geheugen staat, terwijl de bytes van een bestand eerst expliciet (door middel van functies uit `stdio.h`) in het geheugen moeten worden ingelezen; ook voor het wijzigen van de inhoud van een bestand zijn er speciale functies.

Om verschillende bestanden op dezelfde schijf uit elkaar te houden heeft elk bestand een **bestandsnaam**.

Je hebt al de hele tijd met bestanden gewerkt: de programma's die je gemaakt hebt, worden immers als bestand bewaard (bijvoorbeeld het bestand met als naam `hello.c`).

5.1.2 Stromen

Stromen (**streams**) bestaan in twee soorten: leesstromen en schrijfstromen.

Uit een leesstroom komen één voor één bytes te voorschijn. Van waar die bytes komen, hangt af van de stroom zelf. Je kan bijvoorbeeld een stroom aan het toetsenbord koppelen; voor elke toets die ingedrukt wordt zal een byte uit de **standaard invoer**-stroom te voorschijn komen. De standaard invoer-stroom (**standard input**) wordt in C met `stdin` aangeduid. Je kan ook een stroom aan een bestand koppelen. In dat geval komen de bytes van het bestand één voor een langsge"stroomd". Als je een bestand via een leesstroom benadert, kan je het niet veranderen (je mag er dus niet ineens beginnen in schrijven).

Een schrijfstroom doet net het omgekeerde: je kan er bytes ingooien. Waar die bytes naartoe gaan, hangt ook weer af van de stroom zelf. Je kan een schrijfstroom bijvoorbeeld met de printer verbinden; alle bytes die je in de stroom schrijft, zullen door de printer afgedrukt worden. Je kan een schrijfstroom ook weer aan een bestand koppelen. Alle bytes die je in zo'n stroom stopt, worden achter elkaar in een bestand op schijf bewaard. Als je een schrijfstroom koppelt aan een bestand dat al bestond, kan je dat oude bestand laten vernietigen, maar je kan er ook voor kiezen om het oude bestand te laten staan en te beginnen schrijven aan het einde van het bestand. Alle bytes die je in zo'n stroom stopt worden dus in dat geval achteraan het bestand toegevoegd (**append**).

De leesstroom `stdin` heeft ook een schrijf-tegenhanger, `stdout`, de **standaard uitvoer** (**standard output**), die alles wat je erin gooit op het scherm afdrukt. De in- en uitvoerfuncties uit §2.6 maken allemaal achter de schermen gebruik van `stdin` en `stdout`.

Om een bestand naar willekeur te lezen en te schrijven, bestaan er tenslotte ook lees/schrijf-stromen. Je kan dan naar een willekeurige plek van een bestand gaan en er naar keuze beginnen te lezen of schrijven. Het is mogelijk een bestand te verlengen door "voorbij het uiteinde" ervan te schrijven, maar je kan een bestand niet korter maken (wat soms vervelend kan zijn).

5.2 Eenvoudige bestandsverwerking met redirectie

De meeste besturingssystemen maken het mogelijk de standaard in- en uitvoer te koppelen aan bestanden in plaats van toetsenbord en scherm. Op die manier kunnen we met de functies die toetsenbordinvoer en schermuitvoer verzorgden al vrij krachtige programma's schrijven. De manier waarop dat precies gebeurt, hangt af van het besturingssysteem en de shell. Meestal zorgt het intikken van

```
programmanaam >uitbestand
```

in de shell ervoor dat de standaard uitvoer niet meer op het scherm wordt getoond, maar in het bestand met de naam `uitbestand` wordt geschreven. Met

```
programmanaam >>uitbestand
```

wordt de standaard uitvoer achteraan het bestand `uitbestand` geplakt (als dat bestand niet bestond, wordt het aangemaakt). De standaardinvoer kan je aan een bestand hangen met

```
programmanaam <inbestand
```

Beide kan je combineren:

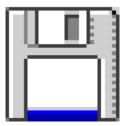
```
programmanaam <in >uit
```

De volgorde waarin `<in` en `>uit` staan, heeft meestal niet veel belang.

Het op deze manier koppelen van standaard in- en uitvoer aan bestanden, wordt **redirectie** genoemd. Nadeel van redirectie is dat je het toetsenbord (bij het redirecten van de standaard invoer) en/of het scherm (bij het redirecten van de standaard uitvoer) niet meer kan gebruiken in je programma; ook kan je bijvoorbeeld niet lezen uit meer dan één bestand tegelijk. Verderop zal ik uitleggen hoe je die beperkingen kunt omzeilen.

5.2.1 Voorbeeld: woorden tellen

Laten we een programma schrijven dat telt uit hoeveel woorden de standaard invoer bestaat.


woorden.c

```
#include <stdio.h>

int iswit(char c)
{
        switch (c) {
        case '\t':
        case '\n':
```

```
        case ' ':
                return 1;
        default:
                return 0;
        }
}

int main(void)
{
        long woorden;
        int inwoord = 0;

        woorden = 0;
        while (!feof(stdin)) {
                char letter;

                letter = getchar();
                if (iswit(letter)) {
                        if (inwoord)
                                /* einde van een woord */
                                inwoord = 0;
                } else {        /* we hebben een letter gelezen */
                        if (!inwoord) {
                                /* begin van een woord */
                                woorden++;
                                inwoord = 1;
                        }
                }
        }
        printf("%ld woorden\n",woorden);
}
```

De opzet van dit programma is eenvoudig: we tellen eigenlijk het aantal keren dat een spatie gevolgd wordt door een letter, waarbij we tabs en regelovergangen als spatie ("witte ruimte") behandelen en al de rest als deel van een woord.

De enige nieuwigheid hier is de functie `feof()` (dit uit `stdio.h` komt). Deze functie test of een stroom ten einde is (EOF betekent End Of File); als we een nul terugkrijgen betekent dat dat er nog bytes klaarstaan om gelezen te worden. Als je `stdin` gewoon aan het toetsenbord koppelt, kan je een EOF genereren door een bepaalde toetscombinatie in te drukken (onder UNIX vaak Ctrl-\ of Ctrl-D, onder MS-DOS vaak Ctrl-Z).

Een leuke manier om dit programma te testen is door het aantal woorden in zijn eigen broncode te tellen met

```
woorden <woorden.c
```

wat als resultaat `87 woorden` oplevert (merk op dat dit programmaatje een beetje primitief is en ook `{` of `=` als een woord beschouwt).

Het vorige programma kan iets efficiënter gemaakt worden als je weet dat `getchar()` eigenlijk geen `char` maar wel een `int` teruggeeft. In normale omstandigheden geeft `getchar()` netjes een letter terug, maar als het einde van het bestand bereikt is, krijg je de waarde

`EOF` terug (die gedefinieerd is in `stdio.h`). Dat geeft meteen aan waarom `getchar()` geen `char` teruggeeft, want dan zou er geen mogelijkheid zijn om een speciale waarde (`EOF`) te onderscheiden van een gewone letter in het bestand. De hoofdlus van het programma van daarnet kan je dus herwerken tot

```
while (1) {
        int letter;

        letter = getchar();
        if (letter == EOF) break;
        else if (iswit(letter)) {
                /***** hier de rest van de lus *****/
}
```

Vaak zul je deze constructie nog compacter geschreven zien als

```
        if ( (letter=getchar()) == EOF) break;
```

Zie ook §4.3.11.

5.2.2 Een rot13-encoder/decoder

Een manier om een tekst onleesbaar te maken is alle letters 13 plaatsen in het alfabet opschuiven. Dus

`abcdefghijklmnopqrstuvwxyz`

wordt

`nopqrstuvwxyzabcdefghijklm`

Als je een 13-geroteerde tekst nog eens 13-roteert, dan zijn alle letters 26 plaatsen opgeschoven, en er is dus niets gebeurd. Je kan dus hetzelfde programma gebruiken om de tekst te coderen en te decoderen.

Het spreekt vanzelf dat dit soort codering bijzonder gemakkelijk te kraken is; als je niet wil dat iemand anders je documenten kan lezen, gebruik je best een gesofistikeerder algoritme ...

Als we veronderstellen dat in de gebruikte karakterset alle letters netjes alfabetisch op elkaar volgen (dus `'a'+1` is `'b'`, enz.) dan kunnen we snel detecteren of een letter een kleine letter is (`letter>='a' && letter<='z'`). Het volgnummer van de letter in het alfabet is dan `letter-'a'` (als we tenminste beginnen bij 0 voor de `a` tot en met 25 voor de `z`). Voor de rot-13 codering moeten we 13 bij dat getal tellen, en er natuurlijk rekening mee houden dat als we voorbij de `z` zitten, we 26 moeten aftrekken. In de praktijk komt het erop neer dat de eerste 13 letters van het alfabet 13 plaatsen naar rechts worden geroteerd en de laatste 13 letters 13 plaatsen naar links. Wat er dus eigenlijk gebeurt is dat de groepjes `abcdefghijklm` en `nopqrstuvwxyz` gewoon van plaats verwisselen (en twee keer wisselen is hetzelfde als niets doen, dus de decoder is hetzelfde als de encoder).

De C versie zo er zo kunnen uitzien:

rot13.c

```
#include <stdio.h>

/* 13-roteer een getal tussen 0 en 25;
 * het resultaat ligt weer tussen 0 en 25 */
int rot13(int i) {
        return i<13 ? i+13 : i-13;
}

int main(void) {
        int letter;

        while ((letter = getchar()) != EOF) {
                if (letter>=’a’ && letter<=’z’)        /* kleine letter */
                        putchar(’a’ + rot13(letter-’a’));
                else if (letter>=’A’ && letter<=’Z’)   /* hoofdletter */
                        putchar(’A’ + rot13(letter-’A’));
                else putchar(letter);                  /* geen letter */
        }
        return 0;
}
```

▸ **OEFENING 5.1**
Pas het programma aan zodat het roteren niet noodzakelijk over een afstand van 13 letters gebeurt. Laat de gebruiker de rotatie-afstand als programma-argument meegeven.

Een rotatie-afstand van 3 staat bekend onder de naam Caesar-sleutel (omdat Julius Caesar op deze manier zijn boodschappen voor ongewenste ogen onleesbaar zou gemaakt hebben).

♢

5.2.3 stderr

Naast de `stdin` en `stdout` stroom krijgt elk programma nog een derde standaardstroom: `stderr`. Het is de bedoeling dat het programma status- en foutboodschappen naar `stderr` stuurt, en niet naar `stdout`. Uitvoer van `stdout` wordt immers nog al eens gerideréct, zodat de gebruiker eventuele foutmeldingen niet zou opmerken.

Redirectie van `stderr` is ook mogelijk. Afhankelijk van de shell waarmee je werkt gaat dat bijvoorbeeld zo:

```
programmanaam 2>errors
```

De tekencombinatie `2>` (in sommige shells `>&`) geeft dus aan naar welk bestand de `stderr` uitvoer gestuurd moet worden.

5.3 Functies voor standaard in- en uitvoer

Functies die met standaard in- en uitvoer werken heb ik al uitgelegd in §2.6.

5.3.1 `printf()`

Ik heb vroeger al iets uitgelegd over het formaatargument van `printf()`. De mogelijkheden zijn echter veel ruimer. Na een `%` teken verwacht `printf()` achtereenvolgens

- een aantal (geen, één of meer) van de volgende tekens:

  `#` om aan te geven dat de uitvoer in een "andere vorm" moet gebeuren. Bij `%o` wordt een `0` vóór het octale getal afgedrukt; bij `%x` en `%X` wordt respectievelijk `0x` of `0X` vóór het hexadecimale getal geschreven. Bij `%e`, `%E`, `%f`, `%g` en `%G` wordt altijd een decimale punt geschreven, ook al volgen er alleen maar nullen achter de komma; bij `%g` of `%G` worden overbodige nullen na de komma toch afgedrukt.

  `0` om aan te geven dat de uitvoer links moet aangevuld worden met nullen in plaats van spaties.

  `-` de uitvoer moet links uitgelijnd worden. Als er ook een `0` was opgegeven, wordt er toch met spaties aangevuld.

  een spatie geeft aan dat positieve getallen een spatie vooraan moeten krijgen (tenzij je ook een `+` schreef)

  `+` geeft aan dat positieve getallen een `+` teken vooraan moeten krijgen

- eventueel een getal dat de minimumbreedte aangeeft. Er wordt indien nodig links of rechts aangevuld met spaties of nullen.

- eventueel een punt gevolgd door een getal (de *precisie*), om het aantal te gebruiken decimalen op te geven. Als de precisie nul is, mag je ook alleen een punt schrijven.

- eventueel de letter `h` die aangeeft dat bij een `%d`, `%i`, `%o`, `%u`, `%x` of `%X` het argument een `short int` of een `unsigned short int` is; of bij een `%n` een pointer naar een `short int`

- eventueel de letter `l` die aangeeft dat bij een `%d`, `%i`, `%o`, `%u`, `%x` of `%X` het argument een `long int` of een `unsigned long int` is; of bij een `%n` een pointer naar een `long int`

- eventueel de letter `L` die aangeeft dat bij een `%e`, `%E`, `%f`, `%g` of `%G` het argument een `long double` is

- een letter die het type van het argument aanduidt en de manier waarop dat afgedrukt moet worden:

  `diouxX` het `int` argument wordt afgedrukt als signed decimaal, (`d` of `i`), unsigned octaal (`o`, unsigned decimaal (`u`), of hexadecimaal (`x` of `X` al naar gelang je de hexadecimale cijfers als `a` t/m `f` of `A` t/m `F` wil). Als de precisie opgegeven is, wordt eventueel links aangevuld met nullen.

  `eE` het `double` argument wordt in wetenschappelijke notatie. Als de precisie ontbreekt, worden zes cijfers na de komma afgedrukt. Het resultaat ziet er als `0.000000e00` of `0.000000E00` uit. De exponent is altijd minstens twee cijfers lang.

  `f` het `double` argument wordt als kommagetal afgedrukt. Als je geen precisie hebt opgegeven, worden zes cijfers na de komma afgedrukt. Het resultaat ziet er als `0.000000` uit. Als de precisie 0 is, wordt geen decimale punt afgedrukt.

- `gG` het `double` argument wordt op de kortste van beide vorige manieren afgedrukt. Overtollige nullen achter de komma (en eventueel ook het decimale punt zelf) worden verwijderd.
- `c` het `int` argument wordt naar `unsigned char` omgezet, en de overeenkomstige letter wordt afgedrukt.
- `s` het `char *` argument wordt afgedrukt. Als er een precisie opgegeven was, bepaalt die het maximaal aantal af te drukken letters (in dat geval moet de string niet noodzakelijk door een nulbyte afgesloten worden).
- `p` Het `void *` argument wordt afgedrukt als pointer. De manier waarop de waarde van een pointer precies voorgesteld wordt, is afhankelijk van het gebruikte systeem.
- `n` Er wordt niets afgedrukt; het argument moet een pointer naar een `int` zijn. `printf()` vult in die `int` het aantal letters in dat het op dat ogenblik afgedrukt heeft.
- `%` Er wordt een `%` afgedrukt; er is geen argument nodig.

Het resultaat van `printf()` is een `int` die aangeeft hoeveel karakters er afgedrukt waren. Varianten op `printf()` zijn `sprintf()` en `fprintf()` die als eerste argument respectievelijk een string (`char *`) of een stroom (`FILE *`) hebben, gevolgd door het formaatargument en de andere argumenten. De uitvoer van deze functies wordt in de aangegeven string bewaard of naar de stroom geschreven. Een manier om een `int` in stringvorm om te zetten is dus:

```
{
        char str[20];
        int getal;

        sprintf(str,"%d",getal);
}
```

5.4 Functies voor bestanden

5.4.1 fopen() en fclose()

Naast `stdin`, `stdout` en `stderr`, die de C compiler automatisch ter beschikking stelt, kan je zelf ook naar hartelust bestanden lezen en schrijven. Om in een bestand te kunnen lezen of schrijven, moet je het eerst openen met `fopen()`. Je krijgt dan een `FILE *` terug, waarmee je allerlei bewerkingen op het bestand kunt uitvoeren. Na gebruik moet je het bestand weer sluiten met `fclose()`. Het type van alle stroomvariabelen in C (en dus ook van `stdin`, `stdout` en `stderr`) is `FILE *`. Dit type wordt in `stdio.h` gedefinieerd; hoe het er precies uitziet hangt af van systeem tot systeem en interesseert ons bijgevolg niet. De prototypes van `fopen()` en `fclose()` zijn:

```
FILE *fopen(char *naam,char *type);
int fclose(FILE *stroom);
```

De functie `fopen()` opent een bestand met een bepaalde `naam` en koppelt dat aan een stroom. Het functieresultaat is die stroom, of `NULL` als het openen niet gelukt is (bijvoorbeeld omdat er geen bestand met de opgegeven `naam` bestaat). Het `type` bevat een string die aangeeft wat voor soort stroom we aan dat bestand willen koppelen:

- `r` Open een bestaand bestand om te lezen. Het lezen begint aan het begin van het bestand; we zeggen ook wel kortweg dat de **positie van de stroom** het begin van het bestand is.
- `r+` Open een bestaand bestand om te lezen en te schrijven. Ook hier wordt de stroom gepositioneerd aan het begin van het bestand.
- `w` Maak een bestand om in te schrijven. Als er al een bestand bestond met de opgegeven `naam`, dan wordt dat eerst vernietigd.
- `w+` Open een nieuw bestand om te lezen en te schrijven. Als er al een bestand bestond met de opgegeven `naam`, dan wordt dat eerst vernietigd.
- `a` of `a+` Open een bestaand bestand om er achteraan gegevens aan toe te voegen (append). De positie van de stroom is dus het einde van het bestand. Als het bestand nog niet bestond, dan wordt het aangemaakt.

In de `type` string mag ook de letter `b` voorkomen, die aangeeft dat we niet met tekstbestanden, maar met binaire bestanden willen werken. Op sommige systemen ondergaan tekstbestanden eerst nog enkele filterbewerkingen vooraleer het C-programma ze ziet. Op de meeste UNIX systemen is er helemaal geen onderscheid tussen binaire of tekstbestanden. Daarentegen bewaren MS-DOS systemen een `'\n'` merkwaardig genoeg niet als één karakter, maar als twee (namelijk een CR gevolgd door een LF). Als je geen `b` in het `type` schrijft, wordt bij het lezen die twee bytes in één enkele `'\n'` omgezet, en als het programma een `'\n'` schrijft, worden er twee bytes van gemaakt. Het totale effect hiervan is dat alles er voor het C programma uitziet alsof einde-regel tekens als één byte bewaard worden.

Als je klaar bent met het gebruiken van een bestand, moet je het weer netjes sluiten met `fclose()`. Het functieresultaat is 0 als het sluiten is gelukt of `EOF` als er iets fout ging, bijvoorbeeld als je een stroom probeert te sluiten die je al een keer gesloten had. In de meeste programma's wordt er niet echt gecontroleerd of het sluiten lukte, omdat er meestal toch weinig aan te doen is.

Op het einde van het programma worden alle open stromen automatisch gesloten. Het is netter om dat expliciet zelf te doen. Bovendien is er meestal een bovengrens aan het maximaal aantal bestanden dat een programma tegelijkertijd kan openen; het is dus zaak bestanden zo snel mogelijk te sluiten wanneer je programma ze niet meer gebruikt.

5.5 Algemene in- en uitvoer

Vanzelfsprekend zal je wel wat meer willen aanvangen met bestanden dan openen en sluiten. Alle in- en uitvoerfuncties die met `stdin` en `stdout` werken, hebben een tegenhanger die op een willekeurige stroom werkt. Die functies zal ik nu bespreken.

5.5.1 In- en uitvoer van strings

```
int fputs(const char *str,FILE *stroom);
char *fgets(char *str,int n,FILE *stroom);
```

Zoals je al kon verwachten schrijft `fputs()` de opgegeven string naar een `stroom`, maar in tegenstelling tot `puts()` wordt geen nieuwe-regel teken toegevoegd (de afsluitende nulbyte van de string wordt natuurlijk ook niet geschreven). `fputs()` geeft een getal groter dan of gelijk aan nul terug als er geen fouten waren, of `EOF` als er wel iets mis ging.

`fgets()` leest hoogstens `n` bytes uit een `stroom` en plaatst die in de opgegeven `string` (natuurlijk samen met een afsluitende nulbyte). Het lezen stopt bij een nieuwe-regel teken of aan het einde van het bestand. Het resultaat is `str` als het lezen lukte, anders `NULL` (bijvoorbeeld aan het einde van een bestand). Een manier om een bestand regel per regel te verwerken is dus

```
char regel[100];

while( fgets(regel,sizeof(regel),bestand) ) {
        /* doe iets met de inhoud van  regel  */
}
```

Merk op dat, in tegenstelling tot `gets()`, je bij `fgets()` kan aangeven hoeveel bytes er maximaal gelezen moeten worden. Je kan `fgets()` dus als een 'veilige `gets()`' gebruiken door als laatste argument `stdin` mee te geven. Dit is inderdaad veiliger omdat je met `gets()` op voorhand nooit kan weten hoeveel bytes er zullen binnen komen en je dus niet weet hoe groot je de array moet maken waarin de gelezen regel terecht zal komen.

▸ **OEFENING 5.2**
Schrijf een zoek-en-vervang programma dat een bestand leest uit de standaard-invoer en datzelfde bestand naar de standaard-uitvoer schrijft, maar waarbij elk voorkomen van een bepaald (deel van een) woord door een ander woord vervangen wordt. Deze twee woorden worden als argumenten aan het programma meegegeven. Voor het gemak vervangen we ook binnen woorden, dus als we het programma `zv` dopen en

```
zv rood groen
```

typen, dan zal het ook `roodkapje` in `groenkapje` vervangen.

♢

5.5.2 In- en uitvoer van karakters

```
int fgetc(FILE *stroom);
int getc(FILE *stroom);
int fputc(int c,FILE *stroom);
int putc(int c,FILE *stroom);
int ungetc(int c,FILE *stroom);
```

`fgetc()` en `getc()` lezen een letter uit een stroom. Als het einde van de stroom bereikt is, of wanneer er iets fout ging, krijg je `EOF` terug (net als bij `getchar()` dus). Het verschil tussen beide is dat `getc()` een macro is, zodat het argument `stroom` verschillende keren kan voorkomen in de expansie (zie hoofdstuk §6). Als het argument `stroom` eenvoudig is (en het dus geen kwaad kan het meerdere malen te evalueren), is `getc()` te verkiezen omdat het iets sneller is. Waar het zou kunnen mis gaan is in situaties als `getc(stromentabel[i++])`: bij het expanderen van de macro zal `i++` meer dan één keer geëvalueerd worden, zodat `i` nogal onvoorspelbare waarden kan aannemen.

Met `fputc()` en `putc()` kan je een letter naar een stroom schrijven. Ook hier is `putc()` een macro; verder zijn beide functies gelijkwaardig.

Erg handig is de functie `ungetc()` waarmee je een letter kunt "ont-lezen": bij de volgende aanroep van `fgetc()` zal geen nieuwe letter uit de stroom gelezen worden, maar krijg je het argument van `ungetc()` terug. Dat is vooral handig bij functies die pas weten dat ze mogen ophouden met lezen als het al te laat is, bijvoorbeeld een functie die een geheel getal inleest:

```
long leesgetal(FILE *stroom)
{
        char buffer[100];
        int gelezen = 0;    /* aantal gelezen cijfers */

        while (buffer[gelezen] = getc(stroom)) {
                if (!isdigit(buffer[gelezen])) {
                        /* We hebben iets gelezen dat geen cijfer is.
                         * Het getal is dus ten einde, maar we
                         * hebben nu wel een letter te veel gelezen! */
                        ungetc(buffer[gelezen],stroom);
                        break;
                }
                gelezen = gelezen + 1;
        }
        /* string afsluiten met een nulbyte */
        buffer[gelezen] = 0;
        return atol(buffer);
}
```

Ik heb hier de `isdigit()` functie uit `ctype.h` gebruikt om te zien of een letter een cijfer is.

▸ **OEFENING 5.3**
Maak een eenvoudig vertaalprogramma. Het programma maakt gebruik van een bestand `woordenlijst` dat er bijvoorbeeld als volgt uit ziet:

```
de      the
het     the
een     a
dag     day
snoepgoed       candy
met     with
```

waarbij zoals je ziet op elke regel een Nederlands woord staat, gevolgd door een tab en de Engelse vertaling ervan. Het programma leest een tekst uit de standaard-invoer en schrijft de woord-voor-woord vertaling ervan uit op de standaard-uitvoer. Hou rekening met hoofd- en kleine letters: als in de originele tekst “Grote” staat, moet de uitvoer “Big” worden; “GROTE” moet “BIG” worden. Je kan het programma gebruiken op bestanden met redirectie (`vertaal < nederl.txt > engels.txt`) of interactief (`vertaal` zonder meer). In het laatste geval wordt telkens je op Enter duwt een regel vertaald, omdat de standaard-invoer standaard per regel gebufferd wordt:

```
Op een mooie dag wandelde Roodkapje door het bos
Op a mooie day wandelde Roodkapje door the bos
met een mandje snoepgoed ...
met a mandje candy ...
```

◇

5.5.3 Blok in- en uitvoer

Om een hele hoop bytes ineens te lezen of te schrijven hebben we de volgende twee functies:

```
size_t fread(void *buffer,size_t grootte,size_t aantal,FILE *stroom);
size_t fwrite(void *buffer,size_t grootte,size_t aantal,FILE *stroom);
```

Het type `size_t` is o.a. gedefinieerd in `stdio.h` en is een of ander unsigned geheel type. Het eerste argument, `buffer`, wijst naar de eerste van de te schrijven bytes bij `fwrite()` of naar de plaats van waaraf `fread()` de ingelezen bytes mag bewaren. Er worden `aantal` elementen van elk `grootte` bytes gelezen of geschreven (wat neerkomt op `grootte*aantal` bytes).

Het resultaat van beide functies is het aantal succesvol gelezen of geschreven elementen (dus niet het aantal bytes, tenzij `grootte` gelijk is aan 1 natuurlijk). Als dat minder is dan `aantal`, is er dus ergens een fout opgetreden (of `fread()` is op het einde van het bestand gebotst).

5.5.4 `fseek()`, `ftell()` en `rewind()`

Met deze functies kan je navigeren door lees/schrijf-stromen.

```
int fseek(FILE *stroom,long positie,int vanwaar);
long ftell(FILE *stroom);
void rewind(FILE *stroom);
```

Met `fseek()` kan je naar een bepaalde positie in het bestand "doorspoelen". De volgende lees- of schrijfoperatie zal dan vanaf die positie beginnen. Het `vanwaar` argument geeft aan op welke manier `positie` moet geïnterpreteerd worden:

- `SEEK_SET` betekent dat `positie` de positie vanaf het begin van het bestand aangeeft.
- `SEEK_CUR` betekent dat `positie` aangeeft hoeveel bytes er moet doorgespoeld worden ten opzichte van de huidige positie. `positie` kan ook negatief zijn om terug te spoelen.
- `SEEK_END` betekent dat `positie` het aantal bytes aangeeft vanaf het einde van het bestand.

Als de `fseek()` gelukt is, krijg je een nul terug, anders -1.

De functie `ftell()` geeft de positie in het bestand aan ten opzichte van het begin van het bestand, waarbij `0` aangeeft dat je aan het begin van het bestand staat.

Met `rewind()` ga je terug naar het begin van het bestand; het doet net hetzelfde als `fseek(stroom,0,SEEK_SET); clearerr(stroom);`.

Er is geen functie om de lengte van een bestand op te vragen. De enige oplossing is naar het einde van het bestand gaan en dan met `ftell()` kijken op welke positie je zit:

```
long flen(FILE *stroom)
{
        long positie = ftell(stroom);    /* originele positie onthouden */
        long lengte;

        fseek(stroom,0,SEEK_END);
        lengte = ftell(stroom);
        /* nu nog netjes terugkeren naar de originele positie */
        fseek(stroom,positie,SEEK_SET);
        return lengte;
}
```

5.5.5 Foutopvang in `stdio.h`

De functies uit `stdio.h` signaleren fouten meestal door een of andere speciale waarde terug te geven (zoals `EOF`). Om te zien welke fout er precies opgetreden is bevat `stdio.h` de volgende functies:

```
void clearerr(FILE *stroom);
int feof(FILE *stroom);
int ferror(FILE *stroom);
```

Van elke stroom wordt bijgehouden of het einde van het bestand bereikt is, en of er fouten opgetreden zijn.

De functie `clearerr()` zet de einde-bestand-indicatie en de fout-indicatie van de `stroom` uit. Dit is de enige manier om de indicatoren uit te zetten; de enige manier waardoor ze aan kunnen geraken is als er een fout is opgetreden.

Met `feof()` en `ferror()` kan je de toestand van beide indicatoren opvragen. Het resultaat is nul als de indicator uit staat, of verschillend van nul als die aan staat.

5.5.6 Pipes

Er zijn twee handige functies die een ander programma op te starten en er een stroom aan koppelen:

```
FILE *popen(const char *programma,const char *type);
int pclose(FILE *stroom);
```

Ze werken net als `fopen()` en `fclose()`. Het enige verschil is dat het `type` alleen maar `r` of `w` mag zijn. Als het type `w` is, wordt alles wat je in de stroom schrijft, naar de standaard invoer van het opgestarte `programma` omgeleid; als het type `r` is, kan je alle standaarduitvoer van het `programma` uit de stroom lezen.

Het resultaat van `popen()` gedraagt zich verder als een normale stroom, behalve dat je hem moet afsluiten met `pclose()` (en *niet* met `fclose()`). Het resultaat van `pclose()` is -1 als de `stroom` niet geopend was met `popen()` of als je de stroom al gesloten had met `pclose()`.

Het nadeel van deze functies is dat je niet tegelijkertijd de in- en uitvoer van een programma kunt onderscheppen.

5.6 Directories

5.6.1 De huidige werkdirectory

Elk programma heeft een **huidige werkdirectory**. Als het programma een bestand wil lezen of schrijven, zeg `probeersel.txt`, dan wordt er gezocht in die huidige werkdirectory. Meestal is er ook wel een manier voorzien in het besturingssysteem om de plaats van een bestand aan te geven zonder dat de huidige werkdirectory er aan te pas komt (bijvoorbeeld `/home/geert/probeersel.txt` onder Unix of `c:\ergens\probeersel.txt` onder dos). De eerste vorm wordt een **relatief pad** genoemd, de andere een **absoluut pad**.

In `unistd.h` leven enkele functies die hiermee te maken hebben:

```
int chdir(const char *path);
char *getcwd(char *buf,size_t size);
```

Met `chdir()` verander je de huidige werkdirectory. Het `path` kan relatief of absoluut zijn. Het resultaat van `chdir()` is 0 als alles goed ging.

Om te kijken wat de huidige werkdirectory is, kan je `getcwd()` gebruiken. Je geeft een stukje geheugen en de lengte ervan op. Als het absolute pad van de huidige werkdirectory erin past, vult `getcwd()` het pad in en het resultaat is `buf`. Als het stukje geheugen te klein bleek, krijg je `NULL` terug.

