HANDLEIDING

PROGRAMMEREN

(5e en 6e kwartaal natuur- en sterrenkunde)

July 20, 2011

Faculteit der Natuurwetenschappen, Wiskunde en Informatica
P.F. Klok
J.M. Thijssen
W.L. Meerts

Radboud Universiteit Nijmegen

Inhoudsopgave

Hoofdstuk 1

Inleiding

PROGRAMMEREN is in de eerste plaats bedoeld om natuurkundestudenten een goede programmeermethodiek aan te leren en om hen een idee van de mogelijkheden en de beperkingen van een computer te geven. In totaal worden gedurende twee kwartalen een gecombineerd college (10 x 2 uur) met practicum (14 x 4 uur) gegeven.

Deze handleiding (literatuurlijst [2]), waarin in het kort de te behandelen onderwerpen besproken worden, beoogt een bondige leidraad en hulpmiddel bij college en practicum te zijn. Het is niet bedoeld als enige te raadplegen litteratuur! Daarnaast is er een website met adres `http://www.mlf.science.ru.nl/leo_meerts/programmeren/` die actuele informatie bevat. Een digitale leeromgeving met adres `http://blackboard.ru.nl/` bevat ook informatie over het vak en geeft daarnaast de mogelijkheid om in een "Discussion Board" via e-mail te communiceren over zaken die dit vak betreffen.
Voor een uitgebreide verwoording van de stof wordt vanuit deze handleiding verwezen naar:
The C Programming Language, 2nd ed. door Brian W. Kernigan en Dennis M. Ritchie.

Het practicum maakt gebruikt van personal computers (PC's) met het Windows of Linux besturingssysteem. Hierop worden tijdens het practicum programma's ontwikkeld met behulp van de programmeertaal **C** (zie literatuurlijst [1]). Voor gebruik van de ontwikkelomgeving en software en andere recente ontwikkelingen wordt verwezen naar `http://www.mlf.science.ru.nl/leo_meerts/programmeren/`.

Beoordeling van de opgedane en toegepaste kennis van college en practicum geschiedt aan de hand van resultaten van een aantal opdrachten.

Afgezien van de gebruikte programmeertaal en een praktijkgerichte toespitsing op het vakgebied Natuurkunde, wordt bij Programmeren voldoende stof behandeld om vervolgvakken op het gebied van de Informatica te kunnen volgen.

De verdere indeling van deze handleiding is als volgt:
- Hoofdstuk 2 behandelt — uitgaande van algoritmen — het zelf maken van programma's, de regels die daarbij gevolgd kunnen worden, de taalelementen die bij het programmeren gebruikt worden en een aantal andere aspecten van het programmeren.
- Hoofdstuk 3 bevat een korte, praktische handleiding voor de programmeertaal C, die op het practicum gebruikt wordt.
  N.B.: Dit hoofdstuk is zeer beknopt. Voor uitgebreide documentatie gebruik het boek: Brian W. Kernighan, Dennis M. Ritchie, *The C Programming Language, second edition*
- Hoofdstuk 4 behandelt een aantal sorteer- en zoektechnieken.
- Hoofdstuk 5 behandelt eenvoudige generatoren voor random numbers.
- Hoofdstuk **??** behandelt grammatica's voor computertalen.
- Hoofdstuk 6 behandelt de techniek van het recursief programmeren.
- Appendix A bevat een literatuurlijst waarin o.a. beschrijvingen van de in deze handleiding aangehaalde referenties opgenomen zijn.
- Appendix B bevat een korte beschrijving van de programmeertaal C met voorbeelden.
- Appendix C tenslotte, bevat een register van voorkomende termen.

Hoofdstuk 2

Programmeren

Het maken van programma's wordt **programmeren** genoemd. Dit hoofdstuk beschrijft een aantal zaken die met dat programmeren te maken hebben.

Wanneer moet er geprogrammeerd worden? Als er een probleem met behulp van een computer opgelost moet worden. Hierbij zijn een aantal stappen te onderscheiden, waarvan het programmeren er eigenlijk maar één is:

- ♡ Eerst moet het probleem goed begrepen worden: wat is de bedoeling precies? *Als het niet precies duidelijk is wat er gedaan moet worden, dan is de kans groot, dat er bij een volgende stap problemen ontstaan!*
- ♡ Vervolgens wordt gekeken of het probleem op een geschikte manier met een computer opgelost kan worden. *Er zijn namelijk ook wel eens efficiëntere oplossingen zonder computer mogelijk!*
- ♡ Daarna moet er een recept of **algoritme** opgesteld worden waarin de wijze van oplossen van het probleem nauwkeurig beschreven wordt. Dit algoritme beschrijft het oplossen tot op een voldoende diep niveau.
- ♡ Vervolgens wordt dat algoritme omgezet in instructies of opdrachten van een programmeertaal (het eigenlijke programmeren). *Na deze stap zou het probleem opgelost moeten zijn!*
- ♡ Het aldus gemaakte programma wordt door de bij de gebruikte programmeertaal en computer behorende compiler vertaald naar machinecode, waarna het uitgevoerd kan worden.
  Alhoewel, het kán zijn dat er taalfouten (fouten tegen de **syntax** van de programmeertaal) door de compiler zijn ontdekt, waardoor het programma niet op de computer uitgevoerd kan worden. Deze taalfouten moeten eerst verbeterd worden, waarna er opnieuw gecompileerd kan worden. En dit moet net zolang herhaald worden totdat het programma taalfoutvrij is en uitgevoerd kan worden.
- ♡ Nu kan het programma uitgevoerd worden. Aan de hand van bepaalde invoergegevens kan gecontroleerd worden of het programma de juiste uitvoergegevens opleverd. Is dit niet zo, dan moeten de bijbehorende fouten in de programmastructuur (fouten in de **logica** van het algoritme) gevonden en verbeterd worden, waarna het programma opnieuw gecontroleerd kan worden. Geeft het programma de juiste uitvoergegevens, dan wordt het in de praktijk als goed programma beschouwd, totdat het tegendeel blijkt.
  Er is een "tak van sport" binnen de Informatica, die zich bezig houdt met het bewijzen van de juistheid van programma's.

In de praktijk blijkt vaak dat het opgestelde algoritme niet helemaal voldoet. Er zijn bijvoorbeeld een aantal dingen over het hoofd gezien, waardoor er bij de volgende stappen dusdanige problemen zijn, dat het nodig is een aantal correcties in het algoritme aan te brengen. In zulke gevallen worden een aantal stappen meerdere malen doorlopen. Aangezien dit extra tijd kost, is het zaak dit soort **iteraties** (en de eventuele bijkomende **irritaties**) zoveel mogelijk te beperken.

Omdat het zoeken naar logische fouten ook een tijdrovend werk kan zijn (en dus economisch niet verantwoord), is het zaak om te proberen algoritmen foutloos op te stellen. Voor een klein probleem is het, na enige ervaring opgedaan te hebben, niet moeilijk om dit te doen. Maar naarmate de op te lossen problemen groter en complexer worden, wordt de kans op fouten ook groter. Daarom zijn er diverse technieken, regels, systemen en zelfs complete

programmeertalen ontwikkeld om de kans op logische fouten zo klein mogelijk en de kans op juiste uitvoering van een programma zo groot mogelijk te maken èn om de ontwerpfase zo veel mogelijk te bekorten.

Concepten en methoden om betere en betrouwbaarder programma's te schrijven worden hierna behandeld. De bedoeling van het practicum is om deze te leren gebruiken.

2.1 Algoritmen

Recepten, voorschriften, handleidingen en dergelijke zijn allemaal onder te brengen onder het begrip **algoritme**: een beschrijving om een bepaalde handeling op een gewenste manier uit te voeren. Maar waaruit bestaat een algoritme bij programmeren precies en waaraan moet een algoritme voldoen? Hieronder volgen een aantal punten die kenmerkend zijn:

- Een algoritme bestaat uit **stappen** die in een bepaalde volgorde **uitgevoerd** moeten kunnen worden. Bij uitvoering wordt die opeenvolging van stappen het door het algoritme beschreven **proces** genoemd. De uitvoerder van zo'n proces is de **processor**. Dit kan bijvoorbeeld een computer zijn die een computerprogramma uitvoert, of een persoon die een kookrecept uitvoert.
- De **beschrijving** van een algoritme moet **eindig** zijn. De **uitvoering** van een algoritme hoeft **niet eindig** te zijn, omdat er onbeperkte herhaling van een aantal stappen van een algoritme voor kan komen.
- Een algoritme moet op een bepaald **abstractieniveau** beschreven zijn. De processor die het algoritme uit moet gaan voeren, moet dit abstractieniveau ook begrijpen.
- Elke stap moet **éénduidig** zijn, dus niet voor meerdere interpretaties vatbaar.
- De volgorde waarin de verschillende stappen uitgevoerd worden, kan precies bepaald zijn: de uitvoering wordt dan **serieel** genoemd.
  De volgorde van uitvoering kan ook willekeurig zijn, het is niet belangrijk welke stap eerst uitgevoerd wordt: de uitvoering wordt **collateraal** genoemd.
  Tenslotte is het ook mogelijk dat een aantal stappen naast elkaar uitgevoerd moeten worden: de uitvoering wordt **parallel** genoemd.
  Een uitvoering wordt **sequentieel** genoemd als er geen parallelliteit in voorkomt.
  Merk op, dat voor echt parallelle uitvoering van een programma op een computer een multi-processor systeem of een aantal via een netwerk gekoppelde systemen nodig is.
- Een algoritme dat één of meer stappen bevat die zelf weer een algoritme bevatten, wordt een **samengesteld algoritme** genoemd. Een algoritme dat dergelijke stappen niet bevat, wordt een **elementair algoritme** genoemd.
- Bij de uitvoering van een algoritme kunnen operaties op objecten uitgevoerd worden. Objecten die aanwezig zijn voordat de uitvoering begint, worden met **invoer** of **input** aangeduid, objecten die na beëindiging van het algoritme overblijven, worden met **uitvoer** of **output** aangeduid. Objecten die alleen tijdens de uitvoering van een algoritme bestaan — bijvoorbeeld om tussenresultaten van een berekening te bevatten — worden **lokale objecten** genoemd.
- Samengestelde algoritmen kunnen **parameters** hebben, die de invoer en/of uitvoer van dat algoritme vormen.
- Objecten zijn — min of meer analoog aan algoritmen — **elementair** als ze uit één element bestaan; objecten zijn **samengesteld** als ze uit een aantal elementen, die weer van verschillend type kunnen zijn, bestaan.

Een programmeur die een algoritme opstelt, moet over dit algoritme een aantal beweringen kunnen doen. Deze betreffen de volgende zaken:

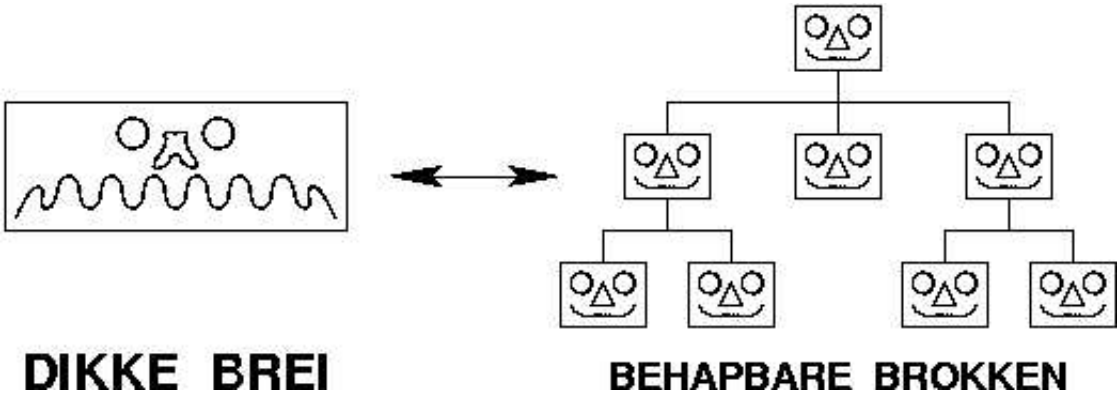
- **Voorwaarden**
  Onder welke voorwaarden werkt het algoritme? Wat zijn de toegestane invoerwaarden? Wat zijn de daarbij behorende mogelijke uitvoerwaarden? Welke beperkingen en veronderstellingen zijn er ten opzichte van het te realiseren proces gemaakt?
- **Correctheid**
  Wordt het proces door het algoritme precies beschreven? Is het verband tussen invoer- en uitvoerwaarden juist?
- **Terminatie**
  Eindigt het door het algoritme beschreven proces voor alle mogelijke invoerwaarden? Is het onmogelijk de processor in een niet eindigende herhaling te brengen?
  Dit geldt dus niet voor niet eindigende processen.

