KULeuven
Technologiecampus De Nayer
Industrieel ingenieur
Opleiding Industriële Ingenieurswetenschappen
afstudeerrichting elektronica-ICT
Bachelor 2e fase en schakelprogramma

P R O G R A M M E E R - T E C H N I E K E N

Academiejaar 2015-16 H. Crauwels

Inhoudsopgave

Permanente evaluatie

Bij permanente evaluatie wordt rekening gehouden met volgende elementen om tot een score te komen:

1. de bijhorende theorie op voorhand bestudeerd hebben;
2. een aantal opgaven voorbereid hebben;
3. op tijd komen;
4. tijdens de praktijkzitting: niet-storend aanwezig zijn;
5. tijdens de praktijkzitting: actief werken aan de opgaves;
6. bij het stellen van vragen blijk geven van toch al kennis te hebben van de behandelde materie; dus, voordat een vraag gesteld wordt, toch al zelf eens nagedacht hebben;
7. aandachtig luisteren wanneer er klassikaal een bijkomende verduidelijkende uitleg gegeven wordt;
8. valabele elementen aanbrengen tijdens een eventuele klassikale discussie;
9. nadat de basisopgave afgewerkt is, met evenveel enthousiasme aan de eventuele uitbreidingen werken;
10. geen plagiaat plegen;
11. na de praktijkzitting: tijdig het verslag met de eventuele oplossingen afgeven aan de begeleidende docent;
12. indien nodig, komen bijwerken aan bepaalde opgaves indien deze niet voldoende afgewerkt zijn tijdens de praktijkzittingen zelf;
13. het ingeleverde werk wordt door de begeleidende docent beoordeeld en eventueel van commentaar voorzien;
14. bij volgende opgaves rekening houden met eventueel ontvangen commentaar op vorige verslagen;
15. het praktijklokaal reglement respecteren (zie agenda, *algemeen reglement praktijklokalen*).

Bijkomende richtlijnen voor de praktijklokalen A202, A203, A213, A214, A217:

- absoluut rook-, eet-, drink- en snoepverbod (herhaling van punt 6 van het algemeen reglement praktijklokalen);
- toestellen en schermen zoveel mogelijk op hun plaats laten staan, dus niet verschuiven of verdraaien;
- netwerk-, muis-, toetsenbord- en voedingskabel laten zitten;
- geen eigen toestellen aansluiten op het netwerk;
- geen *De Nayer*-toestellen meenemen.

Referenties

[1] H.M. Deitel, P.J. Deitel. C: how to program. Pearson Education International/Prentice Hall, 2004.

[2] E.W. Dijkstra. A short introduction to the art of programming. EWD-316, 1971.

[3] B.W. Kernighan, D.M. Ritchie. The C programming language. Prentice Hall, 1988.

[4] R.S. Pressman. Software engineering, a practitioner's approach. McGraw-Hill, 1997.

[5] R.W. Sebesta. Concepts of programming languages. Benjamin Cummings, 1992.

[6] R. Sedgewick. Algorithms in C. Addison Wesley, 0-201-51425-7

[7] M.A. Weiss. Data structures and algorithm analysis. Benjamin Cummings, 1992.

[8] N. Wirth. Algorithms + Data Structures = Programs. Prentice-Hall, 1976.

Denken als discipline

PROF EDSGER W. DIJKSTRA, EMERITUS HOOGLERAAR COMPUTER SCIENCE UNIVERSITY OF TEXAS, AUSTIN:

- Het enige dat van belang is dat je snel iets in elkaar kunt flansen dat je op de markt brengt, wat je verkopen kunt. Hoeft helemaal niet goed te zijn. Als je maar de illusie kunt wekken dat dit een prettig product is, zodat het gekocht wordt, nou ja dan kun je daarna later kijken of je wat betere versies kan maken. Dan krijg je het verschijnsel met de versienummers met zelfs cijfers achter de komma. Versie 2.6 en versie 2.7 en die poespas. Ja. Terwijl als het gewoon goed geweest was versie 1 gewoon het product geweest was.

- Informatica gaat net zo min over computers als astronomie over telescopen.

- Dat was een tijd waarin voor allerlei afdelingen de grootste zorg was: is mijn curriculum wel waterig genoeg? Op hetzelfde ogenblik was de universiteit van Texas in Austin bezig te proberen om de inschrijvingen te doen verminderen en de kwaliteit op te schroeven. Dat was een tegengestelde ontwikkeling die aanmerkelijk aantrekkelijker was dan wat er in het Nederlandse hoger onderwijs gebeurde.

  Universiteiten zal het de moed blijven ontbreken om harde wetenschap te onderwijzen; men zal erin volharden de studenten te misleiden, en elke volgende fase in de infantilisatie van het curriculum zal toegejuicht worden als een onderwijskundige stap voorwaarts.

- Kwaliteit, correctheid en elegantie. Dat zijn de eisen waaraan een computerprogramma volgens Dijkstra hoort te voldoen. In 1954 nam hij zich voor om programmeren tot een wetenschap te maken. Maar sindsdien heeft hij tegen de stroom in moeten roeien.

  Ik lig er niet wakker van dat het bedrijfsleven het gevoel heeft dat ze zich niet kunnen permitteren een eersteklas product af te leveren. Verhindert mij niet om met mijn onderzoek door te gaan.

  Je moet de wereld niet geven waar ze om vraagt, maar wat ze nodig heeft.

- Er zijn heel verschillende ontwerpstijlen. Ik karakteriseer ze altijd als Mozart versus Beethoven. Als Mozart begon te schrijven dan had hij de compositie klaar in zijn hoofd. Hij schreef het manuscript en het was 'aus einem Guss'. En het was bovendien heel mooi geschreven.