5.7 Truukwerk: letter-per-letter invoer

Jammer genoeg heeft de C standaardbibliotheek geen functie om de `stdin` invoer letter per letter te laten binnenkomen. Bovendien verschijnt ingetikte invoer vanzelf op het scherm (**echo**). Daarom geef ik nu enkele functies `toets.c` om in Unix echoloze letter-per-letter invoer te doen:

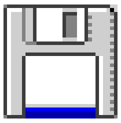
toets.c

```
#include <stdio.h>
#include <fcntl.h>
#include <termio.h>
#include <termios.h>
#include <unistd.h>
#include <sys/ioctl.h>

#include "toets.h"

static struct termio tbufsave;
static int flags;

void initkey(void)
{
        struct termio tbuf;

        /* Als we een letter lezen, en er is er geen beschikbaar
         * (omdat de gebruiker nog niet heeft getikt), dan
         * wacht read() als O_NDELAY uit staat, of geeft 0
         * terug als O_NDELAY aan staat */
        flags=fcntl(0,F_GETFL,0);
        fcntl(0,F_SETFL,flags | O_NDELAY);        /* O_NDELAY aanzetten */

        /* Stel de buffergrootte in op 1 byte. */
        ioctl(0,TCGETA,&tbuf);                    /* Instellingen lezen */
        tbufsave=tbuf;
        /* if you want the key echoed, just remove |ECHO on next line */
        tbuf.c_lflag &= ~(ICANON|ECHO/*|ISIG*/);  /* wis ICANON,ECHO,... */
        tbuf.c_cc[VMIN] = 1;      /* Geef bytes van zodra er 1 klaar staat  */
        ioctl(0,TCSETAF,&tbuf);                   /* Instellingen instellen */
}

void deinitkey(void)
{
        ioctl(0,TCSETA,&tbufsave);
        fcntl(0,F_SETFL,flags);
```

```
}

/* Lees een letter, of geef -2 terug als er geen beschikbaar was */
int readkey(void)
{
        static char buf = 0;

        switch(read(0,&buf,1)) {
                case -1: return -1;       /* ERROR */
                case 0:  return -2;        /* No key */
                default: return buf;
        }
}

/* Lees een letter. Blijf desnoods wachten tot er een komt opdagen. */
int mustreadkey(void)
{
        int key;

        fcntl(0,F_SETFL,fcntl(0,F_GETFL,0) && ~O_NDELAY); /* O_NDELAY uitzetten */
        key=readkey() ;                                   /* Wait for key */
        fcntl(0,F_SETFL,fcntl(0,F_GETFL,0) | O_NDELAY);   /* O_NDELAY aanzetten */
        return key;
}
```

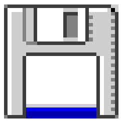
Deze functies maken gebruik van Unix-specifieke internalia; de precieze werking heeft niet zoveel belang. Wat wel van belang is het gebruik van deze functies. `initkey()` schakelt de lijnbuffering van het toetsenbord uit en `deinitkey()` schakelt ze weer in. Met `readkey()` kan je een karakter lezen of kijken of er al een karakter klaar staat. De functie `mustreadkey()` blijft netjes wachten tot er een letter ingetikt wordt en geeft die als functieresultaat terug.

De header file `toets.h` ziet er dan zo uit:


toets.h

```
void initkey(void);
void deinitkey(void);
int readkey(void);
int mustreadkey(void);
```

Een kleine demonstratie:


toetstest.c

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

int main(void)
{
        initkey();
        /* Lees letters totdat op spatie wordt gedrukt */
        while (1) {
                int toets = readkey();

                if (toets == ' ') break;          /* stop bij spatie */
```

```
                else if (toets < 0) putchar('.'); /* niets gedrukt */
                else putchar(toets); /* druk ingetikte letter af */
        }
        /* Wacht op een toets */
        puts("\nDruk op een toets ...");
        mustreadkey();
        deinitkey();
        return 0;
}
```

Hoofdstuk 6

De preprocessor

De preprocessor leest een C programmatekst regel voor regel in en verwerkt alle regels die met een `#` beginnen. Zulke regels heten **preprocessor-aanwijzing**en (**preprocessing directive**s). We hebben al van de preprocessor gebruik gemaakt om eenvoudige macro's te maken. De preprocessor heeft nog wat meer in zijn mars, zoals uit het vervolg zal blijken.

Om beter te zien wat de preprocessor precies doet, kan je alleen de preprocessor laten draaien, zonder dat het programma gecompileerd wordt. Met de GNU C compiler kan dat door de `-E` optie te gebruiken:

```
gcc hello.c -E
```

zorgt ervoor dat de uitvoer van de preprocessor op het scherm verschijnt. (Met de `-o` optie kan je de uitvoer in een bestand bewaren.)

6.1 Overzicht van de werking van de preprocessor

De preprocessor doet de volgende dingen, één voor één, en in deze volgorde:

- De preprocessor begint met alle commentaren te vervangen door een spatie.
- Daarna komt de **trigraph expansie**. Omdat sommige tekens niet op alle toetsenborden makkelijk bereikbaar zijn, kan je ze met de volgende zgn. trigraphs (rijtjes van drie tekens, beginnend met `??`) nabootsen:

  ```
  ??=  #      ??(  [      ??<  {
  ??/  \      ??)  ]      ??>  }
  ??’  ‘      ??!  |      ??-  ˜
  ```

  Om in een string toch ergens letterlijk `??!` te kunnen schrijven, mag je in strings `\?` schrijven voor een vraagteken. Dus blijft de preprocessor van deze regel af:

  ```
  puts("De trigraph voor [ is ?\?(.");
  ```

  OPMERKING: de preprocessor doet dus niets met de backslashes in string- of `char`-constanten. Het is de C compiler zelf die deze backslash-reeksen interpreteert.

  De GNU C compiler houdt niet zo van trigraphs; om ze in te schakelen moet je expliciet de optie `-ansi` of `-trigraphs` gebruiken. Als je ook nog `-Wtrigraphs` gebruikt, geeft de compiler een waarschuwing als hij een trigraph tegenkomt (opdat je kan controleren of er toevallig geen onbedoelde trigraphs geslopen zijn in je programma, zodat het programma ook goed compileert op compilers die wel trigraphs ondersteunen).

- Vervolgens plakt de preprocessor regels die op een backslash eindigen aan elkaar. Zo maakt de preprocessor bijvoorbeeld van

  ```
  int main(void) \
  { \
          puts("Dag grootmoeder!");\
  }
  ```

  de volgende regel:

  ```
  int main(void) {        puts("Dag grootmoeder!");}
  ```

  Een nuttiger toepassing hiervan is het aaneenplakken van lange string-constanten:

  ```
          printf("Deze tekst is veeeel te lang om op een regel te \
  passen. Daarom gebruik ik een \\ op het einde van de re\
  gel. Let op het gebruik van spaties vlak voor de backslash.");
  ```

  Sinds de C compilers strings die naast elkaar staan zelf aaneenplakken, is deze toepassing een beetje in onbruik geraakt:

  ```
          printf("Al dat geknoei met backslashes en dat gepruts met "
                  "spaties is verleden tijd. Ik kan nu het vervolg van "
                  "de string zonder problemen netjes indenteren.");
  ```

  Maar het is wel nuttig om lange macro's te maken:

  ```
  /* Macro om een eventuele \n die achteraan een string staat
   * te verwijderen
   */
  #define ZapCR(str)   if (str[0] != 0       \
          && str[strlen(str)-1] == '\n')     \
          str[strlen(str)-1] = 0
  ```

- De preprocessor-aanwijzingen worden uitgevoerd. De regel waarop een preprocessor-aanwijzing staat, moet beginnen met een #. Er mag witte ruimte rond het #-teken staan. Op die manier kan je geneste preprocessor-aanwijzingen indenteren (het is de gewoonte de # helemaal links te blijven schrijven). De verschillende preprocessor-aanwijzingen zal ik in de rest van dit hoofdstuk bespreken. Een preprocessor-aanwijzing mag uit maar één regel bestaan (je kan regels zo lang maken als je wil door twee regels aaneen te plakken met een backslash, zoals hierboven uitgelegd).

6.2 de lege preprocessor-aanwijzing

Een regel waarop niets anders staat dan een # (eventueel omgeven door witte ruimte), bevat de zgn. lege preprocessor-aanwijzing en doet helemaal niets.

6.3 #include

De preprocessor vervangt de regel waarop `#include` staat door de volledige inhoud van een bestand, dat zelf weer door de preprocessor verwerkt wordt (en dus op zijn beurt `#include`s en zo kan bevatten).

Er zijn twee vormen van `#include`. De eerste heb je al dikwijls zien opduiken:

#include <*bestandsnaam*>

De preprocessor zoekt het bestand hier in een aantal vast ingestelde systeemdirectories. Je moet dat bestand dus niet zelf leveren.

De tweede vorm is

#include "*bestandsnaam*"

Hier gaat de preprocessor eerst zoeken in de huidige directory. Als het bestand daar niet gevonden wordt, zoekt de preprocessor het bestand in de systeemdirectories. Nuttige toepassingen van de tweede vorm zullen we in hoofdstuk 9 zien.

Het is de gewoonte de *bestandsnaam* op `.h` (van "**header file**") te doen eindigen.

Als hetgeen na `#include` niet begint met `<` of `"`, wordt de macro-expansie (zie §6.4) uitgevoerd. Het resultaat hiervan moet dan van één van de twee toegelaten vormen zijn.

6.4 #define en #undef: Macro's

Zoals je al weet uit §2.5 zorgt

```
#define MACRONAAM vervangtekst
```

ervoor dat vanaf die plaats in het programma het losstaande woord `MACRONAAM` door de vervangtekst wordt vervangen. Dat vervangen noemen we ook wel **macro-expansie**.

Met

```
#undef MACRONAAM
```

maak je de definitie van een macro ongedaan. Het blijkt geen kwaad te kunnen een macro meerdere keren te `#undef`en, of een macro te `#undef`en die nog nooit gedefinieerd was.

Als er al een macro met dezelfde naam gedefinieerd was, dan vervalt de oude definitie en krijg je een waarschuwing (netter is eerst de macro expliciet te `#undef`en, zodat het duidelijk is dat je de macro wil herdefiniëren).

Macro-expansie gebeurt niet in preprocessor-aanwijzingen zelf. Dat blijkt uit het volgende stukje code:

```
#include <stdio.h>
#define TEKST        "Hallo!"
#define DRUK_TEKST   printf(TEKST)

int main(void)
{
        DRUK_TEKST;       /* Drukt "Hallo!" af */
#undef TEKST
        /* de volgende regel wordt als
```

```
         * printf(TEKST);
         * geexpandeerd. We hebben geen variabele TEKST
         * gemaakt, dus geeft de compiler een foutmelding.
         */
        DRUK_TEKST;
#define TEKST("Test");
        DRUK_TEKST;       /* Drukt "Test" af */
}
```

De vervangtekst van `DRUK_TEKST` is dus niet `printf("Hallo!")` maar letterlijk `printf(TEKST)`.

6.4.1 Macro's met parameters

De preprocessor kan ook een iets gesofistikeerder soort "zoek en vervang" aan. Bijvoorbeeld:

```
#define KWADRAAT(x) x*x
#define GEMIDDELDE(x,y) (x+y)/2
```

Als er nu verderop in het C programma ergens `KWADRAAT(4)` staat, maakt de preprocessor er `4*4` van.

Let wel: als je `#define KWADRAAT (x) x*x` schrijft, interpreteert de preprocessor dat als "vervang overal `KWADRAAT` door `(x) x*x`", dus het is van vitaal belang geen spatie na `KWADRAAT` te schrijven. In het programma zelf echter zal `KWADRAAT (4)` net zo goed vervangen worden door `4*4` (`KWADRAAT(4)` is natuurlijk ook in orde).

Iets om voor op te letten is dat de preprocessor geen flauw benul heeft van de syntax van C (zoals eerder al gezegd). Als je bijvoorbeeld `KWADRAAT(x+y)` zou schrijven, maakt de preprocessor daar `x+y*x+y` van, wat hetzelfde is als `x+(y*x)+y` omdat vermenigvuldigingen vóór optellingen worden uitgevoerd. Daarom is het verstandig veel haakjes op te nemen in een `#define` (een programma wordt nooit trager door er extra haakjes in te schrijven; hoogstens duurt het compileren een paar microseconden langer):

```
#define KWADRAAT(x) ((x)*(x))
#define GEMIDDELDE(x,y) (((x)+(y))/2)
```

▸ **OEFENING 6.1**
Bekijk alle haakjesparen die ik toegevoegd heb in de definities van `KWADRAAT` en `GEMIDDELDE`. Laat elk haakjespaar eens om de beurt weg, en vind een voorbeeld waaruit blijkt dat het haakjespaar wel degelijk nodig is.

♢

▸ **OEFENING 6.2**
In §3.3.3 heb ik de functie `void wissel(int *a,int *b)` gemaakt, die je op een vervelende manier moest gebruiken, namelijk met een hoop ampersands erin: `wissel(&a,&b);`. Schrijf een macrootje waardoor die ampersands niet meer nodig zijn.

♢

Zoals je ziet schrijf ik alle ge`#define`de woorden in hoofdletters. Dat is niet verplicht, maar het is een algemeen aanvaarde afspraak. Op die manier kan je ook duidelijk in het programma zelf zien dat het om een macro gaat.

We kunnen dus het cirkelprogramma van §2.4.3 herwerken tot

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

#define SCHERM_BREED 80
#define SCHERM_HOOG 24
#define STRAAL 10
#define STAP 0.01
#define PI 3.141592

#define gotoxy(x,y) printf("\033[%d;%dH",y,x)

void main(void)
{
        double teller;

        teller = 0.0;
        while (teller <= 2*PI) {
                /* Cursor naar de juiste plek */
                gotoxy( SCHERM_BREED/2 + (int)(STRAAL*cos(teller)),
                        SCHERM_HOOG/2 + (int)(STRAAL*sin(teller)) );
                printf("*");
                teller += STAP;
        }
}
```

Ik had natuurlijk net zo goed een *functie* `gotoxy()` kunnen maken:

```
void gotoxy(int x,int y)
{
        printf("\033[%d;%dH",y,x);
}
```

Het nadeel hiervan is dat er een extra functie-aanroep moet plaatsvinden. Functie-aanroepen zijn "duurder" in geheugen en uitvoeringstijd: het programma moet weten van waar het komt (m.a.w. naar waar het moet terugkeren op het einde van `gotoxy()`) en die administratie kost geheugen en tijd.

`#undef` werkt ook op macro's met parameters, bijvoorbeeld: `#undef gotoxy`.

6.4.2 De komma-operator in macro's

Een macro heeft altijd een vast aantal argumenten. Dat kan soms nogal vervelend zijn, bijvoorbeeld als je een printf-achtige macro wil maken in de trant van

```
waarsch("probeer %d bytes naar %s te schrijven",aant,naam);
```

die als effect zou moeten hebben

```
printf("***");
printf("probeer %d bytes naar %s te schrijven",aant,naam);
putchar('\n');
```

Een lepe truuk is ervoor te zorgen dat de macro maar één argument heeft en dus zo gebruikt zal worden:

```
waarsch(("probeer %d bytes naar %s te schrijven",aant,naam));
```

Deze macro heeft inderdaad maar één argument, namelijk

```
("probeer %d bytes naar %s te schrijven",aant,naam)
```

De komma's die in het argument staan, zijn komma-operatoren. Maar als we nu de macro `waarsch()` definiëren als

```
#define waarsch(x) printf("***");    \
                   printf x;         \
                   putchar('\n')
```

dan worden de haakjes van het argument ineens de haakjes van een functieaanroep-operator, en de komma's zijn dus geen komma-operatoren meer, maar scheiden gewoon de argumenten van elkaar.

Nadeel van deze aanpak is dat je de macro op een nogal vreemde manier moet gebruiken (een dubbel haakjespaar rond de argumenten) en ook dat we geen extra argument aan de lijst kunnen toevoegen. Het is dus niet mogelijk om in plaats van de uitvoer naar het scherm te sturen met `printf x`, de uitvoer naar een bestand te sturen met `fprintf(meldingen,x)` of zo, omdat de haakjes rond `x` problemen geven.

6.4.3 Voorgedefinieerde symbolen

De preprocessor maakt zelf een aantal macro's aan:

`__STDC__` is gedefinieerd als `1` als de compiler de C standaard ondersteunt. Deze macro wordt gebruikt om oude compilers te detecteren.

`__LINE__` Bevat het regelnummer van de huidige regel in de broncode.

`__FILE__` Bevat de naam van het huidige `.c` of `.h` bestand als string. De laatste twee macro's zijn bruikbaar om fouten in een programma helpen op te sporen, door het regelnummer van de plaats waarop de fout optrad af te drukken. `assert()` (zie §10.1) maakt intern gebruik van `__LINE__` en `__FILE__`.

`__DATE__` Bevat de huidige datum, in de vorm `mmm dd jjjj`.

`__TIME__` Bevat de huidige tijd, in de vorm `uu:mm:ss`

`__cplusplus` Is `1` als de compiler C++ compileert. Je kan op die manier een functiebibliotheek maken die zowel onder C als C++ bruikbaar is. Het gros van de code zal wel voor C en C++ bruikbaar zijn, en waar nodig kan je aparte versies maken met

```
#ifdef __cplusplus
        /* C++ versie */
#else
        /* C versie */
#endif
```

De uitleg van `#ifdef`, `#else` en `#endif` komt in §6.5.

GNU C onder Linux definieert ook nog `linux`, `__unix__`, `__i386__`, `__linux__`, `__unix`, `__i386` en `__linux`. Als je niet wil dat één van deze symbolen gedefinieerd is, kan je de `-U`*symbool* optie gebruiken, bijvoorbeeld `gcc -U__unix__ bozewolf.c -o bozewolf`.

6.4.4 De pasting- en quoting-operatoren ## en

Tijdens de macro-expansie worden de # en ## operatoren uitgevoerd. Deze operatoren mogen alleen in de vervangtekst van een macro gebruikt worden (dus niet in het midden van gewone programmatekst, waar ze overigens vrij nutteloos zijn, zoals we zullen zien).

De ## operator plakt de twee woorden die er rond staan aan elkaar. We kunnen deze operator in de volgende situatie gebruiken: de definitie

```
#define wissel(a,b)    {     int hulp;          \
                             hulp = a;          \
                             a = b;             \
                             b = hulp;   }
```

werkt goed, tenzij één van de argumenten zelf hulp heet: wissel(hulp,x) wordt dan geëxpandeerd als

```
        {        int hulp;
                 hulp = hulp;
                 hulp = x;
                 x = hulp;          }
```

Probleem is dat de int hulp; ervoor zorgt dat we binnen het blok niet meer aan de hulp van buiten het blok kunnen komen. Met de pasting operator kunnen we deze situatie netjes opvangen:

```
#define wissel(a,b)    {     int hulp ## a;          \
                             hulp ## a = a;          \
                             a = b;                  \
                             b = hulp ## a;   }
```

zodat wissel(hulp,x) nu de variabele hulphulp gebruikt.

▸ **OEFENING 6.3**
Wat kan hier misgaan? En schrijf natuurlijk een versie die helemaal foutvrij is.

◇

De # operator maakt een string van het woord dat (eventueel na wat witte ruimte) erop volgt:

```
#define DEBUGFUN(f,arg) puts("Probeer " #f "(" #arg ") ...");     \
                f(arg)
```

zorgt ervoor dat DEBUGFUN(sorteer,tabel) de tekst Probeer sorteer(tabel) ... op het scherm zet en sorteer(tabel) aanroept (handig om het verloop van een programma dat raar doet in het oog te houden).

Let wel: de # operator mag alleen inwerken op een macro-argument. Dus

```
#define INHOUD lekkers
#define MANDJE "een mandje met " # INHOUD
```

geeft een foutmelding bij de definitie van MANDJE; wat wel mag is

```
#define MANDJE(inhoud) "een mandje met " # inhoud
```

De preprocessor is slim genoeg om dubbele aanhalingstekens en backslashes goed te verwerken (dus # te\s"t wordt "te\\s\"t").

6.4.5 Macro-expansie

Om te vermijden dat macro's als

```
#define OEPS OEPS x
```

als effect zouden hebben dat `OEPS` geëxpandeerd zou worden tot `OEPS x`, wat weer expandeert tot `OEPS OEPS x`, tot `OEPS OEPS OEPS x` en zo verder totdat heel het geheugen van je computer vol zit met `OEPS`en, wordt een macronaam maar één keer in een expansie vervangen. Als je dus `OEPS` schrijft, wordt dat geëxpandeerd tot `OEPS x`, en de expansie stopt omdat `OEPS` al eens gebruikt is in de expansie.

We kunnen dat iets verder doortrekken tot

```
#define DIT DAT
#define DAT NOGIETS
#define NOGIETS DIT x
```

De macro `DIT` wordt als volgt geëxpandeerd:

$$\texttt{DIT} \longrightarrow \texttt{DAT} \longrightarrow \texttt{NOGIETS} \longrightarrow \texttt{DIT x}$$

en de expansie stopt omdat `DIT` al eens geëxpandeerd is.

6.5 Voorwaardelijke compilatie met #if, #else, #elif en #endif

De preprocessor kan bepaalde stukken code uit het programma wegfilteren, zodat de C compiler er niets van te zien krijgt. Dit wordt **voorwaardelijke compilatie** genoemd. De algemene syntax hiervoor is:

`#if` *preprocessoruitdrukking*
⋮ `/*` *Wordt gecompileerd als de uitrdukking waar is* `*/`
`#elif` *preprocessoruitdrukking*
⋮ `/*` *Het #elif-gedeelte is optioneel, en er mag ook meer dan één #elif voorkomen* `*/`
`#else` *preprocessoruitdrukking*
⋮ `/*` *het #else-gedeelte is ook optioneel.* `*/`
`#endif`

Een preprocessor-uitdrukking mag gehele constanten bevatten en C operatoren die op dat soort uitdrukkingen werken (je kan bijvoorbeeld de functieaanroepoperator `()` niet gebruiken, omdat het nogal moeilijk is een functie aan te roepen van een programma voordat de compilatie ervan gedaan is ...). Verder is er ook nog de `defined` *naam* operator (die je alleen maar in preprocessor-uitdrukkingen mag gebruiken), die `1` geeft als een macro met een bepaalde *naam* gedefinieerd is, en anders `0`. Bijvoorbeeld:

```
#if ! defined NULL
        /* NULL is nog niet gedefinieerd, dus vlug doen */
        #define NULL (void *) 0
#endif
```

Er zijn twee handige afkortingen voor vaak gebruikte constructies: `#ifdef naam` doet hetzelfde als `#if defined naam`, en `#ifndef naam` doet hetzelfde als `#if ! defined naam`.

Merk op dat een macro met lege vervangtekst ook als gedefinieerd wordt beschouwd:

```
#define LEEG
#ifdef LEEG
        /* dit wordt verwerkt */
        printf("De macro is gedefinieerd, maar leeg");
#else
        /* dit wordt niet verwerkt */
        ik kan hier schrijven wat ik wil ...
        de preprocessor eet het toch op
#endif
```

In de praktijk is voorwaardelijke compilatie handig om verschillende versies van een programma te maken. Je zou bijvoorbeeld een versie met en zonder debug-boodschappen kunnen maken:

```
#ifndef NDEBUG
        printf("Probeer bestand %s te openen",naam);
#endif
        f = fopen(naam,"r");
```

Die boodschappen kunnen handig zijn tijdens het testen van het programma, maar in de definitieve versie zijn ze minder welkom. Je kan omschakelen tussen beide versies door ergens bovenaan in het programma al dan niet een `#define NDEBUG` regel te schrijven. Een elegantere oplossing is van de broncode af te blijven en de compiler te vragen het symbool te definiëren. De meeste compilers hebben daar voorzieningen voor; bij de GNU C compiler kan je met de `-D`*symbool* optie vragen dat een bepaald symbool gedefinieerd moet worden. Je kan dus de versie zonder debug-boodschappen maken met `gcc -DNDEBUG prog.c -o prog`.

Ook handig is om meerdere de hoeveelheid gegenereerde debug-boodschappen te kunnen instellen, bijvoorbeeld door `DEBUGLEVEL` waarden te geven tussen `0` (geen debug-info uitsturen) en `100` (zoveel mogelijk debug-info genereren):

```
#if DEBUGLEVEL >= 20
        /* Een vrij belangrijke debug-melding */
        puts("Probeer harddisk te formatteren ...");
#endif

#if DEBUGLEVEL >= 90
        /* Deze boodschap wordt alleen getoond als er vrij veel
         * debug-info gevraagd wordt */
        puts("Optelling gelukt");
#endif
```

6.5.1 Idempotentie

Het is handig om een `.h` bestand meerdere keren te mogen `#include`n, zonder dat er rare neveneffecten optreden.

Een veelgebruikte methode is ervoor te zorgen dat de inhoud van het bestand alleen bij de eerste keer verwerkt wordt, en alles over te slaan als het bestand nog eens ge`#include`d wordt. Dat kan met de volgende constructie:

```
/*** bestand mijnprog.h ***/
#ifndef MIJNPROG_H
```

```
#define MIJNPROG_H

/* verdere inhoud van het bestand */

#endif
```

Bij de eerste `#include mijnprog.h` is de macro `MIJNPROG_H` niet gedefinieerd. Het bestand wordt dus volledig verwerkt, en de macro `MIJNPROG_H` wordt gedefinieerd, zodat bij een volgende `#include mijnprog.h` er eigenlijk niets meer gebeurt (alles in `mijnprog.h` wordt gewoon overgeslagen).

6.6 Pragma's

It is basically a mistake to use `#pragma` *for *anything**
—GNU C documentatie

De syntax van pragma-aanwijzingen is

```
#pragma pragmatekst
```

Pragma's worden gebruikt om compiler-afhankelijke opties in te stellen. Er zijn geen standaard pragma's. Als een compiler een `#pragma` tegenkomt die hij niet herkent, dan slaat hij die gewoon over. (De GNU C compiler gebruikt pragma's alleen in de C++ stand.)

Bovendien heeft het concept van pragma's twee belangrijke nadelen:

- Het is onmogelijk `#pragma` in een macro te gebruiken. Als je immers iets probeert als `#define PRAG #pragma blabla` dan wordt `PRAG` wel netjes geëxpandeerd tot `#pragma blabla`, maar de preprocessor zoekt niet meer naar preprocessoraanwijzingen in geëxpandeerde tekst, zodat `#pragma blabla` gewoon blijft staan.

- Een andere compiler kan een bepaalde `#pragma` misschien wel op een heel andere manier interpreteren dan je bedoeld had.

6.7 `#line`

De `#line` n aanwijzing zorgt ervoor dat de preprocessor de huidige regel als de n-de regel van het bestand beschouwt. Dit heeft effect op de `__LINE__` macro (zie §6.4.3) en op de verdere waarschuwingen en foutmeldingen die de compiler afdrukt. Je kan de preprocessor zelfs doen denken dat hij in een ander bestand zit met `#line` n *"bestandsnaam"*.

De `#line` preprocessoraanwijzing wordt vooral gebruikt door **programmageneratoren** (programma's die als uitvoer een C programma hebben).

6.8 `#error`

Als de preprocessor een `#error` aanwijzing tegenkomt, stopt het compileren met een optionele foutmelding:

```
#ifdef MSDOS
        #error "Dit programma compileert niet onder ms-dos."
#endif
```

Hoofdstuk 7

De C standaardbibliotheek

In de vorige hoofdstukken heb ik een aantal functies uit de standaard functiebibliotheek van C gebruikt. Deze functies zijn verspreid over 15 header bestanden. Voor het gemak zal ik wel de prototypes herhalen van de functies die ik al eerder uitgelegd heb.

Bij sommige functies kan ik jammer genoeg niet genoeg uitleg en voorbeelden geven als ik wel zou willen zonder deze cursus al te dik te laten worden ... Gelukkig zijn er nog altijd de man pages (hum).

7.1 `stdio.h`

7.1.1 Bestandsfuncties

`FILE *fopen(const char *bestandsnaam,const char *mode);` Zie §5.4.1.

`FILE *freopen(const char *bestandsnaam,const char *mode,FILE *stroom);`

`int fflush(FILE *stroom);`

`int fclose(FILE *stroom);` Zie §5.4.1.

`int remove(const char *bestandsnaam);`

Verwijdert een bestand. Als alles goed ging, is het functieresultaat 0.

`int rename(const char *oudenaam,const char *nieuwenaam);`

Geeft een bestand een nieuwe naam. Als er al een bestand met de `nieuwenaam` bestond, dan wordt dat verwijderd. Als alles goed ging, is het functieresultaat 0.

`FILE *tmpfile(void);`

`char *tmpnam(char s[L_tmpnam]);`

`int setvbuf(FILE *stroom,char *buffer,int mode,size_t buffergrootte);`

`void setbuf(FILE *stroom,char *buffer);`

7.1.2 Geformatteerde I/O

`void fprintf(FILE *stroom,const char *formaat,...);` Zie §5.3.1.

`int printf(const char *formaat,...);` Zie §5.3.1.

`int sprintf(char *doelstring,char *formaat,...);` Zie §5.3.1.

`int fscanf(FILE *stroom,const char *formaat,...);`

`int scanf(const char *formaat,...);` Zie §2.6.2.

`int sscanf(char *doelstring,const char *formaat,...);`

7.1.3 Eenvoudige I/O

`int fgetc(FILE *stroom);` Zie §5.5.2.
`int fgets(char *s,int n,FILE *stroom);` Zie §5.5.1.
`int fputs(int c,FILE *stroom);` Zie §5.5.1.
`int getc(FILE *stroom);` Zie §5.5.2.
`int getchar(void);` Zie §2.6.2.
`char *gets(char *doelstring);` Zie §2.6.2.
`int putc(int c,FILE *stroom);` Zie §5.5.2.
`int fputc(int c,FILE *stroom);` Zie §5.5.2.
`int putchar(int c);` Zie §2.6.1.
`int puts(const char *string);` Zie §2.6.1.
`int ungetc(int c,FILE *stroom);` Zie §5.5.2.

7.1.4 Blok I/O

```
size_t fread(void *buffer,size_t grootte,size_t aantal,FILE *stroom);
size_t fwrite(void *buffer,size_t grootte,size_t aantal,FILE *stroom);
```
Zie §5.5.3 voor uitleg over beide functies.

7.1.5 Positiebepaling in bestanden

`int fseek(FILE *stroom,long offset,int oorsprong);` Zie §5.5.4.
`long ftell(FILE *stroom);` Zie §5.5.4.
`void rewind(FILE *stroom);` Zie §5.5.4.
`int fgetpos(FILE *stroom,fpos_t *positie);`
`int fsetpos(FILE *stroom,const fpos_t *positie);`

7.1.6 Foutafhandeling

`void clearerr(FILE *stroom);` Zie §5.5.5.
`int feof(FILE *stroom);` Zie §5.5.5.
`int ferror(FILE *stroom);` Zie §5.5.5.
`int perror(const char *foutstring);`

7.2 `errno.h`

7.3 `string.h`

Op sommige systemen heet deze header file `string.h` en op andere `strings.h`, en geen van beide is dus echt portabel te noemen.

Overigens, als je echt op veilig wilt spelen, en super-portabele programma's wil maken, gebruik je het best enkel de functies `strcpy`, `strncpy`, `strcat`, `strncat`, `strlen`, `strcmp`, `strncmp`, `strchr` en `strrchr`. De andere functies zijn relatief nieuwe (ANSI) stringfuncties en worden niet altijd overal ondersteund.

7.3.1 Stringfuncties

`char *strcpy(char *doel,const char *bron);` Zie §3.6.2.
`char *strncpy(char *doel,const char *bron,int n);` Zie §3.6.2.
`char *strcat(char *voor,const char *achter);` Zie §3.6.3.
`char *strncat(char *voor,const char *achter,int n);` Zie §3.6.3.
`int strcmp(const char *string1,const char *string2);` Zie §3.6.1.
`int strncmp(const char *string1,const char *string2,int n);` Zie §3.6.1.

```
char *strchr(const char *string,int letter);
size_t strspn(const char *string,const char *aanvaard);
size_t strcspn(const char *string,const char *wijsaf);
char *strpbrk(const char *string,const char *aanvaard);
char *strstr(const char *hooiberg,const char *naald);
```

Doorzoekt de string `hooiberg` naar de plaats waar `naald` voor het eerst voorkomt. Als de string `naald` gevonden wordt in `hooiberg`, dan is het resultaat van `strstr()` een pointer naar de eerste letter ervan (binnen `hooiberg`). Anders is het resultaat `NULL`.

`size_t strlen(const char *string);` Zie §3.6.

```
char *strerror(int foutnummer);
char *strtok(char *string,const char *begrenzing);
```

7.3.2 Functies voor geheugenblokken

```
void *memcpy(void *doel,const void *bron,size_t aantal);
void *memmove(void *doel,const void *bron,size_t aantal);
```

Deze twee functies kopiëren geheugenblokken. Beide functies kopiëren `aantal` bytes beginnend vanaf `bron` naar `doel`. Het enige verschil is dat bij `memcpy()` bron en doel niet mogen overlappen. Bij `memmove()` mag dat wel, maar `memmove()` is iets trager dan `memcpy()` als beide gebieden niet overlappen.

```
int memcmp(const void *blok1,const void *blok2,size_t bytes);
void *memchr(const void *blok,int c,size_t bytes);
void *memset(void *blok,int c,size_t bytes);
```

7.4 stdlib.h

`double atof(const char *string);` Zie §2.4.5.
`int atoi(const char *string);` Zie §2.4.5.
`long atol(const char *string);` Zie §2.4.5.

```
double strtod(const char *s,char **eindptr);
long strtol(const char *s,char **eindptr,int talstelsel);
unsigned long strtoul(const char *s,char **eindptr,int talstelsel);
```

7.4.1 Willekeurige getallen

Deze functies heb ik al uitgelegd in §2.9.

```
int rand(void);
int srand(unsigned int seed);
```

7.4.2 Geheugenbeheer

`void *malloc(size_t grootte);` Zie §3.5.2.