- **Efficiëntie**
  Hoe ingewikkeld is de uitvoering van dit algoritme? Is er een verband tussen de kosten van uitvoering en het aantal invoerwaarden? Zo ja, wat is dit verband? Hoe groot (of klein) is het programma? Gaat de grootte ten koste van de snelheid? Gaat de snelheid ten koste van de grootte?

2.2 Top-down methode

Een manier om problemen met behulp van een computerprogramma op te lossen, is het toepassen van de **top-down methode**. Hierbij wordt een groot probleem in deelproblemen opgesplitst, die op hun beurt weer in kleinere deelproblemen opgesplitst worden, enz. Er worden op deze manier **verfijningen** aangebracht zodanig dat elk niveau menselijkerwijze te overzien is. En uiteindelijk is de oplossing van het probleem tot in dusdanig detail beschreven, dat die gemakkelijk in een programmeertaal omgezet kan worden.

Duidelijk is, dat deze methode prima geschikt is om **algoritmen** op te stellen, omdat een structuur analoog aan samengestelde algoritmen gevolgd wordt.
Nadeel van deze methode is, dat er meestal te veel **iteratief** gewerkt moet worden. Want bij complexe problemen blijkt vaak pas op een lager niveau — dus na een aantal malen verfijnen — dat er op een hoger niveau iets in het ontwerp veranderd moet worden. Als het goed is, is dit echter al in de ontwerpfase (tijdens het opstellen van het algoritme) onderkend en blijkt het niet pas in de implementatiefase (tijdens het omzetten in een programmeertaal of het uittesten van het programma).



Figuur 2.1: *"Rommel-maar-aan" vs. top-down methode.*

2.3 Bottom-up methode

Wordt bij de top-down methode het probleem van bovenaf aangepakt, bij de **bottom-up methode** wordt het probleem aan de basis aangepakt. Bij grote projecten (om grote of complexe problemen op te lossen) zijn er vaak al een aantal **basismodulen** beschikbaar. Deze dienen dan als basis om de rest van het programma op te bouwen.

Voordeel is dat er — uitgaande van dergelijke bestaande èn werkende basismodulen — onderdelen van het programma afzonderlijk getest kunnen worden. Als de gegevensoverdracht en de logica tussen de verschillende delen van te voren in de ontwerpfase juist gedefinieerd is, dan zullen de delen ook als geheel werken.
Nadeel is dat er, beginnend aan de basis, bepaalde zaken of relaties vergeten of uit het oog verloren worden. Om dit te corrigeren, is er op zijn minst één nieuwe "ontwerpslag" gevolgd door een nieuwe "programmeerslag" nodig.

NB: *In de praktijk zal vaak een* **combinatie** *van top-down en bottom-up methoden gebruikt worden, waardoor meer van de voordelen van beide geprofiteerd kan worden.*

2.4 Voorbeeld

Stel dat de ledenadministratie van een sportvereniging geautomatiseerd moet worden. De korfbalvereniging M.E.E. (Mik 'Em Erin) is vrij plotseling gegroeid van 12 naar 120 leden en diverse leden van het bestuur hebben het dusdanig druk gekregen dat ze hun hoop op de PC van het zoontje van de penningmeester gevestigd hebben om hun werklast te verlichten. Dus de koppen worden bij elkaar gestoken om een oplossing te vinden.

Eerst wordt met de betrokkenen de **probleemstelling** duidelijk bepaald. Resultaat hiervan is dat men van mening is dat per lid de volgende gegevens opgeslagen moeten worden:
- naam
- adres
- telefoonnummer
- e-mailadres
- lidnummer
- ingangsdatum lidmaatschap
- opzegdatum lidmaatschap
- datum, geeft aan tot wanneer de contributie betaald is
- soort lid (junior, senior)

Verder moeten de volgende bewerkingen op de gegevens uitgevoerd kunnen worden:
- leden toevoegen
- mutaties op ledengegevens uitvoeren
- leden afvoeren
- ledengegevens afdrukken (alfabetisch op naam of op postcode)
- contributiefacturen afdrukken
- overzicht van nog te ontvangen contributies (wanbetalers!) afdrukken

De adviseur van de vereniging (het zoontje van de penningmeester dus, die nu als **systeemanalist** aan de slag gaat) heeft net een programmeerpracticum achter de rug en is van mening dat de geschetste problemen met behulp van een computer opgelost kunnen worden. Het bijhouden van de ledengegevens kan nog zonder problemen met een "papieren" kaartenbak gedaan worden, daar is echt geen computer bij nodig, maar het berekenen van de contributies is dermate tijdrovend, dat dat het gebruik van een computer rechtvaardigd. Bovendien kost het geen geld, omdat de penningmeester genereus aanbiedt dat PC èn zoontje best voor dit karweitje gebruikt kunnen worden.

De adviseur, die nu de pet van **systeemontwerper** opzet, maakt het volgende van het probleem:
- Er zullen twee gegevensbestanden gemaakt worden: het ene bevat de gegevens van het huidige ledenbestand, het andere bevat de gegevens van oud-leden (archieffunctie, van belang bij het zoveel-jarig bestaan van de vereniging). *Deze scheiding wordt aangebracht vanwege inhoud en om de grootte beperkt te houden.*
- Elk bestand bevat één regel (of record) per persoon, die de gegevens van de betreffende persoon bevat. Voor elk gegeven (of veld, of item) in het record is een vast aantal posities gereserveerd.
- Het invoeren, muteren en overhevelen van ledengegevens zal in eerste instantie met behulp van een tekstverwerkingsprogramma gedaan worden. Later zal hier een apart programma voor gemaakt worden, waarmee het mogelijk is invoer en mutaties op een aantal punten te controleren (bijvoorbeeld postcode: vier cijfers gevolgd door twee letters).
- Het sorteren van ledengegevens zal met een deelprogramma gedaan worden.
- Het afdrukken van ledengegevens zal met een deelprogramma gedaan worden.
- Het berekenen van contributie en het bedrukken van acceptgirokaarten zal met een deelprogramma gedaan worden.
- Het financiële overzicht van betaalde en nog te ontvangen contributies zal met een deelprogramma gedaan worden.

Bij dit ontwerp kan het volgende opgemerkt worden:
- Alle gegevens zijn op een **centraal** punt beschikbaar, namelijk in het huidige ledenbestand. Voordelen hiervan zijn dat de gegevens niet over verschillende bestanden versnipperd zijn en dat ze maar éénmaal voorkomen. Nadeel is dat er enorm veel gegevens bij elkaar kunnen komen te staan, waardoor bewerkingen, die maar op een klein deel van de gegevens gedaan worden, veel langer duren dan nodig is. Hier is dus een afweging gemaakt — afhankelijk van de te verwachten hoeveelheid gegevens — tussen één groot of meerdere kleinere bestanden. Dergelijke bestanden met onderlinge relaties worden ook een **database** genoemd.

- De verschillende afzonderlijke bewerkingen zijn ondergebracht in **deel**programma's. Hierdoor is een zekere opdeling gemaakt, waardoor het hoofdprobleem in kleinere problemen opgedeeld is. Voordelen hiervan zijn dat er bij het opstellen van algoritme en programma minder fouten gemaakt worden en dat de programma's overzichtelijker zijn.
- Een andere mogelijkheid is het maken van **aparte** (en dus kleinere) programma's voor de verschillende functies. Nadeel hiervan is dat een aantal gemeenschappelijke programma-onderdelen in elk programma (dus in n-voud) voorkomen, wat extra ruimte en onderhoud kost.
- Om het werken voor gebruikers gemakkelijk te maken, zal er een **menu** gemaakt worden, van waaruit de verschillende deelprogramma's door aanklikken gestart kunnen worden.

Nu gaat de **programmeur** (jawel, nog steeds hetzelfde zoontje!) aan de slag om deelalgoritmen op te stellen die de volledige oplossing van het probleem vormen.

Het deelprogramma *bereken de te betalen contributies en druk deze af* wordt hier verder uitgewerkt door het op te delen in (of te verfijnen tot) de volgende deelproblemen:
(1) lees de gegevens van alle leden uit het bestand in
(2) bereken de te betalen contributie voor elk lid
(3) druk per lid het te betalen bedrag met naam en adres af

Bij verdere verfijning blijken er twee mogelijkheden te zijn: eerst alle gegevens inlezen en daarna verwerken, of per persoon werken door gegevens van een persoon te lezen, deze meteen te verwerken en daarna de gegevens van de volgende persoon te lezen. Omdat er meerdere deelprogramma's zijn en omdat de omvang van de bestanden niet exorbitant groot zal worden, wordt er gekozen voor de eerste mogelijkheid. De functie "inlezen van gegevens" zal door alle deelprogramma's gebruikt kunnen worden.

Tijdens het verder verfijnen blijkt ook dat er een betere eerste verfijning gemaakt kan worden, die meer algemeen gesteld is, namelijk:
(1) vraag de naam van en open het gegevensbestand
(2) lees de gegevens van alle leden uit het gegevensbestand
(3) verwerk de gegevens (hier: bereken de te betalen contributie)
(4) druk de verwerkte gegevens af (hier: druk facturen af)

Deelprobleem 1 wordt verfijnd tot:
(1.1) vraag naam van bestand
(1.2) probeer het bestand te openen
- *als dit niet lukt, wordt een foutboodschap afgedrukt en wordt er naar 1.1 gegaan*
- *als dit wel lukt, wordt doorgegaan naar 1.3*

(1.3) controleer of het een goed bestand is
- *als dit niet zo is, wordt een foutboodschap afgedrukt en wordt er naar 1.1 gegaan*
- *als dit wel zo is, wordt doorgegaan naar 1.4*

(1.4) druk een boodschap af dat het bestand geopend is

Deelprobleem 2 wordt verfijnd tot:
(2.1) zolang er nog gegevens zijn
- *lees de gegevens van de huidige persoon*
- *sla de gegevens op*
- *ga naar de volgende persoon*

(2.2) druk een boodschap af dat de gegevens ingelezen zijn

Deelprobleem 3 wordt verfijnd tot:
(3.1) zolang er nog personen zijn
- *bereken de te betalen contributie voor de huidige persoon*
- *sla de gegevens op*
- *ga naar de gegevens van de volgende persoon*

(3.2) druk een boodschap af dat de gegevens verwerkt zijn

Deelprobleem 4 wordt verfijnd tot:
(4.1) zolang er nog personen zijn
- *druk de gegevens van de huidige persoon af*
- *ga naar de gegevens van de volgende persoon*

(4.2) druk een boodschap af dat de gegevens afgedrukt zijn

Bij deze — nog niet erg diepgaande — verfijningen gekomen, kan het volgende opgemerkt worden:
Er worden ergens gegevens vandaan gehaald (gegevensbestand), deze gegevens worden tijdelijk bewaard (sla de gegevens op), er worden nieuwe gegevens uit de oude gemaakt en er worden gegevens afgescheiden (druk gegevens af).
Maar **hoe** worden de gegevens tussendoor bewaard? Dat kan op verschillende manieren en is onder meer afhankelijk van hoeveelheid en soort gegevens en van gebruikte computer. Er kan bijvoorbeeld gebruik gemaakt worden van tijdelijke bestanden (werkt erg traag) of van opslag in het primaire geheugen van de computer (hoeveelheid gegevens kan niet enorm groot zijn).
De manier waarop dit opgelost wordt, heeft ook invloed op het algoritme!