`void free(void *gebied);` Zie §3.5.2.

`void *calloc(size_t aantal,size_t grootte);`

Reserveert een stuk geheugen van `aantal*grootte` bytes. Het verschil met `malloc()` is dat het geleverde stuk geheugen eerst gevuld wordt met nulbytes (de **c** van `calloc` staat voor **clear**: het geheugenblok wordt eerst gewist). Het niet portabel met `calloc()` een pointertabel te alloceren. De tabel zit immers vol met pointers met als bitpatroon allemaal nullen, en het is niet gegarandeerd dat de nulpointer hiermee overeenkomt. Jammer genoeg bestaat er geen standaard-functie die een nulpointertabel reserveert ...

`void *realloc(void *gebied,size_t grootte);`

Verandert de grootte van een eerder met `malloc()` gereserveerd geheugen`gebied`. Het is mogelijk dat het blok hiervoor verplaatst moet worden; het resultaat van `realloc()` geeft een pointer naar het nieuwe gebied. Het is dus niet verstandig na de `malloc()` aanroep de waarde van de pointer `grootte` nog verder te gebruiken. Als het blok groter is geworden, dan bevatten de extra bytes onbepaalde waarden. Als er niet genoeg geheugen meer is, is het resultaat `NULL` en blijft het originele geheugen`gebied` onaangeroerd.

Een paar bijzondere gevallen: `realloc(gebied,0)` doet hetzelfde als `free(gebied);` en `realloc(NULL,grootte)` hetzelfde als `malloc(grootte);`.

7.4.3 Programmabeëindiging

`void abort(void);`

Beëindigt het programma; deze functie keert dus nooit terug. Alle open bestanden worden geflusht en gesloten, maar functies die met `atexit()` opgegeven waren, worden niet uitgevoerd.

`void exit(int foutcode);`

Beëindigt het progrmamma. De functies die via `atexit()` opgegeven waren, worden in omgekeerde volgorde van het opgeven uitgevoerd. Daarna worden alle open bestanden geflusht en gesloten. Tenslotte wordt de foutcode doorgegeven aan het besturingssysteem. Voorgedefinieerde foutcodes zijn `EXIT_SUCCESS` en `EXIT_FAILURE`.

`int atexit(void (*functie)(void));`

De `functie` wordt aan een interne lijst toegevoegd. Al deze functies zullen in omgekeerde volgorde van het aanmelden bij `atexit()` worden aangeroepen wanneer het programma eindigt door een `return` in `main()` of een `exit()`. Als er iets fout ging, geeft `atexit()` een waarde verschillend van nul terug.

7.4.4 Communicatie met het systeem

`int system(const char *commando);`

Voert het `commando` uit met de shell van het systeem. Programma's die van deze functie gebruik maken zijn lastig om portabel te schrijven—zo werkt op sommige systemen `system("dir");` en op andere dan weer `system("ls");` ... Als alles goed gaat, is het functieresultaat de foutcode die het opgeroepen `commando` doorgaf. Met `system(NULL)` kan je controleren of er wel een shell aanwezig is (je kan maar nooit weten dat je programma ooit op een broodrooster wordt gedraaid): `system()` geeft dan een nul terug als er geen shell is.

`char *getenv(const char *naam);` Zie §4.8.2.

7.4.5 Sorteren en zoeken

`void *bsearch(const void *sleutel,const void *array, size_t n, size_t grootte, int (*vergelijk)(const void *,const void *));`

`void qsort(void *array,size_t n,size_t grootte, int (*vergelijk)(const void *,const void *));`

7.4.6 Wiskundige functies

```
int abs(int x);
long labs(long x);
div_t div(int noemer,int teller);
ldiv_t ldiv(long noemer,long teller);
```

7.5 time.h

'Let me tell you the whole story. It'll take a little time.'
'Time,' said Arthur weakly, 'is not currently one of my problems.'
—Douglas Adams, "The Hitch Hiker's Guide to the Galaxy"

Deze header file definieert een

```
struct tm {
        int tm_sec;    /* seconden (0..59) */
        int tm_min;    /* minuten  (0..59) */
        int tm_hour;   /* uren     (0..23) */
        int tm_mday;   /* dag      (1..31) */
        int tm_mon;    /* maand    (0 januari..11 december) */
        int tm_year;   /* jaren sinds 1900 */
        int tm_wday;   /* dagen    (0 zondag..6 zaterdag) */
        int tm_yday;   /* dag van het jaar (0..365) */
        int tm_isdst;  /* zomertijd (>0) of niet (0) of onbekend (<0) */
}
```

waarvan sommige functies uit deze header file gebruik maken om met datums te werken.

```
clock_t clock(void);
time_t time(time_t *tijd);
```

Zoals al uitgelegd in §2.9, geeft deze functie het aantal seconden sinds 1 januari 1970. (Ik heb toen een beetje gelogen en gezegd dat het returntype een `int` was; op sommige systemen kan een `int` maar hoogstens 32767 bevatten, wat duidelijk veel te weinig is.) Als `tijd` niet `NULL` is, wordt het functieresultaat ook nog eens in `*tijd` geschreven.

```
double difftime(time_t tijd2,time_t tijd1);
time_t mktime(struct tm *tijd);
char *asctime(const struct tm *tijd);
char *ctime(const time_t *tijd);
struct tm *gmtime(const time_t *tijd);
struct tm *localtime(const time_t *tijd);
```

`size_t strftime(char *string,size_t smax,const char *formaat,const struct tm *tijd);`

7.6 math.h

Al deze functies zitten in een aparte functiebibliotheek. Bij de GNU C compiler moet je expliciet vragen die wiskundige bibliotheek te raadplegen door de optie `-lm` op te geven.

```
double sin(double x);
double cos(double x);
double tan(double x);
double asin(double x);
double acos(double x);
double atan(double atan x);
double atan2(double y,double x);
```

Berekent de hoek die het punt (x, y) met de positieve X-as maakt. Deze hoek wordt in radialen uitgedrukt. Dit komt neer op het berekenen van `atan(y/x)` en dan het teken van het resultaat aanpassen naargelang het kwadrant waarin (x, y) ligt, met nog een extra controle voor het geval `x` gelijk is aan nul.

```
double sinh(double x);
double cosh(double x);
double tanh(double x);
double exp(double x);
double log(double x);
double log10(double x);
```

`double sqrt(double x);` Berekent $\sqrt{x}$.

```
double ceil(double x);
double floor(double x);
double fabs(double x);
double ldexp(double x,int n);
double frexp(double x,int *exp);
double modf(double x,double *ip);
double fmod(double x,double y);
```

`double pow(double x,double y);` Berekent de machtsverheffing x^y.

7.7 assert.h

`void assert(int uitdrukking);` Zie §10.1.

7.8 ctype.h

Deze header file beschrijft een aantal functies die van een `char` kunnen bepalen of het al dan niet een kleine of een hoofdletter is, ... Het voordeel van bijvoorbeeld `islower(c)` boven `(c>='a' && c<='z')` is dat `islower()` ook correct blijft werken in andere karaktersets dan ASCII, waar het niet altijd zo is dat de letters in het alfabet opeenvolgende codenummers hebben.

Het functieresultaat van al deze functies is nul als de letter niet van het geteste type is.

De precieze omschrijving van begrippen als "hoofdletter" of "witte ruimte" hangt af van de gebruikte locale (zie §7.10). Zo zal Ä niet als hoofdletter herkend worden in de **"C"** locale.

`int iscntrl(int c);` Gaat na of `c` een controlekarakter (bijvoorbeeld `'\n'` of `'\t'`) is.

`int isdigit(int c);` Gaat na of `c` een cijfer (`0` t/m `9`) is.

`int isgraph(int c);` Gaat na of `c` een afdrukbaar teken is maar geen spatie.

`int islower(int c);` Gaat na of `c` een kleine letter (`a` t/m `z`) is.

`int isupper(int c);` Gaat na of `c` een hoofdletter (`A` t/m `Z`) is.

`int isalpha(int c);` Gaat na of `c` een letter is; doet hetzelfde als `islower(c) || isupper(c)`.

`int isalnum(int c);` Gaat na of `c` een letter of cijfer is; doet hetzelfde als `isalpha(c) || isdigit(c)`.

`int isprint(int c);` Gaat na of `c` een afdrukbaar teken is; ook spaties worden als afdrukbaar beschouwd.

`int ispunct(int c);` Gaat na of `c` een afdrukbaar teken is dat geen spatie, letter of cijfer is.

`int isspace(int c);` Gaat na of `c` witte ruimte voorstelt; in de **"C"** locale zijn dat `' '`, `'\f'`, `'\n'`, `'\r'`, `'\t'` en `'\v'`.

`int isxdigit(int c);` Gaat na of `c` een hexadecimaal cijfer is. (`0` t/m `9`, `a` t/m `f` en `A` t/m `F`)

7.9 Systeemafhankelijke constanten

Deze header file bevat informatie over het waardenbereik van sommige datatypen op het systeem waarop het programma gecompileerd wordt.

7.9.1 `float.h`

#define	kleinst mogelijke waarde	betekenis
`FLT_RADIX`	2	grondtal bij exponentweergave
`FLT_ROUNDS`		floating point afrondingsgedrag bij optellen
`FLT_DIG`	6	aantal decimale cijfers precisie
`FLT_EPSILON`	1E-5	kleinste getal waarvoor `1.0 + FLT_EPSILON` $\neq$ `1.0`
`FLT_MANT_DIG`		aantal cijfers in mantisse bij grondtal `FLT_RADIX`
`FLT_MAX`	1E+37	maximumwaarde van een `float`
`FLT_MAX_EXP`		grootste exponent waarvoor $\texttt{FLT\_RADIX}^{\texttt{FLT\_MAX\_EXP}} - 1$ nog in een `float` past
`FLT_MIN`	1E-37	
`FLT_MIN_EXP`		
`DBL_DIG`	10	
`DBL_EPSILON`	1E-9	
`DBL_MANT_DIG`		
`DBL_MAX`	1E+37	
`DBL_MAX_EXP`		
`DBL_MIN`	1E-37	
`DBL_MIN_EXP`		

7.9.2 limits.h

#define	kleinst mogelijke waarde	betekenis
CHAR_BIT	8	aantal bits in een `char`
CHAR_MAX	UCHAR_MAX of SCHAR_MAX	maximumwaarde van een `char`
CHAR_MIN	0 of SCHAR_MIN	minimumwaarde van een `char`
INT_MAX	32767	maximumwaarde van een `int`
INT_MIN	-32767	minimumwaarde van een `int`
LONG_MAX	2147483647L	maximumwaarde van een `long`
INT_MIN	-2147483647L	minimumwaarde van een `long`
SCHAR_MAX	127	maximumwaarde van een `signed char`
SCHAR_MIN	-127	minimumwaarde van een `signed char`
SHRT_MAX	32767	maximumwaarde van een `short`
SHRT_MIN	-32767	minimumwaarde van een `short`
UCHAR_MAX	255U	maximumwaarde van een `unsigned char`
UINT_MAX	65535U	maximumwaarde van een `unsigned char`
ULONG_MAX	4294967295UL	maximumwaarde van een `unsigned char`
USHRT_MAX	65535U	maximumwaarde van een `unsigned char`

7.10 locale.h

Een “**locale**” is een verzameling taal- en cultuur-conventies in verband met onder andere de taal voor boodschappen, karaktersets, lexicografische conenties. Met de functies uit deze header file kan een programma bepalen binnen welke locale het draait en zich daaraan aanpassen.

```
char *setlocale(int categorie,const char *locale);
struct lconv *localeconv(void);
```

7.11 setjmp.h

```
int setjmp(jmp_buf env);
void longjmp(jmp_buf env,int waarde);
```

7.12 signal.h

```
void (*signal(int signaal,void (*handler)(int)))(int);
int raise(int signaal);
```

7.13 stdarg.h

In deze header file worden de hulpmiddelen geleverd om zelf functies te schrijven die een variabel aantal argumenten kunnen hebben, zoals bijvoorbeeld `printf()`. Zie §4.6.

```
va_start(va_list ap,laatstearg);
type va_arg(va_list ap,type);
void va_end(va_list ap);
```

7.13.1 printf-achtige functies

```
int vprintf(const char *formaat,va_list arg);
int vfprintf(FILE *stroom,const char *formaat,va_list arg);
int vsprintf(char *s,const char *formaat,va_list arg);
```

7.14 stddef.h

Hier wordt onder andere het type `ptrdiff_t` gedefinieerd, dat zeker groot genoeg is om het (signed) verschil van twee willekeurige pointers te kunnen bevatten.

Ook het type `size_t` wordt hier gedefinieerd. Dit is het type dat de `sizeof()` operator teruggeeft. Het is altijd een unsigned geheel datatype.

Hoofdstuk 8

Netwerkprogramma's

Een hoop computers die met elkaar kunnen communiceren wordt een **(computer)netwerk** genoemd. Vanzelfsprekend moeten alle computers in een netwerk dezelfde communicatiemethode of **protocol** gebruiken. Tegenwoordig is het **Internet Protocol** (IP) heel populair. Ik zal in dit hoofdstuk dan ook uitleggen hoe je programma's kunt maken die communiceren met elkaar over het Internet heen.

Over bepaalde details zal ik nogal vlug heenspringen. Een netwerk van miljoenen computers zit nu eenmaal niet simpel in elkaar, maar voor eenvoudige toepassingen is het gebruik ervan erg eenvoudig. Als ik ook alle achterliggende details uit zou willen leggen, dan zou deze cursus minstens in omvang moeten verdubbelen ...

8.1 IP adressen en host names

Elke computer op het Internet (**host**) heeft een uniek identificatienummer, het zogenaamd **IP adres**, dat uit vier bytes bestaat en meestal geschreven wordt als vier decimale getallen (elk tussen 0 en 255) met een punt ertussen (een **dotted quad**) Zo heeft de computer `zeus.rug.ac.be` het dotted quad `157.193.41.38` als IP-adres. IP-adressen zijn voor mensen nogal lastig te onthouden, en daarom hebben de computers op het internet ook een naam in letters (zoals daarnet `zeus.rug.ac.be`).

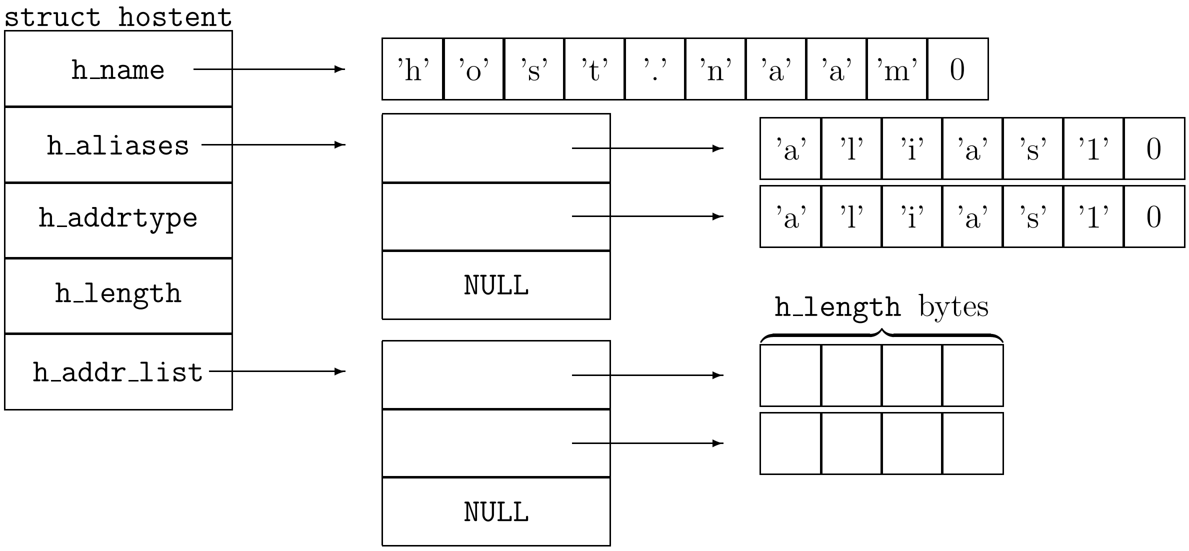
Het omzetten van de ene vorm naar de andere gebeurt door aparte computers die aan het netwerk hangen, zogenaamde **nameservers**. Aan zo'n nameserver kan je dus vragen wat de naam is van de computer met adres `157.193.41.38`, of welk adres `zeus.rug.ac.be` heeft. Er is niet één grote centrale nameserver, maar een hoop nameservers die als het nodig is aan elkaar kunnen informatie doorsturen. In de praktijk is alles zo georganiseerd dat je er als programmeur (en zeker als gebruiker) niet veel van merkt.

De functies `gethostbyname()` en `gethostbyaddr()` zorgen voor het opzoeken in beide richtingen. Ze geven een pointer naar een `hostent` structuur terug. Als er geen host met de opgegeven naam of adres gevonden werd, is het resultaat `NULL`. Deze twee functies en de `hostent` structuur zijn te vinden in `netdb.h`.

De `hostent` structuur is flexibel genoeg om hosts met verschillende namen en adressen aan te kunnen:

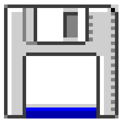
```
struct hostent {
        char *h_name;       /* officiele naam van de host */
        char **h_aliases;   /* lijst andere namen van de host */
        int h_addrtype;     /* bijvoorbeeld AF_INET voor een IP adres */
        int h_length;       /* lengte van een adres */
```

```
        char **h_addr_list; /* lijst adressen; elk adres is h_length bytes lang */
};
```



De hostent structuur

Het volgende programmaatje demonstreert de `gethostbyname()` en `gethostbyaddr()` functies:



hostnaam.c

```
#include <stdio.h>
#include <netdb.h>
#include <netinet/in.h>         /* voor AF_INET */

void printhost(struct hostent *host)
{
        if (!host) {
                puts("geen hostent!");
                return;
        }

        printf("hostnaam: %s\n",host->h_name);

        /* aliases */
        {
                int aantal = 0;
                while (1) {
                       char *alias = host->h_aliases[aantal];
                       if (!alias) break;
                       printf("alias %s\n",alias);
                       aantal++;
                }
        }

        /* ip adressen */
        {
                int aantal = 0;
                while (1) {
```

```
                        unsigned char *ipadres = host->h_addr_list[aantal];
                        int tel;
                        if (!ipadres) break;

                        printf("ip adres ");
                        /* het adres bestaat uit host->h_length bytes */
                        for (tel = 0; tel < host->h_length-1; tel++)
                                printf("%d.",ipadres[tel]);
                        printf("%d\n",ipadres[tel]);
                        aantal++;
                }
        }
}

int main(void)
{
        unsigned char adres[] = { 157,193,41,38 };

        printhost(gethostbyname("zeus.rug.ac.be"));
        puts("--------");
        printhost(gethostbyaddr(adres,4,AF_INET));
        return 0;
}
```

Het argument van `gethostbyname()` spreekt voor zich, en de argumenten van `gethostbyaddr()` zijn achtereenvolgens een pointer naar het adres, de lengte van het adres in bytes, en het gebruikte protocol.

Het resultaat van beide `printhost()` aanroepen is telkens

```
hostnaam: zeus
alias zeus.rug.ac.be
ip adres 157.193.41.38
```

tenminste als ik het programma draai vanop een computer binnen `rug.ac.be` zelf. Als je `zeus.rug.ac.be` vervangt door bijvoorbeeld `ftp.funet.fi` dan krijg je:

```
hostnaam: ftp.funet.fi
ip adres 128.214.248.6
```

8.2 Poorten

Eén host kan terzelfdertijd verbinding maken met tientallen computers. Als je dus een verbinding wil maken met een host is het niet genoeg het IP adres ervan te kennen—om al die connecties uit elkaar te houden heeft elke host 65536 **poorten**. Hopelijk zit dan op die computer een ander programma te luisteren naar binnenkomende aanvragen op die poort.

8.3 Sockets

Een **socket** is een “communicatie-eindpunt”. Als twee programma’s een socket hebben gemaakt, dan kunnen ze die met elkaar verbinden en zo met elkaar communiceren. Een socket is dus een beetje te vergelijken met een telefoontoestel: als we alle twee een telefoon hebben, dan moeten we om te kunnen communiceren alleen nog op een of andere manier aan het (telefoon)netwerk duidelijk maken dat we beide toestellen willen verbinden.

8.4 Clients schrijven

Ik zal eerst er van uitgaan dat er al een programma draait op een of andere host dat staat te wachten op verbindingen. Zo'n programma wordt een **server**-programma genoemd: het biedt een of andere dienst aan aan al wie er verbinding mee maakt (de **client**s). Een server staat dus de hele tijd passief niks te doen totdat er een client langskomt, terwijl een client actief een verbinding opbouwt.

Je kan bijvoorbeeld een WWW-serverprogramma draaien, zodat anderen WWW-verbindingen kunnen maken met je computer. Het WWW-serverprogramma wacht op poort 80 op aanvragen van pagina's. Analoog draait een mail-server meestal op poort 25.

Hoe je een serverprogramma schrijft, zal ik later pas uitleggen; nu is het er ons om te doen een verbinding te maken met een al bestaande server. Het eerste wat we moeten doen, is een socket maken. Dat gaat met de functie `socket()`:

```
#include <sys/types.h>
#include <sys/socket.h>

int socket(int domein,int type,int protocol);
```

Het `domein` geeft aan welk communicatieprotocol voor de socket gebruikt moet worden. In ons geval is dat het IP protocol, wat je kan opgeven met `AF_INET`. (De AF staat voor **Address Family**, wat hier ongeveer synoniem is met protocol.)

Het `type` geeft aan op welke manier de communicatie moet verlopen. De meest primitieve manier is `SOCK_DGRAM` waarbij je kleine pakketjes informatie per keer moet doorsturen. Zo'n pakketje (ook wel **datagram** genoemd) komt dan in één keer toe bij de bestemmeling, *als* het toekomt, want datagram-sockets garanderen niet dat verzonden pakketjes effectief aankomen. Datagram-sockets zijn ook **connectieloos**: je moet eerst geen verbinding maken met de bestemmeling, maar je kan de pakketjes wanneer je maar wil afvuren. Datagram-sockets komen dus meer overeen met een brief versturen dan met een telefoongesprek: je kan de brief versturen wanneer je wil, maar je weet niet zeker dat hij aangekomen is (of ooit aan zal komen). En als je meerdere pakketten verstuurt, weet je ook niet in welke volgorde die zullen aankomen.

Ik zal enkel `SOCK_STREAM` sockets gebruiken. Een stream-socket heeft veel weg van een telefoonverbinding. Je moet eerst een verbinding maken (met `connect()`, zie verder), net zoals je een bij een telefoon eerst een nummer moet draaien. Daarna kunnen beide kanten bytes in de socket gieten, die er aan de andere kant gegarandeerd uitkomen, en wel in dezelfde volgorde als de zender ze er heeft in gestopt.

Het laatste argument, `protocol`, kan gebruikt worden om varianten te kiezen van het protocol zoals opgegeven in het `domein` argument. Meestal wordt hier `0` ingevuld, wat aangeeft dat het systeem het zelf maar moet uitzoeken.

Het resultaat van `socket()` is een `int`, die de andere socket-functies nodig hebben. Als er iets fout ging, is het resultaat `-1`.

De situatie is een beetje vergelijkbaar met de `FILE *` die je van `fopen()` terugkrijgt: wat die `int` precies voorstelt is niet belangrijk voor ons, zolang het systeem daar maar genoeg aan heeft om te weten dat dat onze socket is. De algemene term hiervoor is **magic cookie**: het ding (die `FILE *` of `int`) heeft wel ergens een betekenis voor de functies die ermee werken, maar voor ons is het een ding dat magischerwijze blijft werken zolang we er maar afblijven.

Vervolgens moeten we de socket nog verbinden met de socket van het gewenste serverprogramma. Dat gaat met de functie `connect()`:

```
#include <sys/types.h>
#include <sys/socket.h>

int connect(int socketnr,struct sockaddr *serverAdres,int adresLen);
```

Als `socketnr` moet je het resultaat van `sock()` invullen.

In `serverAdres` komt een pointer naar een `struct sockaddr_in` te staan, en niet een `struct sockaddr *`. Dat komt omdat `connect()` onafhankelijk is van het gebruikte protocol; een `struct sockaddr` beschrijft een algemeen adres voor een niet nader aangeduid protocol:

```
struct sockaddr {
        unsigned short sa_family;
        char     sa_data[14];
};
```

De waarde van `sa_family` geeft aan om welk protocol het precies gaat. Internet adressen worden met een `struct sockaddr_in` aangeduid:

```
struct sockaddr_in {
        unsigned short sin_family;     /* bevat AF_INET */
        unsigned short sin_port;       /* poortnummer */
        struct in_addr sin_addr;       /* IP adres */
        char sin_zero[8];              /* ongebruikt */
};
```

Het IP adres wordt als volgt beschreven:

```
struct in_addr {
        unsigned long s_addr;
}
```

Het resultaat van `connect()` is 0 als alles goed ging, of -1 als er een fout optrad.

De functies die kunnen schrijven en lezen in een socket, zijn nogal primitief. Gelukkig kunnen we er een gewone `FILE *` stroom van maken met de functie `fdopen()`.

Dat alles wordt gedemonstreerd in de volgende functie:

fopensock.c

```
#include <stdio.h>
#include <netdb.h>
#include <netinet/in.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <string.h>        /* voor memcpy() */

#include "fopensock.h"

FILE *fopenSocket(struct hostent *host,int poort)
```

```
{
        int pries;
        struct sockaddr_in priesAddr;
        FILE *resultaat;

        if (!host) return NULL;
        pries = socket(AF_INET,SOCK_STREAM,0);
        if (pries < 0) return NULL;

        memcpy(&priesAddr.sin_addr,host->h_addr,host->h_length);
        priesAddr.sin_family = AF_INET;
        priesAddr.sin_port = htons(poort);
        if (connect(pries,(struct sockaddr *)&priesAddr,
                sizeof(struct sockaddr_in)) < 0)
                return NULL;

        resultaat = fdopen(pries,"r+");
        setbuf(resultaat,NULL); /* buffering uitzetten */
        return resultaat;
}
```

De volgorde waarin de verschillende bytes in een `short` worden opgeslagen, hangt af van het gebruikte computersysteem. De volgorde die op het Internet wordt gebruikt heet **network byte order**; de volgorde die jouw computer gebruikt heet **host byte order**. Vandaar de naam `htons`: **h**ost **to n**etwork **s**hort. Er bestaan ook nog `ntohs()` die de omgekeerde omzetting doet, en `htonl()` en `ntohl()` die hetzelfde met `long`s doen. Blijkbaar moeten poortnummers in een `struct sockaddr_in` in netwerkvolgorde staan; vandaar de `htons()` aanroep.

Het voordeel van deze functie `fopenSocket()` is dat we ons vanaf nu niets meer van de interne werking van sockets moeten aantrekken. Alle internalia van sockets worden netjes achter de schermen afgehandeld door `fopenSocket()`, en het resultaat is een `FILE *` stroom waarin je kunt lezen en schrijven met de gekende I/O-functies.

In `fopensock.h` staat alleen maar het prototype van `fopensock()`, zodat ik deze functie verderop makkelijk kan gebruiken in de toepassingen.

8.4.1 Voorbeeld: e-mail sturen

Een eenvoudige toepassing is het versturen van e-mail. Ik zal uitleggen hoe dat gaat via het Simple Mail Transfer Protocol (SMTP). Het mailserver-programma luistert op poort 25. Alle communicatie gebeurt door stukken tekst uit te wisselen; er zitten dus geen niet-afdrukbare tekens tussen (bijvoorbeeld die met ASCII code kleiner dan 32).

Om uit te zoeken hoe dat protocol werkt, kan je het programma `telnet` handig gebruiken. Op mijn netwerk draait er een mailserver op de host `thanatos`. Als ik `telnet thanatos 25` in de shell tik, maakt `telnet` een verbinding met de mailserver die op `thanatos` draait. Alles wat ik intik, wordt naar de server gestuurd, en alles wat de server terugstuurt, komt op het scherm. Een goeie gok is altijd het intikken van `HELP`, zoals blijkt uit de volgende telnet-sessie (wat ik intik staat *cursief*; wat het systeem antwoordt staat in `schrijfmachineletters`):

telnet thanatos 25

`220 thanatos.rug.ac.be ESMTP Sendmail 8.8.3/8.8.3; Mon, 2 Nov 1998 18:45:05 +0100` *help*

```
214-This is Sendmail version 8.8.3
214-Topics:
214-    HELO    EHLO    MAIL    RCPT    DATA
214-    RSET    NOOP    QUIT    HELP    VRFY
214-    EXPN    VERB    ETRN    DSN
214-For more info use "HELP <topic>".
214-To report bugs in the implementation send email to
214-    sendmail-bugs@sendmail.org.
214-For local information send email to Postmaster at your site.
214 End of HELP info
```

De mailserver is heel vriendelijk: hij antwoordt netjes met onder andere een lijst van alle commando's die hij begrijpt! Vier commando's hiervan volstaan om mail te sturen:

help mail

```
214-MAIL FROM: <sender> [ <parameters> ]
214-    Specifies the sender.  Parameters are ESMTP extensions.
214-    See "HELP DSN" for details.
214 End of HELP info
```

help rcpt

```
214-RCPT TO: <recipient> [ <parameters> ]
214-    Specifies the recipient.  Can be used any number of times.
214-    Parameters are ESMTP extensions.  See "HELP DSN" for details.
214 End of HELP info
```

help data

```
214-DATA
214-    Following text is collected as the message.
214-    End with a single dot.
214 End of HELP info
```

help quit

```
214-QUIT
214-    Exit sendmail (SMTP).
214 End of HELP info
```

Eens proberen of het lukt om een mailtje te sturen:

MAIL FROM: ikke

```
250 ikke... Sender ok
```

RCPT TO: geert@thanatos

```
250 geert@thanatos... Recipient ok
```

DATA

```
354 Enter mail, end with "." on a line by itself
```

Dag Geert!
Hoe gaat het nog met jezelf?
.

```
250 TAA01185 Message accepted for delivery
```

QUIT

```
221 thanatos.rug.ac.be closing connection
```

Ook als de ontvanger email-adres `thanatos` zelf heeft maar op een andere server, komt de mail toe. De mail-servers kunnen immers mail naar elkaar door sturen.

Dat alles in een programmaatje gieten is geen probleem:

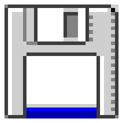
mail.c

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

#include "fopensock.h"

/* Een ietwat roekeloze versie, die niet controleert of
 * de mail-server wel antwoordt dat alles ok is ... */
void sendmail(FILE *socket,char *van,char *naar,char *bericht)
{
        char buff[255];

        /* De mailserver vindt het blijkbaar absoluut noodzakelijk
         * dat zijn eerste regel daadwerkelijk gelezen wordt door de client.
         * Dus doen we dat maar ... */
        fgets(buff,sizeof(buff),socket);

        fprintf(socket,"MAIL FROM: %s\n",van);
        fprintf(socket,"RCPT TO: %s\n",naar);
        /* Op de volgende regel wordt het eigenlijke bericht verstuurd.
         * De mail-server veronderstelt dat het bericht gedaan is als
         * hij een regel met alleen maar een punt erop tegenkomt.
         * Als er in het bericht dus al zo'n regel zit, is dat dikke pech ... */
        fprintf(socket,"DATA\n%s\n.\n",bericht);
        fputs("QUIT\n",socket);
}

int main(void)
{
        FILE *mailsocket;

        mailsocket = fopenSocket(gethostbyname("thanatos"),25);
        sendmail(mailsocket,"een.bewonderaar","geert@thanatos",
                "Subject: ik wacht ...\n"
                "Wanneer is die nieuwe versie van de C cursus nu klaar?\n"
                "Ik wacht al een maand!");
        fclose(mailsocket);
        return 0;
}
```

Hier kan je ook zien dat een eventueel onderwerp van de mail gewoon in de boodschap zelf opgenomen kan worden.

8.4.2 Voorbeeld: een HTTP verbinding

Een van de meer populaire toepassingen op Internet is ongetwijfeld WWW. Om WWW-connecties te maken, wordt meestal poort 80 gebruikt. Het gebruikte protocol wordt HTTP (HyperText Transfer Protocol) genoemd. Ook dit protocol is volledig tekstgebaseerd. De telnet-truuk werkt dus ook hierop: `telnet thanatos 80` geeft, in de veronderstelling dat op host `thanatos` een http-serverprogramma draait dat luistert op poort 80. Als je bijvoorbeeld de pagina `http://thanatos/~geert/` wil lezen, volstaat het de regel *GET /~geert/* door te sturen. De HTTP-server antwoordt dan door de hele pagina terug te zenden, en sluit dan de verbinding.


printhttp.c

```
#include <stdio.h>
#include <netdb.h>
#include "fopensock.h"

/* Om bijvoorbeeld http://zeus.rug.ac.be/C/ te lezen moet
 * host wijzen naar een hostent die met zeus.rug.ac.be overeenkomt
 * en url de string "/C/" bevatten. */
void printhttp(struct hostent *host,char *url)
{
        FILE *sock = fopenSocket(host,80);
        int letter;

        fprintf(sock,"GET %s HTTP/1.0\n\n",url);
        while ((letter = fgetc(sock)) != EOF) {
                putchar(letter);
        }
        fclose(sock);
}

int main(void)
{
        printhttp(gethostbyname("localhost"),"/~geert/");
        return 0;
}
```

De server stuurde domweg de inhoud van de pagina door. Je kan meer informatie vragen over door aan de aanvraag *HTTP/1.0* toe te voegen, gevolgd door een lege regel, dus door bijvoorbeeld *GET /~geert/ HTTP/1.0* door te sturen, en daarna een lege regel. In dat geval ziet de uitvoer er ongeveer zo uit:

```
HTTP/1.0 200 OK
Date: Mon, 02 Nov 1998 21:25:18 GMT
Server: Apache/1.0.0
Content-type: text/html
Content-length: 210
Last-modified: Sat, 25 Jul 1998 12:34:12 GMT

```

`<HTTP><HEAD`*enzovoort: de inhoud van de pagina*

Als er iets fout ging, bijvoorbeeld als de gevraagde pagina niet bestaat, is de eerste regel anders en bevat een foutcode zoals `HTTP/1.0 404 Not found`.

Je zou de `printhttp()` functie hier rekening mee kunnen doen houden.

8.5 Servers schrijven

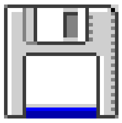
Zoals al eerder gezegd is voor alle communicatie een socket nodig. De eerste stap is dus precies als voorheen `socket()` te gebruiken. Vanaf hier houden de gelijkenissen met client-programma's echter op.

Aangeven op welke poort we willen luisteren gebeurt met `bind()`. Deze functie heeft dezelfde syntax als `connect()`. Het adres in de `struct sockaddr_in` geeft aan op welke poort we willen luisteren, en ook van welke netwerkverbinding in de computer (bijvoorbeeld een modem of een ethernetkaart) we verzoeken aanvaarden. De meeste computers hebben maar één netwerkverbinding. Het veiligst is de waarde `INADDR_ANY` in te vullen, die aangeeft dat we van alle netwerkverbindingen verzoeken willen aanvaarden. Het resultaat van `bind()` is een socket, of -1 als er iets fout ging.

Vervolgens moeten we het systeem verwittigen dat we met deze socket willen luisteren naar verbindingen. Dat gaat met `listen(socket,aantal)`, waarbij het `aantal` aangeeft hoeveel wachtende verbindingen er maximaal kunnen zijn. Op de meeste systemen is dat maximum beperkt tot 5 (als je meer probeert op te geven, wordt er toch 5 van gemaakt), dus het is zaak als server om snel genoeg "de hoorn op te nemen". Als er iets fout gaat, is het resultaat -1, anders 0.

Tenslotte moeten we de binnenkomende verbindingen verwerken. De functie `accept()` blijft wachten tot er een client verbinding maakt met onze socket. Het resultaat is een *nieuwe* socket, waarlangs we met de client kunnen communiceren. De originele socket (die die we een `bind()` behandeling hebben gegeven) kunnen we blijven gebruiken om nieuwe client-verbindingen te aanvaarden. Een vaak gemaakte en lastig op te sporen fout is proberen te communiceren langs de eerste socket, wat gegarandeerd tot onverwachte resultaten leidt.

Dit alles wordt in de praktijk gebracht door het volgende programma:



netecho.c

```
#include <stdio.h>
#include <netdb.h>
#include <netinet/in.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <string.h>
#include <stdlib.h>     /* voor atoi() */

int main(int argc,char **argv)
{
        int poort = 4000;
        int sock;
        int quiet = 0;

        /* Drie manieren van aanroepen: ofwel
         * netecho              (gebruik poort 4000)
         * netecho 1234         (gebruik poort 1234)
         * netecho 1234 quiet   (gebruik poort 1234 en antwoord niets
         *                       terug aan de client; schrijf enkel op
         *                       het scherm wat de client doorstuurt) */
        if (argc >= 2) poort = atoi(argv[1]);
        if (argc >= 3) {
                if (!strcmp(argv[2],"quiet")) quiet = 1;
        }

        sock = socket(AF_INET,SOCK_STREAM,0);
        {       /* poort van de socket instellen */
                struct sockaddr_in adres;

                adres.sin_family = AF_INET;
                adres.sin_addr.s_addr = INADDR_ANY;
```

```
                adres.sin_port = htons(poort);
                if (bind(sock,(struct sockaddr *)&adres,sizeof(adres))<0) {
                        puts("bind error");
                        return 10;
                }
        }

        /* begin te luisteren naar clients die een verbinding willen */
        if (listen(sock,5)< 0) {
                puts("listen error");
                return 11;
        }

        while (1) {
                struct sockaddr_in client;
                int clientSock;
                char regel[255];
                FILE *klant;
                int clientlen = sizeof(client);

                printf("Wacht op verbindingen op poort %d ...\n",poort);
                clientSock = accept(sock,(struct sockaddr *)&client,&clientlen);
                puts("Een klant! Een klant!");
                klant = fdopen(clientSock,"r+");
                setbuf(klant,NULL);
                while (fgets(regel,sizeof(regel),klant)) {
                        if (!quiet) fprintf(klant,"Je zei: %s",regel);
                        printf(">\t%s",regel);
                }
                /* client heeft de verbinding verbroken */
                fclose(klant);
        }

        return 0;
}
```

Dit programma wacht gewoon tot er iemand verbinding maakt, en herhaalt dan alle door de client doorgestuurde regels aan de client. Op het scherm wordt ook uitgeschreven wat er allemaal gebeurt.

Het nadeel van dit serverprogramma is dat het maar één client tegelijk kan bedienen. Alle andere clients moeten wachten tot de server klaar is met de vorige client. Zo'n server wordt een **iteratieve server** genoemd.

Je kan dit programma uittesten door te `telnet`ten naar poort 4000 van je computer. In dit geval gebruiken we dus het bestaande programma `telnet` als client-programma.

Het is interessant om zien wat er gebeurt als je met een browser een verbinding maakt met het programma, door bijvoorbeeld naar de pagina `http://thanatos:4000/hallo.html` te gaan. De meeste browsers sturen naast de `GET /hallo.html` regel nog wat extra informatie door, bijvoorbeeld welke browser de gebruiker gebruikt (op een regel die met `User-Agent:` begint). Dit is bijvoorbeeld wat Netscape 3.01 blijkt door te sturen:

```
>       GET /hallo.html HTTP/1.0
>       Connection: Keep-Alive
```

```
>       User-Agent: Mozilla/3.01 (X11; I; Linux 2.0.34 i586)
>       Host: thanatos:4000
>       Accept: image/gif, image/x-xbitmap, image/jpeg, image/pjpeg, */*
>
```

Het schrijven van servers die meerdere verbindingen tegelijkertijd aankunnen is andere koek en valt jammer genoeg buiten het bereik van dit inleindende hoofdstuk.

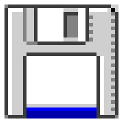
8.6 CGI programma's

Een WWW-site wordt vaak opgebouwd als een hoop HTML bestanden die het WWW-serverprogramma dan aan alle clients doorstuurt. Maar het is ook mogelijk om in plaats van een bestand de uitvoer van een programma mee te geven aan de client. Op die manier kan je interactieve toepassingen op het WWW maken.

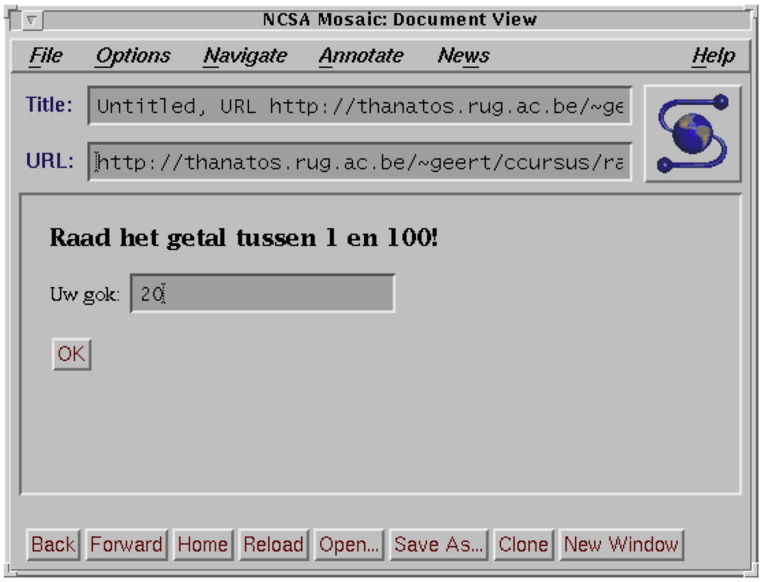
Het voordeel hiervan is dat de communicatie met het programma voor de gebruiker redelijk aantrekkelijk verloopt (de meeste tegenwoordige WWW-browsers kunnen genoeg grafische hoogstandjes aan), en dat er geen aparte software moet gedownload worden: een browser is genoeg; alle verwerking gebeurt door de server.

Er is een standaard koppeling tussen de WWW server en je eigen programma die Common Gateway Interface genoemd wordt.

Ik zal het raad-het-getal spelletje herprogrammeren met CGI. Om te beginnen hebben we een startpagina in HTML nodig, die ik in een bestand `raadhetgetal.html` bewaar:



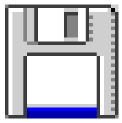

```
<H1>Raad het getal tussen 1 en 100!</H1>
<FORM METHOD=POST ACTION=raadhetgetal.cgi>
Uw gok: <INPUT NAME=gok>
<P>
<INPUT TYPE=submit VALUE=OK>
</FORM>
```



De startpagina van Raad Het Getal

Belangrijk is het stuk tussen `<FORM>` en `</FORM>`: een CGI programma krijgt de parameters uit het HTML formulier door via `stdin` of langs een environment variabele, afhankelijk van de gebruikte methode in het formulier (bij `METHOD=POST` is het langs `stdin`). De enige parameter die we gemaakt hebben is `gok`.

Alle magisch achter-de-schermen werk zit verborgen in `cgisup`, een verzameling CGI hulproutines die al jaren lang haar diensten bewijst:



cgisup.c

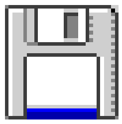
```
/*      This file is Copyright (C) GVE Software - Geert Vernaeve.
 *      You may use it for non-commercial purposes only.
 *      Distribution is freely permitted, as long as no fee is charged.
 *
 *      For comments, mail me at geert.vernaeve@rug.ac.be
 *                      or geert@einstein.rug.ac.be
 *
 *              CGI SUPPORT MODULE V1.1d
 *
 *      HISTORY:
 *      V1.1b: Fixed some bugs from the V1.1 release.
 *      V1.1c: Added documentation in the source code.
 *      V1.1d:
 *
 *         _agd#######.      ,d#########
 *       _J############.    d###########
 *      .###O"      "###.   ####
 *      d###         "###.  ####
 *      ####          "###. ##########
 *      ####           "###.##########
 *      Q###    ####    "#######
 *      "###b   ####     "######
 *       `Q########D      "#############
 *          "O###O"        "###########
 */
#include <stdio.h>
#include <stdlib.h>
#include "cgisup.h"

/* We maintain two linked lists of arguments, i.e. items
 * of the type "value=data".
 * The first is ArgForm and contains all things from a HTML form
 * (e.g. <INPUT NAME=Name> or <SELECT NAME=name>)
 * and the second list, ArgURL, contains all things from the URL
 * of our page (i.e. http://.../cgiwrap?value1=name1&value2=name2&...).
 */
struct Argument {
        struct Argument *Next;
        char *Name;
        char *Data;
};

static struct Argument *ArgForm = NULL, *ArgURL = NULL;

/* Scan the list for a certain Name and return the Data associated with it.
 * If the Name is not found, NULL is returned. */
static char *getarg(char *Name, struct Argument *Scan)
{
        for (; Scan; Scan = Scan->Next)
                if (!strcmp(Name,Scan->Name)) return Scan->Data;
        return NULL;
```

```
}

/* Using this function, you can find out which values the user filled in
 * into your HTML form.
 * NOTE: if the user did not enter data, you may get a NULL return
 * value (and not "" which you would expect). */
char *GetArg(char *Name)
{
        return getarg(Name,ArgForm);
}

/* This function is used to check values which are filled in into the
 * document's URL. */
char *GetURLArg(char *Name)
{
        return getarg(Name,ArgURL);
}

/* hex digit conversion */
static int Hex(char c)
{
        if (c>='0' && c<='9') return c-'0';
        if (c>='a' && c<='f') return c-'a'+10;
        if (c>='A' && c<='F') return c-'A'+10;
}

/* HTTP encodes all nonstandard ASCII in the following way:
 * + --> space
 * %xx -> ASCII hex xx
 * This function performs an in-place translation of this data.
 */
static void translate(char *str)
{
        register char *write = str;

        while (*str) {
                if (*str == '+') {
                        *write++ = ' ';
                        str++;
                } else if (*str == '%') {
                        register unsigned char OutChar;

                        str++;
                        OutChar = Hex(*str++)<<4;
                        OutChar += Hex(*str++);
                        *write++ = OutChar;
                } else *write++ = *str++;
        }
        *write = 0;
}

/* This is the workhorse function.
 * Take an input string in the form
 * value1=data1&value2=data2&...
 * and create a linked list containing this data.
 * ArgList is a pointer to the anchor of the list to be created.
 * On the fly, all "value" and "data" strings are converted to ASCII
 * (see translate()) */
static void interpretargs(char *input,struct Argument **ArgList)
```

```
{
        char *String;

        *ArgList = NULL;

        /* input MAY be non modifiable (e.g. the result of getenv()).
         * Create a buffer large enough to work on. */
        String = malloc(strlen(input) + 1);
        if (!String) return;

        while (*input) {
                struct Argument *NewArg = malloc(sizeof(struct Argument));
                char *CurrentChar = String;

                if (!NewArg) break;

                /* Get the Name, terminated by = */
                while (*input) {
                        if (*input == '=') {
                                input++; break;
                        }
                        *CurrentChar++ = *input++;
                }
                *CurrentChar = 0;
                translate(String);
                NewArg->Name = malloc(strlen(String)+1);
                strcpy(NewArg->Name,String);

                /* Get the Data, terminated by EOF or & */
                if (*input) {
                        CurrentChar = String;
                        while (*input) {
                                if (*input == '&') {
                                        input++; break;
                                }
                                *CurrentChar++ = *input++;
                        }
                        *CurrentChar = 0;
                        translate(String);
                        NewArg->Data = malloc(strlen(String)+1);
                        strcpy(NewArg->Data,String);
                } else NewArg->Data = NULL;

                NewArg->Next = *ArgList;
                *ArgList = NewArg;
        }
        free(String);
}

/* Fetch the form arguments from stdin and put them in the linked list
 * named ArgForm.
 * The environment variable CONTENT_LENGTH contains the exact
 * number of bytes we are allowed to read from stdin. */
static void GetStdin(void)
{       char *CLen = getenv("CONTENT_LENGTH");
        char *Buffer;
        int Length;

        if (!CLen) return;
```

```
        Length = atoi(CLen);
        Buffer = malloc(Length+1);        /* Allow for a null terminator */
        if (!Buffer) return;
        {
                char *Writer = Buffer;

                while (Length--) *Writer++ = fgetc(stdin);
                *Writer = 0;        /* Null-terminate the string */
        }
        interpretargs(Buffer,&ArgForm);
        free(Buffer);
}

/* Fetch the URL arguments from the environment and put them
 * in the linked list named ArgURL. */
static void GetURL(void)
{
        char *String = getenv("QUERY_STRING");

        if (String) interpretargs(String,&ArgURL);
}

/* Your program MUST call this function in order to use
 * GetArg() and GetURLArg() since this function gets the
 * form arguments from stdin, fetches the URL arguments from the
 * environment, and creates two nice linked lists containing them.
 */
void InterpretArgs(void)
{
        GetStdin();
        GetURL();
}
```

De enige drie extern zichtbare functies staan gedeclareerd in `cgisup.h`:



cgisup.h

```
char *GetArg(char *Name); /* Use this to get content of a FORM field */
char *GetURLArg(char *Name); /* Use this to parse URL options */
void InterpretArgs(void); /* Call this once at begin of program */
```

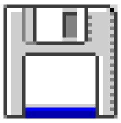
Aan het begin van het programma moet `InterpretArgs()` aangeroepen worden. Die functie leest en verwerkt de binnenkomende parameters. Hierna kan je naar hartelust `GetArg()` aanroepen. De functie `GetURLArg()` hebben we hier niet nodig. De precieze details van deze functies zijn niet zo belangrijk.

Het eerste dat een CGI programma *moet* zeggen is

```
Content-type: text/html
```

gevolgd door een lege regel. Doe je dat niet, dan kan je de meest vreemde foutboodschappen te zien krijgen. Alles wat je daarna naar `stdout` afdrukt, zal aan de gebruiker door de browser getoond worden.

We moeten op een of andere manier ervoor zien te zorgen dat het programma een willekeurig getal kiest, en dat dat getal niet verandert tijdens het raden. Dat is niet zo vanzelfsprekend als het lijkt, want het CGI programma draait alleen heel eventjes telkens als de gebruiker haar gok wil laten verwerken en op OK drukt. De truuk is om het getal in de HTML uitvoer te verbergen in een verborgen parameter `juist`. De eerste keer bestaat er nog geen parameter `juist`, wat voor ons programma het teken is om een getal te kiezen, stel 85. Vanaf dan wordt het gekozen getal in het HTML formulier verborgen door deze pagina naar `stdout` af te drukken:

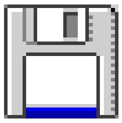


raadhetgetalbis.html

```
<H1>Te laag!</H1>
<FORM METHOD=POST ACTION=raadhetgetal.cgi>
Uw gok: <INPUT NAME=gok>
<P>
<INPUT NAME=juist TYPE=hidden VALUE=85>
<INPUT TYPE=submit VALUE=OK>
</FORM>
```

Door de toevoeging `TYPE=hidden` wordt de parameter `juist` niet aan de gebruiker getoond door de browser.

Het programma dat voor alles zorgt:



raadhetgetal.c

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

#include "cgisup.h"

int main(void)
{
        int juist,gok;

        InterpretArgs();
        puts("Content-type: text/html\n");

        {       /* lees gok van de speler */

                char *gokArg = GetArg("gok");

                if (gokArg == NULL) {
                        puts("<H1>Fout</H1> Parameter 'gok' niet gevonden!");
                        return 10;
                }
                gok = atoi(gokArg);
        }

        {       /* bepaal de oplossing */

                char *juistArg = GetArg("juist");

                if (juistArg == NULL) {
                        /* Dit is de eerste gok.
                         * Kies een willekeurig getal ... */
                        srand(time(NULL));
                        juist = rand() * 100.0 / (RAND_MAX+1.0) + 1;
                } else {
```

```
                        /* Dit is niet de eerste gok.
                         * Haal het eerder gekozen willekeurig getal op. */
                        juist = atoi(juistArg);
                }
        }

        if (juist == gok) {
                puts("<H1>Juist!</H1>\n"
                        "<A HREF=raadhetgetal.html>Nog eens proberen?</A>");
        } else {
                if (gok < juist) puts("<H1>Te laag!</H1>");
                else puts("<H1>Te hoog!</H1>");
                puts("<FORM METHOD=POST ACTION=raadhetgetal.cgi>\n"
                        "Uw gok: <INPUT NAME=gok>\n"
                        "<P>");
                printf("<INPUT NAME=juist TYPE=hidden VALUE=%d>\n",juist);
                puts("<INPUT TYPE=submit VALUE=OK>");
                puts("</FORM>");
        }
        return 0;
}
```

Dit programma moet onder de naam `raadgetal.cgi` gecompileerd worden. Het is het veiligst om een naam te kiezen die op `.cgi` eindigt (sommige WWW serverprogramma's zijn nogal strikt). Om alles aan de praat te krijgen op mijn Unix machine moet ik ook het shell-commando `chmod a+x raadgetal.cgi` geven (anders weigert de WWW server het CGI programma te starten) en een bestandje `.htaccess` maken met de volgende inhoud:

```
Options FollowSymLinks ExecCGI
AddType application/x-httpd-cgi .cgi
```

Deze details hangen een beetje af van het gebruikte systeem.

Hoofdstuk 9

Grote programma's

9.1 Programma's over meerdere bestanden

Eens je programma's wat groter beginnen worden, is het handig ze te bewaren in meer dan één `.c` bestand. Dat heeft een aantal voordelen:

- Je kan ervoor zorgen dat wanneer je het programma wijzigt, alleen de gewijzigde `.c` bestanden moeten opnieuw gecompileerd worden. Dat kan een hoop tijd uitsparen bij het compileren. Het programma `make`, beschreven in §9.2, maakt dit mogelijk.
- Elk bestand kan zijn eigen static extern variabelen hebben, die alleen maar in dat bestand zichtbaar zijn (zie §4.4.4).
- Het is makkelijker kleinere bestandjes te bewerken, waarin bij elkaar horende functies gegroepeerd zitten, dan één monsterbestand waar alle routines in staan.

9.1.1 Voorbeeld: een prettyprinter

Om te laten zien hoe dat alles in de praktijk gebeurt, zal ik een programmaatje schrijven dat een programmatekst op een aantrekkelijke manier afdrukt (een zogenaamd **prettyprinter** programma). Om het simpel te houden zal ik enkel de commentaren in vet laten afdrukken (meer geavanceerde prettyprinters zouden bijvoorbeeld ook alle sleutelwoorden vet kunnen maken). Ik maak eerst een bestand `escape.h` waar de escape sequenties in zitten die omschakelen tussen vetgedrukte en normale tekst:

```
#define TXT_VET     "\033[1m"
#define TXT_NORMAAL "\033[0m"
```

pretty/escape.h

Het programma zal de volgende opties aanvaarden:

- een bestandsnaam, die aangeeft welk bestand het programma moet verwerken
- `tab`, gevolgd door de breedte van de tabstops. Dus `pretty prog.c tab 4` zou ervoor moeten zorgen dat het programma `prog.c` wordt afgedrukt waarbij een tab vier spaties breed is. Als de gebruiker niets opgeeft, gebruiken we 8.
- `breed` en `hoog`, gevolgd door respectievelijk de breedte van het blad (standaard 80 letters) en de hoogte (standaard 66 regels).

- `vanblz` en `totblz`, gevolgd door respectievelijk het bladzijdenummer van de eerste en laatste af te drukken bladzijden.

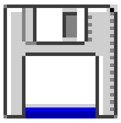
In `argumenten.c` stop ik alles wat met het verwerken van de argumenten te maken heeft:

pretty/argumenten.c

```
#include <string.h>       /* voor strcmp() en strcpy() */
#include <stdlib.h>       /* voor atoi() */
#include <stdio.h>        /* voor FILE */

#include "argumenten.h"

int tab = 8;
int breed = 80;
int hoog = 66;
int vanblz = 1;
int totblz = 10000;
char bestandsnaam[255];

#define strgelijk(a,b) (strcmp(a,b) == 0)

/* Geeft 0 terug als alles OK was */
FILE *verwerkArgumenten(int argc,char **argv)
{
        int tel;
        FILE *bestand = NULL;

        for (tel = 1; tel < argc; tel++) {
                if (strgelijk(argv[tel],"tab"))
                        tab = atoi(argv[++tel]);
                else if (strgelijk(argv[tel],"breed"))
                        breed = atoi(argv[++tel]);
                else if (strgelijk(argv[tel],"hoog"))
                        hoog = atoi(argv[++tel]);
                else if (strgelijk(argv[tel],"vanblz"))
                        vanblz = atoi(argv[++tel]);
                else if (strgelijk(argv[tel],"totblz"))
                        totblz = atoi(argv[++tel]);
                else {  /* argv[tel] bevat bestandsnaam */
                        if (bestand) {
                                /* Als er meer dan een bestand opgegeven
                                 * is, gebruik dan het eerste
                                 * en sla de andere over */
                                printf("Waarschuwing: meer dan een bestandsnaam"
                                        " opgegeven (%s overgeslagen)\n",
                                        argv[tel]);
                        } else {
                                bestand = fopen(argv[tel],"r");
                                strcpy(bestandsnaam,argv[tel]);
                                if (!bestand)
                                        printf("Waarschuwing: kon bestand %s "
                                                "niet lezen\n",argv[tel]);
                        }
```

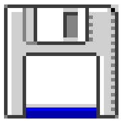
```
                }
        }
        return bestand;
}
```

Alles wat in dit bestand `extern` is (dat zijn hier alle globale variabelen en de functie `verwerkArgumenten()`, ik heb immers geen `static extern` variabelen of functies gemaakt) som ik netjes op in `argumenten.h`:


pretty/argumenten.h

```
#include <stdio.h>    /* voor FILE */

extern int tab,breed,hoog,vanblz,totblz;
extern char bestandsnaam[255];

extern FILE *verwerkArgumenten(int argc,char **argv);
```

Merk op dat ik in `argumenten.c` ook nog een `#include "argumenten.h"` heb geschreven. De compiler ziet dan eerst de `extern` declaraties, en komt vervolgens de declaraties nog een keer tegen (zonder `extern`) zodat hij kan controleren of beide wel overeenstemmen. Ik heb nog altijd geen functie `main()` gemaakt. Die zal ik bewaren in het bestand `main.c`:


pretty/main.c

```
#include <stdio.h>

#include "argumenten.h"
#include "letter.h"

int main(int argc,char **argv)
{
        FILE *bestand;
        int letter;

        if (argc <= 1) {
                /* Er zijn geen argumenten opgegeven; geef een
                 * korte uitleg over het gebruik van het programma */
                puts("pretty bestandsnaam [tab 8][breed 80][hoog 66]"
                        "[vanblz #][totblz #]");
                return 0;
        }
        bestand = verwerkArgumenten(argc,argv);
        if (!bestand) {
                puts("pretty: geen bestand opgegeven, "
                        "of bestand niet leesbaar");
                return 10;
        }
        while( (letter=getc(bestand)) != EOF)
                verwerkLetter(letter);
        return 0;
}
```

Ik doe hier `#include argumenten.h`, zodat de compiler weet welke types de dingen hebben die extern zijn in `argumenten.c`. Ik zou natuurlijk gewoon alleen maar

```
extern FILE *verwerkArgumenten(int argc,char **argv);
```

kunnen schrijven en de `#include` kunnen weglaten (omdat ik de andere dingen uit `argumenten.h` toch niet gebruik in `main.c`). Dat gaat goed tot wanneer je het aantal of het type van de argumenten van `verwerkArgumenten()` verandert tijdens het ontwikkelen van het programma, want dan moet je manueel gaan zoeken in welke bestanden je allemaal `verwerkArgumenten()` nog gebruikt hebt. Je kan jezelf dus heel wat administratie uitsparen door `.h` bestanden te gebruiken. Er ontbreekt nog één ding, namelijk de functie `verwerkLetter()`, die ik in `letter.c` maak:

pretty/letter.c

```
#include "argumenten.h"
#include "escape.h"
#include "letter.h"

static int blz = 1;
static int regel = 0;     /* loopt van 0 t/m hoog-1 */
static int positie = 0;   /* loopt van 0 t/m breed-1 */

/* geeft 1 als deze pagina afgedrukt moet worden, anders 0 */
static int drukmij(void)
{
        return vanblz<=blz && blz<=totblz;
}

static void hoofding(void)
{
        /* %40.40s: maximum 40 letters en aanvullen met spaties tot 40 letters */
        if (drukmij()) printf("%40.40s Bladzijde %5d\n",bestandsnaam,blz);
        regel++;    /* de hoofding neemt 1 regel in */
}

static void drukc(char c)
{
        if (drukmij()) putchar(c);
}

static void druks(char *s)
{
        if (drukmij()) printf("%s",s);
}

void verwerkLetter(char c)
{
        static char vorige = 0;
        static int inCommentaar = 0;

        if (regel == 0) {
                /* Bovenste regel van het blad: druk een hoofding */
                /* Indien nodig vet even onderbreken */
                if (inCommentaar) druks(TXT_NORMAAL);
                hoofding();
```

```
                if (inCommentaar) druks(TXT_VET);
        }
        switch(c) {
                case '\n':
                        drukc(c);
                        positie = 0;
                        regel++;
                        break;
                case '\t':
                        do {
                                drukc(' '); positie++;
                        } while (positie%tab != 0);
                        break;
                case '*':
                        drukc('*');
                        if (!inCommentaar && vorige=='/') {
                                /* Commentaartekst begint */
                                inCommentaar = 1;
                                druks(TXT_VET);
                        }
                        positie++;
                        break;
                case '/':
                        drukc('/');
                        if (inCommentaar && vorige == '*') {
                                /* Einde commentaartekst */
                                inCommentaar = 0;
                                druks(TXT_NORMAAL);
                        }
                        positie++;
                        break;
                default:
                        drukc(c);
                        positie++;
                        break;
        }
        vorige = c;    /* vorige letter onthouden */
        /* Kijk of we over het einde van een regel zijn */
        if (positie == breed) {
                drukc('\n');
                positie = 0;
                regel++;
        }
        /* Kijk of we over het einde van een bladzijde zijn */
        if (regel == hoog) {
                drukc('\f');
                regel = 0;
                blz++;
        }
}
```

en de bijhorende header file `letter.h`:

```
extern void verwerkLetter(char c);
```

pretty/letter.h

Een ongelukkig gevolg van het idee om een functie `verwerkLetter()` te gebruiken, die het bestand letter per letter te zien krijgt, is dat een `/*` niet in het vet wordt afgedrukt, maar `*/` wel. Het was misschien beter geweest het bestand regel per regel te verwerken, waardoor het makkelijker zou zijn om vooruit te kijken. Het probleem is immers dat `verwerkLetter()` de `/` van een `/*` niet in het vet kan drukken omdat er ook nog iets anders dan een `*` kan op volgen. Het lijkt ook nogal moeilijk om `verwerkLetter()` om te bouwen zodat C sleutelwoorden ook in het vet komen, om dezelfde reden.

9.1.2 Compileren

Hoe je zo'n programma dat in verschillende bestanden zit precies compileert, hangt van de compiler af. Bij sommige compilers (bijvoorbeeld die van Borland) moet je een **projekt-bestand** maken (dat je bijvoorbeeld `pretty.prj` kan noemen), waarin alle te compileren bestanden opgesomd staan:

```
argumenten.c
letter.c
main.c
```

Of je kan het je helemaal gemakkelijk maken en volledig via menu's een nieuw project definiëren en daar de `.c` bestanden één voor één aan toevoegen.

Bij de GNU C compiler moet je

```
gcc argumenten.c letter.c main.c -o pretty
```

tikken. Als je alle bestanden van het programma in één directory bewaart (en er zitten dus geen `.c` bestanden in de huidige directory die geen deel uitmaken van het programma), kan je er je snel vanaf maken met

```
gcc *.c -o pretty
```

Bij deze aanpak worden telkens *alle* `.c` bestanden van het programma gecompileerd, ook al heb je maar in één ervan iets veranderd. Je kan ook elk bestand eerst apart compileren:

```
gcc -c letter.c -o letter.o
gcc -c argumenten.c -o argumenten.o
gcc -c main.c -o main.o
```

De `-c` optie geeft aan dat `gcc` alleen maar mag compileren en nog niet mag **link**en. Linken is het aan elkaar plakken van alle stukjes programma (hier de drie `.o` bestanden en natuurlijk ook nog een aantal bibliotheekfuncties die we gebruikt hebben, zoals `printf()` en dergelijke). De ongelinkte stukjes programma heten **objectbestand**en (**object files**) en worden bewaard in bestanden die eindigen op `.o`. Meer over linken vind je in §9.3. (Eigenlijk hoef je de naam van het uitvoerbestand niet te vermelden als je met `-c` compileert—de compiler is slim genoeg om te weten dat je `letter.c` in `letter.o` wilt compileren.) Linken doe je met

```
gcc letter.o argumenten.o main.o -o pretty
```

Als je nu één bestand wijzigt, moet je enkel dat ene bestand hercompileren met `gcc -c` en dan natuurlijk niet vergeten alles opnieuw te linken. Deze aanpak ziet er vrij omslachtig uit, en gelukkig is er het hulpprogramma `make` dat veel werk uit handen kan nemen.