$\Longrightarrow$ *Door goed te begrijpen wat het probleem is en wat de randvoorwaarden zijn, kan niet alleen goed bepaald worden hoe, maar ook op welk niveau (van de verfijningen) oplossingen gemaakt moet worden. En hierdoor worden onnodige complicaties op een hoger of lager niveau vermeden.*

Bij het opstellen van een algoritme is er dus steeds een terugkoppeling, waardoor het algoritme aangepast kan worden. Als een probleem goed verfijnd wordt, zal deze terugkoppeling niet zo ingrijpend zijn en zal er nauwelijks bijstelling op hoger gelegen niveau's nodig zijn.

Als het algoritme goed uitgewerkt is, zal het programmeren daarna nauwelijks problemen geven. Maar als opstellen van het algoritme en programmeren **gelijktijdig** gedaan worden, wordt het bij grotere problemen al gauw een janboel, waardoor de terugkoppeling veel ingrijpender is en daardoor sneller extra (logische) fouten in het ontwerp veroorzaakt.

In de uitwerking van dit voorbeeld is een aantal keren sprake van **herhaling**. De meeste programmeertalen hebben speciale instructies die een herhaling bevatten. Deze instructies zijn een soort keurslijf, maar zorgen er wel voor dat het programma een goede structuur krijgt. In de volgende sectie worden de herhaling en de andere basisstructuren behandeld.

2.5 Taalstructuren van computertalen

Bij het programmeren kunnen een aantal verschillende **structuren** onderscheiden worden. De eenvoudigste hiervan is de **reeks**, het uitvoeren van een aantal instructies na elkaar. Verder zijn er de **keuze**, het conditioneel uitvoeren van instructies, en de **herhaling**, het herhaald uitvoeren van dezelfde instructies.

- Een **reeks** bestaat uit een aantal instructies die na elkaar uitgevoerd worden:
  instructie 1
  instructie 2
  instructie 3

  Voorbeeld van een reeks:
  lees een getal A in
  lees een getal B in
  tel de waarde van A bij B op en plaats het resultaat in C
  druk de waarde van C af

- Een **keuze** is een structuur waarin, afhankelijk van een bepaalde voorwaarde, een reeks instructies al dan niet uitgevoerd wordt:
  ALS voorwaarde waar is DAN
  instructie 1
  instructie 2
  instructie 3
  ANDERS
  instructie 4
  instructie 5
  instructie 6
  EINDE

Hierbij worden, als er aan de voorwaarde voldaan wordt, **alleen** de instructies 1 t/m 3, tussen DAN en ANDERS, uitgevoerd. De instructies 4 t/m 6 worden in dit geval **niet** uitgevoerd.
Als er **niet** aan de voorwaarde voldaan wordt, gebeurt het omgekeerde. De instructies 4 t/m 6, tussen ANDERS en EINDE, worden nu **wel** uitgevoerd en de instructies 1 t/m 3 **niet**.

Voorbeeld van een keuze:

ALS het regent DAN
doe regenpak aan
pak de fiets
ga naar werk
ANDERS
neem regenpak mee
pak de fiets
ga naar werk
EINDE

NB: *De twee opdrachten "pak de fiets" en "ga naar werk" worden in beide gevallen uitgevoerd en kunnen daarom beter uit de keuze-instructie gehaald worden en er na gezet worden.*

- Een **herhaling** is een structuur waarin, afhankelijk van een bepaalde voorwaarde, een reeks instructies meerdere malen uitgevoerd wordt:

  ZOLANG voorwaarde waar is DOE
  instructie 1
  instructie 2
  instructie 3
  EINDE

  of:

  DOE
  instructie 1
  instructie 2
  instructie 3
  ZOLANG voorwaarde waar is

  Voorbeelden van dergelijke herhalingen:

  ZOLANG het regent DOE
  wacht een kwartier
  EINDE

  en:

  DOE
  wacht een kwartier
  ZOLANG het regent

  Hieruit is het verschil tussen de twee structuren te zien: in het eerste geval wordt er eerst naar de voorwaarde gekeken en worden de instructies mogelijk niet uitgevoerd (als er niet aan de voorwaarde voldaan wordt), terwijl in het tweede geval de instructies altijd minstens één keer uitgevoerd worden (daar wordt pas na uitvoering naar de voorwaarde gekeken).

In hoofdstuk 3 zullen we zien hoe deze structuren er in C uitzien.

2.6 Programmeerstijl

Programma's kunnen globaal in twee categoriën verdeeld worden. Er zijn éénmalig gebruikte weggooiprogramma's en er zijn programma's die minstens een aantal malen uitgevoerd worden. De laatste categorie bevat ook programma's die na enige tijd verbeterd of veranderd moeten worden. Belangrijk hierbij is dan, dat de manier waarop het programma werkt snel te begrijpen is.

NB: *De opdrachten die voor het practicum gemaakt moeten worden, vallen onder de laatste categorie.*

Het is dus belangrijk om de **structuur van het programma** ook in de programmacode uit te laten komen. Dit kan op een aantal eenvoudige manieren gedaan worden:
- Door op de juiste plaats één of meerdere blanko regels in de code tussen te voegen.
- Door bij keuzen en herhalingen in te springen. Hierdoor is het bereik van een herhaling duidelijk te zien, zeker als er een herhaling binnen een andere herhaling gebruikt wordt.

Daarnaast is het belangrijk om — ook weer op de juiste plaatsen — **goed commentaar** in een programma te zetten. Onder andere op de volgende plaatsen:
- Aan het begin van een programma kan de functie van het programma uitgelegd worden. Verder kan de naam van de programmeur (m/v) vermeld worden, tezamen met de datum dat het programma klaar was. Later, als het programma veranderd is, moet deze informatie bijgewerkt worden.
- Aan het begin van een programma kan voor de gebruikte variabelen een beschrijving van de inhoud gegeven worden.
- Bij afgeronde gedeelten in de code — zoals het begin van verfijningen — kan uitgelegd worden wat er precies gebeurt.

Tenslotte kan, door **sprekende namen** aan variabelen en constanten toe te kennen, veel duidelijk gemaakt worden over wat er in het programma gebeurt. Noem variabelen niet “A”, “B” en “C”, maar bijvoorbeeld “hoekA”, “hoekB” en “hoekC”.

2.7 Debuggen

Een programma kan helemaal volgens de regels opgesteld worden, maar desondanks foute resultaten geven. In computerjargon wordt dan gezegd dat “het programma een **bug** (Engelse uitspraak) bevat” (er optimistisch vanuit gaande, dat er slechts één fout in zit). Het vinden van een fout wordt **debuggen** (Engelse uitspraak met vernederlandste uitgang) genoemd.

Een eerste fase van debuggen is het nalopen van het programma. Hierbij is van belang een idee te hebben waar de fout optreedt. Het is soms mogelijk om uit de uitvoerwaarden die bij verschillende invoerwaarden horen af te leiden wat er fout gaat en waar dat in het programma gebeurt.

Een volgende fase is het uitschrijven van tussenresultaten. Op bepaalde punten in het programma worden waarden van variabelen afgedrukt, met de bedoeling uit deze waarden op te maken of er al iets misgegaan is of niet. Hiermee wordt het programmagedeelte waarin de fout optreedt steeds kleiner gemaakt.

Een laatste fase is het gebruik maken van een **debugger**, een programma dat speciaal ontworpen is voor het vinden van fouten. De belangrijkste mogelijkheden van een dergelijk programma zijn:

- De uitvoering van een programma kan op van te voren aangegeven plaatsen (**break points**) gestopt worden.
- De waarde van een variabele kan op een break point bekeken en zo nodig veranderd worden.
- Programma-instructies kunnen per instructie uitgevoerd worden, dat wil zeggen dat de uitvoering van het programma na uitvoering van een instructie bij de volgende instructie stopt. Dit heet **single step** mode.

2.8 Nauwkeurigheid

Computers maken nooit fouten. Dat klopt meestal, uitgezonderd kapotte computers en computers die in een te warme omgeving staan. Maar betekent nauwkeurig ook exact? Want toch kunnen er bijvoorbeeld uit **precies dezelfde** berekeningen, maar op verschillende computers uitgevoerd, **verschillende** antwoorden komen! Hiervoor zijn een aantal redenen te geven.

Een computer werkt **binair**.
Gevolg hiervan is bijvoorbeeld dat de waarde van de breuk 1/3 niet exact weergegeven kan worden. Wordt de breuk in een berekening gebruikt, dan wordt er een afrondingsfout geïntroduceerd, die bij verdere berekeningen door zal werken.

Ook een computer heeft **grenzen**!
Het bereik van de waarden van getallen (of het nu gehele getallen zijn of getallen met een decimale punt) in een computer is aan grenzen gebonden. Dus er zijn getallen die te groot of te klein zijn om weergegeven te worden. De uitkomst van een berekening kan dus gewoon een fout resultaat opleveren!

De ene computer is de andere niet, ze kunnen intern **verschillen**!
Naast het bereik kan de interne weergave van waarden met een decimale punt op kleine punten verschillen voor verschillende typen computers. En hierdoor kan de weergaven van hele kleine getallen verschillen.

Het zijn meestal dus geen grote verschillen, maar een klein verschil kan in grote berekeningen veel uit gaan maken...

2.9 Efficiëntie

Als laatste en paar opmerkingen over de efficiëntie van programma's. Want hoewel computers tamelijk snel zijn, zijn ze niet zo snel dat alles in een handomdraai gedaan wordt. Een programma goed maken is het belangrijkste, maar een programma efficiënt maken is vaak ook erg belangrijk. Nu is het zo dat het ene programma zich beter leent om sneller gemaakt te worden dan het ander. Een programma dat bestaat uit een aantal instructies die na elkaar als reeks uitgevoerd worden, heeft niet veel mogelijkheden om versneld te worden, maar een programma met een aantal herhalingen (liefst nog een aantal binnen elkaar) biedt mogelijkheden. Voor de manier waarop kunnen geen vaste regels gegeven worden, het is vaak een kwestie van iets "zien" (het wordt ook "creatief zijn" genoemd).

Een paar regels die gegeven kunnen worden zijn:
- Let op of er toekenningen gedaan worden in een herhaling, die ook buiten die herhaling gedaan kunnen worden.
- Let op of er meerdere malen toekenningen gedaan worden, die eenmaal via een hulpvariabele gedaan kunnen worden.
- Let op of een herhaling voortijdig afgebroken kan worden (de herhaling wordt dus niet zoveel maal voor nop uitgevoerd).

Afhankelijk van de gebruikte compiler, wordt er door die compiler vaak ook al aan optimalisatie gedaan.

NB: *Commentaar bij een programma dat gecompileerd wordt, heeft* **niet** *tot gevolg dat dat programma langzamer loopt. Dit is dus* **absoluut geen reden** *om geen commentaar in een programma te zetten.*

Hoofdstuk 3

De programmeertaal C

Dit hoofdstuk behandelt heel beknopt de programmeertaal C. Voor een uitgebreide behandeling wordt verwezen naar [1]. Een korte beschrijving van C instructies wordt gegeven in appendix B van deze handleiding.