9.2 make

Met `make` kan je het hele compilatieproces automatiseren; het enige dat je zal moeten intikken is

```
make
```

en `make` zal zelf uitzoeken welke bestanden het moet hercompileren en hoe het dat moet doen. Daarvoor gebruikt het informatie uit een bestand dat `makefile` moet heten.

Een `makefile` bestaat uit een aantal *regels* (rules) die als volgt opgebouwd zijn:

doel : *afhankelijke bestanden*
    shell-commando
    ⋮
    shell-commando

De betekenis hiervan is: als één of meer van de *afhankelijke bestanden* gewijzigd zijn, moet *doel* gehercompileerd worden. (`make` bepaalt dat aan de hand van de tijdstippen waarop het doelbestand en de afhankelijke bestanden laatst gewijzigd zijn.) Om het bestand *doel* te compileren zal `make` de opgegeven *shell-commando's* één voor één uitvoeren. Het is jouw verantwoordelijkheid om ervoor te zorgen dat de opgegeven commando's effectief een nieuwe versie van *doel* maken; `make` voert de commando's domweg uit en veronderstelt dat alles dan in orde is.

Belangrijk: vóór de shell-commando's *moet* een tab komen (spaties werken niet)!

Een makefile voor het prettyprinter-programma zou er als volgt kunnen uitzien:

```
pretty: main.o letter.o argumenten.o
	gcc main.o letter.o argumenten.o -o pretty

main.o: main.c escape.h argumenten.h letter.h
	gcc -c main.c -o main.o

argumenten.o: argumenten.c argumenten.h
	gcc -c argumenten.c -o argumenten.o

letter.o: letter.c argumenten.h escape.h letter.h
	gcc -c letter.c -o letter.o
```

`make` zal altijd proberen het doel van de eerste regel te compileren. In ons geval is dat `pretty`; daarvoor moeten eerst `main.o`, `letter.o` en `argumenten.o` in orde zijn. Er bestaan regels voor die drie bestanden, dus zal `make` eerst kijken of `main.o` in orde is; als er sinds de laatste compilatie iets aan `main.c`, `escape.h`, `argumenten.h` of `letter.h` veranderd is, hercompileert `make` het bestand `main.o`. Hetzelfde gebeurt voor de twee andere `.o` bestanden. Als één of meerdere `.o` bestanden gehercompileerd zijn, zal `pretty` ook opnieuw gemaakt worden.

9.2.1 Macro's in make

Met

```
macronaam = vervangtekst
```

kan je een `make`-macro maken; je gebruikt die macro door ergens `$(macronaam)` te schrijven. Als je ergens een echt `$`-teken nodig hebt, moet je `$$` schrijven. De makefile kan dus korter geschreven worden als pretty/makefile

```
CC = gcc -Wall -c
OBJECTEN = main.o argumenten.o letter.o

pretty: $(OBJECTEN)
	gcc $(OBJECTEN) -o pretty

main.o: main.c escape.h argumenten.h letter.h
	$(CC) main.c -o main.o

argumenten.o: argumenten.c argumenten.h
	$(CC) argumenten.c -o argumenten.o

letter.o: letter.c argumenten.h escape.h letter.h
	$(CC) letter.c -o letter.o

clean:
	rm *.o
```

Het is dus mogelijk om bij elke compilatie bepaalde opties aan de compiler te geven: hier heb ik `-Wall` gebruikt; ook `-ggdb` komt wel eens van pas, om overal debug-informatie te genereren (zie §10.2.1).

Tot slot is er nog de regel voor `clean`. Er is geen enkele regel die afhangt van `clean`; als je dus `make` draait, zal nooit `rm *.o` uitgevoerd worden. Je kan echter als argument van `make` opgeven welk doel het moet proberen te maken. Dus `make letter.o` zal alleen het bestand `letter.o` compileren als dat nodig was (maar niet `pretty` opnieuw linken), en `make clean` zal alle `.o` bestanden opkuisen (die zijn immers niet meer nodig eens het programma volledig gelinkt is). Op die manier kan je ingewikkelde reeksen commando's handig inpakken.

9.2.2 `touch`

Als je een bestand met programmacode wijzigt, zal `make` het over het algemeen (als de `makefile` goed opgesteld is natuurlijk) hercompileren. Meestal is dat precies wat je wilt dat er gebeurt, maar soms is het overbodig. Als je bijvoorbeeld enkel een stukje commentaar toegevoegd hebt aan een bestand met C-code, zeg `roodkapje.c`, kan je ervoor zorgen dat `make` niet hercompileert door `touch roodkapje.o` te typen. Het `touch` commando zorgt ervoor dat de datum waarop het aangegeven bestand laatst gewijzigd is, op 'nu' gezet wordt. De datum van laatste wijziging van `roodkapje.o` is daardoor later dan die van `roodkapje.c`; `make` besluit hieruit dat `roodkapje.o` niet opnieuw gecompileerd hoeft te worden.

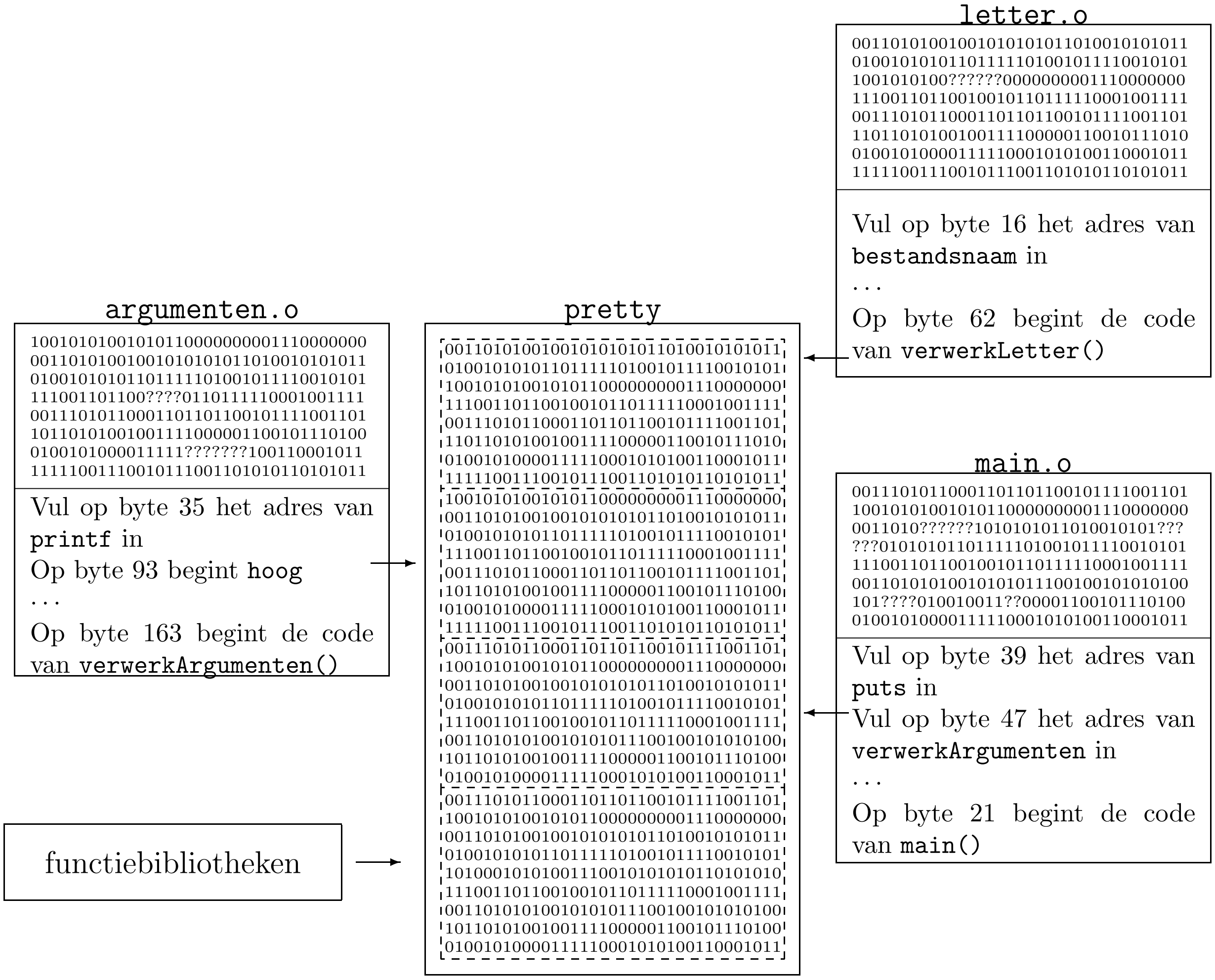
9.3 De linker

In wat voorafging heb je geleerd hoe je variabelen kunt maken die voor meerdere bestanden zichtbaar zijn (maak ze in één bestand met bv `int globaal;` en gebruik ze dan in andere bestanden met `extern int globaal;`) of die alleen maar zichtbaar zijn in het bestand waar ze gemaakt worden (met `static alleenhier;`).

In deze paragraaf onderzoeken we hoe één en ander precies verloopt.

De uitvoer van `gcc -o` is een gecompileerd C bestand. Het bevat uitvoerbare code (binaire dingen die de processor direkt verstaat), die nog niet helemaal 'af' is. Er zitten nog verwijzingen in naar variabelen en functies (kortweg **symbolen** genoemd) uit andere bestanden. In `letter.o` van de prettyprinter wordt bijvoorbeeld gebruik gemaakt van `putchar()`, `printf()`, `bestandsnaam`, ... Naast de binaire codes voor de instructies die de processor begrijpt, zit er in `letter.o` dus ook nog een tabel met informatie als "op byte 16 moet het adres van `bestandsnaam` ingevuld worden" (in de figuur heb ik dat voorgesteld door wat vraagtekens te schrijven tussen de enen en nullen) of "de variabele `breed` begint op byte 60".

Helemaal op het einde worden alle `.o` bestanden en functiebibliotheken aan elkaar geplakt. Het resultaat is een **uitvoerbaar programma** (`pretty` in ons geval) waarin alle verwijzingen ingevuld zijn. Dit aaneenplakken en invullen heet **linken** en wordt gedaan door een programma dat de **linker** heet. De linker van GNU C heet `ld`, maar je zal die waarschijnlijk nooit zelf opstarten, omdat de compiler `gcc` dat zelf doet wanneer het nodig is.



Figuur 9.1: Werking van de linker

De linker vindt bijvoorbeeld dat in `letter.o` ergens het adres van het symbool `bestandsnaam` (of het nu een functie of een variabele is, is onbelangrijk voor de linker) moet ingevuld worden. De linker zoekt in alle opgegeven `.o` bestanden en in enkele standaard functiebibliotheken naar dit symbool. In `argumenten.o` wordt dat symbool inderdaad gedefinieerd. De linker weet waar de code van `argumenten.o` zal terechtkomen in het uitvoerbaar bestand `pretty` en kan dus het adres invullen.

Hieruit blijkt ook wat er gebeurt als je een functie gebruikt die je nergens gemaakt hebt, of ergens `extern int bleh` schrijft en in geen enkel bestand een variabele `bleh` maakt: de compiler genereert netjes een `.o` bestand met een verwijzing naar het symbool. Daarna pas doet de linker zijn werk en geeft een foutmelding dat hij een symbool niet kon vinden. De linker trekt zich ook niets aan van het type van zo'n symbool. Vandaar kan je perfect ergens een `char` variabele `bleh` maken en in een ander bestand `extern int bleh` schrijven, met de nodige gevolgen vandien. (Dit probleem is natuurlijk op te lossen door het gebruik van header files, zoals in het voorbeeld.)

9.3.1 `nm`

Met het programma `nm` kan je zelf eens gaan kijken welke symbolen er zoal in een `.o` bestand zitten. Zo geeft `nm letter.o` dit resultaat:

```
         U bestandsnaam
00000000 d blz
         U breed
0000006c t drukc
00000000 t drukmij
00000098 t druks
00000000 t gcc2_compiled.
00000034 t hoofding
         U hoog
00000010 d inCommentaar.16
00000008 d positie
         U printf
         U putchar
00000004 d regel
         U tab
         U totblz
         U vanblz
000000c4 T verwerkLetter
0000000c d vorige.15
```

De linkerkolom geeft het adres van een symbool weer (in hexadecimaal); de tweede kolom geeft het type aan, en de laatste kolom de naam van het symbool. Lokale symbolen (die dus alleen maar binnen dit bestand geldig zijn) hebben een kleine letter in de tweede kolom; symbolen die ook in andere bestanden gebruikt kunnen worden hebben een hoofdletter.

De symbolen `bestandsnaam`, `breed`, `hoog`, `printf`, ... hebben type U (undefined): de linker moet deze symbolen zien te vinden in de andere `.o` bestanden of in de functiebibliotheken.

Het symbool `verwerkLetter` heeft type T. De T staat voor "text". Een programma bestaat namelijk uit twee stukken binaire data: een "text" gedeelte en een "data" gedeelte. Alles in het "text" gedeelte mag tijdens de loop van het programma niet gewijzigd worden. In dit gedeelte staan bijvoorbeeld stringconstanten, maar ook alle machinecode van de functies zelf, zodat een programma niet per ongeluk zijn eigen instructies kan overschrijven.

Een voorbeeld van een symbool in het "data" gedeelte vinden we in de uitvoer van `nm argumenten.o` waar we onder andere deze regels vinden:

```
000000ff C bestandsnaam
00000004 D breed
00000008 D hoog
00000000 D tab
00000010 D totblz
0000000c D vanblz
```

Het D type staat overduidelijk voor “data”. In de binaire soep van `argumenten.o` zijn dus ergens een aantal bytes gereserveerd voor de inhoud van de variabelen `breed`, `hoog`, ... Dit is dan ook de manier waarop globale variabelen geïnitialiseerd worden: het stuk geheugen waar ze staan wordt rechtstreeks uit de binary binnengelezen! Dit verklaart meteen waarom globale variabelen standaard met nul gevuld worden: de compiler moet toch iets in de binary schrijven, en dus kunnen we in één moeite door nullen schrijven. Lokale variabelen daarentegen worden voortdurend aangemaakt wanneer het programma een blok binnengaat, en het zou te veel tijd kosten om die telkens automatisch met nul te vullen.

Het C type tenslotte staat voor “common” en geeft aan dat dit symbool moet wijzen naar een blok niet-geïnitialiseerd geheugen. De grootte ervan in bytes staat overigens in de eerste kolom. Als we in `argumenten.c`

char bestandsnaam[255] = ”test”;

zouden geschreven hebben, dan was dit symbool ook van het type D geweest, en dan zouden er 255 bytes in de binary `pretty` gestaan hebben met daarin de string `test` gevolgd door een paar honderd nulbytes.

9.4 Functiebibliotheken

Een functiebibliotheek is eigenlijk een bestand waarin een aantal `.o` bestanden verpakt zitten. Er zijn twee varianten:

- **Statische bibliotheken**. Als je een programma linkt met een statische bibliotheek, dan wordt de inhoud ervan in de binary van het programma opgenomen. Een statische bibliotheek wordt in een bestand opgeslagen waarvan de naam op `.a` eindigt.

- **Dynamische bibliotheken**. Programma’s die met zo’n bibliotheek gelinkt worden bevatten alleen maar een verwijzing naar de bibliotheeknaam. Zo’n programma is dus eigenlijk nog niet helemaal volledig gelinkt na het compileren: tijdens het opstarten ervan zal het systeem naar de dynamische bibliotheek zoeken en ze in het geheugen (de binary op schijf blijft ongewijzigd) aan het programma linken. Dynamische bibliotheken hebben bestandsnamen die op `.so` of `.sa` eindigen (de `s` staat voor ‘shared’).

Als je tien programma’s maakt die een bepaalde statische bibliotheek gebruiken, dan zullen ze alle tien een kopie van de inhoud ervan bevatten. Op je schijf staat dus tien keer hetzelfde. Erger dan deze plaatsverspilling is dat, wanneer je een fout vindt in de bibliotheek, je al die tien programma’s opnieuw moet compileren. Als je daarentegen een dynamische bibliotheek gebruikt, zullen de gecompileerde programma’s veel kleiner zijn (ze bevatten enkel een verwijzing naar de bibliotheeknaam), en als er fouten verbeterd worden in de bibliotheek, dan profiteren alle programma’s die ervan gebruik maken daar onmiddellijk van. Een nadeel van dynamische bibliotheken is dat de gebruiker natuurlijk wel die bibliotheek op zijn systeem moet geïnstalleerd hebben als hij jouw programma wil kunnen gebruiken.

In de praktijk zijn bijna alle functiebibliotheken van de dynamische soort.

9.4.1 Zelf een functiebiblioheek maken

We zullen hier zelf een functiebibliotheek ontwikkelen die we de welluidende naam `libtest` zullen geven. Het is de gewoonte dat de naam van een functiebibliotheek met `lib` begint (van **library**).

In de bibliotheek zal ik twee bestanden zetten: `libtest.o` en `testfunctie.o`. Ze zijn afkomstig van de overeenkomstige C bestanden:

dl/libtest.c

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

void beschrijving(void) {
        puts("Dit is een testlibrary.");
}
```

dl/testfunctie.c

```
#include <string.h>
#include "testfunctie.h"

int testfunctie(char *str) {
        return strlen(str);
}
```

De bestanden `libtest.h` en `testfunctie.h` bevatten elk één regel tekst met daarin het prototype van de functie `beschrijving()` resp. `testfunctie()`.

De `.o` bestanden maken we op de klassieke manier, maar we moeten wel de optie `-fPIC` vermelden:

```
gcc -Wall -fPIC -c libtest.c
gcc -Wall -fPIC -c testfunctie.c
```

De library maken we met

```
gcc -shared -Wl,-soname,libtest.so.1 -o libtest.so.1.0 libtest.o testfunctie.o
```

Dat is een hele boterham. De `-o libtest.so.1.0` geeft net als vroeger aan wat de naam van het resulterende bestand (onze functiebibliotheek dus) zal zijn: `libtest.so.1.0` (de 1.0 is het versienummer). Daarna volgen alle objectbestanden die in de bibliotheek moeten gestopt worden. Merk op dat ik niet `libtest.so` als bestandsnaam heb gekozen, maar `libtest.so.1.0`. Daarom moeten er ook een paar links gelegd worden met deze Unix commando's:

```
ln -s libtest.so.1.0 libtest.so.1
ln -s libtest.so.1 libtest.so
```

Dit zorgt ervoor dat als het systeem het bestand `libtest.so.1` wil lezen, er in werkelijkheid bij `libtest.so` wordt gekeken, en als het `libtest.so.1.0` wil lezen, het bij `libtest.so.1` gaat kijken (en dus uiteindelijk weer bij `libtest.so` uitkomt). Deze ietwat ingewikkelde constructie is nodig om alles goed te doen blijven werken als er nieuwe versies van de library ontwikkeld worden. Het idee is dat kleine aanpassingen alleen het laatste getal van het versienummer ophogen (1.1, 1.2, enzovoort). Met 'kleine aanpassingen' wordt bedoeld dat programma's die met de vorige versie werkten ook de nieuwe versie kunnen gebruiken.

Aanpassingen die ervoor zorgen dat de library niet meer compatibel is (bijvoorbeeld omdat een functie verdwijnt of niet meer hetzelfde aantal argumenten heeft) verhogen het eerste getal van het versienummer en zetten het laatste getal van het versienummer op nul (dus van versie 1.4 naar 2.0 bijvoorbeeld).

Even testen met dit programma:

dl/linkmij.c

```
#include "libtest.h"

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

Compileren gaat met

```
gcc -Wall -ltest -L. linkmij.c -o linkmij
```

De optie `-ltest` zorgt ervoor dat de linker naar de `libtest.so` bibliotheek zal kijken. Normaal gezien wordt enkel in een paar standaard directories naar bibliotheken gezocht; de `-L.` optie zorgt ervoor dat de linker ook in de huidige directory (die altijd `.` heet in Unix) zal kijken.

Als je dit programma probeert te draaien, verschijnt er

`linkmij: error in loading shared libraries: libtest.so.1: cannot open shared object file: No such file or directory`

Oeps. We moeten het systeem nog wijsmaken dat het de shared libraries ook in de huidige directory moeten gaan zoeken met

```
export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:.
```

Het alternatief is natuurlijk `libtest.so` naar een van de standaard directories kopiëren (`/lib` of `/usr/lib` bijvoorbeeld), iets dat je als gewone gebruiker meestal niet zomaar mag.

9.4.2 Dynamic loading

Een programma kan ook zelf beslissen wanneer het een bibliotheek inlaadt. In zitten een aantal handige functies:

`void *dlopen (const char *bestandsnaam, int opties);`

Opent een bibliotheek met de opgegeven bestandsnaam. `opties` moet `RTLD_NOW` zijn of `RTLD_LAZY`; in ons geval maakt het niet zoveel uit. Het resultaat is een pointer, die we moeten bijhouden om verderop iets met de bibliotheek te kunnen doen.

`const char *dlerror(void);`

Geeft een string terug die de laatst opgetreden fout bij de dynamic loading functies aangeeft.

`void *dlsym(void *bibliotheek, char *symbool);`

Zoekt een symbool in de opgegeven bibliotheek. `bibliotheek` moet het resultaat van een `dlopen()` aanroep zijn.

`int dlclose (void *bibliotheek);`

Sluit een bibliotheek, zodat het systeem eventueel wat geheugen kan vrijmaken.

In het volgende programma worden al deze functies gedemonstreerd. Het interessante is dat de gebruiker zelf kan kiezen welke bibliotheek er gebruikt wordt door een argument aan het programma mee te geven:

dl/probeer.c

```
#include <dlfcn.h>
#include <stdio.h>
#include <string.h>
#include "libtest.h"
#include "testfunctie.h"

int main(int argc,char **argv) {
        void *library;
        char libname[255];

        if (argc != 2) {
                puts("Gebruik: probeer libnaam");
                return 20;
        }

        sprintf(libname,"%s.so",argv[1]);

        library = dlopen(libname,RTLD_LAZY);
        if (!library) {
                puts("Kan library niet openen");
                puts(dlerror());
                return 10;
        }

        {       void (*symbool)(void); /* pointer naar ftie */

                symbool = dlsym(library,"beschrijving");
                if (!symbool) {
                        puts("Kan symbool 'beschrijving' niet vinden");
                        return 15;
                }
                (*symbool)();
        }

        {       int (*symbool)(char *); /* pointer naar ftie */

                symbool = dlsym(library,"testfunctie");
                if (!symbool) {
                        puts("Kan symbool 'testfunctie' niet vinden");
                        return 15;
                }
                printf("%d\n",(*symbool)("testfunctie"));
        }

        dlclose(library);
        return 0;
}
```

Compileren gaat met

```
gcc -Wall -ldl probeer.c -o probeer
```

Al de dynamic loading-functies zitten in de `dl` library (vandaar `-ldl`). Merk op dat ik niet link met onze testlibrary: er staat nergens een aanroep van `beschrijving()` of `testfunctie()` in dit programma.

Het programma kan dan gestart worden met `probeer libtest`. We openen de `libtest.so` library en kijken of ze een symbool `beschrijving` bevat. Hopelijk is dat een pointer naar een functie die een string teruggeeft, want er is geen enkele mogelijkheid om dat te controleren. We doen dan hetzelfde met `testfunctie`.

Waarom al die moeite? Deze aanpak laat toe dat iemand anders eigen modules voor het programma `probeer` kan schrijven, zonder dat die de broncode ervan moet hebben. Op soortgelijke manier kan je de WWW-browser Netscape voorzien van eigen **plugins**, waardoor de browser nieuwe bestandstypes kan herkennen. Op die manier hoeven de programmeurs van de browser zich ook niets aan te trekken van de vele makers van de plugins (en omgekeerd), zolang er maar duidelijke afspraken zijn over welke symbolen de plugin-libraries precies moeten bevatten.

9.5 Versiecontrole-systemen

Grote programma's worden vaak door meer dan één programmeur tegelijkertijd geschreven. Het kan hierbij voorkomen dat twee programmeurs eenzelfde broncodebestand willen wijzigen. Zonder speciale tools geeft deze situatie nogal wat problemen; als je bijvoorbeeld met een gewone teksteditor zou werken, dan "wint" diegene die het laatst zijn tekst bewaart. Zulke programma's worden **versiecontrole-systemen** (version control systems) genoemd. Er bestaan een hoop zulke systemen; voorlopig zal ik enkel een van de populairste aanraken:

9.5.1 RCS (Revision Control System)

RCS beheert revisies van bestanden; het maakt het mogelijk om o.a. terug te gaan naar oudere versies, versies samen te voegen, ... RCS zorgt ervoor dat twee programmeurs niet tegelijkertijd hetzelfde bestand kunnen wijzigen. Oudere versies worden op een compacte en efficiënte manier opgeslagen. RCS kan gebruikt worden om gelijk welke tekst te onderhouden die vaak verandert, dus niet alleen programma's maar ook documentatie of grafische bestanden.

RCS bewaart alle bestanden van een project in een subdirectory die `RCS` heet; het eerste dat je dus moet doen om RCS te gaan gebruiken is `mkdir RCS` typen in de shell.

Als je een bestand `roodkapje.c` aan het project wilt toevoegen, moet je het commando `ci roodkapje.c` (check-in) gebruiken. RCS zal het bestand `roodkapje.c` nu beginversie 1.1 geven, bewaren in de RCS subdirectory, en het originele bestand verwijderen. Het idee is dat je dit bestand "aan RCS gegeven hebt". Als jij of een andere programmeur later dit bestand wil zien, doe je een check-out: `co roodkapje.c`. Als je er ook aan wil veranderen, moet je `co -l roodkapje.c` doen. Dit zorgt ervoor dat jij de enige bent die nu een `co -l` van dit bestand kan doen; anderen kunnen het enkel maar lezen, totdat je klaar bent met wijzigen en het bestand weer aan de zorgen van RCS toevertrouwt met een `ci`. Dit systeem heet **vergrendeling** (**locking**, vandaar de `-l`): alleen diegene die het bestand vergrendeld heeft kan eraan werken. Na elke check-out/check-in cyclus wordt het versienummer verhoogd.

Er is natuurlijk nog veel meer over RCS te vertellen, maar dat zou buiten het kader van deze cursus vallen.

9.5.2 CVS—Concurrent Versions System

CVS is een soort geavanceerde RCS. CVS gebruikt hetzelfde bestandsformaat als RCS om de bestanden van een project in te bewaren.

Eén van de belangrijkste voordelen is dat CVS wel toestaat dat twee of meer programmeurs tegelijkertijd aan hetzelfde bestand werken. CVS probeert de verschillende veranderingen zelf samen te voegen; als dat niet lukt, word je gewaarschuwd en kan je zelf ingrijpen.

CVS wordt vaak gebruikt voor grote projecten waarbij veel programmeurs betrokken zijn. Een voorbeeld hiervan is het GNOME Project (waarbij geprobeerd wordt een gratis gebruikersvriendelijke desktop te ontwikkelen), waarbij iedereen die geïnteresseerd is langs kan komen en beginnen meewerken, zonder dat ze elkaar allemaal in de weg lopen.

Meer info kan je op de CVS homepage vinden op `http://www.cyclic.com/cvs/info.html`. CVS is gratis te downloaden op `http://download.cyclic.com/pub/`.

Hoofdstuk 10

Debuggen

This program has something for everyone.
Some people want to find bugs, so I added some.

Het is in de praktijk onmogelijk om volledig foutloze programma's te maken. In dit hoofdstuk bespreek ik technieken en hulpprogramma's om fouten op te sporen.

Fouten in programma's worden **bug**s (letterlijk betekent “bug”: insect) genoemd. Een beroemd verhaaltje hierover vertelt dat in de Mark II, één van de eerste computers, daadwerkelijk een mot in een relais verantwoordelijk was voor een bug. Het logboek waarin het voorval genoteerd stond (*“1545 Relay #70 Panel F (moth) in relay. First actual case of bug being found”*) is zelfs een tijdlang tentoongesteld, samen met de onfortuinlijke mot zelf. Ondanks dit verhaaltje staat het vast dat het woord “bug” al veel vroeger in deze betekenis gebruikt werd.

10.1 `assert.h`

In `assert.h` zit de `assert()` macro die zich gedraagt als

```
        void assert(int uitdrukking);
```

Als de `uitdrukking` niet waar is (m.a.w. nul) dan wordt een foutboodschap afgedrukt naar `stderr` en breekt het programma af. De precieze foutboodschap hangt af van de gebruikte compiler, en bevat meestal een aanduiding van de plaats in het programma waar de `assert()` opdracht staat:

```
prog: prog.c:5: main: Assertion ‘uitdrukking’ failed.
```

geeft aan dat `assert()` een niet-nul `uitdrukking` kreeg in de functie `main()` in het bestand `prog.c` op regel 5.

Deze macro is handig om te testen op “onmogelijke” waarden van bepaalde variabelen:

```
void verwerkstring(char *str)
{
        /* het programma is zo gebouwd dat het nooit verwerkstring(NULL)
         * zou mogen doen; deze assert() is er voor het geval dat ... */
        assert(str != NULL);

        /* doe hier van alles met str */
}
```

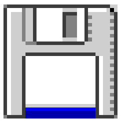
Eens het programma voldoende stabiel draait, zijn de `assert()` opdrachten in feite overbodig (ze vertragen het programma enkel maar en maken het groter). Je kan de `assert()` opdrachten uitschakelen door de macro `NDEBUG` te definiëren vóór de `#include <assert.h>` regel.

10.2 De GNU debugger gdb

Er bestaan hulpprogramma's om bugs helpen op te sporen: debuggers. Ik zal hier `gdb` (versie 4.16) bespreken; de algemene principes hiervan gaan natuurlijk ook op voor andere debuggers.

10.2.1 Post-mortem debuggen

Laten we het foute programmaatje uit §3.6 eens debuggen. Ik onderstel dat het bestand `stringconst.c` er zo uitziet:



stringconst.c

```
#include <stdio.h>
int main(void)
{
        char *s = "strinch";

        s[5] = 'g';
        s[6] = 0;
        puts(s);
        return 0;
}
```

Compileren gaat met `gcc stringconst.c -o stringconst` en de debugger starten met

`gdb` *programmanaam*

in ons geval dus `gdb stringconst`. Merk op dat `gdb` dus op uitvoerbare programmabestanden (binaries) werkt en niet met de broncode. Na de welkomstmelding verschijnt de `gdb` prompt `(gdb)`. Je kan nu een commando ingeven (bijvoorbeeld `help`); om het programma te starten tik je `run` en druk je op Enter. Bijna onmiddelijk treedt de segmentation fault op, die ditmaal netjes door `gdb` onderschept wordt:

```
Program received signal SIGSEGV, Segmentation fault.
0x8048487 in main ()
```

`gdb` toont in welke functie en in welke instructie de fout optrad. In ons geval was dat de instructie op adres `0x8048487`, ergens in de functie `main()`. Het zou natuurlijk leuk zijn om te weten met welke instructie uit de broncode dat overeenkwam. Maar omdat `gdb` alleen maar het bestand `stringconst` heeft (een binary) kan `gdb` dat niet weten. Gelukkig is er een oplossing: je kan de compiler extra debug-informatie aan de binary laten toevoegen met de optie `-ggdb`. De binary wordt daardoor natuurlijk wel flink groter, dus eens de debug-fase voorbij schakel je deze optie het best weer uit. (Je kan ook manueel de debug-informatie verwijderen met het `strip` commando.) Ga dus uit `gdb` door `quit` te tikken. Je krijgt de melding

```
The program is running.  Quit anyway (and kill it)? (y or n)
```

`gdb` heeft het programma immers niet beëindigd maar alleen maar tijdelijk stilgelegd. Geef `y` en hercompileer het programma met `gcc stringconst.c -ggdb -o stringconst` Als je nu het programma weer debugt, is de foutmelding dit maal:

```
Program received signal SIGSEGV, Segmentation fault.
0x8048147 in main () at stringconst.c:5
5               s[5] = 'g';
```

Je krijgt dus netjes de plaats van de fout te zien: regel 5 van bestand `stringconst.c`.

We kunnen ook eens kijken naar de inhoud van de variabelen met het `print` commando. Tik bijvoorbeeld `print s` en je krijgt

```
$1 = 0x8055438 "strinch"
```

Hieruit blijkt dat de opdracht `s[5] = 'g';` mislukt is. De uitdrukking na `print` kan redelijk complex zijn; je kan zelfs subroutines aanroepen, wat handig kan zijn als je een routine hebt die een complexe datastructuur afdrukt (bijvoorbeeld `print drukGelinkteLijst(l)` in een programma dat met lijsten werkt).

Deze manier van debuggen wordt **post-mortem** genoemd (letterlijk: na het overlijden) omdat het programma pas geïnspecteerd wordt nadat het gecrasht is.

10.2.2 Interactief debuggen

Als voorbeeld zal ik het prettyprinter programma debuggen, en meer bepaald de `verwerkArgumenten()` routine. Ik wil natuurlijk dat het programma niet netjes doorloopt tot het crasht, maar dat het stopt van zodra het aan de functie `verwerkArgumenten()` begint. Dat kan door een **breakpoint** te plaatsen:

```
(gdb) break verwerkArgumenten
```

Je kan ook een breakpoint plaatsen op een willekeurige regel in een willekeurig bestand door `break` *bestandsnaam*`:`*regelnummer* te geven.

Het programma starten we weer met `run`, waarbij we ook argumenten kunnen doorgeven:

```
(gdb) run tab 4 breed 60 letter.c
```

Het programma draait totdat het op het breakpoint botst:

```
20              FILE *bestand = NULL;
```

De debugger toont de regel waar het programma stopt, dat is de eerste regel die uitgevoerd zal worden als het programma verder loopt. Hier is `bestand` dus nog niet geïnitialiseerd. Met `step` kan je de loop van het programma regel per regel volgen:

```
(gdb) step
22              for (tel = 1; tel < argc; tel++) {
```

Op dit punt is `bestand = NULL` dus uitgevoerd. Met `step` volg je ook het programma als het in een subroutine springt. Wil je alleen het verloop binnen de huidige routine volgen, dan kan je `next` gebruiken. Na `step` of `next` kan je een getal schrijven om meer dan één instructie per keer te volgen; dus `step 5` doet hetzelfde als vijf keer een `step` geven.

Je kan alleen variabelen bekijken van de "huidige" functie. In dit geval kan ik bijvoorbeeld niet kijken naar de `letter` variabele van `main()`. Met `bt` (backtrace) kan je zien welke functie de huidige functie heeft aangeroepen:

```
(gdb) bt
#0  verwerkArgumenten (argc=6, argv=0xbffff70c) at argumenten.c:22
#1  0x804864d in main (argc=6, argv=0xbffff70c) at main.c:19
#2  0x80480eb in ___crt_dummy__ ()
```

Hieruit kan je zien dat `main()` op regel 19 de functie `verwerkArugmenten()` heeft aangeroepen. (Je ziet ook dat `main()` zelf ook aangeroepen werd door een functie, die `___crt_dummy__()` blijkt te heten—`crt` staat voor C RunTime).

Met `up` en `down` kan je tussen de verschillende functies overspringen:

```
(gdb) print letter
No symbol "letter in current context.
(gdb) up
#1  0x804864d in main (argc=6, argv=0xbffff70c) at main.c:19
19              bestand = verwerkArgumenten(argc,argv);
(gdb) print letter
$9 = 0
```

Je kan tijdens het debuggen breakpoints toevoegen of verwijderen (door `disable` gevolgd door het nummer van het breakpoint te geven). De verdere loop van het programma kan je dan instructie per instructie volgen (met `step` en/of `next`) of tot aan het volgende breakpoint door `continue` te geven.

10.3 xxgdb

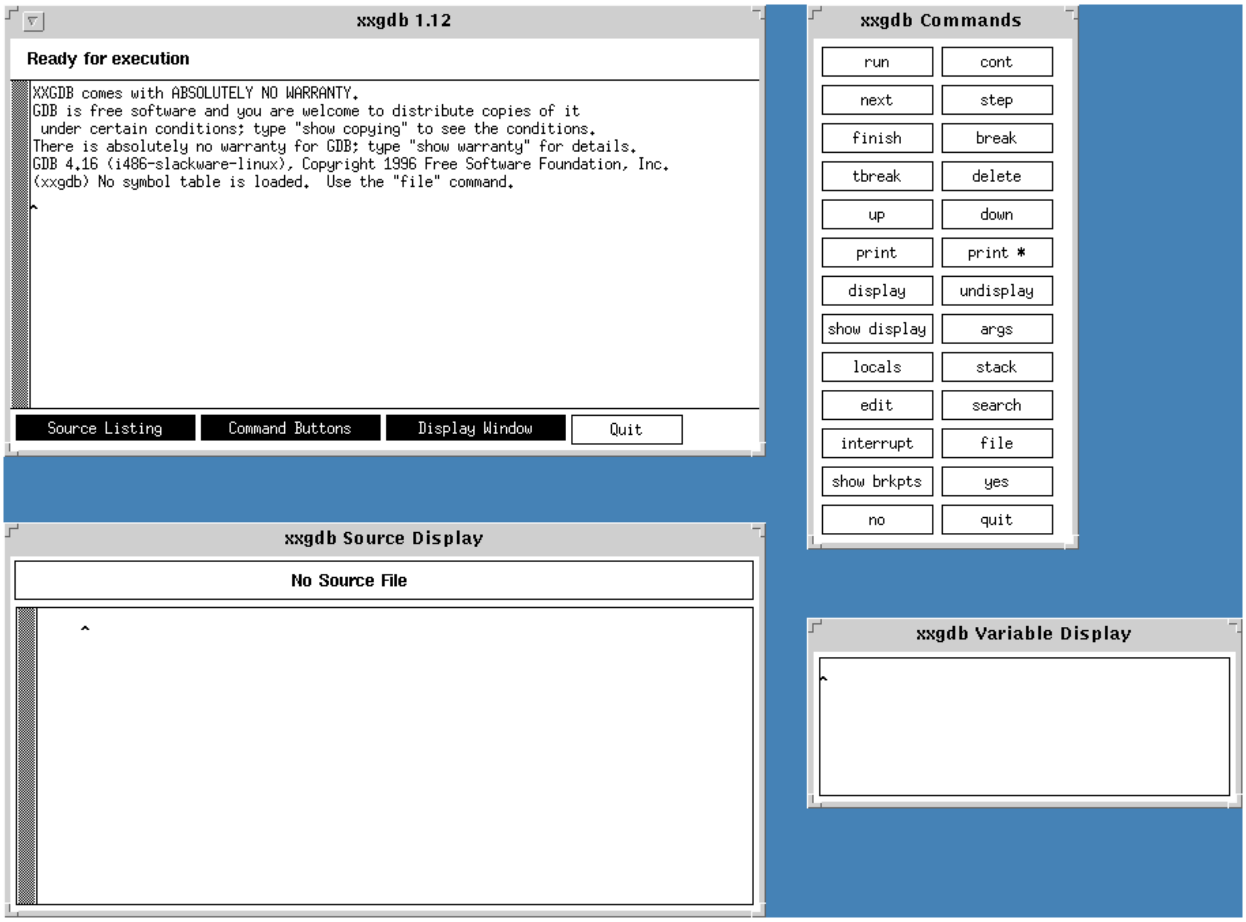
`xxgdb` is een grafische interface voor `gdb` onder X-Windows. De knoppen *Source Listing*, *Command Buttons* en *Display Window* doen drie vensters open en dicht (zie figuur 10.3).

In het *Source Display* venster zie je de broncode van het programma. De plaats waar het programma staat wordt met een blauw pijltje aangeduid (de aangeduide regel is de regel die nog net niet uitgevoerd is), en breakpoints met een rood handje.

Het *Variable Display* venster is handig om variabelen en uitdrukkingen constant in het oog te houden; zo kan je in de `verwerkArgumenten()` routine van daarnet het commando

```
(xxgdb) display argv[tel]
```

commando geven om deze uitdrukking te bekijken terwijl je het programma verder volgt. Met `undisplay` gevolgd door het bijhorende nummer verwijder je een uidrukking uit de lijst.



Figuur 10.1: `xxgdb` klaar voor actie

Hoofdstuk 11

Hardware aansturen met C

In dit hoofdstuk zal ik kort laten zien wat er zoal komt kijken bij het rechtstreeks aansturen van hardware in C. Als voorbeeld zal ik het gebruik van de parallelle poort op intel-systemen nemen.

W A A R S C H U W I N G:

Het spreekt vanzelf dat rechtstreeks aansturen van hardware een slecht idee is als je wil dat meerdere programma's er tegelijkertijd van kunnen gebruik maken: de betere methode is gebruik maken van het **besturingssysteem**. Dat kan er immers voor zorgen dat geen twee programma's tegelijkertijd iets met de parallelle poort proberen doen (stel je voor dat er een printer aan hangt en twee programma's sturen tegelijkertijd iets door). Bovendien ben je zo ook zeker dat het programma blijft werken, ook al verandert de hardware zelf drastisch (als de gebruiker via een netwerk wil printen op een printer die aan een andere computer hangt bijvoorbeeld).

Dit gezegd zijnde: soms is er geen andere mogelijkheid dan rechtstreeks knoeien met bepaalde apparaten, omdat er domweg geen voorzieningen voor zijn in je favoriete besturingssysteem.

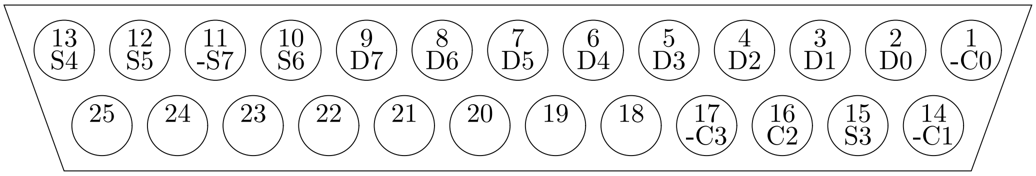
11.1 De hardware van de parallelle poort

De parallelle poort bestaat uit 25 pinnetjes. Elk pinnetje kan aan of uit zijn; er staat dan een spanning op van 0 volt (uit) of +5 volt (aan). Enkele pinnen kan je zelf aan en uit zetten (uitvoerpinnen); van andere pinnen kan je dan weer kijken of ze aan of uit staan (invoerpinnen). Ik zal ook wel eens **lijn** zeggen in plaats van **pin**.

Er zijn een aantal versies van de parallelle poort-specificatie in omloop; de allereerste PC's waren beperkter dan de huidige. Tegenwoordig kan je de meeste uitvoerpinnen ook omschakelen tot invoerpinnen, er zijn voorzieningen om aan extra hoge snelheid informatie te kunnen doorpompen, enzovoort. Wij zullen ons hier beperken tot een vrij conservatief gebruik van de parallelle poort:

- Acht pinnen voor uitvoer, D0 t/m D7 genoemd (D staat voor Data).
- Vier pinnen voor uitvoer, C0 t/m C7 genoemd (C staat voor Control).
- Vijf pinnen voor invoer, S3 t/m S7 genoemd (S staat voor Status).

De pinnen zijn geordend zoals op deze figuur:



Figuur 11.1: Pinnen van de parallelle poort-connector

11.2 Poort-instructies

De intel-processoren sturen de parallelle poort aan met speciale poort-instructies. Andere processoren maken gebruik van een truuk die **memory mapped** invoer/uitvoer heet: de poort wordt gestuurd door op een bepaalde geheugenplaats bepaalde waarden te schrijven of te lezen. Intel gebruikt echter zoals gezegd poort-instructies, die bytes kunnen lezen en schrijven naar en van bepaalde adressen. Het is dus net alsof er naast het gewone geheugen een speciaal 'poorten-geheugen' bestaat, dat alleen maar met de speciale poort-instructies kan bewerkt worden. Blijkbaar vonden de mensen van intel dit een goed idee toen ze hun processor ontwierpen ... Verwarrend genoeg wordt het begrip **poort** dus in twee betekenissen gebruikt: in hardware-zin (een hoop pinnetjes die achteraan je computer te zien zijn) en in software-zin (dingen met adressen waar je bytes kunt instoppen en uithalen).

Een parallelle poort wordt bestuurd door drie bytes met opeenvolgende poort-adressen. Deze drie bytes worden **register**s genoemd. Het beginadres hangt af van welke poort je wil gebruiken (het is immers mogelijk dat je computer meer dan één zo'n poort heeft). Mogelijke waarden zijn meestal `0x3bc`, `0x378` en `0x278`.

> Waarom heten die dingen registers? We zagen eerder al dat de processor ook registers heeft: variabelen die in een stukje geheugen binnenin de processor zelf worden bewaard (en dus niet in één of andere geheugenchip). De registers van de parallelle poort worden binnenin een aparte chip bewaard die zorgt voor het aansturen van de pinnen van de parallelle poort. Vandaar.

We moeten aan het besturingssysteem nu toestemming vragen om die drie poort-adressen te mogen gebruiken. Systemen als ms-dos hebben daar niet veel moeite mee: iedereen mag naar eigen goeddunken poorten lezen en schrijven. Linux is beschaafder en biedt de volgende functie:

```
#include <sys/io.h>
int ioperm(unsigned long beginadres, unsigned long aantal, int aanuit);
```

Hiermee kunnen we vragen een blok poort-adressen te mogen gebruiken. Het eerste adres is `beginadres` en het laatste `beginadres+aantal-1`. Als `aanuit` 1 is, vragen we om deze poort-adressen te mogen gebruiken; als `aanuit` 0 is, geven we aan dat we klaar zijn met het gebruik van de poort-adressen, zodat een ander programma er gebruik van kan maken. Als het programma stopt, worden alle poort-adressen overigens automatisch vrijgegeven.

Het resultaat is 0 als alles goed ging. Alleen gebruiker `root` kan met poorten werken; als een gewone gebruiker dat probeert, zal `ioperm()` zeker mislukken.

In ons geval moeten we dus iets schrijven als

```
    int portbase = 0x3bc;
    if (ioperm(portbase,3,1)) {
          puts("Kan poorten niet krijgen, sorry");
```

```
                return;
        }
```

Merk op dat `ioperm()` een typische Linux-functie is: onder andere besturingssystemen moet je op andere manieren toestemming vragen. (Nog een reden om zo weinig mogelijk proberen rechtstreeks de hardware te gebruiken!)

11.3 De poortregisters

De opbouw van de poortregisters is als volgt:

Adres	Bit 7	6	5	4	3	2	1	Bit 0
0	D7	D6	D5	D4	D3	D2	D1	D0
1	-S7	S6	S5	S4	S3			
2					-C3	C2	-C1	-C0

Vanzelfsprekend moet je bij het adres het beginadres optellen. Bit 7 heeft een waarde van 128, bit 6 een waarde van 64, ..., bit 1 een waarde van 2 en bit 0 een waarde van 1. Zie §4.3.7 voor meer informatie over binaire getallen en binaire operatoren.

De bits waar niets staat ingevuld, hebben geen functie, of dienen voor geavanceerde parallelle poort-functies, die ik niet zal bespreken. De bits waar wel iets staat ingevuld werken als volgt: een 0 in de bit betekent 0V op de overeenkomstige pin, een 1 in de bit betekent +5V, behalve bij de signalen S7, C3, C1 en C0 (er staat telkens een minteken voor). Bij deze vier signalen is het net omgekeerd: een 0 is +5V en een 1 is 0V. Misschien hadden de ontwerpers van de eerste IBM PC de avond ervoor een bijzonder wild feestje gehad of zo ...

11.4 Aansturen van de poorten

11.4.1 Lezen

Deze functie leest een statuslijn:

```
int leesS(const int lijnnr,const int poortbase)
{
        int bitwaarde;
        int reg;   /* inhoud poortregister */

        if (lijnnr<3 || lijnnr>7)
                return -1;   /* ook naar een feestje geweest? */
        bitwaarde = 1 << lijnnr;
        reg = inb(poortbase + 1);
        if (lijnnr == 7) reg = !reg;
        if (reg & bitwaarde) return 1;
        else return 0;
}
```

Met `inb(poortadres)` kan je een byte lezen uit een poortadres. De truuk om te zien of een bepaalde bit aanstaat in `reg` is te kijken wat `reg & bitwaarde` bevat. Dit getal bevat overal nullen behalve op de ene plaats waar `bitwaarde` een 1-bit heeft. Op die plaats is de uitkomst van `&` afhankelijk van de overeenkomende bit van `reg`. Raadpleeg desnoods §4.3.7 om de werking van de bitsgewijze operatoren even op te frissen. Merk op dat de functie rekening houdt met het feit dat S7 geïnverteerd is: als je een 1 terugkrijgt, betekent dat dat er op die pin +5V staat, een 0 betekent dat er 0V op staat.

11.4.2 Schrijven

Met `outb()` kan je een byte schrijven in een poortadres. Omdat we maar één bit per keer willen veranderen, moeten we weten wat de inhoud van de overige 7 bits was:

```
void schrijfD(const int lijnnr,const int aan,const int poortbase)
{
        int bitwaarde;
        int reg;

        if (lijnnr<0 || lijnnr>7) return -1;   /* oeps */
        bitwaarde = 1 << lijnnr;
        reg = inb(poortbase);
        if (aan)   /* bit op 1 zetten */
                outb(reg | bitwaarde,poortbase);
        else        /* bit op 0 zetten */
                outb(reg & ~bitwaarde,poortbase);
}
```

Afhankelijk van de waarde van `aan` zetten we een bit op 0 of op 1. Ook hier moeten we wat goochelen met bits om het juiste effect te bereiken; zie §4.3.7.5 voor meer uitleg.

Merk op dat ik hier veronderstel dat ik de toestand van de datapinnen ook kan *lezen* op hetzelfde poortadres waar ik de toestand in *geschreven* heb. Dit hoeft niet noodzakelijk zo te zijn bij hardware (bij de meeste parallelle poorten is het gelukkig wel zo): sommige registers zijn alleen schrijfbaar (write only) of alleen leesbaar (read only). Het zou ook kunnen dat het schrijven in een register iets helemaal anders doet dan het lezen; de ontwerpers van de parallelle poort zouden ook er net zo goed kunnen voor gekozen hebben dat het register waarin de datalijnen geschreven worden hetzelfde adres had als dat waaruit de status van de statuslijnen gelezen worden.

Bijlage A

ASCII tabel

Op de volgende pagina staat een tabel met de ASCII ("American Standard Code for Information Interchange") codes. In string- en `char`-constanten kan je elk karakter maken met een backslash gevolgd door de octale code; zo is `"\124\145\163\164\056"` net hetzelfde als `"Test."`. Na de backslash mogen naar keuze één, twee of drie octale cijfers komen.

De eerste 32 codes uit de tabel zijn controlecodes. Een aantal veelgebruikte codes zijn als volgt te schrijven in C:

Code	C schrijfwijze	Betekenis
NUL	`\000`	Nul
BEL	`\a`	Audible bell / Alert (maakt een biepgeluid)
BS	`\b`	Backspace
HT	`\t`	Horizontale Tab
LF	`\n`	Line Feed (regelovergang)
VT	`\v`	Vertikale Tab
FF	`\f`	Form Feed (paginaovergang)
CR	`\r`	Wagenterugloop
SP		Spatie
DEL	`\177`	Delete

Wat er precies gebeurt als je een **controlekarakter** afdrukt, hangt af van het uitvoerapparaat en het gebruikte computersysteem. Zo zal op UNIX systemen een LF naar het begin van de volgende regel gaan, terwijl op MS-DOS systemen een LF alleen maar naar de volgende regel gaat (maar de horizontale positie niet verandert), en je expliciet nog een CR moet geven om naar het begin van de regel te gaan. (Meer over de LF/CR problematiek in het hoofdstuk over bestanden.) Op een printer kan je een Backspace gebruiken om twee letters over elkaar af te drukken (bijvoorbeeld `=\b/` om een $\neq$ te maken), maar als je hetzelfde op een scherm probeert, verdwijnt de `=` en er blijft alleen een `/` over. Een Form Feed zorgt ervoor dat de printer een nieuw blad pakt, of het scherm gewist wordt en de cursor linksboven het scherm geplaatst wordt. (Hoewel—op sommige systemen gaat de cursor alleen maar een positie naar beneden(!)) Meestal kan een `char` ook nog andere waarden dan 0 t/m 127 bevatten. De lettertekens die daarmee overeenkomen zijn niet-standaard uitbreidingen van de ASCII karakterset.

b7 b6 b5	0 0 0	0 0 1	0 1 0	0 1 1	1 0 0	1 0 1	1 1 0	1 1 1
BITS b4 b3 b2 b1	CONTROLE		SYMBOLEN CIJFERS		HOOFDLETTERS		KLEINE LETTERS	
0 0 0 0	0 NUL 0 0	16 DLE 10 20	32 SP 20 40	48 0 30 60	64 @ 40 100	80 P 50 120	96 ‘ 60 140	112 p 70 160
0 0 0 1	1 SOH 1 1	17 DC1 11 21	33 ! 21 41	49 1 31 61	65 A 41 101	81 Q 51 121	97 a 61 141	113 q 71 161
0 0 1 0	2 STX 2 2	18 DC2 12 22	34 ” 22 42	50 2 32 62	66 B 42 102	82 R 52 122	98 b 62 142	114 r 72 162
0 0 1 1	3 ETX 3 3	19 DC3 13 23	35 # 23 43	51 3 33 63	67 C 43 103	83 S 53 123	99 c 63 143	115 s 73 163
0 1 0 0	4 EOT 4 4	20 DC4 14 24	36 $ 24 44	52 4 34 64	68 D 44 104	84 T 54 124	100 d 64 144	116 t 74 164
0 1 0 1	5 ENQ 5 5	21 NAK 15 25	37 % 25 45	53 5 35 65	69 E 45 105	85 U 55 125	101 e 65 145	117 u 75 165
0 1 1 0	6 ACK 6 6	22 SYN 16 26	38 & 26 46	54 6 36 66	70 F 46 106	86 V 56 126	102 f 66 146	118 v 76 166
0 1 1 1	7 BEL 7 7	23 ETB 17 27	39 ’ 27 47	55 7 37 67	71 G 47 107	87 W 57 127	103 g 67 147	119 w 77 167
1 0 0 0	8 BS 8 10	24 CAN 18 30	40 (28 50	56 8 38 70	72 H 48 110	88 X 58 130	104 h 68 150	120 x 78 170
1 0 0 1	9 HT 9 11	25 EM 19 31	41) 29 51	57 9 39 71	73 I 49 111	89 Y 59 131	105 i 69 151	121 y 79 171
1 0 1 0	10 LF A 12	26 SUB 1A 32	42 * 2A 52	58 : 3A 72	74 J 4A 112	90 Z 5A 132	106 j 6A 152	122 z 7A 172
1 0 1 1	11 VT B 13	27 ESC 1B 33	43 + 2B 53	59 ; 3B 73	75 K 4B 113	91 [5B 133	107 k 6B 153	123 { 7B 173
1 1 0 0	12 FF C 14	28 FS 1C 34	44 , 2C 54	60 < 3C 74	76 L 4C 114	92 \ 5C 134	108 l 6C 154	124 | 7C 174
1 1 0 1	13 CR D 15	29 GS 1D 35	45 – 2D 55	61 = 3D 75	77 M 4D 115	93] 5D 135	109 m 6D 155	125 } 7D 175
1 1 1 0	14 SO E 16	30 RS 1E 36	46 . 2E 56	62 > 3E 76	78 N 4E 116	94 ^ 5E 136	110 n 6E 156	126 ~ 7E 176
1 1 1 1	15 SI F 17	31 US 1F 37	47 / 2F 57	63 ? 3F 77	79 O 4F 117	95 _ 5F 137	111 o 6F 157	127 DEL 7F 177

LEGENDE: dec CHAR hex oct

Victor Eijkhout
Dept. of Mathematics
UCLA
Los Angeles CA 90024, USA

Bijlage B

C sleutelwoorden

In deze paragraaf komen alle sleutelwoorden van C aan bod. Van elk geef ik een kort voorbeeld. Meer details vind je in de cursus zelf.

B.1 Controlestructuren

Controlestructuren doorbreken de van-boven-naar-onder volgorde van het programma.

B.1.1 if

Test een voorwaarde en voert die alleen uit als aan die voorwaarde voldaan is. Eventueel kan je ook een `else`-blok opgeven, dat uitgevoerd wordt als aan de voorwaarde niet voldaan is.

```
if (idioot(gebruiker))
        printf("Je bent een idioot, %s. Een complete nietsnut.\n",gebruiker);
else
        printf("Hallo, %s.\n",gebruiker);
```

B.1.2 switch

Voert een blok uit naargelang de waarde van een bepaalde variabele. De variabele mag geen pointer (dus ook geen string) of kommagetal zijn. In het voorbeeld zijn er maar twee blokken, maar het kunnen er zoveel zijn als je maar wilt. De `default` is optioneel.

```
char letter;

switch (letter) {
case 'a': case 'e': case 'i': case 'o': case 'u': case 'y':
        printf("klinker");
        break;
default:
        printf("medeklinker");
        break;
}
```

Met `break` kan je de `switch` verlaten. (`continue` kan je hier overigens niet gebruiken).

B.1.3 return

Verlaat de functie waar het programma zich in bevindt. Als de functie een resultaat teruggeeft, moet na `return` een uitdrukking komen, die als resultaat van de functie wordt gebruikt.

```
int grootste(int a,int b) {
        if (a > b) return a;
        return b;
}
```

B.1.4 Lussen

B.1.4.1 for

`for` is de meest veelzijdige lusconstructie. Je kan achtereenvolgens een initialisatie (wordt geëvalueerd vóór het binnengaan van de lus), een voorwaarde (die getest wordt vóór elke doorloop of ze waar is), en een uitdrukking (die geëvalueerd wordt na elke doorloop).

```
int x;

for (x = 1; x <= 4 ; x++)
        printf("%d ",x);
printf("hoedje van, hoedje van, ...");
```

B.1.4.2 while

```
int x;

x = 1;
while (x <= 4)
        printf("%d ",x++);
```

B.1.4.3 do

```
int x;

x = 1;
do
        printf("%d ",x++);
while (x < 4);
```

In alle drie de lussen kan je een `break` opdracht gebruiken, waardoor de lus direkt stopt, en `continue`, waardoor het programma direkt naar het eind van het lusblok springt en er een nieuwe doorloop begint (er wordt natuurlijk eerst wel gecontroleerd of nog aan de lusvoorwaarde voldaan is).

B.1.5 goto

Niet gebruiken. Niet. Nooit dus.

B.2 Datatypes

B.2.1 Enkelvoudige datatypes

Gehele datatypes, van minst naar meest aantal cijfers:

```
char short int long
```

Standaard zijn ze `signed` (behalve bij `char`); door `unsigned` vóór het type te schrijven kies je voor types waar enkel positieve getallen in passen. Wil je ook negatieve getallen kunnen opslaan, dan kan je expliciet `unsigned` schrijven.

Niet-standaard is `long long`.

Kommagetal-datatypes, van minst naar meest nauwkeurig:

```
float double
```

Niet-standaard is `long double`.

Tenslotte is er nog `void`. Het kan alleen in combinatie met pointers of functies gebruikt worden: er bestaan geen variabelen van het type `void`, maar wel functies die `void` teruggeven of pointers naar `void`.

B.2.2 Samengestelde datatypes

Met `*` kan je pointer-datatypes maken; zo is

```
char *
```

een pointer naar een `char`.

Met `[]` maak je arrays:

```
char []
```

maak je een array van `char`s. Merk op dat een eventuele variabelnaam vóór de haakjes komt:

```
char letters[3];
```

maakt een array van drie `char`s. Let erop dat ik hier een variabele maak (met naam `letters`), terwijl ik in de eerste twee voorbeelden enkel namen van types heb opgeschreven.

B.2.3 Eigen datatypes

Met `struct` kan je een datatype maken dat bestaat uit een aantal andere types achter elkaar:

```
struct mandje {
        float gewicht;
        char *eigenaar;
};
```

maakt een nieuw datatype met als naam `struct mandje`.

`union` gedraagt zich net als `struct`, maar alle deelvariabelen ervan liggen 'op' elkaar en niet 'achter' elkaar. Er kan dus maar één zo'n deelvariabele 'actief' zijn, en ze door elkaar gebruiken kan vreemde resultaten opleveren.

Met `typedef` kan je een andere naam geven aan een type:

```
typedef char ** pointerNaarPointerNaarChar;
```

maakt een nieuw type aan, dat je zo kan gebruiken:

```
pointerNaarPointerNaarChar c;
```

Deze regel doet in alle opzichten hetzelfde als `char **c;`

Met `enum` kan je types maken die synoniem zijn van `int`:

```
enum keuze { JA, NEE, MISSCHIEN };
```

maakt een type `enum keuze` dat bedoeld is om in principe enkel de waarden `JA`, `NEE` en `MISSCHIEN` te bevatten.

B.2.4 Opslagklassen

Elke variabele heeft altijd een opslagklasse, die bepaalt hoe en wanneer ze aangemaakt en vernietigd wordt.

B.2.4.1 Lokale variabelen

- `auto`: dit is de standaard opslagklasse voor lokale variabelen. De variabele wordt gemaakt aan het begin van een blok en verdwijnt vanzelf aan het eind ervan.
- `register`: hetzelfde als `auto`, maar vraagt de compiler om extra moeite te doen deze variabele niet ergens in het geheugen te plaatsen, maar in een register van de processor zelf. `register` variabelen hebben bijgevolg geen adres.
- `static`: de variabele blijft leven, ook nadat het blok waarin ze gemaakt wordt verlaten is. Als het blok later nog een keer uitgevoerd wordt, heeft de variabele de waarde van de vorige keer behouden.

B.2.4.2 Globale variabelen en functies

Voor het gemak zal ik het altijd over variabelen hebben; alles gaat ook op voor functies. De benamingen van de opslagklassen komen niet altijd overeen met de sleutelwoorden die je ervoor moet intikken, wat een beetje verwarrend kan zijn.

- **extern**: dit is de standaard opslagklasse. De variabele is niet alleen zichtbaar binnen het bestand waar ze gemaakt is, maar ook in andere bestanden. Andere bestanden die zo'n variabele willen kunnen zien, moeten expliciet het `extern` sleutelwoord vermelden. Het bestand waarin de variabele gemaakt wordt, mag geen opslagklasse-sleutelwoord hebben.
- **static extern**: deze opslagklasse wordt aangeduid door het `static` sleutelwoord. Variabelen van deze opslagklasse zijn enkel zichtbaar binnen het bestand waarin ze gemaakt worden.

B.2.4.3 `const` en `volatile`

Dit zijn niet geen opslagklassen in de echte zin van het woord (ze bepalen helemaal niet wanneer variabelen beginnen of ophouden te bestaan) maar deze sleutelwoorden komen wel voor op dezelfde plaatsen als de echte opslagklassen, vandaar. Ze geven aan dat een variabele niet meer van waarde kan veranderen (`const`) of dat veranderingen aan de waarde ervan allemaal effectief naar het geheugen moeten worden geschreven (`volatile`).

B.2.5 Grootte van een datatype

Met `sizeof()` kan je te weten komen hoeveel bytes een bepaalde variabele of een bepaald type inneemt. Tussen de haakjes moet de naam van de variabele of het type komen:

```
char **c;
typedef char ** ppc;

void test() {
        /* Dit programma drukt drie keer het aantal bytes af
         * dat een pointervariabele inneemt */
        printf("%d\n",sizeof(c));
        printf("%d\n",sizeof(char **));
        printf("%d\n",sizeof(ppc));
}
```

Bijlage C

Oplossingen van de oefeningen

He wished that someone would come and tell him that it was all wrong so that he could shout at them and feel better.
—DOUGLAS ADAMS, "The Hitch Hiker's Guide to the Galaxy"

▸ **OEFENING 2.1** **Blz. 11**

Een syntactisch frivoliteitje van C is dat "**lege declaraties**" toegestaan zijn. Dat zoiets niet helemaal zo gek is als op het eerste gezicht lijkt, zou uit de volgende uitleg moeten blijken.

Je mag immers schrijven:

```
struct mijnstruct { int x; } y;  /* een declaratie */
```

Maar de variabelnaam `y` is niet verplicht:

```
struct mijnstruct { int x; };    /* definitie van een struct */
```

dus de variabelnaam mag weggelaten worden. Aan de andere kant mag het type van een variabele óók weggelaten worden; de compiler kiest dan `int`. Dit is een restant van de manier waarop functies gedefinieerd werden in pre-ANSI C, waar het sleutelwoord `void` nog niet bestond. Een functiedefinitie als

```
druksterren()
{
        int i;
        for (i = 0; i < 80; i++) putchar('*');
        putchar('\n');
}
```

maakt eigenlijk een functie die een `int` teruggeeft.