De taal die voor het practicum gebruikt wordt, stamt uit het begin van de jaren zeventig en is dus niet de modernste. Er is echter voor gekozen, omdat alle taalelementen en -constructies die bij Programmeren behandeld worden aanwezig zijn en omdat de kans op gebruik van de taal later groot is (bij practica, tijdens stages bij afdelingen en in de wetenschappelijke wereld).

3.1 Programmastructuur en taalregels

Een C programma is volgens een bepaalde structuur opgebouwd. Bovendien moeten een aantal taalregels gevolgd worden om een wat syntax betreft juist C programma te schrijven. In deze sectie worden programmastructuur en taalregels behandeld.

Eerst iets algemeens over de **programmastructuur** (details komen later). Een programma bestaat uit twee delen:
- Een **globaal gedeelte** dat globale definities en declaraties bevat.
- Een **functiegedeelte** dat een functie met de vaste naam `main` bevat.

Het idee hierachter is dat globale dingen ook lokaal (in ons geval in de functie `main`) bekend zijn en gebruikt kunnen worden. Lokale dingen zijn (zoals de naam zegt) slechts lokaal bekend en te gebruiken. Dat lijkt nu nogal overbodig, maar later zullen we zien dat dit een goed idee is.

Veel voorkomende dingen zijn **variabelen**. Een variabele bevat een bitpatroon van nullen en enen. Afhankelijk van het **type** van de variabele wordt dit bitpatroon op een bepaalde wijze geïnterpreteerd.
Voorbeelden van typen zijn gehele getallen (hiervoor wordt `int` van integer gebruikt), getallen met een decimale punt (`float` van floating point), tekens (`char` van character).
Twee verschillende variabelen die precies hetzelfde bitpatroon bevatten maar van verschillend type zijn, hebben dus waarschijnlijk geheel verschillende waarden in een programma!

Alle dingen die in een programma gebruikt worden, moeten bekend zijn. Variabelen, bijvoorbeeld, moeten stuk voor stuk "aangegeven" worden. Dit **declareren** houdt in dat de naam van de variabele aan een bepaald type gekoppeld wordt. Hierdoor is het bijvoorbeeld mogelijk om te controleren of bepaalde bewerkingen op bepaalde variabelen wel legaal zijn en of er tikfouten in de namen van gebruikte variabelen zitten.

Nu een paar opmerkingen over de **syntax** van een C programma (zie appendix A van [1]).
- **Instructies** (inclusief declaraties) worden altijd afgesloten met een puntkomma, waardoor een instructie of declaratie meerdere regels kan beslaan.
  Onderdelen van instructies worden gescheiden door één of meer spaties, één of meer TAB's of door de overgang naar een nieuwe regel.
- **Identifiers** (bijvoorbeeld namen van variabelen) zijn "woorden" die bestaan uit een combinatie van letters, cijfers en het teken _ (underscore) maar die niet met een cijfer mogen beginnen.

- **Keywords** zijn woorden die al een bepaalde betekenis hebben (bijvoorbeeld `int`) en mogen/kunnen niet voor eigen doeleinden gebruikt worden.
- Bij **declaratie** van een variabele wordt het **type** van de variabele gevolgd door de naam van de variabele en een puntkomma om af te sluiten. Eventueel kunnen meerdere variabelen van hetzelfde type tegelijk gedeclareerd worden door de namen ervan door komma's te scheiden.
- Een **toekenning** bestaat uit de naam van een variabele waaraan een waarde toegekend gaat worden, gevolgd door een gelijkteken `=`, gevolgd door een uitdrukking om de waarde te berekenen, gevolgd door de afsluitende puntkomma.
  De uitdrukking kan een enkele variabele zijn, een constante, of een combinatie van bewerkingen tussen één of meer constanten en/of variabelen.
- **Constanten** worden gedefinieerd door het woord `#define`, gevolgd door de naam van de constante, gevolgd door de waarde van de constante (die meteen het type bepaalt) en dit keer NIET gevolgd door een afsluitende puntkomma.
- Het **tussenvoegen** van tekst- of programmadelen in een ander bestand gebeurt door het woord `#include` gevolgd door een naamspecificatie van dat bestand en wederom NIET gevolgd door een afsluitende puntkomma.
  Staat de naam tussen `< >` dan gaat het om een systeembestand, staat de naam tussen `" "` dan kan een eigen bestand opgegegven worden.
- **Commentaar** begint met de combinatie `/*` (slash asterisk) en eindigt met de eerstvolgende `*/` (asterisk slash). Commentaar kan dus ook meerdere regels beslaan.

De functie `main` bestaat uit een **programmakop** (of "**header**") en een **codegedeelte** waarin het het eigenlijke werk gedaan wordt. Het codegedeelte staat tussen twee accoladen {} ingeklemd en bestaat uit een declaratiegedeelte voor lokale variabelen, gevolgd door uitvoerbare instructies. Elke in een functie gebruikte variabele moet gedeclareerd worden, d.w.z. er moet aangegeven worden wat voor type variabele het is (details later).
De **header** van een functie bestaat uit een type-indicator (in het geval van `main` is dit `int` voor integer of geheel getal), de naam van de functie (in het geval van `main` is dit `main`), en een (mogelijke lege) lijst van parameters van de functie tussen ronde haken ((in het geval van `main` een lege lijst). Functies worden later behandeld.

Bij het uitvoeren van een C programma wordt vanuit het besturingssysteem de functie `main` gestart en worden achtereenvolgens de instructies van die functie uitgevoerd. Een `return`-instructie zorgt ervoor dat het programma beëindigd wordt en dat de controle teruggaat naar het besturingssysteem. Als er geen `return` in de code staat, wordt die impliciet door de compiler aan het eind van de uitvoerbare code toegevoegd.

Voorbeeldprogramma `progstruct.c`:

```
/*************************************************************************/
/* Status:   03-apr-2002, pfk.
/* Functie:  Bereken omtrek van gegeven rechthoek.
/*************************************************************************/

/* Globaal gedeelte. *****************************************************/

#include <stdio.h>          /* definities voor standaard I/O-functies */

/* Functiegedeelte. ******************************************************/

int main ()                 /* header van functie "main" */
{                           /* begin van het codegedeelte van "main" */
  int breedte,                 /* declaratie van lokale variabelen */
      lengte,
      omtrek;

  breedte = 10;                /* berekeningen e.d. */
  lengte  = 11;
  omtrek  = 2 * (breedte + lengte);
  printf ("\nOmtrek van rechthoek: %d\n", omtrek);

  return (0);                  /* einde van uitvoering van "main" */
```

```
}                           /* einde van het codegedeelte van "main" */
```

3.2 I/O van/naar toetsenbord/beeldscherm

Meestal heeft een programma gegevens nodig die in het programma bewerkt worden, waarna de resultaten weer teruggegeven worden. **Invoer** (of **input**) van gegevens en **uitvoer** (of **output**) van gegevens worden tezamen met de Engelse term **I/O** aangeduid.

In deze sectie wordt iets over I/O via toetsenbord en beeldscherm verteld, namelijk het deel dat op **formatted I/O** betrekking heeft. Hierbij worden **conversiecodes** gebruikt om aan te geven welk formaat gegevens hebben. Later, in een volgende sectie, zal I/O van en naar bestanden behandeld worden.

Inlezen van gegevens van het toetsenbord gaat met de functie `scanf` (scan formatted). Dexe heeft als eerste argument een textstring (een aantal tekens ingesloten in dubbele aanhalingstekens), die conversiecodes bevat om aan te geven hoeveel waarden van welk type er ingelezen moeten worden. De ingevoerde tekens van het toetsenbord worden volgens de conversiecodes geïnterpreteerd.
De volgende argumenten zijn de adressen van de variabelen waarin de waarden gezet moeten worden. Het aantal en het type van de opgegeven variabelen moet kloppen met de opgegeven conversiecodes om verrassingen te voorkomen!

Afdrukken van gegevens op het scherm gaat met de functie `printf` (print formatted), die als eerste argument ook een textstring heeft, waarin, naast gewone tekst die afgedrukt moet worden, ook speciale codes voor conversie van variabelen staan.

De **conversiecodes** bestaan uit een procent `%` gevolgd door een letter die aangeeft welke conversie er gedaan moet worden. Bij het inlezen van gegevens worden er tekens ingelezen die volgens de opgegeven conversiecode omgezet worden naar de weergave van het gewenste type. Dit resultaat wordt daarna in een variabele van dat type gezet.

Voorbeeldprogramma `io.c`:

```
/*************************************************************************
/* Status:   08-apr-2002, pfk.
/* Functie:  Programma om het oppervlak van twee cirkels uit te rekenen.
/*           Voorbeeld van formatted I/O.
/*************************************************************************/

#include <stdio.h>

#define PI 3.141593  /* constante met benaderingswaarde van pi */

int main ()
{
  /* declaraties */
  float oppervlak;         /* oppervlak van cirkel */
  float straal1, straal2;  /* stralen van cirkels */

  /* vraag en lees waarde voor straal */
  printf ("Programma om het oppervlak van twee cirkels uit te rekenen:\n");
  printf ("  geef de stralen van de twee cirkels [float float]: ");
  scanf  ("%f %f", &straal1, &straal2);

  /* bereken oppervlak apart en toon uitkomst eerste cirkel */
  oppervlak = PI * straal1 * straal1;
  printf ("Oppervlak van eerste cirkel = %10.2f\n", oppervlak);

  /* bereken oppervlak in functie en toon uitkomst tweede cirkel */
  printf ("Oppervlak van tweede cirkel = %10.2f\n", PI * straal2 * straal2);
```

```
  return (0);
}
```

3.3 Taalstructuren

In hoofdstuk 2 zijn een aantal structuren besproken die in computertalen voorkomen: de reeks, de keuze en de herhaling. In deze sectie worden de laatste twee voor C behandeld en wordt er aandacht besteed aan inspringen om de structuur van een programma duidelijk te maken.