Het blijkt dus dat zowel type- als variabelnaam in sommige gevallen weggelaten mogen worden. Om de syntax niet te ingewikkeld te maken, mogen ze ook allebei weggelaten worden. Als je dus een kommapunt na een functiedefinitie schrijft, doe je in feite hetzelfde als `int ;` schrijven, wat een "**lege declaratie**" genoemd wordt.

▸ **OEFENING 2.2** **Blz. 12**

`zoekZakdoek` is een variabele van het type "pointer naar een functie" (om precies te zijn: het type is `void (*)(void)`), dus de opdracht `zoekZakdoek;` is net zo nutteloos als `x;`, `"grootmoeder";` of 5+3; en een goede compiler zal dan ook helemaal geen code genereren voor die opdracht. (De GNU C compiler geeft desgevraagd de waarschuwing `statement with no effect`—zie bijlage E).

Veel moeilijker om vinden is

```
        if (zakdoekGevonden)
                snuitNeus();
        else
                abort();
```

want de voorwaarde is steeds waar (`zakdoekGevonden` is een pointer naar een bestaande functie, en dus niet gelijk aan `NULL`), en de `abort()` zal nooit aangeroepen worden. (Ook hier zou een goede compiler een waarschuwing moeten geven—GNU C trapt erin en zwijgt zedig.)

▸ **OEFENING 2.3** **Blz. 14**

De functie `printf()` zou bijvoorbeeld niet echt kunnen bestaan, maar een macro in `stdio.h` kunnen zijn die er zo uitziet:

```
#define printf __internal__printf
```

Een reden voor deze aanpak zou kunnen zijn dat de compiler zo bepaalde controles kan doorvoeren (bijvoorbeeld controleren of alle argumenten van het type zijn dat de formaatstring aangeeft) of optimalisaties kan aanbrengen (bijvoorbeeld bij `printf("test");` moet het hele argument niet op `%`-tekens gescand te worden en zou een snellere variant kunnen gebruikt worden).

Over het algemeen zal de functie `printf()` gewoon in de standaard functiebibliotheek zitten. Zonder `stdio.h` zal de compiler wel geen type checking kunnen doen (in dit geval: controleren of het eerste argument een string is).

▸ **OEFENING 2.4** **Blz. 14**

De functie `printf()` krijgt uitvoer op het scherm door op een of andere manier een functie van het besturingssysteem zelf aan te roepen. Dat gebeurt in elk besturingssysteem op een andere manier. De vitters in het publiek zouden misschien kunnen opwerpen dat `printf()` net zo goed de functie `putchar()` zou kunnen gebruiken, maar dat is gewoon het probleem verschuiven, want ... hoe krijgt `putchar()` dan een letter op het scherm?

▸ **OEFENING 2.5** **Blz. 15**

`test` is een commando dat in de shell zelf ingebakken is, d.w.z. in tegenstelling tot gewone commando's (zoals `ls` of `roodkapje`) gaat de shell *niet* op zoek naar het programma `test`. (Doe maar eens `man test`.) De oplossing is expliciet aan te geven dat de shell in *deze* directory moet gaan zoeken (en dus geen interne commando's mag gebruiken) door `./test` te tikken.

Dit soort eigenaardigheden van de shell kan wel even duren om door te hebben als je er niet op bedacht bent ...

▸ **OEFENING 2.6** **Blz. 19**

De truuk is een derde tijdelijke variabele te gebruiken:

```
{
        int x;
        int y;
        int wissel;

        wissel = x;
        /* De waarde van x is in veiligheid gebracht
```

```
         * dus we mogen x gerust overschrijven: */
        x = y;
        y = wissel;
}
```

▸ OEFENING 2.7 **Blz. 21**

Het programma gaat `main()` in en maakt de variabele `str` aan. De inhoud ervan is volledig onbepaald:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?

Na `strcpy(str,"Dag grootmoeder!\n"))`:

’D’	’a’	’g’	’ ’	’g’	’r’	’o’	’o’	’t’	’m’	’o’	’e’	’d’	’e’	’r’	’!’	’\n’	0	?	?

Merk op dat `\n` wel degelijk maar één byte in beslag neemt. Je kan alles natuurlijk ook zo schrijven:

68	97	103	32	103	114	111	111	116	109	111	101	100	101	114	33	10	0	?	?

Je kunt zelf wel uitmaken welke vorm je het duidelijkst vindt ... (Bovendien is in de tweede versie verondersteld dat het systeem de ASCII code gebruikt, dus portabel is het ook al niet).

Vervolgens het resultaat van `str[12] = str[15]` (`str[15]` is het uitroepteken na `grootmoeder`):

’D’	’a’	’g’	’ ’	’g’	’r’	’o’	’o’	’t’	’m’	’o’	’e’	’!’	’e’	’r’	’!’	’\n’	0	?	?

`strcpy(str,"Aan")` overschrijft de eerste *vier* bytes van de string (denk om de nulbyte):

’A’	’a’	’n’	0	’g’	’r’	’o’	’o’	’t’	’m’	’o’	’e’	’!’	’e’	’r’	’!’	’\n’	0	?	?

De string is nu dus drie letters lang (de nulbyte geeft immers het einde van de string aan; wat er achter komt heeft geen belang voor C strings); `str[3] = ' '` maakt die string weer 17 letters lang:

’A’	’a’	’n’	’ ’	’g’	’r’	’o’	’o’	’t’	’m’	’o’	’e’	’!’	’e’	’r’	’!’	’\n’	0	?	?

en tenslotte nog `str[13]=0`:

’A’	’a’	’n’	’ ’	’g’	’r’	’o’	’o’	’t’	’m’	’o’	’e’	’!’	0	’r’	’!’	’\n’	0	?	?

De variabele `str` bevat dus op het einde nog de string `"Aan grootmoe!"` met nog wat rommel erachter, die voor de string zelf niets uitmaakt.

▸ OEFENING 2.8 **Blz. 21**

Je zou natuurlijk op dezelfde manier kunnen te werk gaan als hiervoor en alleen maar `=` vervangen door `strcpy()`:

```
{
        char zus[20];
        char zo[20];
        char wissel[20];

        strcpy(wissel,zus);
```

```
        strcpy(zus,zo);
        strcpy(zo,wissel);
}
```

Het nadeel van deze aanpak is dat we een extra string moeten aanmaken. Als de strings lang zijn kan dat nogal wat geheugen vragen, en verderop zullen we soms niet eens op voorhand weten hoe lang de om te wisselen strings zijn. Efficiënter is om letter per letter om te wisselen:

```
{
        int teller;
        char zus[20];
        char zo[20];
        int zusGedaan; /* 1 als we het einde van deze string bereikt hebben */
        int zoGedaan;

        stringsGewisseld = 0;
        zusGedaan = zoGedaan = 0;
        teller = 0;
        /* de while-lus blijft draaien zolang minstens een van beide
         * strings nog niet volledig verwerkt is */
        while (!zusGedaan || !zoGedaan) {
                /* wissel de 'teller'-ste letter van zus en zo om */
                char wissel;
                wissel = zus[teller];
                zus[teller] = zo[teller];
                zo[teller] = wissel;

                if (zus[teller] == 0) zusGedaan = 1;
                if (zo[teller] == 0) zoGedaan = 1;
                teller = teller + 1;
        }
}
```

We moeten hier natuurlijk wel opletten dat alles goed blijft werken als beide strings niet even lang zijn. In dat geval doet dit stukje programma eigenlijk te veel werk, want het blijft doorwisselen tot ook de langste string eindigt, terwijl het eigenlijk voldoende is om alleen maar de letters van de langste string te blijven kopiëren van zodra de kortste string afgewerkt is.

▸ **OEFENING 2.9** **Blz. 23**

Eenvoudig, je hoeft enkel een `1` in een `3` te veranderen:

```
#include <stdio.h>

void main(void)
{
        int teller;

        teller = 1;
        while (teller <= 100) {
```

```
                printf("%d ",teller);
                teller = teller + 3;
        }
        printf("\n");
}
```

▸ **OEFENING 2.10** **Blz. 23**

Hier is een (heel) klein beetje meer aanpaswerk aan:

```
#include <stdio.h>

void main(void)
{
        int teller;

        teller = 100;
        while (teller >= 1) {
                printf("%d ",teller);
                teller = teller - 1;
        }
        printf("\n");
}
```

▸ **OEFENING 2.11** **Blz. 23**

Er wordt een sterretje afgedrukt voor elke waarde van `teller` tussen 1 en 79, wat dus 79 (en dus geen 80) sterretjes oplevert.

▸ **OEFENING 2.12** **Blz. 24**

```
#include <stdio.h>

void main(void)
{
        int teller;

        teller = 1;
        while (teller <= 10) {
                printf("%d\t%d\t%f\n", teller, teller*teller,
                        1.0/teller);
                teller = teller + 1;
        }
}
```

▸ **OEFENING 2.13** **Blz. 31**

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

```
        while ( teller < 10 )
        {
                printf("Hello");
                teller := teller + 1
        ; }
; }
```

Merk op dat de na de `while`-opdracht er nog een extra kommapunt staat. Dat is geen fout in C omdat je overal **lege opdracht**en mag invoegen.

▸ **OEFENING 2.14** **Blz. 31**

Als er %-tekens in de string zouden staan, verwacht `printf()` extra argumenten. Met

```
#define write(str) printf("%s",str)
```

is het probleem opgelost.

▸ **OEFENING 2.15** **Blz. 32**

`(double)(SCHERM_HOOG/2)` maakt eerst de deling van twee `int`s; het resultaat is dus ook weer een `int`, dus de waarde na de komma wordt afgekapt. Daarna pas wordt er een kommagetal van gemaakt. Als `SCHERM_HOOG` bijvoorbeeld 79 zou zijn, krijgen we `(double)49` wat ons dus `49.0` oplevert.

De uitdrukking `SCHERM_HOOG/2.0` deelt een `int` door een `double`, en `((double)SCHERM_HOOG)/2` een `double` door een `int`. Als minstens één van de argumenten van `/` een `double` is, wordt de deling als kommagetal berekend en het resultaat is natuurlijk ook `double`, dus we zouden hier netjes de waarde `49.5` krijgen.

▸ **OEFENING 2.16** **Blz. 32**

Het effect is dat er `while (teller <= 2*3.141592;)` komt te staan, en een `;` mag niet binnen een voorwaarde staan. De compiler zal dus een foutmelding geven bij het compileren van de `while` opdracht.

Dit soort fouten is heel verraderlijk (de kommapunt staat daar zo natuurlijk en onschuldig) omdat de C compiler de fout pas vindt op de plaats waar `PI` gebruikt wordt, en dus iets zal zeggen als `fout in "while (teller <= 2*PI)"` terwijl je eigenlijk moet gaan zoeken in de `#define` die een eeuwigheid daarvóór in de listing staat . . .

▸ **OEFENING 2.17** **Blz. 32**

Stel bijvoorbeeld dat in `str` een aantal `%`-tekens voorkomen, zoals `"Dit is \%s een \%c slecht voorbeeld"`, dan zou `printf()` nog twee extra argumenten verwachten, die er niet zijn.

> `printf()` zal dan tóch ergens uit het geheugen die twee extra argumenten halen, wat vooral nefast kan zijn voor de `%s` (dat zou toevallig een string van 10000 letters kunnen zijn . . .).

De oplossing is natuurlijk `printf("%s",str)`, of eventueel (als er een return achter mag komen) `puts(str)`.

▸ **OEFENING 2.18** **Blz. 32**

Niet zo moeilijk:

```
void putchar(char letter)
{
        printf("%c",letter);
}
```

▸ **OEFENING 2.19** **Blz. 33**

De functie `printf()` moet eerst controleren of er geen `%`-tekens in de string staan. Bovendien krijgt `printf()` een pointer naar een string door, terwijl `putchar()` al meteen de af te drukken `char` ziet. En `putchar()` weet al op voorhand dat er maar één letter moet afgedrukt worden, terwijl `printf()` op zoek moet naar het einde van de string.

In de praktijk scheelt dat natuurlijk maar fracties van milliseconden, maar als een dergelijke functieaanroep binnen lussen voorkomt, kan dat aardig oplopen ...

▸ **OEFENING 2.20** **Blz. 34**

Om te beginnen wordt de variabele `totaal` niet geïnitialiseerd. Bovendien gaat het programma flink de mist in als de argeloze gebruiker iets anders dan getallen intikt (bijvoorbeeld `help`) ...

Je kan dit oplossen als je weet dat `scanf()` eigenlijk een resultaat teruggeeft, namelijk het aantal verwerkte argumenten. Als de invoer in orde was, is dat in ons geval dus 1 (één goed verwerkte `%f`), anders 0. (Zie §4.3.5.1 voor meer over het weggooien van functieresultaten.)

Een betere versie:

```
#include <stdio.h>

void main(void)
{
        float getal;
        float totaal;

        totaal = 0;
        getal = 1;
        while (getal != 0.0) {
                int ok;

                puts("Tik een getal in (0 om te stoppen)");
                ok = scanf("%f",&getal);
                if (ok == 0) {
                        /* Invoer is geen getal. Eet een letter op: */
                        char eetop;

                        scanf("%c",&eetop);
                } else {
                        /* Invoer was OK; getal bevat de ingetikte waarde */
                        totaal = totaal + getal;
                }
        }
        printf("Totaal: %f\n",totaal);
}
```

Dit is een voorbeeld van **defensief programmeren**: het programma moet niet alleen correct werken met de invoer die het verwacht; het moet ook zinvol reageren op foutieve invoer.

▸ **OEFENING 2.21** **Blz. 37**

Wat dacht je van

```
#define Sin(graden) sin((graden) / 180.0 * PI)
```

Vergeet vooral de haakjes rond `graden` niet (iets als `Sin(1+2)` moet immers liefst ook correct werken) …

▸ **OEFENING 2.22** **Blz. 44**

Laten we voor het gemak onderstellen dat `RAND_MAX` gelijk is aan 10000.

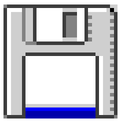
Als `rand()` een getal tussen 0 en 1999 is, dan is het resultaat van `dobbelsteen3()` een 1. Er zijn dus 2000 kansen op 10001 (er zijn 10001 mogelijke resultaten van `rand()`) dat het resultaat een 1 is.

Als `rand()` een getal tussen 2001 en 4000 geeft, dan is het resultaat een 2, dus er is weer een kans van 2000 op 10001 dat het resultaat een 2 is. Hetzelfde geldt voor 3, 4 en 5.

De enige mogelijkheid om een 6 te krijgen is als `rand()` tussen 10000 en 11999 ligt, maar `rand()` is hoogstens `RAND_MAX`. Dus is de enige mogelijkheid om 6 te krijgen dat `rand()` toevallig 10000 teruggeeft.

Met `dobbelsteen3()` hebben we dus een heel oneerlijke dobbelsteen gemaakt: er is maar één kans op 10000 om een zes te gooien, en alle andere getallen hebben ongeveer één kans op vijf om voor te komen.

▸ **OEFENING 2.23** **Blz. 46**


rekenen.c

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

int main(void)
{
        srand(time(NULL));

        while (1 == 1) {
                int oplossing;

                {       /* Opgave afdrukken en oplossing bepalen */
                        int getal1,getal2,bewerking;

                        /* Een getal tussen 0 en 3 om te kiezen tussen
                         * optellen, aftrekken, delen en vermenigvuldigen */
                        bewerking = rand() * 4.0 / (RAND_MAX+1.0);

                        /* Twee getallen tussen 0 en 10 */
                        getal1 = rand() * 11.0 / (RAND_MAX+1.0);
                        getal2 = rand() * 11.0 / (RAND_MAX+1.0);
```

```
                        if (bewerking == 0) {
                                printf("%d + %d",getal1,getal2);
                                oplossing = getal1 + getal2;
                        } else if (bewerking == 1) {
                                printf("%d - %d",getal1+getal2,getal2);
                                oplossing = getal1;
                        } else if (bewerking == 2) {
                                printf("%d * %d",getal1,getal2);
                                oplossing = getal1 * getal2;
                        } else if (bewerking == 3) {
                                printf("%d / %d",getal1*getal2,getal1);
                                oplossing = getal2;
                        }
                }
                printf(" = ");
                {       /* Invoer gebruiker lezen en verwerken */
                        int antwoord;

                        scanf("%d",&antwoord);
                        if (antwoord < 0) break;
                        if (antwoord == oplossing) printf("OK!\n");
                        else printf("Jammer ... het was %d\n",oplossing);
                }
        }
        return 0;
}
```

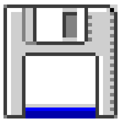
De gebruiker stopt het programma door een negatief getal in te geven.

Aftrekken en delen zijn zo gemaakt dat de oplossing nooit negatief is (bij het aftrekken) of een kommagetal (bij het delen).

Als extraatje kan je het programma de stand laten bijhouden (zoveel oefeningen goed gemaakt, zoveel fout).

▸ OEFENING 2.24

Blz. 46



vularray.c

```
#include <stdlib.h>

/* Vul een array van 'lengte' ints met getallen
 * tussen 0 en max-1 */
void vularray(int array[],int lengte,int max)
{
        int i;

        i = 0;
        while (i < lengte) {
                array[i] = (int)(rand() * (float)max / (RAND_MAX+1.0));
                i = i + 1;
        }
}
```

Uittesten gaat met dit programma:

sorttest.c

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

/* Properder is deze define in een eigen .h file (bubble.h of zo)
 * te steken, maar dat hebben we op dit punt van de cursus nog niet gezien */
#define LENGTE 20

extern int tabel[LENGTE];

void bubbleSort(void);
void selectionSort(void);
void vularray(int arr[],int len,int max);

void printarray(int arr[],int len) {
int i;

        i = 0;
        while (i < len-1) {
        printf("%d ",arr[i]);
                i++;
        }
printf("%d\n",arr[i]);
}

void main(void) {
srand(time(NULL));

puts("Bubble sort");
        vularray(tabel,LENGTE,1000);
        printarray(tabel,LENGTE);
bubbleSort();
printarray(tabel,LENGTE);

puts("Selection sort");
        vularray(tabel,LENGTE,1000);
printarray(tabel,LENGTE);
selectionSort();
printarray(tabel,LENGTE);
}
```

Compileren kan je bijvoorbeeld doen met

```
gcc -Wall sorttest.c vularray.c bubble.c selectionsort.c
```

Het was natuurlijk properder geweest als `sorttest.c`, `bubble.c` en `selectionsort.c` niet drie keer apart `LENGTE` zouden definen, maar in plaats daarvan een header bestand zouden gebruiken waarin die define stond. Dan zou je de lengte van de arrays kunnen veranderen door in dat ene header bestand de aanpassing te maken, in plaats van apart in de drie. Het programma wordt er ook robuuster op, want je kan niet meer per ongeluk de lengte op verschillende waarden instellen in de drie bestanden.

▸ **OEFENING 3.1** **Blz. 48**

```
int main(void)
{
        char namen[10][40];
        int teller;

        teller = 0;
        while (teller < 10) {
                namen[teller][0] = 0; /* of: strcpy(namen[0],""); */
                teller = teller + 1;
        }
        return 0;
}
```

▸ **OEFENING 3.2** **Blz. 49**

```
#define LENGTE 20

int tabel[LENGTE];

double gemiddelde(void)
{
        double totaal;
        int teller;

        totaal = 0;
        teller = 0;
        while (teller < LENGTE) {
                /* in de volgende regel wordt een int automatisch
                 * naar double omgezet door de compiler */
                totaal = totaal + tabel[teller];
                teller = teller + 1;
        }
        return totaal / LENGTE;
}
```

Ik gebruik voor `totaal` geen `int`, want het totaal zou wel eens te groot kunnen zijn om nog in een `int` te passen. In `double`s kan je over het algemeen veel grotere getallen dan in `int`s kwijt (hoewel rekenen ermee wat trager gaat).

OPMERKING: de aanpak die ik in deze oplossing volgde, is niet echt de elegantste. Het is netter om de functie `gemiddelde()` te herschrijven zodat ze op gelijk welke array van `int`s kan werken. Vandaar:

Bouw de functie om tot `double gemiddelde(int tabel[],int lengte)` (in §3.2.3 wordt uitgelegd hoe je arrays als functie-argumenten kunt meegeven).

▸ OEFENING 3.3 **Blz. 49**

```
#define LENGTE 20

int tabel[LENGTE];

int grootste(void)
{
        int grootste;
        int teller;

        grootste = tabel[0];
        teller = 1;
        while (teller < LENGTE) {
                if (tabel[teller] > grootste)
                        grootste = tabel[teller];
                teller = teller + 1;
        }
        return grootste;
}
```

▸ OEFENING 3.4 **Blz. 49**

In de opdracht `x = y;` heeft `y` als type `int *` (want de naam van een array in een uitrdukking geeft een pointer naar het eerste element ervan), en je kan geen pointer stoppen `x`, die een array van `int`s is. Omdat rechts van `=` dus nooit een uitdrukking van het type “array” kan staan, is het zinloos iets proberen in een variabele van een array-type te stoppen.

▸ OEFENING 3.5 **Blz. 51**

Dat zou min of meer duidelijk moeten zijn uit de beschrijving van het algoritme: kleine getallen kunnen maar één plaats per doorloop naar boven bubbelen.

▸ OEFENING 3.6 **Blz. 51**

```
#define LENGTE 20

int tabel[LENGTE];

void bubbleSort(void)
{
        int doorloop;

        doorlooop = 1;
        while (1 == 1) {
                int teller;
                int omgewisseld;

                omgewisseld = 0;    /* houdt bij of er tijdens deze */
                                    /* doorloop iets gebeurd is */
```

```
                teller = 0;
                while (teller < LENGTE-doorloop) {
                        if (tabel[teller] > tabel[teller+1]) {
                                /* omwisselen */
                                int wissel;

                                wissel = tabel[teller];
                                tabel[teller] = tabel[teller+1];
                                tabel[teller+1] = wissel;
                                omgewisseld = 1;
                        }
                        teller = teller + 1;
                }
                if (omgewisseld == 0) break;
                doorloop = doorloop + 1;
        }
}
```

▸ **OEFENING 3.7** **Blz. 58**

```
#include <ctype.h>
void telhoofdklein(char *string,int *hoofd,int *kleine) {
        *hoofd = 0;
        *kleine = 0;

        while (*string) {
                if (isupper(*string)) *hoofd++;
                if (islower(*string++)) *kleine++;
        }
```

Merk op dat ik van de variabele `tel` af ben door handig gebruik te maken van het feit dat `string` zich gedraagt als een lokale variabele die automatisch ingevuld wordt aan het begin van de functie.

Merk ook op dat klein niet altijd duidelijk is; zo had het duidelijker geweest als ik `islower(*string)` had geschreven en een derde opdracht `string++` achteraan aan de lus had toegevoegd.

Kortere oplossingen zijn ook altijd welkom ...

▸ **OEFENING 3.8** **Blz. 60**

Zoals je al kon vermoeden, wordt de waarde 21 in `Tabel1[21]` gestopt. Je kan `daarheen` behandelen *alsof* het een array van 97 `ints` was. En `daarheen[-3]` werkt ook (het is dezelfde `int` als `Tabel1[0]`)!

Uit dit soort voorbeelden blijkt nog maar eens dat het voor de C compiler bijna ondoenbaar is om fouten als `daarheen[-4]` of `daarheen[97]` te signaleren.

Meer gedetailleerde uitleg vind je verder in de tekst.

▸ **OEFENING 3.9** **Blz. 61**

Ik zal gewoon de uitdrukking uiteenrafelen:

`wijs` is een pointer naar de `double` op adres 10000

`wijs+2` is een pointer naar de `double` twee plaatsjes verder, dus de `double` op 10012 (elk

plaatsje is 6 bytes breed)
`*(wijs+2)` is "het ding waar `wijs+2` naar wijst", dus "de `double` op 10012".

Maar wacht eens: dat is net hetzelfde als `wijs[2]`!

Met andere woorden: de `[]` operator hebben we in feite niet echt nodig, we hebben genoeg met `*` en `+`. In het algemeen geldt trouwens: `p[n]` is hetzelfde als `*(p+n)` (hoewel sommige compilers iets als `2[wijs]` weigeren te compileren).

▸ **OEFENING 3.10** **Blz. 63**

De array is `sizeof(tabel)` bytes groot, en elk element neemt `sizeof(int)` bytes in. We krijgen dus

```
int tabel[20];

/* ... */
        while (teller < sizeof(tabel) / sizeof(int)) {
                /* ... */
        }
```

Het is natuurlijk maar de vraag of het handiger tikt dan `LENGTE` :-)

▸ **OEFENING 3.11** **Blz. 65**

De functie `vullen()` krijgt als argument `tabel` een *kopie* van de variabele getallen uit `main()` mee. Als we dus `tabel` doen wijzen naar een vers gereserveerd gebied geheugen, verandert `getallen` helemaal niet. In C worden alle argumenten immers **by value** (zie ook §2.2) doorgegeven aan functies: de functie krijgt een lokale kopie van de argumenten. Als ze die lokale kopie (hier `tabel`) verandert, blijft de variabele uit de aanroepende functie (hier `getallen`) onveranderd.

De *inhoud* van een tabel wordt wel *schijnbaar* by reference doorgegeven:

```
void vullen(int tabel[]) {
        int tel;
        for (tel = 0; tel < 10; tel++)
                tabel[tel] = tel;
}

int main(void) {
        int *getallen = malloc(sizeof(int) * 10);
        vullen(getallen);
        printf("Het 2de element bevat %d\n",getallen[1]);
        return 0;
}
```

Hier krijgt `vullen()` een kopie van de pointer `getallen`. De functie `vullen()` blijft van deze kopie (`tabel`) af, maar verandert wel de inhoud van de array waar `tabel` naar wijst. Vanzelfsprekend is dit hetzelfde stukje geheugen als dat waar `getallen` naar wijst. We kunnen nu iets soortgelijks doen om het probleem op te lossen:

```
tabel *vullen(void) {
        int aantal,tel;
        int *tabel;
```

```
        puts("Aantal getallen?");
        scanf("%d",&aantal);
        tabel = malloc(aantal * sizeof(int));
        /* tabel opvullen */
        for (tel = 0; tel < aantal; tel = tel + 1) {
                int getal;

                printf("Getal %d?\n",tel+1);
                scanf("%d",&getal);
                tabel[tel] = getal;
                /* Kortere maar onduidelijkere versie:
                   scanf("%d",&(tabel[tel])); */
        }
        return tabel;
}

int main(void)
{
        int *getallen;
        getallen = vullen();
        /* hier volgt dan een stuk programma dat
         * de elementen van ‘getallen’ gebruikt */
        return 0;
}
```

Merk op dat in beide versies `main()` op geen enkele manier te weten kan komen hoe groot de tabel is. Om dit probleem op te lossen, zouden we de functiehoofding kunnen aanpassen tot

```
int *vullen(int *grootte)
```

en dan ergens in `vullen()` de opdracht `*grootte = aantal;` schrijven. De tabel zou dan gevuld worden met

```
        int *getallen;
        int grootte;
        getallen = vullen(&grootte);
```

Hierbij heb ik van `vullen()` eigenlijk een functie gemaakt die twee resultaten teruggeeft: een pointer naar een verse tabel en een `int` die de grootte ervan aangeeft. Deze techniek om meerdere functieresultaten terug te geven heb je al gezien in §3.3.3.

▸ **OEFENING 3.12** **Blz. 68**

Eerst wordt de string achterwaarts in `buffer` gemaakt. Dan weten we hoe groot de string moet zijn, zodat we een `malloc()` kunnen doen.

maakString2.c

```
char *maakString(long getal)
{
        char buffer[20];
        int lengte;
```

```
        char *resultaat;

        /* Bijzonder geval. Als we deze regel zouden weglaten,
         * zou het resultaat van maakString(0) een lege string zijn
         */
        if (getal == 0) {
                resultaat = malloc(2);
                strcpy(resultaat,"0");
                return resultaat;
        }

        lengte = 0;

        /* Eerst de string achterstevoren maken
         * (getal%10 geeft het laatste cijfer)
         */
        while (getal != 0) {
                /* We veronderstellen dat de code van '0'
                 * 1 minder is dan '1', 2 minder dan '2' enz.
                 * (dit is het geval in de ASCII codering)
                 */
                buffer[lengte] = getal%10 + '0';
                getal = getal / 10;
                lengte = lengte + 1;
        }

        /* lengte + 1 bytes, want we mogen de nulbyte niet vergeten */
        resultaat = malloc(lengte + 1);

        {       /* Resultaat omkeren */
                int tel, telaf;

                tel = 0;
                telaf = lengte-1;
                while (tel < telaf) {
                        resultaat[tel] = buffer[telaf];
                        resultaat[telaf] = buffer[tel];

                        tel = tel + 1;
                        telaf = telaf - 1;
                }
        }

        /* De string netjes afsluiten met een nulbyte */
        resultaat[lengte] = 0;

        return resultaat;
}
```

Merk ook op dat ik het bijzondere geval waarbij het getal nul is een beetje herschreven heb. De gebruiker van `maakString()` moest in de vorige versie zelf het resultaat `free()`en, tenzij het getal nul was. In deze versie is dat onderscheid niet meer nodig: in alle gevallen moet de gebruiker het resultaat vrijgeven na gebruik met `free()`. (Een stuk geheugen vrijgeven dat je niet met `malloc()` gekregen hebt, staat altijd garant voor vuurwerk—hoewel sommige systemen gewoon doorgaan alsof er geen vuiltje aan de lucht is.)

▸ **OEFENING 3.13** **Blz. 74**

De `.` operator heeft een hogere prioriteit dan de indirecie-operator. `*ik.naam` is dus hetzelfde als `*(ik.naam)`, en aangezien `ik` geen `struct` is (maar een pointer (naar een `struct`)) kan je er de `.` operator niet op loslaten en zal de C compiler een foutmelding geven.

▸ **OEFENING 3.14** **Blz. 77**

```
struct Persoon *zoekPersoon(struct Persoon *lijst,
                        char *naam,char *voornaam)
{
        struct Persoon *zoek;

        zoek = lijst;
        while (zoek != NULL) {
                if (strcmp(zoek->naam,naam) != 0)
                        ; /* naam komt niet overeen,
                          /* dus gewoon voort gaan */
                else if (strcmp(zoek->voornaam,voornaam) != 0)
                        ; /* voornaam komt niet overeen */
                else /* naam en voornaam kloppen: gevonden! */
                        return zoek;
                zoek = zoek->link;
        }
        /* komt niet voor in de lijst */
        return NULL;
}
```

▸ **OEFENING 3.15** **Blz. 78**

```
struct Persoon *verwijderPersoon(struct Persoon *lijst,char *naam,
                                char *voornaam)
{
        struct Persoon *zoek;
        struct Persoon *vorige;

        vorige = NULL;
        zoek = lijst;
        while (zoek != NULL) {
                if (strcmp(zoek->naam,naam) != 0
                  || strcmp(zoek->voornaam,voornaam) != 0) {
                        /* naam en/of voornaam komen niet overeen */
                        vorige = zoek;
                        zoek = zoek->link;
```

```
                } else if (vorige == NULL) {
                        /* Verwijder het eerste element uit de lijst */
                        struct Persoon *resultaat;

                        resultaat = lijst->link;
                        free(lijst);
                        return resultaat;
                } else {
                        /* verwijder *zoek uit de lijst */
                        vorige->link = zoek->link;
                        free(zoek);
                        return lijst;
                }
        }
        /* komt niet voor in de lijst */
        return lijst;
}
```

▸ OEFENING 3.16 **Blz. 78**

```
struct Persoon *voegAchteraanToe(struct Persoon *lijst,
                        struct Persoon *nieuw)
{
        struct Persoon *zoek;

        nieuw->link = NULL;   /* zal laatste element van de lijst zijn */

        if (lijst == NULL)
                /* De lijst was leeg; het resultaat is dus een lijst
                 * met 1 element erin, namelijk het toe te voegen element
                 */
                return nieuw;
        /* De lijst was niet leeg; zoek dus het laatste element
         * en plak het nieuwe element er achter
         */
        zoek = lijst;
        while (zoek != NULL) {
                if (zoek->link == NULL) {
                        zoek->link = nieuw;
                        return lijst;
                }
                zoek = zoek->link;
        }
}
```

▸ OEFENING 3.17 **Blz. 78**

Twee lijsten aaneen plakken is bijna hetzelfde als een element achteraan een lijst toevoegen: het laatste element van `lijst1` moet nu wijzen naar het eerste element van `lijst2` in plaats van naar het nieuwe element.

```
struct Persoon *plakLijst(struct Persoon *lijst1,struct Persoon *lijst2)
{
        struct Persoon *zoek;

        if (lijst1 == NULL) return lijst2;

        /* De lijst was niet leeg; zoek dus het laatste element
         * en plak het eerste element van de tweede lijst er achter
         */
        zoek = lijst1;
        while (zoek != NULL) {
                if (zoek->link == NULL) {
                        zoek->link = lijst2;
                        return lijst1;
                }
                zoek = zoek->link;
        }
}
```

Hiermee kan je een kortere versie van `voegAchteraanToe()` schrijven:

```
struct Persoon *voegAchteraanToe(struct Persoon *lijst,struct Persoon *nieuw)
{
        nieuw->link = NULL;
        return PlakLijst(lijst,nieuw);
}
```

Wat ik hier doe, is in feite van `nieuw` een lijstje met één element maken en dat achteraan de bestaande lijst plakken.