3.3.1 Keuze: `if` en `switch`

Er zijn een aantal mogelijkheden voor de keuze bij C. De eenvoudigste is één instructie die alleen uitgevoerd wordt als de uitkomst van de uitdrukking tussen ronde haken waar is:
if (expression) clausule_statement_if_true;

Hetzelfde geval, maar nu worden meerdere instructies in de clausule uitgevoerd:
if (expression)
{ *clausule_statements_if_true;*
}

Het geval waarbij er ook instructies uitgevoerd worden als de uitkomst van de uitdrukking tussen ronde haken NIET waar is:
if (expression)
{ *clausule_statements_if_true;*
}
else
{ *clausule_statements_if_false;*
}

Voorbeeld om absolute waarde van `x` te berekenen:

```
if (x<0)  x = -x;
```

Een voorbeeld van "**geneste**" `if`'s. Hierbij worden meerdere uitdrukkingen na elkaar getest, totdat er een uitdrukking waar is. De daarbij behorende clausule wordt uitgevoerd en de geneste `if` wordt beëindigd:

```
if (x<0)                              if (expression1)
   printf ("x is negatief");             clausule_statement1;
else                                  else
   if (x==0)                             if (expression2)
      printf ("x is nul");                  clausule_statement2;
   else                                   else
      printf ("x is positief");             clausule_statement3;
```

Als er heel veel uitdrukkingen getest worden, wordt er ook steeds verder ingesprongen en om dit te voorkomen wordt het volgende gedaan:

```
if (x<0)                              if (condition)
   printf ("x is negatief");             clausule_statement;
else if (x==0)                        else if (expression2)
   printf ("x is nul");                  clausule_statement2;
else                                  else if (expression3)
   printf ("x is positief");             clausule_statement3;
                                      else
                                         clausule_statement4;
```

Voor een keuze met meer dan twee mogelijkheden en voor specifieke waarden kan ook de `switch` gebruikt worden:
switch (expression)
{ *case l1: statements*$_{l1}$*;* * *hier kunnen meerdere statements staan* *\

break; * *einde* `case`, *verlaat* `switch` *\
...
case ln: $statements_{ln}$*;*
break;
default: $statements_{default}$*;*
break;
}

Voorbeeld:

```
int maand;
scanf ("%d", &maand);
switch (maand)
{ case 1:  printf ("januari");
           break;
  ...
  case 12: printf ("december");
           break;
  default: printf ("onbekende maand!");
           break;
}
```

3.3.2 Herhaling: `while`, `do while` en `for`

De `while` herhalingsstructuur met één statement:
while (expression) statement;

De `while` herhalingsstructuur met meerdere statements:
while (expression)
{ *statements;*
}

De `do while` herhalingsstructuur:
do
{ *statements;*
}
while (expression);

De `for` herhalingsstructuur ziet er als volgt uit:
for (expression1; expression2; expression3)
{ *clausule_statements;*
}

De uitvoering van de `for` herhalingsstructuur wordt gecontroleerd door een loopvariabele. De uitvoering van de loop gaat dan als volgt:

`initialisatie:`
expression1 wordt uitgevoerd
`loop:`
expression2 wordt uitgevoerd
levert deze `true` dan
wordt de loopclausule uitgevoerd
wordt *expression3* uitgevoerd
wordt naar `loop:` gegaan voor de volgende herhaling
levert deze `false` dan wordt de loop beëindigd

In de praktijk wordt de uitvoering van de `for` herhalingsstructuur meestal gecontroleerd door een loopvariabele. Dit gaat dan als volgt:
`initialisatie:`
de loopvariabele wordt geïnitialiseerd door *expression1*

`loop:`
de loopvariabele wordt getest door *expression2*
levert deze `true` dan
wordt de loopclausule uitgevoerd
wordt de loopvariabele aangepast door *expression3*
wordt naar `loop:` gegaan voor de volgende herhaling
levert deze `false` dan wordt de loop beëindigd

Voorbeeld om de getallen 1 tot en met 10 af te drukken:

```
int i;
for (i=1; i<=10; i++)  printf ("Waarde is %d\n", i);
```

3.3.3 Onderbreken van herhalingen: `break` en `continue`

Er zijn twee mogelijkheden om vanuit de herhalingsclausule uit te breken uit de "sleur" van de herhaling. Hiervoor zijn twee mogelijkheden, ten eerste het volledig verlaten van de herhaling met `break` (en doorgaan met de volgende instructie) en ten tweede het onderbreken van de huidige herhaling en doorgaan met de volgende herhaling met `continue` (afhankelijk van de algemene voorwaarden waaronder de herhaling werkt).

Een voorbeeld hiervan:

```
for ( i = 0; i < count; i++ )
{ if (i==j)  continue;     \* volgende herhaling *\
   if (i<j)  break;        \* verlaat for-loop *\
   j--;
}
```

3.3.4 Inspringen

Door op de juiste manier in te springen, kan de structuur van een programma duidelijk gemaakt worden en kan er bovendien gemakkelijk op fouten in die structuur gecontroleerd worden.
Sommige editors gaan automatisch inspringen als ze denken dat er een C-programma gemaakt wordt.

Door de statements in de clausules van keuze- en herhalingsinstructies in te laten springen, is meteen te zien waar die clausules beginnen, waar ze ophouden en welke statements ertoe behoren.

Slecht voorbeeld van inspringen, de laatste instructie wordt ALTIJD uitgevoerd, maar dit is niet goed te zien door het in laten springen van die instructie:

```
if (x<0)
   x = -x;
   y = -y;
```

Goed voorbeeld (nu is aan het inspringen te zien dat de laatste instructie NIET bij de `if`-clausule hoort):

```
if (x<0)
   x = -x;
y = -y;
```

3.4 Array's

Bij het verwerken van gegevens voor een ledenbestand — zie het voorbeeld in het vorige hoofdstuk — werd in eerste instantie gekozen voor het **eerst** inlezen van de gegevens van alle leden en het **daarna** berekenen van de contributies. Dat betekent dat de gegevens voor de verschillende leden in een aantal variabelen van het programma opgeslagen moeten zijn. Voor drie leden is dat geen probleem, bijvoorbeeld LIDNR1, LIDNR2, LIDNR3, CONTRLID1, CONTRLID2, CONTRLID3 en nog wat andere variabelen (allemaal in drievoud) om de nodige gegevens te bewaren en de nodige bewerkingen te kunnen doen. Maar nu voor 120 leden! Het is geen doen als alles in 120-voud moet en dat kan dan ook op een handiger manier.

Het is namelijk mogelijk om in een declaratie in één keer een rij variabelen van éénzelfde type aan te geven. Zo'n rij of **array** heeft een naam en de verschillende elementen worden aangegeven met die naam gevolgd door een **index** tussen rechte haken. In de declaratie van het array wordt het bereik van de index aangegeven,bijvoorbeeld:

```
int lidnrs[120];
```

Nu is ook in te zien dat herhaling en array een ijzersterke combinatie vormen, want met één statement kunnen alle lidnummers in één keer ingelezen worden:

```
#define AANTAL_LEDEN 120
int lidnrs[AANTAL_LEDEN];
int i;
for (i=0; i<AANTAL_LEDEN; i++) scanf ("%d", &lidnrs[i]);
```

Bij de voorgaande code kan het volgende opgemerkt worden:
- Dat het erg handig is de constante `AANTAL_LEDEN` te gebruiken, want als het aantal leden verandert, hoeft alleen deze constante in het programma aangepast te worden om na compileren de waarde in het gehele programma goed te hebben.
- Dat het aanpassen van deze ene constante bij frequente mutaties in het ledenaantal op zich weer niet erg handig is. Hiervoor komen later betere, meer flexibele oplossingen!

Het is ook mogelijk om meerdere indices te declareren en zo hogerdimensionale arrays te maken. Als voorbeeld hier een matrixvermenigvuldiging:

```
float m[3][3], m1[3][3], m2[3][3];
float sum;
int   i, j, k;

/* code om m1 en m2 met waarden te vullen */

/* bereken m = m1 * m2 */
for ( i = 0; i < 3; i++ )
{ for ( j = 0; j < 3; j++ )
  { sum = 0.0;
    for ( k = 0; k < 3; k++ )
      sum += m1[i][k] * m2[k][j];
    m[i][j] = sum;
  }
}
```

3.5 Tekst

Een character constante wordt tussen enkele aanhalingstekens aangegeven `'a'` en een string constante wordt aangegeven tussen dubbele aanhalingstekens `"Dit is een tekststring!"`.

Voor variabelen is het type `char` voor een enkel teken en array van `char` voor een string (een aantal op elkaar volgende tekens).

Hierbij de volgende opmerkingen:
- De variabele `teken` die als `char teken` gedeclareerd is, bevat één teken en er kan dus bijvoorbeeld het volgende mee gedaan worden:
  ```
  teken = 'a';
  if (teken == 'a') doe_iets_nuttigs;
  ```
- De string constante `"Dit is een tekststring!"` bevat 23 tekens, maar is intern 24 tekens lang, omdat er een "afsluitingsteken" toegevoegd wordt, namelijk het teken `'\0'` (backslash nul).
  Strings worden geacht altijd afgesloten te zijn met dit teken, bij string constanten wordt het automatisch gedaan, bij eigen maaksels moet je dit teken zelf toevoegen!
- De bovengenoemde string constante kan bijvoorbeeld als volgt in een array van `char` opgeslagen worden:

```
char regel[24];
regel = "Dit is een tekststring!";
```

- Een ander voorbeeld:

```
char woord[4];
woord[0] = 'H';
woord[1] = 'i';
woord[2] = '!';
woord[3] = '\0';
```

dit kan – alleen bij declaratie – korter gedaan worden (de dimensie wordt automatisch vastgesteld):

```
char woord[] = "Hi!";
```

- Functies om bewerkingen op strings uit te voeren, gaan er vanuit dat de aangeboden strings met `'\0'` afgesloten zijn.
  Is er geen `'\0'` aanwezig, dan wordt er net zolang doorgelezen totdat het teken gevonden wordt, totdat er een maximum aantal tekens bereikt is of totdat er een andere fout optreedt!
- Hieronder een paar string functies (zie appendix B van [1]):
  String kopiëren: `strcpy (string_kopie, string_origineel);`
  Lengte van string opvragen: `lengte = strlen (string_origineel);`

3.6 Pointers en dynamische alloceren

3.6.1 Pointers

Elke variabele kan beschouwd worden als een hokje met inhoud. De inhoud is de **waarde** van de variabele en via het **adres** van het hokje kan die waarde gebruikt of veranderd worden.
Na declaratie van een variabele wordt het adres impliciet gebruikt bij bewerkingen met of op die variabele. Maar het is in bepaalde gevallen ook handig om het adres van een variabele te kunnen manipuleren in plaats van de waarde.

Er kunnen ook **pointer** variabelen gedeclareerd worden, dat zijn variabelen die als waarde het adres van een andere variabele bevatten.

Enige opmerkingen hierover:

- Declaratie van een pointer variabele wordt gedaan door tussen het type en de naam een asterisk `*` te plaatsen (mag tegen het type aan, of tegen de naam aan, maar er moet wel ergens een spatie tussen type en naam zitten).
- Het adres van een "gewone", niet-pointer variabele wordt aangegeven door een ampersand `&` voor de naam van de variabele te zetten.
- De inhoud van de lokatie waar een pointer variabele naar wijst, wordt aangegeven met een asterisk `*` vóór de naam van de pointer variabele. De pointer wordt dan dus echt als pointer gebruikt.
- Het type van een pointer en van de variabele waarnaar die pointer wijst, moeten hetzelfde zijn!
- Voorbeeld waarbij een pointer variabele en een gewone variabele gedeclareerd worden, het adres van de gewone variabele in de pointer variabele gezet wordt en de inhoud van de gewone variabele via de pointer variabele afgedrukt wordt. Daarna wordt de waarde van de gewone variabele veranderd, waarna die gewijzigde inhoud weer via de pointer variabele afgedrukt wordt:

```
int *ptr_naar_gewone_var;
int gewone_var = 0;
ptr_naar_gewone_var = &gewone_var;
printf ("Waarde van gewone variabele is %d.\n", *ptr_naar_gewone_var);
gewone_var = 10;
printf ("Waarde van gewone variabele is nu %d.\n", *ptr_naar_gewone_var);
```

- Ander voorbeeld:

```
ptr_naar_x = &x;
y = *ptr_naar_x;
```

Dit is dus hetzelfde als:

```
y = x;
```

- Pointers en arrays hebben ook wat met elkaar. Per definitie is de naam van een array (zonder index en zonder de rechte haken) een pointer naar het eerste element van het array.
  Als een pointer naar een arrayelement wijst en de pointer wordt met de waarde 1 verhoogd, dan wijst de pointer daarna naar het volgende element. Dit wordt impliciet gedaan en gaat goed mits pointer en array hetzelfde type hebben (waardoor de pointerwaarde dus met het goede aantal bytes verhoogd wordt om naar de juiste lokatie van het volgende arrayelement te wijzen).
  Bij de volgende declaraties:
  ```
  int *ptr_naar_int;
  int arr_van_int[10];
  ```
  geldt dus dat de volgende twee statements precies hetzelfde doen, namelijk het adres van het eerste arrayelement in de pointer zetten:
  ```
  ptr_naar_int = &arr_van_int[0];
  ptr_naar_int = arr_van_int;
  ```
  Een enkel arrayelement gedraagt zich dus als een normale variabele.

3.6.2 Dynamisch alloceren

Mooi, pointers, maar levert het meer op dan alleen maar gegoochel met ampersands en asterisks? Jawel, tezamen met het dynamisch alloceren van geheugen vormen pointers een andere ijzersterke combinatie (vergelijk: herhaling met array)!

Bij het declareren van variabelen wordt er een stukje geheugen gereserveerd waarin de waarde van die variabele opgeborgen wordt. Deze stukjes geheugen zijn tijdens de uitvoering van het programma niet meer anders te gebruiken, ze zijn aan het begin van de uitvoering van een programma **statisch** gealloceerd.

Door een deel van het geheugen vrij te houden (zowel van statische allocatie als van programmacode) en door functies beschikbaar te stellen om tijdens de uitvoer van een programma ruimte voor variabelen in dat vrije deel te alloceren, is deze vrije ruimte enorm flexibel te gebruiken.
Een matrix van een van te voren onbekende grootte inlezen? *Geen probleem!*
Gegevens van een snel wisselend aantal leden inlezen? *Geen probleem!*

Voorbeeld van een allocatie van een array van N integers:
```
int *ptr_naar_array_van_int;
ptr_naar_array_van_int = (int *) malloc (N*sizeof(int));
```

De functie `sizeof(type)` geeft het aantal bytes dat per element van `type` gereserveerd moet worden.
De `(int *)` zorgt ervoor dat de functie `malloc` een functiewaarde van het type pointer-naar-int terug geeft.

3.7 Functies

In voorbeelden hebben we al kennis gemaakt met functies. Dit zijn stukjes programma die een bepaalde bewerking uitvoeren.

Als een bepaalde bewerking een aantal malen op verschillende plaatsen in een programma voorkomt, dan moeten de opdrachten voor die bewerking evenzovele malen in dat programma voorkomen. Het is echter mogelijk om de betreffende opdrachten éénmalig apart te zetten en vervolgens op elke plaats in het programma waar die opdrachten uitgevoerd moeten worden, te verwijzen naar die apart gezette opdrachten. Het volgende gebeurt dan: vanuit het programma wordt naar de aparte opdrachten gegaan, deze worden uitgevoerd, er wordt na afloop weer teruggegaan naar de juiste plaats in het programma om met de volgende opdracht verder te gaan.

Ook gekompliceerde bewerkingen kunnen op deze manier afgehandeld worden: de gewenste — en mogelijk door iemand anders gemaakte — bewerkingen kunnen zo vanuit een programma gebruikt worden.
Later zullen we zien hoe nuttige functies in een “bibliotheek” gezet kunnen worden, waarna die bibliotheek gebruikt kan worden door programma’s om de benodigde functies te leveren (analoog aan de I/O-functies uit de `stdio` bibliotheek).

Een **functie** is een stukje programma dat een bepaalde bewerking uitvoert en het resultaat daarvan eventueel als **functiewaarde** teruggeeft.

3.7.1 Algemeen

De syntax van een functie in C ziet er als volgt uit:
functietype functienaam (parameterlijst)
{ *declaraties;*
instructies;
}

Enige verfijningen hierop:
- Het functietype is een bestaand type, zoals `int` of `float`. Maar het is ook mogelijk een functie te maken die **geen** functiewaarde teruggeeft. Een dergelijke functie heeft type `void`.
- De parameterlijst hoeft geen parameters te bevatten, maar de ronde haken moeten in ieder geval aanwezig zijn.
- Een parameter in een parameterlijst wordt voorafgegaan door het bij die parameter behorende type.
- Meerdere parameters in een parameterlijst worden gescheiden door komma's.
  Op deze manier kan de compiler "type-checking" doen, d.w.z. controleren of de typen bij aanroep van een functie gelijk zijn.
- Het `return` statement dient om de functie te verlaten.
  Instructies die op een `return` *statement volgen zonder dat daar op een andere manier bij te komen is, worden dus niet meer uitgevoerd!*
- De functiewaarde van een functie wordt teruggegeven als argument van het `return` statement. Dit argument mag tussen ronde haken staan, maar dit hoeft niet.
  Bij type `void` *is er geen functiewaarde die teruggegeven wordt en heeft het* `return` *statement dus geen argument.*
- Declaraties die in een functie gedaan worden, gelden alleen binnen die functie.
- Bij het uitvoeren van een functie worden er kopieën van de parameters gemaakt en deze kopieën worden bij het uitvoeren van de functie gebruikt.
  Gevolg hiervan is, dat de waarde van een parameter, die in de functie veranderd wordt, buiten die functie dus niet verandert!

Een voorbeeld van een functie die het berekenen van de kracht tussen twee puntladingen volgens de wet van Coulomb uitvoert:

```
#include <stdio.h>

double CoulombsLaw (double q1, double q2, double r)
/*********************************************************************
/* Coulomb's law for the force acting on two point charges
/* q1 and q2 at a distance r.
/* MKS units are used.
/*********************************************************************/
{
  double k = 8.9875e9;
  return k * q1 * q2 / (r * r);
}

int main ()
{ printf ("%e newtons\n", CoulombsLaw (1.6e-19, 1.6e-19, 5.3e-11));
  return 0;
}
```

3.7.2 Call by value, call by reference

In de vorige sectie is opgemerkt dat de argumenten van functies bij de aanroep van een functie ingevuld worden met een **kopie van de waarde** van de variabele die in de aanroep opgegeven wordt. Hierdoor is het dus niet mogelijk om de waarde van een parameter die in de functie veranderd wordt, "terug te geven" na afloop van die functie. De parameters worden "**by value**" aangeroepen.

De functiewaarde dient dus om het resultaat van een functie terug te geven. Maar soms zijn meer resultaten die niet via één functiewaarde teruggegeven kunnen worden. Om dit toch te kunnen doen, kunnen parameters ook "**by reference**" aangeroepen worden. In dit geval wordt niet de waarde van de parameter gekopieerd, maar wordt het **adres** van de parameter als argument meegegeven.

Enige opmerkingen hierover:
- Het **adres** van een variabele wordt aangegeven door een ampersand `&` voor de naam van de variabele te zetten.
- Wordt het adres van een variabele meegegeven, dan moet in de functie zelf een `*` voor de variabele gezet worden als het om de **inhoud** van die variabele gaat.
  De `*` (dit wordt dereferencing genoemd) is dus de omgekeerde operator van de `&`.
- Een array wordt altijd met een pointer aangegeven, dus als een array als argument van een functie gebruikt wordt, zijn de elementen van dat array wèl in de functie te veranderen!
  Er hoeft dus ook geen `&` voor de naam van het array te staan.
  Dit kan wel, maar dan wordt het adres van het adres van het array meegegeven!.
- Een enkel array-element wordt niet met een pointer aangegeven, dus hiervoor geldt hetzelfde als voor "niet-array" variabelen.

Voorbeeld met verschillende functie-aanroepen om "call by value" en "call by reference" duidelijk te maken:

```
#include <stdio.h>

int veranderNietInFunctie (int geheel)
{ geheel += 1;
  return (geheel);
}

int veranderWelInFunctie (int *geheel)
{ *geheel += 1;
  return (*geheel);
}

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

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

void main ()
{
  int var1, var2;

  var1 = 10;
  printf ("Waarde van var1 voor veranderNietInFunctie:       %d\n", var1);
  printf ("Waarde van functie veranderNietInFunctie:         %d\n",
           veranderNietInFunctie(var1));
  printf ("Waarde van var1  na  veranderNietInFunctie:       %d\n\n", var1);

  var1 = 10;
  printf ("Waarde van var1 voor veranderWelInFunctie:        %d\n", var1);
  printf ("Waarde van functie veranderWelInFunctie:          %d\n",
```

```
          veranderWelInFunctie(&var1));
  printf ("Waarde van var1  na  veranderWelInFunctie:        %d\n\n", var1);

  var1 = 11;
  var2 = 22;
  printf ("Waarde var1 en var2 voor verwisselZonderPointers: %d %d\n", var1, var2);
  verwisselZonderPointers (var1, var2);
  printf ("Waarde var1 en var2  na  verwisselZonderPointers: %d %d\n", var1, var2);

  var1 = 11;
  var2 = 22;
  printf ("Waarde var1 en var2 voor verwisselMetPointers:    %d %d\n", var1, var2);
  verwisselMetPointers (&var1, &var2);
  printf ("Waarde var1 en var2  na  verwisselMetPointers:    %d %d\n", var1, var2);

}
```

Uitvoer van dit programma:

```
Waarde van var1 voor veranderNietInFunctie:       10
Waarde van  functie  veranderNietInFunctie:       11
Waarde van var1  na  veranderNietInFunctie:       10

Waarde van var1 voor veranderWelInFunctie:        10
Waarde van  functie  veranderWelInFunctie:        11
Waarde van var1  na  veranderWelInFunctie:        11

Waarde var1 en var2 voor verwisselZonderPointers: 11 22
Waarde var1 en var2  na  verwisselZonderPointers: 11 22

Waarde var1 en var2 voor verwisselMetPointers:    11 22
Waarde var1 en var2  na  verwisselMetPointers:    22 11
```

3.7.3 Bereik

In het voorgaande is al duidelijk geworden dat niet alle gedeclareerde variabelen overal in het programma bekend hoeven te zijn. Een bron van fouten in grote programma's is het gebruik van dezelfde variabele voor verschillende doelen. Een variabele wordt bijvoorbeeld gebruikt om iets te berekenen dat pas veel later in het programma gebruikt wordt, terwijl in de tussentijd diezelfde variabele gebruikt wordt om "even" iets anders te te berekenen, waardoor de oorspronkelijke waarde dus veranderd wordt.

In C is het mogelijk om variabelen "globaal" en "lokaal" te gebruiken. Een variabele die buiten een functie (dus ook `main`) gedeclareerd is, is globaal bekend in het gehele programma. Een variabele die in een functie of in een compound statement (tussen accoladen) gedeclareerd is, is alleen in die functie of dat compound statement bekend en heeft dus een beperkt of lokaal bereik.

Voorbeeld om globaal en lokaal duidelijk te maken. De waarden van globale en lokale variabelen worden afgedrukt en daarna verhoogd:

```
#include <stdio.h>

int glob = 11,              /* initialiseer globale variabele */
    idem;

void allesGlobaal ()
{
  printf ("allesGlobaal      waarde glob: %d\n", glob++);
  printf ("allesGlobaal      waarde idem: %d\n\n", idem++);
```

```
  return;
}

void nietAllesGlobaal ()
{
  int idem = 0;              /* initialiseer lokale variabele */
  printf ("nietAllesGlobaal  waarde glob: %d\n", glob++);
  printf ("nietAllesGlobaal  waarde idem: %d\n\n", idem++);
  return;
}

int main ()
{
  idem = 11;                 /* initialiseer globale variabele */
  printf ("main              waarde glob: %d\n", glob++);
  printf ("main              waarde idem: %d\n\n", idem++);
  allesGlobaal ();
  nietAllesGlobaal ();
  printf ("main              waarde glob: %d\n", glob++);
  printf ("main              waarde idem: %d\n\n", idem++);

  {                          /* compound statement */
    int idem = 0;            /* initialiseer lokale variabele */
    printf ("main              waarde glob: %d\n", glob++);
    printf ("main              waarde idem: %d\n\n", idem++);
  }

  return (0);
}
```

Uitvoer van dit programma:

```
main              waarde glob: 11
main              waarde idem: 11

allesGlobaal      waarde glob: 12
allesGlobaal      waarde idem: 12

nietAllesGlobaal  waarde glob: 13
nietAllesGlobaal  waarde idem: 0

main              waarde glob: 14
main              waarde idem: 13

main              waarde glob: 15
main              waarde idem: 0
```

3.8 I/O van/naar bestanden op schijf

Als er veel externe gegevens door een programma gebruikt of gegenereerd worden, kunnen deze gegevens in files opgeslagen worden. Vanuit het programma kunnen gegevens dan ingelezen of weggeschreven worden.