▸ **OEFENING 3.18** **Blz. 78**

Ik zei toch dat het niet zo gemakkelijk was . . . :-)

▸ **OEFENING 4.1** **Blz. 89**

```
        char *f(int x[10],int (*y)[10]) { ... }
```

Inderdaad: `(*y)[10]` is een `int`, dus `(*y)` of `*y` is een array van tien `ints`, wat uiteindelijk betekent dat `y` een pointer is naar een array van tien `ints`. Het type van `y` krijgen we door zoals gewoonlijk `y` uit de declaratie te schrappen: `int (*)[10]`.

▸ **OEFENING 4.2** **Blz. 90**

Vies hè?
—wat het konijntje zei tegen de bakker die na de derde keer aandringen eindelijk worteltaart had gebakken

Het is het handigst eerst een declaratie van het aangeduide type te schrijven door er op de juiste plek een variabelnaam tussen te schrijven.

a) `char *x` betekent zoals we al vaak gezien hebben dat `x` een pointer naar een `char` is.

b) In `char *x[42]` heeft `[42]` de hoogste prioriteit ("plakt het hardst"), dus staat er in feite `char *(x[42])`, hetgeen we als volgt uiteenrafelen:

- `*(x[42])` is een `char`
- `(x[42])` of dus `x[42]` is een pointer naar een `char`
- `x` is een array van 42 pointers naar `char`.

c) `char ***x` betekent "gewoon" dat `***x` een `char` is, dus dat `**x` een pointer is naar een `char`, dus dat `*x` een pointer naar een pointer naar een `char` is en dat `x` een pointer naar een pointer naar een pointer naar een `char` is. Pointers naar pointers gebruiken wordt wel eens met een groot woord **meervoudige indirectie** genoemd.

d) `char (*x)[42]` betekent:

- `(*x)[42]` is een `char`
- `(*x)` of dus `*x` is een array van 42 `char`s
- `x` is een pointer naar een array van 42 `char`s

e) `char *x(void)` betekent dat `x(void)` een pointer naar een `char` is, dus is `x` een functie die een pointer naar een `char` teruggeeft.

f) `char (*x)(void)` betekent dat `*x` een functie is die een `char` teruggeeft, dus is `x` een pointer naar een functie die `char` teruggeeft.

g) `char * (*x)[42](void)` is hetzelfde als `char * ((*x)[42](void))` (het sterretje plakt niet zo hard), en maar weer uitrafelen:

- `(*x)[42](void)` is een `char *`, een pointer naar een `char` dus
- `(*x)[42]` is een functie die een pointer naar een `char` teruggeeft
- `*x` is een array van 42 functies die een pointer naar een `char` teruggeven
- `x` is een pointer naar een array van 42 functies die een pointer naar een `char` teruggeven

h) `char * (*x)(void)[42]` lijkt goed op de vorige, maar betekent iets heel anders (een beetje zoals een colafles geen fles cola is):

- `(*x)(void)[42]` is een pointer naar een `char`
- `(*x)(void)` is een array van 42 pointers naar `char`
- `*x` is een functie die een array van 42 pointers naar `char` teruggeeft
- `x` is een pointer naar een functie die een array van 42 pointers naar `char` teruggeeft

In alle voorbeelden waar ik zei "een functie die ... teruggeeft" had ik eigenlijk "een functie die geen argumenten heeft en ... teruggeeft" maar het was zo al wangedrochtelijk genoeg :-)

▸ OEFENING 4.3 **Blz. 92**

Het is een array van **drie** pointers naar `char`s. De eerste wijst naar de string `roodkapje`, de tweede naar `wolfhouthakker` en de derde naar `heks`. Als je in C immers twee strings na elkaar schrijft, plakt de compiler die aaneen tot één lange string. Dit soort fouten kan bijzonder “subtiel” zijn :-)

▸ OEFENING 4.4 **Blz. 100**

Deze oefening komt inderdaad niet toevallig na de uitleg van `continue` ...

Het verschil tussen bijvoorbeeld

```
{
        int x;

        for (x=0; x<10; x++) {
                if (x%2 == 0) continue;
                printf("%d ",x);
        }
}
```

dat de oneven getallen tussen 0 en 10 afdrukt, en

```
{
        int x;

        x = 0;
        while (x < 10) {
                if (x%2 == 0) continue;
                printf("%d ",x);
                x++;
        }
}
```

is dat in het laatste programma `continue` ook de opdracht `x++;` overslaat, zodat `x` op nul blijft staan en het programma klem loopt.

▸ OEFENING 4.5 **Blz. 100**

```
#include <stdio.h>

int main(void)
{
        float totaal;

        totaal = 0;
        while (1) {
                int ok;
                float getal;

                ok = scanf("%f",&getal);
                if (ok != 0) {
                        char eetop;

                        scanf("%c",&eetop);
                        continue;
```

```
                }
                if (getal == 0) break;
                totaal = totaal + getal;
        }
        printf("Totaal: %f\n",totaal);
        return 0;
}
```

De variabele `getal` is hier een lokale variabele geworden. Door het gebruik van `break` en `continue` is deze versie iets duidelijker dan de oude.

▸ OEFENING 4.6 **Blz. 105**

dingbottel.c

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

#define DING    5
#define BOTTEL  7

/* hulpfunctie */
int hoeveelkeer(int wat,int getal)
{
        int resultaat;

        /* deelbaar door ‘wat’? */
        if (getal%wat == 0) resultaat = 1;
        else resultaat = 0;
        /* hoeveel keer komt er het cijfer ‘wat’ in voor? */
        while (getal != 0) {
                /* het laatste cijfer is  getal%10 */
                if (getal%10 == wat) resultaat = resultaat + 1;
                /* knip het laatste cijfer eraf */
                getal = getal / 10;
        }
        return resultaat;
}

/* hoeveel keer moet er ‘ding’ gezegd worden voor dit getal? */
int hoeveelding(int getal)
{
        return hoeveelkeer(DING,getal);
}

int hoeveelbottel(int getal)
{
        return hoeveelkeer(BOTTEL,getal);
}

/* druk een letter een aantal keer af
 * bv. drukzoveel(5,’*’) drukt vijf sterretjes */
void drukzoveel(int aantal, int letter)
{
```

```
        while (aantal > 0) {
                putchar(letter);
                aantal = aantal - 1;
        }
}

void computerzegt(int getal)
{
        int dings;
        int bottels;

        dings = hoeveelding(getal);
        bottels = hoeveelbottel(getal);
        if (dings == 0) {
                if (bottels == 0) {
                        /* geen dings, geen bottels,
                         * gewoon het getalletje zeggen */
                        printf("%d\n",getal);
                        return;
                }
        }
        /* in het andere geval zijn er dings en/of bottels */
        drukzoveel(dings,'d');
        drukzoveel(bottels,'b');
        putchar('\n');
}

/* Lees wat de speler zegt.
 * Geeft 1 terug als het antwoord OK was */
int spelerzegt(int getal)
{
        char invoer[80];   /* de invoer van de speler */
        int dings;
        int bottels;
        int spelerdings;
        int spelerbottels;
        int tel;

        dings = hoeveelding(getal);
        bottels = hoeveelbottel(getal);
        gets(invoer);

        /* Kijk eerst of het aantal dings en bottels klopt */
        spelerdings = 0;
        spelerbottels = 0;
        for (tel = 0;  invoer[tel] != 0;  tel = tel + 1) {
                switch (invoer[tel]) {
                case 'b': spelerbottels = spelerbottels + 1;
                          break;
                case 'd': spelerdings = spelerdings + 1;
                          break;
                }
```

```
        }
        /* Vergelijk met de juiste aantallen */
        if (dings != spelerdings) return 0;
        if (bottels != spelerbottels) return 0;

        /* Aantallen dings en bottels kloppen.
         * Nu eventueel controleren of het getal juist is.
         * Dat moet natuurlijk alleen als dings==0 en bottels==0 */
        if (dings != 0) return 1;
        if (bottels != 0) return 1;
        /* Geen dings, geen bottels. Dus moet het antwoord een getal zijn */
        if (atoi(invoer) == getal) return 1;
        else return 0;
}

int main(void)
{
        int getal;

        getal = 1;
        while (1) {
                /* Om de beurt de computer ... */
                computerzegt(getal);
                getal = getal + 1;
                /* ... en de speler laten spelen */
                if (spelerzegt(getal) != 1) break;
                getal = getal + 1;
        }
        printf("Jammer ... het was: ");
        computerzegt(getal);
        return 0;
}
```

Wat commentaar bij dit programma: zoals je ziet, zijn de functies `hoeveelding()` en `hoeveelbottel()` bijna de moeite van het opschrijven niet waard. Hun enige functie is het programma duidelijker maken.

Merk verder op dat ik in de functie `drukzoveel()` gebruik maak van het feit dat `aantal` zich gedraagt als een gewone lokale variabele om een extra lusteller uit te sparen. Ook belangrijk is dat deze functie correct blijft werken als het `aantal` nul is.

Wie zin heeft kan dit programma nog uitbreiden zodat het vraagt of de computer begint of de speler, de mogelijkheid voorzien om met meerdere spelers tegen meerdere computers te spelen, ...

▸ **OEFENING 4.7** **Blz. 108**

`!!x` kan alleen maar `0` of `1` opleveren. Als `x` niet `0` of `1` is, geeft `!!x` als resultaat `1`.

▸ **OEFENING 4.8** **Blz. 113**

Met de `++` en `--` operatoren kan je wel zoiets bedenken:

```
x[i++] += 5;
```

Wat `x[i++] = x[i++] + 5;` doet, is niet duidelijk omdat je niet weet welke `++` het eerst geëvalueerd wordt; bovendien zou het resultaat in elk geval iets anders geweest zijn.

▸ **OEFENING 4.9** **Blz. 113**

```
        i = i + x;
        tabel[i] = tabel[i] + j;
        x = x * tabel[i];
```

▸ **OEFENING 4.10** **Blz. 120**

En déze oefening staat niet toevallig na de uitleg van de komma-operator. Daar zit dan ook het probleem: `tab[x,y,z] = 0;` heeft als effect `tab[z] = 0;` (de komma-operator evalueert eerst `x` en `y`, maar die uitdrukkingen hebben geen neveneffecten, en doen dus niets). Wat we willen is natuurlijk `tab[x][y][z] = 0`.

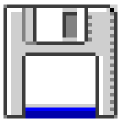
▸ **OEFENING 4.11** **Blz. 121**

De functie-versie werkt alleen met `int`s; de macro-versie werkt met alle numerieke datatypes (dus ook bijvoorbeeld met `double`s). En natuurlijk was het beter

```
#define abs(x) ( ((x)<0) ? -(x) : (x) )
```

te schrijven om rare nevenwerkingen met ingewikkelder uitdrukkingen te voorkomen ...

▸ **OEFENING 5.1** **Blz. 137**



caesar.c

```
#include <stdio.h>
#include <stdlib.h>           /* voor atol() */

/* roteer een getal tussen 0 en 25;
 * het resultaat ligt weer tussen 0 en 25 */
int rot(int i,int afstand)
{
        /* het resultaat is  i+afstand,
         * tenzij dat >= 26 is; dan is het  i+afstand-26 */
        return i+afstand<26 ? i+afstand : i+afstand-26;
}

int main(int argc, char **argv)
{
        int letter;
        int afstand;

        if (argc != 2) {
                puts("gebruik: caesar <rotatie-afstand>");
                return 10;
        }

        afstand = atol(argv[1]);
        if (afstand<0 || afstand>26) {
                puts("afstand moet tussen 0 en 25 liggen");
        }
```

```
        while ((letter = getchar()) != EOF) {
                if (letter>='a' && letter<='z')        /* kleine letter */
                        putchar('a' + rot(letter-'a',afstand));
                else if (letter>='A' && letter<='Z')   /* hoofdletter */
                        putchar('A' + rot(letter-'A',afstand));
                else putchar(letter);                  /* geen letter */
        }
        return 0;
}
```

▸ **OEFENING 5.2** **Blz. 141**

zv.c

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

#define ZOEK argv[1]
#define VERVANG argv[2]

int main(int argc,char **argv) {
        char buff[1000];

        if (argc != 3) {
                puts("Gebruik: zv zoektekst vervangtekst");
                return 10;
        }
        while (fgets(buff,sizeof(buff),stdin)) {
                char *toprint = buff;

                /* resultaat van fgets() eindigt op een \n */
                buff[strlen(buff)-1] = 0;     /* verwijder die \n */
                while (1) {
                        char *pos = strstr(toprint,ZOEK);

                        if (!pos) {   /* zoektekst niet gevonden */
                                puts(toprint);
                                break;
                        }
                        *pos = 0;
                        printf("%s%s",toprint,VERVANG);
                        /* Nu de rest van de string doorzoeken */
                        toprint = pos + strlen(ZOEK);
                }
        }
        return 0;
}
```

Merk op dat dit programma niet goed werkt als er regels voorkomen die langer zijn dan 998 letters (niet 1000; er zijn twee `char`s nodig voor de `\n` en de nulbyte).

▸ OEFENING 5.3 **Blz. 142**

vertaal.c

```
#include <stdio.h>
#include <string.h>
#include <ctype.h>
#include <malloc.h>

struct Woord {
        struct Woord *next;
        char *van;
        char *naar;
};

struct Woord *leeswoorden(char *fnaam) {
        struct Woord *result = NULL;
        char lijn[255];
        FILE *wdb = fopen(fnaam,"r");
        if (!wdb) return NULL;
        while(fgets(lijn,sizeof(lijn),wdb)) {
                struct Woord *nieuw = malloc(sizeof(struct Woord));
                char *split = strstr(lijn,"\t");
                char *end = lijn + strlen(lijn)-1;     /* wijst naar \n */
                if (!split) continue;
                nieuw->next = result;
                result = nieuw;
                nieuw->van = malloc(split-lijn+1);
                strncpy(nieuw->van,lijn,split-lijn);
                nieuw->van[split-lijn] = 0;
                split++;
                nieuw->naar = malloc(end-split+1);
                strncpy(nieuw->naar,split,end-split);
                nieuw->naar[end-split] = 0;
        }
        fclose(wdb);
        return result;
}

/* Druk de vertaling van het woord af */
void vertaalwoord(char *woord,struct Woord *woordenlijst) {
        struct Woord *snuffel;
        for (snuffel = woordenlijst; snuffel; snuffel = snuffel->next) {
                if (!strcasecmp(snuffel->van,woord)) {
                        /* hoe staat het originele woord er? */
                        if (islower(*woord)) {         /* kleine letters */
                                printf("%s",snuffel->naar);
                        } else if (isupper(woord[1])) {  /* HOOFDLETTERS */
                                char *walk;
                                for (walk = snuffel->naar; *walk; walk++)
                                        putchar(toupper(*walk));
                        } else {     /* Alleen eerste letter hoofdletter */
                                printf("%c%s",toupper(snuffel->naar[0]),
                                        snuffel->naar+1);
```

```
                        }
                        return;
                }
        }
        /* geen vertaling gevonden in woordenboek ... */
        printf("%s",woord);
}

/* Lees een woord van stdin en druk de vertaling af op stdout */
void doewoord(struct Woord *woordenlijst) {
        char woord[255];
        char *lees;

        for (lees = woord; ; lees++) {
                int c = getchar();
                if (c == EOF) break;
                if (!isalpha(c)) {        /* einde woord */
                        ungetc(c,stdin);
                        break;
                }
                *lees = c;
        }
        *lees = 0;        /* string afsluiten met nulbyte */
        vertaalwoord(woord,woordenlijst);
}

int main(int argc,char **argv) {
        int c;
        struct Woord *woordenlijst;

        woordenlijst = leeswoorden("woordenlijst");
        while ((c = getchar())) {
                if (c == EOF) break;
                if (isalpha(c)) {
                        ungetc(c,stdin);
                        doewoord(woordenlijst);
                } else putchar(c);
        }
        return 0;
}
```

▸ **OEFENING 6.1** **Blz. 152**

`#define KWADRAAT(x) (x*(x))` en `#define KWADRAAT(x) ((x)*x)` gaan de mist in bij `KWADRAAT(x+y)` zoals uitgelegd in de tekst.

`#define KWADRAAT(x) (x)*(x)` geeft een verkeerd resultaat bij `KWADRAAT(y/KWADRAAT(x))` (het resultaat is $\frac{y}{x} \cdot x$ of dus gewoon y, en niet $\frac{y}{x \cdot x}$)

Bij `GEMIDDELDE` kunnen we iets soortgelijks uithalen door de `<<` operator te gebruiken: `GEMIDDELDE(x<<2,y)` geeft `(x<<2+y)/2` wat hetzelfde doet als `((x<<2)+y)/2` en dat is niet echt wat we willen.

▸ **OEFENING 6.2** **Blz. 152**

```
#define verwissel(x,y) (wissel(&(x),&(y)))
```

zorgt ervoor dat `verwissel(a,b);` de inhoud van `a` en `b` verwisselt.

▸ **OEFENING 6.3** **Blz. 155**

Als `hulp ## a` en `b` hetzelfde zouden opleveren, krijgen we natuurlijk weer problemen, zoals in `wissel(x,hulpx);`. De oplossing ligt voor de hand:

```
#define wissel(a,b)    {     int hulp ## a ## b;          \
                             hulp ## a ## b = a;          \
                             a = b;                       \
                             b = hulp ## a ## b;   }
```

De hulpvariabele heet nu `hulp ## a ## b` en dat kan nooit hetzelfde geven als `a` of `b`.

Misschien lijkt deze situatie nogal bij het haar gegrepen. Kwaliteitsmacro's moeten echter rekening houden met alle mogelijkheden—het kan je later hoofdbrekens besparen, want op het eerste zicht *kan* de fout gewoonweg niet bij `wissel()` liggen, want die macro gedroeg zich toch altijd zo netjes …

Bijlage D

Verwijzingen

D.1 Bibliografie (en aanbevolen lectuur)

Brian W. Kernighan, Dennis M. Ritchie, *The C Programming Language*, Prentice-Hall, Englewood Cliffs, NJ, 1978; ISBN 0-113-110163-3.
De officiële beschrijving van C tot vóór ANSI C*, geschreven door de makers van C zelf. Dit zeer invloedrijke boek legt heel helder de taal uit voor wie al vertrouwd is met programmeren in andere talen. Het wordt bijgenaamd* **White Book** *(naar de kleur van de kaft) en* **Old Testament***—de tweede uitgave beschrijft* ANSI C *en wordt dan ook vaak* **New Testament** *genoemd (Prentice-Hall,* 1988*; ISBN 0-13-110362-8).*

Morris L. Bolsky, *Handboek voor de C-programmeur (The C Programmer's handbook)*, Prentice-Hall/Academic Service, Schoonhoven, 1988; ISBN 90-6233-294-3.
*Een handig ringband-naslagwerkje over pre-*ANSI C*.*

Samuel P. Harbison, Guy L. Steele Jr., *C: a Reference Manual*, 2nd Ed., Prentice-Hall, Englewood Cliffs, NJ, 1987; ISBN 0-13-109802-0.
Een erg volledige beschrijving van C en de header files errond. Dit boek werd geschreven omdat tijdens het schrijven van een familie C compilers voor uiteenlopende soorten computers het duidelijk werd dat sommige aspecten van C onvoldoende gedocumenteerd waren.

Mark A. Terribile, *Practical C++*, McGraw-Hill, 1994; ISBN 0-07-063738-5.
Dit uitstekende boek over C++ bevat ook een beknopt overzicht van C zelf.

Rex Jaeschke, *The Dictionary of Standard C*, McGraw-Hill, 1991; ISBN 1-878956-07-8.
Dit boekje bevat een hoop informatie over C, alfabetisch geordend van `abort` *tot* `X3J16`*.*

Eric S. Raymond, *The new Hacker's Dictionary*, 2nd Ed., MIT Press, 1993.
Programmeren zelf wordt hier niet uitgelegd, maar wel de hele hacker-cultuur die errond hangt, de anekdotes, de typische tussen-de-regels-door humor, ... Dit boek bestaat ook in elektronische versie (the Jargon File) en is te vinden op `http://www.ccil.org/jargon`.

Chris Brown, *Programmieren verteilter Unix-anwendungen*, Prentice Hall, München, 1994; ISBN 3-930436-14-0. Originele titel: *UNIX Distributed Programming.*
Een uitgebreid overzicht over communicatie tussen verschillende programma's op allerlei manieren, onder andere via sockets over het Internet.

D.2 Internet adressen

De auteur van deze cursus is te bereiken via het e-mail adres `geert@zeus.rug.ac.be` (suggesties en verbeteringen zijn natuurlijk altijd welkom); errata en programmalistings zal je op `http://zeus.rug.ac.be/C/` kunnen vinden.

Er zijn ook nieuwsgroepen over C, o.a. `comp.lang.c` en `comp.lang.c.moderated`.

De GNU C compiler is op vele FTP-sites te vinden. De MS-DOS versie heet DJGPP en is onder andere op `http://www.delorie.com` te vinden. De officiële distributie van `gcc` voor Linux staat in gebruiksklare vorm op `ftp://tsx-11.mit.edu/pub/linux/packages/GCC`.

Interessant is de ontwikkelomgeving Cygwin. Het is een Unix-achtige omgeving voor windows 95, 98 en nt. Een groot deel van de Unix systeemfuncties wordt nagebootst onder windows. Het bevat ook vele populaire GNU hulpmiddelen (`gcc`, `make`, `gdb`, ...) Het mooiste van al is dat je het gratis kan downloaden op `http://sourceware.cygnus.com/cygwin/`. Je kan de windows-specifieke functies ook blijven gebruiken, wat handig is om bijvoorbeeld een Unix programma naar een windows omgeving over te zetten en te voorzien van een grafische gebruikersinterface (door de windows functies te gebruiken om met vensters en dergelijke te werken).

Bijlage E

Foutmeldingen en waarschuwingen van de GNU C compiler

Ik beschrijf hier de foutmeldingen en waarschuwingen die de GNU C compiler `gcc` versie 2.7.2 genereert, samen met een korte omschrijving van de oorzaak ervan.

E.1 Foutmeldingen

- `conflicting types for` ‘*variabele*’
  Je hebt een variabele twee maal op een andere manier gedeclareerd. Je krijgt ook nog een `previous declaration of` ‘*variabele*’, voorafgegaan door de plaats waar je de variabele de eerste keer hebt gedeclareerd.

- `invalid macro name`
  Je hebt in een `#define` een ongeldige macronaam gebruikt, bijvoorbeeld

  ```
  #define := =
  ```

- `invalid operands to binary %`
  Je hebt de `%` of `%=` operator proberen te gebruiken op iets dat geen geheel datatype is.

- `parse error before` ‘*woord*’
  De compiler verwacht het *woord* op deze plaats niet.
  Deze foutmelding kan je krijgen als je een variabele of een `struct` niet aan het begin van een blok declareert, maar ergens tussenin de opdrachten stopt. Als je geen accolades rond de opdrachten van een functiedefinitie schrijft, krijg je ook deze foutmelding.

  Deze foutmelding kan ook veroorzaakt worden doordat je een blok vergeet af te sluiten in een functiedefinitie:

  ```
  void streepke(void)
  {
          int tel;

          for (tel = 0; tel < 20; tel++) {
                  putchar('-');
  }
  ```

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

De compiler denkt dat de regel `int main(void)` nog in het blok van de functie `streepke()` staat, en zal dus de melding `parse error before 'int'` geven. Dit soort fouten is een beetje lastig op te sporen omdat de oorzaak (het blok van de `for`-lus is niet afgesloten) soms vrij ver uit de buurt staat, en omdat het vaak niet duidelijk is welk blok je precies vergat af te sluiten (vooral als de functie `streepke()` flink lang zou zijn en veel geneste blokken zou bevatten).

- `redeclaration of '`*variabele*`'`
  Je hebt een variabele twee maal gedeclareerd. Je krijgt ook nog een `'`*variabele*`' previously declared here`, voorafgegaan door de plaats waar je de variabele de eerste keer hebt gedeclareerd.

- `redefinition of '`*functienaam*`'`
  Je hebt twee functies gemaakt met de naam *functienaam*. Je krijgt ook hier een `'`*functienaam*`' previously declared here` die aanduidt waar de eerste definitie van de functie staat.

- `request for member '`*lid*`' in something not a structure or union`
  Je hebt `naam.lid` geschreven, maar `naam` is geen `struct` of `union` variabele.
  Deze fout krijg je ook als `naam` wel een pointer naar een `struct` of `union` is en je `*naam.lid` schrijft (zie oefening 3.13).

- `syntax error before '`*tekst*`'`
  Meestal veroorzaakt doordat je ergens een kommapunt vergeten bent, nog vóór de aangegeven plaats. Als je bijvoorbeeld

  ```
  {
          int tabel[20]
          char *s;
  }
  ```

  probeert te compileren, krijg je een `syntax error before 'char'` terwijl de fout op de regel ervoor zit (er ontbreekt een kommapunt).

- `two or more data types in declaration of '`*functienaam*`'`
  Je hebt iets als `int int main()` geschreven als eerste regel van een functiedefinitie. Een veel voorkomende oorzaak van deze foutmelding is het vergeten van een kommapunt bij de definitie van een `struct` of `union`:

  ```
  struct Datum {
          int jaar;
          char maand;
          char dag;
  }
  ```

```
int main(void) {
        /* ... */
}
```

Hier heb je in feite `struct datum int main(void)` geschreven, en een functie mag natuurlijk maar één returntype hebben ...

- `‘`*varnaam*`’ undeclared (first use in this function)`
  `(Each undeclared identifier is reported only once`
  `for each function it appears in.)`
  Je hebt een variabele met de naam *varnaam* gebruikt, maar er is geen variabele met die naam gedeclareerd. Als je in dezelfde functie nog ergens *varnaam* gebruikt, zal de compiler dat maar één keer signaleren (om te vermijden dat er schermen vol foutmeldingen langsfloepen).

E.2 Waarschuwingen

Waarschuwingen (**warning**s) zijn boodschappen die het gebruik van constructies signaleren die niet echt fout zijn, maar waar vaak fouten tegen gemaakt wordt (bijvoorbeeld het gebruik van `=` in de voorwaarde van een `if`). Met een aantal opties van `gcc` kan je de instellen waar de compiler op let tijdens het compileren. Sommige waarschuwingen worden alleen gegenereerd als je een bepaalde optie opgeeft bij het compileren. Die optie staat dan tussen haakjes rechts van de tekst van de waarschuwing aangegeven.

De `-Wall` optie zorgt ervoor dat de compiler waarschuwingen genereert bij de meest voorkomende constructies.

Bij sommige warnings krijg je eerst nog een aanduiding van de plaats van de fout, zoals `At top level:` (buiten de functiedefinities, bijvoorbeeld een globale variabele) of `In function ‘`*naam*`’:` (in een bepaalde functie).

- `assignment of read-only location`
  Je hebt iets dat `const` is proberen te wijzigen (zie §4.1.6).

- `built-in function ‘`*naam*`’ used without declaration`
  Je hebt de functie `sin()` of `cos()` gebruikt zonder `math.h` te includen.

- `comparison is always 0 due to limited range of data type`
  Je hebt iets geschreven als `if (c == 1000)` met `c` een `char` variabele. Een `char` kan nooit 1000 zijn omdat de grootste waarde die in een `char` past kleiner is dan 1000, en het resultaat van `==` is dus altijd vals. Het `char` type is afhankelijk van de compiler signed of unsigned, en `if (c == 200)` is alleen OK op een compiler die voor unsigned `char`s kiest. Het is dus beter in zulke gevallen `c` exipliciet als `unsigned char` te declareren.

- `control reaches end of non-void function` (`-Wreturn-type`)
  Je hebt een functie gemaakt die een resultaat terug moet geven, maar je hebt ze niet afgesloten met een `return`.

- `‘`*naam*`’ defined but not used` (`-Wunused`)
  Je hebt een globale `static` variabele gedefinieerd in het bestand zonder ze verderop te gebruiken.

- `double format, different type arg (arg` *n*`)` (`-Wformat`)
  In het formaatargument van `printf()` of `scanf()` staat `%f`, maar je hebt een argument opgegeven dat geen kommagetal-type is.

- `empty declaration`
  Je hebt ergens iets als `int ;` geschreven, en de compiler verwacht eigenlijk `int variabelnaam;` of zo. Syntactisch gezien is het toegelaten een lege declaratie (zie §C) te schrijven, maar het is waarschijnlijker dat er een tikfout in je programma is geslopen.

- `format argument is not a pointer` (`-Wformat`)
  In het formaatargument van `printf()` staat `%s`, maar het argument dat je opgegeven hebt, is geen pointer.
  Een argument van `scanf()` is geen pointer.

- `function returns address of local variable`
  Je hebt een functie gemaakt die als resultaat het adres van een lokale variabele geeft. Omdat het geheugengebied waar de lokale variabelen van een functie bewaard worden na het einde van de functie vrijgegeven wordt, wijst het functieresultaat dus naar een vrijgegeven stuk geheugen, wat waarschijnlijk niet je bedoeling is: dat stuk geheugen kan plots wel gereserveerd worden (dat kan zelfs door een ander programma gebeuren in sommige besturingssystemen!) en dus in het gunstigste geval iets helemaal anders bevatten dan je er zelf in stak. Zie ook §3.6.

- `implicit declaration of function` (`-Wimplicit`)
  Je hebt een functie gebruikt zonder dat er een prototype (zie §2.8.3.2) voor beschikbaar was.

- `no semicolon at end of struct or union`
  Je hebt een kommapunt vergeten te schrijven na de afsluitende accolade van `struct` of `union`.

- `‘`*macronaam*`’ redefined`
  Je hebt een `#define` *macronaam* gedaan van een macro die al eens gedefinieerd was. De compiler toont op de volgende regel (`this is the location of the previous definition`) de plaats waar je de eerste keer die macro definieerde.

  Netter is de oude macro eerst te `#undef`en; eventueel kan je de volgende constructie gebruiken:

  ```
  #ifdef macronaam
  # undef macronaam
  #endif
  #define macronaam lalala
  ```

- `return-type defaults to ‘int’` (`-Wreturn-type`)
  Je hebt een functie gedeclareerd zonder een return-type op te geven; de compiler onderstelt dat het `int` is (zie §2.8.3.2).

- `‘return’ with a value, in function returning void`
  Je probeert een waarde terug te geven met `return` in een functie die `void` als resultaattype heeft.

- `'return' with no value, in function returning non-void` (`-Wreturn-type`)
  Je hebt `return;` geschreven in een functie die een resultaat moet teruggeven.

- `statement with no effect` (`-Wunused`)
  Je hebt een resultaat berekend in een uitdrukking, maar je gebruikt dat resultaat niet, zoals in de opdracht `sin;`. De opdracht doet dus eigenlijk niets.

- `suggest parentheses around assignment used as truth value` (`-Wparentheses`)
  Je hebt het resultaat van een toekenningsopdracht (bijvoorbeeld `x = 10`) gebruikt als voorwaarde in bijvoorbeeld een `if`, `while`. Misschien bedoelde je wel `==`. Als er wel degelijk `=` moet staan, kan je de compiler dat duidelijk maken door een extra paar haakjes rond de toekenningsopdracht te schrijven. (Zie ook §4.3.5.1 en §2.4.)

- `too many arguments for format` (`-Wformat`)
  In het formaatargument van `printf()` of `scanf()` staan minder `%`-tekens dan er argumenten opgegeven zijn.

- `unused variable '`*naam*`'` (`-Wunused`)
  Je hebt een variabele gedeclareerd in een functie maar ze nooit gebruikt.

Index

Inhoudsopgave