Het gebruik van een file voor I/O vanuit een C programma gaat als volgt:

- Er moet een variabele van het type `FILE` gedeclareerd worden. Deze variabele, de **file identifier**, dient als "tussenpersoon" tussen programmacode en filesysteem.
- Een file wordt "geopend" door aanroep van functie `fopen`. Doel hiervan is de verbinding tussen file identifier en file te maken. Verder wordt aangegeven hoe de file gebruikt gaat worden, namelijk voor lezen, schrijven

of toevoegen.
- Geformatteerd kan er gelezen en geschreven worden door aanroep van: `fscanf` en `fprintf`. Deze functies hebben als extra eerste argument de file identifier.
- Teksten (strings, dus) kunnen met speciale functies gelezen en geschreven worden: `fgets` en `fputs`.
- Files worden gesloten door aanroep van functie `fclose`.

Zie appendix B van deze handleiding.

3.9 Structures

Met arrays worden elementen van hetzelfde type aan elkaar gekoppeld. Maar het is ook wel eens handig om datastructuren te gebruiken die per element een aantal variabelen van hetzelfde of van verschillend type bevatten.

Dit is mogelijk door de declaratie van nieuwe typen, de zogenaamde **structures**. Een structure heeft één of meer **members** of velden, die een eerder gedefinieerd type hebben. Zie hoofdstuk 6 van [1].

Voorbeeld van declaratie en gebruik van een structure (in dit geval met twee velden van hetzelfde type):

```
#include <stdio.h>

struct point   /* globale declaratie van structure van type 'struct point' */
{ int x;       /* eerste veld of 'member' van structure */
  int y;       /* tweede veld of 'member' van structure */
};

int main ()
{
  /* declaratie van variabelen van type 'struct point' */
  struct point hoekpuntenDriehoek[3];
  struct point hoekpuntenVierhoek[4] = {{-1, 0,}, {0, 2,}, {1, 0,}, {-1, 0}};

  int teller;

  /* adresseer afzonderlijke velden van de array-elementen */
  hoekpuntenDriehoek[0].x = -1;
  hoekpuntenDriehoek[0].y =  0;
  hoekpuntenDriehoek[1].x =  1;
  hoekpuntenDriehoek[1].y =  0;
  hoekpuntenDriehoek[2].x =  0;
  hoekpuntenDriehoek[2].y =  1;

  printf ("Hoekpunten van driehoek:\n");
  /* print array-elementen stuk voor stuk */
  for (teller=0; teller<3; teller++)
    printf ("  %4d %4d\n", hoekpuntenDriehoek[teller]);

  printf ("Hoekpunten van vierhoek:\n");
  for (teller=0; teller<4; teller++)
    printf ("  %4d %4d\n", hoekpuntenVierhoek[teller]);

  return (0);
}
```

Hoofdstuk 4

Sorteren en zoeken

Sorteren en zoeken vormen één van de meest fundamentele en één van de meest gebruikte technieken in computerprogramma's. Daarom is er vrij veel onderzoek naar gedaan, met als gevolg dat er een groot aantal algoritmen op dit gebied bestaat en dat er veel bekend is over het "gedrag" van de verschillende algoritmen. In dit hoofdstuk wordt alleen op sorteren in het werkgeheugen ingegaan en niet op sorteren van gegevens in bestanden.
Voor een uitgebreide behandeling van sorteren en zoeken wordt verwezen naar [5].

Bij het sorteren van gegevens in arrays in het werkgeheugen kunnen drie verschillende wijzen onderscheiden worden:

- Sorteren door **uitwisseling** (of **by exchange**).
  Hierbij worden elementen die niet in de goede volgorde staan, paarsgewijs verwisseld, totdat alle elementen geordend zijn.
- Sorteren door **keuze** (of **by selection**).
  Hierbij wordt eerst het kleinste element gezocht, dit wordt vooraan gezet (verwisseld met het eerste element), vervolgens wordt het één na kleinste element gezocht, totdat alle elementen geordend zijn.
- Sorteren door **tussenvoegen** (of **by insertion**).
  Hierbij wordt het eerste element als het geordende deel van de elementen beschouwd. Het volgende element wordt op de juiste plaats toegevoegd aan het geordende deel, enzovoorts, totdat alle elementen aan het geordende deel toegevoegd zijn.

De implementatie van een sorteeralgoritme kan, afhankelijk van het soort gegevens, op twee verschillende manieren uitgevoerd worden:

- De te sorteren elementen worden zélf **verplaatst**.
  Zijn de te sorteren elementen van een basistype, dan is dit de meest efficiënte manier.
- De te sorteren elementen worden zélf niet verplaatst, maar hun positie wordt in een speciaal array, het zogenaamde **pointerarray**, bijgehouden. Hierin wordt de plaats van het n-de element bijgehouden in het n-de array-element. Verwisselen twee elementen van plaats, dan hoeven in dit geval alleen maar de twee bijbehorende pointerarray-elementen omgewisseld te worden.
  Bestaan de te sorteren elementen bijvoorbeeld uit records (zie volgende hoofdstuk van deze handleiding), arrays of strings, dan hoeft alleen maar de plaats van twee elementen in het pointerarray veranderd te worden om de twee elementen van plaats te laten verwisselen. Hierbij is minder tijd nodig dan het verwisselen van de twee elementen in hun geheel.

Wat efficiëntie van de implementatie betreft kan er ook nog een onderscheid gemaakt worden tussen sorteren waarbij elementen onderling **verwisseld** worden ("in situ") en tussen sorteren waarbij elementen naar een ander gebied **verplaatst** worden. Het verwisselen kost minder ruimte (er is bijvoorbeeld maar één array voor de elementen nodig), maar meer tijd (drie maal kopiëren per verwisseling), terwijl het verplaatsen meer ruimte kost (er is een extra array nodig waarnaar verplaatst wordt), maar minder tijd (één maal kopiëren per verplaatsing).

Bij het zoeken naar een sorteermethode — en zeker als er zeer veel elementen gesorteerd worden — moet er naar een aantal kriteria gekeken worden, omdat bij gebruik van een verkeerde methode de benodigde tijd enorm op kan lopen:

- Hoe snel zal er in een gemiddeld geval gesorteerd worden?
- Hoe snel zal er in het beste geval (alle elementen zijn al gesorteerd) en in het slechtste geval (alle elementen

staan in verkeerde volgorde) gesorteerd worden?
- Zal het algoritme sneller zijn als de elementen beter gesorteerd zijn en langzamer als de elementen minder gesorteerd zijn?
- Worden gelijke elementen ook verwisseld?

Globaal kan wat uitvoering betreft een verdeling gemaakt worden in **niet-recursieve** en in **recursieve** uitvoering. Merk hierbij echter op dat een recursieve uitvoering ook niet-recursief uitgevoerd kan worden.

4.1 Niet-recursief sorteren

Een aantal veelgebruikte methoden zijn:

- **Selection sort**.
  Hierbij wordt het kleinste element gezocht en dit wordt meteen op de goede plaats gezet (door middel van verwisselen of verplaatsen).
  De benodigde tijd is ongeveer $\frac{1}{2} \times n^2$, dus evenredig met n^2. De sorteertijd is onafhankelijk van beste, gemiddelde of slechtste geval.

- **Insertion sort**.
  Hierbij worden de elementen na elkaar in een gesorteerd gedeelte gezet.
  Gemiddeld evenredig met n^2, maar er zijn extra bewerkingen nodig omdat de elementen in het gesorteerde gedeelte opgeschoven moeten worden, als er een element tussengevoegd wordt. De sorteertijd is in dit geval afhankelijk van beste, gemiddelde of slechtste geval, omdat de oorspronkelijke mate van gesorteerdheid meetelt bij het tussenvoegen.

- **Bubble sort**.
  Hierbij worden naastliggende elementen langs de hele rij paarsgewijs met elkaar vergeleken en in de goede volgorde gezet. Als de rij op deze manier één maal langsgelopen is, staat het grootste element achteraan. Hierna wordt dezelfde procedure herhaald voor de ongesorteerde rij, die één element korter is, totdat alle elementen gesorteerd zijn.
  Evenredig met n^2. De sorteertijd is onafhankelijk van beste, gemiddelde of slechtste geval.

- **Shaker sort**.
  Hetzelfde als bubble sort, maar dan alternerend. De rij wordt nu ook van achter naar voor doorlopen, waarbij het kleinste element op de eerste plaats terecht komt. Deze methode is iets efficiënter dan de bubble sort.
  Evenredig met n^2. De sorteertijd is onafhankelijk van beste, gemiddelde of slechtste geval.

- **Shell sort**.
  Dit is een soort veredelde bubble sort, die veel efficiënter werkt. In plaats van steeds twee naastliggende elementen te vergelijken, wordt er een stapgrootte groter dan één gebruikt. Deze stapgrootte S wordt gebruikt om de elementen op posities 1 en 1+S met elkaar te vergelijken, vervolgens 2 en 2+S, enzovoorts. Indien nodig worden elementen verwisseld. Nadat de elementen doorlopen zijn, wordt de stapgrootte verkleind en wordt de bewerking opnieuw uitgevoerd. Dit gaat zo door tot en met stapgrootte gelijk aan één (à la bubble sort), waarna de elementen in de goede volgorde staan.
  De winst ten opzichte van de bubble sort wordt behaald doordat er met grote stapgrootte minder bewerkingen uitgevoerd worden, terwijl er over het algemeen efficiënter verplaatst wordt. Gemiddeld evenredig met $n^{1.2}$. De sorteertijd is onafhankelijk van beste, gemiddelde of slechtste geval.

4.2 Recursief sorteren

De volgende sorteermethode wordt tot de in het algemeen snelste en meest efficiënte gerekend:

- **Quicksort**.
  Hierbij wordt ernaar gestreefd om een element (de mediaan) zodanig te plaatsen, dat alle elementen die

er voor staan ook er voor horen en dat alle elementen die er na staan, ook er na horen. Als de elementen zo staan, wordt links en rechts van de mediaan dezelfde bewerking herhaald (hier kan van recursie gebruik gemaakt worden!). Dit gaat door tot er geen herhalingen meer mogelijk zijn: alle elementen staan in de goede volgorde.
Gemiddeld evenredig met $n * log(n)$.

Hoofdstuk 5

Random numbers

Voor een groot aantal toepassingen is het nodig om over een serie willekeurige getallen (**random numbers**) te beschikken. In vrijwel alle programmeertalen is er een functie beschikbaar, die als een soort superdobbelsteen (min of meer) willekeurige getallen produceert. Het is erg moeilijk om met eenvoudige middelen op een computer een betrouwbare, écht willekeurige serie getallen te generen, maar voor niet al te grote toepassingen voldoen de meeste "ingebouwde" **random number generatoren**.

Een eenvoudige random number generator kan met de volgende formule gemaakt worden:

```
nieuw := ( (oud * factor) + toevoeging ) MOD modulus
```

Hierbij is `nieuw` de geleverde waarde en `oud` de vorige waarde. Een dergelijke generator is een functie met één argument (`oud`), die de functiewaarde `nieuw` opleverd. De eerste keer moet `oud` zelf verzonnen worden, daarna wordt de laatste waarde van `nieuw` steeds als nieuwe waarde van `oud` genomen. De waarden van `factor`, `toevoeging` en modulus zijn interne waarden van de functie. In het algemeen geldt: hoe groter de waarde van `modulus`, hoe beter de serie geleverde waarden op een serie echte random getallen lijkt.

Hoofdstuk 6

Recursief programmeren

In een aantal gevallen kan een probleem beschreven worden door een algoritme dat een aanroep naar zichzelf bevat. Bekijk bijvoorbeeld het berekenen van N! voor waarden van N groter dan of gelijk aan nul:

$$N! = 1 \times 2 \times 3 \times ... \times (N-1) \times (N)$$

Hiervoor kan het volgende algoritme opgesteld worden:

Berekening waarde N! :
    ALS N groter is dan 0 DAN
        waarde N! = N × waarde (N − 1)!
    ANDERS
        waarde N! = 1

Bij de eerste uitvoering van het algoritme heeft N een bepaalde waarde, stel 10. Tijdens de uitvoering wordt het algoritme zelf opnieuw aangeroepen, maar nu met waarde N=9. Bij de tweede aanroep wordt het algoritme nogmaals aangeroepen, maar nu met waarde N=8, enzovoorts, totdat het algoritme aangeroepen wordt met waarde N=1. Tot nu toe is er eigenlijk nog niets gerekend aan de waarde van N!, er is alleen maar in een diep gat gedoken door vanuit de "DAN" clausule steeds weer een nieuwe versie van hetzelfde algoritme aan te roepen.

Maar voor N=1 wordt de "ANDERS" clausule uitgevoerd, waardoor het algoritme de waarde van N berekent en de uitvoering beëindigt. Vervolgens wordt de "DAN" clausule in versie van N=2 verder uitgevoerd: de waarde van N wordt berekend en deze versie van het algoritme beëindigt de uitvoering. Hierna komt de versie van N=3 aan de beurt, enzovoorts, totdat uiteindelijk de versie van N=10 uitgevoerd is. En deze levert het resultaat van de gehele berekening — in dit geval 10! — op. We zijn uit het gat tevoorschijn gekomen met het resultaat.

Belangrijk bij het opstellen van een recursief algoritme is het hebben van een goed **stopcriterium**. Zonder de "ANDERS" clausule in het aangehaalde voorbeeld is het niet mogelijk om uit het gat tevoorschijn te komen: het is als het ware een zwart gat geworden.

Voorbeeld van recursief programmeren:

```
/*****************************************************************************/
/* Voorbeeld van conventioneel en recursief programmeren.
/*****************************************************************************/

#include <stdio.h>

/* Bereken de Nde macht van een getal X op conventionele manier. */

double X_tot_de_Nde_macht_conv (double x, int n)
{
  double product = 1.0;
  int    macht;

  if (n >= 0)               /* macht is positief: bereken antwoord in loop */
  { for (macht=1; macht<=n; macht++) product *= x;
  }
  else     /* macht is negatief: bereken voor positieve macht en deel op 1
                  is nul:      geen loop, deel wel op 1                  */
  { for (macht=1; macht<=-n; macht++) product *= x;
    product = 1.0 / product;
  }

  return product;
}


/* Bereken de Nde macht van een getal X op recursieve manier. */

double X_tot_de_Nde_macht_rec (double x, int n)
{
  double product = 1.0;
  int    macht;

  if (n == 0)          /* macht is nul: klaar! */
  { return 1.0;
  }
  else
  { if (n > 0)       /* macht is positief: bereken voor macht-1 */
    { return X_tot_de_Nde_macht_rec(x,n-1) * x;
    }
    else               /* macht is negatief: bereken voor positieve waarde */
    { return 1.0 / X_tot_de_Nde_macht_rec(x,-n);
    }
  }
}


main ()
{
  double X = 1.000001;
  int    N = 20000;

  printf ("Conventioneel:            X, N, X_tot_de_Nde_macht_conv(X,N));

  printf ("Recursief:                X, N, X_tot_de_Nde_macht_rec(X,N));
}
```

Appendix A

Literatuurlijst

1. Brian W. Kernigan, Dennis M. Ritchie,
   The C Programming Language, 2nd ed.,
   Prentice Hall, 1988, ISBN 0-13-110362-8.

2. P.F. Klok en J.M. Thijssen,
   Handleiding Programmeren.

3. O.-J. Dahl, E.W. Dijkstra en C.A.R. Hoare,
   Structured Programming,
   Academic Press, 1972.

4. D.E. Knuth,
   The Art of Computer Programming, Vol. 1: Fundamental Algorithms,
   Addison-Wesley, 1973.

5. D.E. Knuth,
   The Art of Computer Programming, Vol. 3: Sorting and Searching,
   Addison-Wesley, 1973.

6. Link naar website "Inleiding Computergebruik",
   `http://www.hef.kun.nl/~pfk/teaching/ic/`

Appendix B

Korte beschrijving van C statements

Dit appendix bevat een zeer korte samenvatting van verschillende statements en constructies van de ANSI-versie van de taal C (zie [1]). Voorafgaande hieraan een paar opmerkingen:

- Een C-programma heeft de volgende structuur:

  ```
  /* Kop met beschrijving (niet verplicht, maar wel van harte aanbevolen!) */
  #include ...
  #define ...
  declaratie van globale variabelen
  declaratie van functies
  int main ()
  {
    declaratie van lokale variabelen
    statements
  }
  ```

- De **index van een array** in C begint altijd bij 0. In een array van tien elementen loopt de index dus van 0 tot en met 9 en niet van 1 tot en met 10 zoals bij de meeste andere talen.
- In een **printf**-statement wordt een nieuwe regel aangegeven met `\n` (een backslash gevolgd door een kleine letter n).
- Het compileren en linken van een C programma wordt gedaan met het volgende commando:
  `make` *programmanaam*
  De bijbehorende file Makefile is bij de assistent te verkrijgen.

Hierna volgt een korte beschrijving van de verschillende constructies van de taal C.
Syntax is cursief, **voorbeelden zijn vet**.

Identifiers	Sequence of letters, digits and underscores starting with a letter or underscore; case sensitive.
Comments	*/* ... */* **/* Dit is commentaar! */**
Data types	**int** **long int** **short int** **int** **char** **float** **double**
Declaration of variables	*type variable_list;* **int i;** **char c, d;** **float x, y;**
Assignment	*value = expression;* **i = 2;** **c = 'a';** **x = x + 2 * x;**
Main program	*int main ()* { *statement*$_1$*;* ... *statement*$_n$*;* *return (0);* }
Compound statement	{ *statement*$_1$*;* ... *statement*$_n$*;* }

Arithmetic operators		
	Addition	`+`
	Subtraction	`-`
	Multiplication	`*`
	Division	`/`
	Modulo	`%`
	Unary +	`+`
	Unary -	`-`

Logical operators		
	Less than	$<$
	Less than or equal to	`<=`
	Greater than	$>$
	Greater than or equal to	`>=`
	Equality	`==`
	Inequality	`!=`
	Logical and	`&&`
	Logical or	`||`
	Negation	`!`

Writing formatted output	*printf (formats_and_text_string, list_of_vars);* **printf ("Waarde integer is: %d\n", intvar);** **printf ("Waarde float is: %f\", floatvar);**

Printf conversion codes

`d`	`int,` signed decimal notation
`c`	`int,` single character
`s`	`char *,` string
`f`	`double,` floating-point number

Reading formatted input

scanf (formats_string, list_of_vars);

scanf ("%d", &intvar);

scanf ("%4d%4.1f", &intvar, &floatvar);

Scanf conversion codes

`d`	decimal integer, `int *`
`c`	characters, `char *`
`s`	string of non-white space characters, `char *`
`f`	floating-point number, `float *`

If statement

if (*expression*)
    statement_if_true;

```
if (x < 1)
    y = 2;

if ( (x == 1) && (y != 0) )
{
    y = 2;
    x++;
}
```

if (*expression*)
    statement_if_true;
else
    statement_if_false;

```
if (x < 1)
    y = 2;
else
    y = 3;
```

While statement

while (expression) statement;

```
while (x < 1) x++;

while (x < 1)
{
    x++;
}
```

Repeat statement

do
    statements;
while (condition);

```
do
    x++;
    y += x;
while (x ¡= 0);
```

For statement

for (expression1; expression2; expression3)
    statement;

```
for (i=1; 1<=10; i++)
    printf ("%d", i);
```

Switch statement

```
switch (expression)
{
    case l1:
    ...
    case lk: statementsl;
             break;
    ...
    case m1:
    ...
    case mn: statementsm;
             break;
    default: statements;
             break;
}
```

```
switch (c = getchar())
{
    case 'a':
    case 'b':
    case 'c': putchar ('1');
              break;
    case 'd': putchar ('2');
              break;
    default : putchar ('3');
}
```

Read from file

```
FILE *fin; /* file identifier /*
int m1k, m1r;
/* open input file */
if ((fin=fopen("nameOfFile", "r")) == NULL)
{
    printf ("File bestaat niet!\n");
    return (1);
}
/* read from input file */
fscanf (fin, "%d%d", &m1k, &m1r);
/* close input file */
fclose (fin);
```

Write into file

```
FILE *fout; /* file identifier /*
nt m1k, m1r;
float v1k, v1r;
/* open output file */
if ((fout=fopen("nameOfFile", "w")) == NULL)
{
    printf ("File kan niet geopend worden!\n");
    return (1);
}
/* write into output file */
fprintf (fout, "%8d%8d", m1k, m1r);
fprintf (fout, "%8.2f%8.2f', v1k, v1r);
/* close output file */
fclose (fout);
```

Testing for end-of-line

```
while ((c = getchar()) != '\n') statement;
```

Testing for end-of-file

```
while ((c = getchar()) != EOF) statement;
```

Function definition

```
int max (int a, int b)
{
    return (a>b?a:b);
}

void test ()
{
    int i;
    i = 2;
    ...
}
```

Pointer variable definition

```
type *identifier;

int *pointerToInt;
int normalInt;
    normalInt = 33;
    *pointerToInt = normalInt;
    printf ("Int value = %d\n", *pointerToInt);
```

Dereferencing

```
*identifier
```

Pointer type

```
typedef int * pint;
pint p;
```

Nil pointer

```
NULL
```

Memory allocation

```
type *identifier;
identifier = (type*) malloc (sizeof(type));
```

Memory deallocation

```
free ((char*)identifier);
```

Array

```
type array-identifier[size];
```

Array type

```
typedef knownID newID[size];

typedef int VEC5[5];
typedef float FVEC[10];
VEC5 v5;
FVEC f;
```

Two-dimensional array

```
type array-identifier[index1][index2];

int x[2][3];

/* dynamic allocation and use of matrix */
int **ptrToMat;
int k, r; /* loop variables */
int K = 5; /* number of columns */
int R = 6; /* number of rows */
/* allocate rows */
ptrToMat = (int **) malloc (R*sizeof(int));
/* allocate columns */
for (r=0; r<R; r++);
    ptrToMat[r] = (int *) malloc (K*sizeof(int));
```

```
/* fill matrix row by row */
for (r=0; r<R; r++);
    for (k=0; k<K; k++);
        ptrToMat[r][k] = r*k;
```

Two-dimensional array type

```
typedef int TWO[3][4];

typedef int SINGLE[5];
typedef SINGLE TWO[4];
```

Structure variable

struct
{ *components;*
} *identifier;*

```
struct
{ int i;
  float f;
} a;
```

Structure type

typedef struct type
{ *components;*
} *type;*

```
typedef struct student
{ char name[30];
  int StudNum;
} STUDENT;
typedef STUDENT CLASS[100];
CLASS Stats;
```

Field access using pointers

pointer -> field

Appendix C

Register