  Beethoven was een aarzelaar en een worstelaar en die schreef voordat hij de compositie klaar had en overplakte dan iets om het te veranderen. En er is een bepaalde plaats geweest waar hij negen keer heeft overplakt en men heeft het voorzichtig losgehaald om te zien wat er gebeurd was en toen bleek de laatste versie gelijk te zijn aan de eerste.

- Ik herinner me de eerste keer, dat was in 1970, de eerste keer dat ik echt de markt opging om te laten zien hoe je programma's kon ontwikkelen zodat je ze stevig in je intellectuele greep kon houden. Ik ging eerst naar Parijs en daarna naar Brussel. In Parijs gaf ik een voordracht voor de Sorbonne en het publiek was razend enthousiast. En toen op weg naar huis hield ik hetzelfde verhaal bij een groot softwarehuis in Brussel. Het verhaal viel volkomen plat op zijn gezicht. Ik heb nog nooit zo'n slechte voordracht gegeven in zekere zin. En later ontdekte ik waarom: het managament was niet geinteresseerd in feilloze programma's want het waren de maintenance contracts waaraan het bedrijf z'n stabiliteit aan ontleende. En de programmeurs waren er ook niet in geïnteresseerd omdat bleek dat die een groot stuk van hun intellectuele opwinding ontleenden aan het feit dat ze NIET precies begrepen wat ze deden. Zij hadden het gevoel dat als je allemaal precies wist wat je deed en je geen riscico's liep, dat het dan een saai vak was.

- Als je in de fysica iets niet begrijpt, kun je je altijd verschuilen achter de ongepeilde diepte van de natuur. Je kunt altijd God de schuld geven. Je hebt het zelf niet zo ingewikkkeld gemaakt. Maar als je programma niet werkt, heb je niemand achter wie je je kunt verschuilen.

Je kunt je niet verschuilen achter onwillige natuur, neen, een nul is een nul en een één is een één en als het niet werkt, heb je het gewoon fout gedaan.

- In de zestiger jaren zag Dijkstra hoe de complexiteit van de programma's de programmeurs boven het hoofd groeide. En dat ook de meest prestigieuze projecten erdoor bedreigd werden.

  Dat was een ervaring in 1969, dat was vlak nadat de eerste geslaagde maanlanding achter de rug was. Het was een conferentie in eh, een NATO-conference on software engineering in Rome en daar kwam ik Joel Aron tegen die het hoofd was van IBM's federal systems division en dat was de afdeling die verantwoordelijk was geweest voor de software van het maanschot. En ik wist dat elke Apollovlucht iets van 40.000 new lines of code nodig had, nou doet het er niet precies toe wat voor eenheid een line of code is, 40.000 is veel en ik was diep onder de indruk dat ze zoveel programmatuur goed en in orde hadden gekregen, dus toen ik Joel Aron tegenkwam zei ik: "how do you do it?" "Do what?" vroeg hij. Nou zei ik, "getting that software right." "Right ?!?" he said. En toen vertelde hij dat in een van de berekeningen van de baan van de Lunar Module de maan in plaats van aantrekkend afstotend was en die tekenfout hadden ze bij toeval, moet je nagaan: bij toeval, vijf dagen van te voren ontdekt. Ik verschoot van kleur en zei: "Those guys have been lucky." "Yes!" was het antwoord van Joel Aron.

- Het testen van een programma is een effectieve manier om de aanwezigheid van fouten in een programma aan te tonen, maar het is volkomen inadequaat om hun afwezigheid te bewijzen. - EWD 340

- Elegantie is geen overbodige luxe, maar vormt het onderscheid tussen succes en falen. - EWD 1284

- Eén van de dingen die ik in de zestiger jaren al ontdekt heb, is dat wiskundige elegance, mathematical elegance, dat dat niet een esthetische kwestie is, een kwestie van smaak of mode of wat dan ook, maar dat je het vertalen kunt in een technisch begrip. Want in bijvoorbeeld de Concise Oxford Dictionary daar vind je als een van de betekenissen van "elegant": ingeniously simple and effective.

  In de programmeerpraktijk is het te hanteren doordat als je een echt elegant programma maakt dat het, nou, ten eerste korter is dan z'n meeste alternatieven, ten tweede uit duidelijk gescheiden onderdelen bestaat waarvan je het ene onderdeel door een alternatieve implementatie kunt vervangen zonder dat dat de rest van het programma beïnvloedt, en verder merkwaardigerwijze zijn de elegante programma's ook heel vaak de efficientste.

- Zolang er geen computers waren, was programmeren helemaal geen probleem. Toen we een paar kleine computers hadden, werd programmeren een klein probleem. Nu we gigantische computers hebben, is het programmeren een gigantisch probleem geworden.

- Ik denk dat ik professioneel ernstig beïnvloed ben door mijn moeder. Zij was een briljant wiskundige en ik herinner me toen ik in de zomervakantie de boeken voor het volgende jaar gekocht had, ik het boek goniometrie zag en ik vond het er heel griezelig uitzien allemaal met griekse letters en ik vroeg mijn moeder of goniometrie moeilijk was. Zei ze: nee, helemaal niet. Je moet zorgen dat je al je formules goed kent en als je meer dan vijf regels nodig hebt, dan ben je op de verkeerde weg.

- De vraag is waarom elegance in den brede zo weinig is aangeslagen. Het is inderdaad weinig aangeslagen. Het nadeel van elegance is, als je het een nadeel wilt noemen trouwens, het vergt hard werk en toewijding om het te bereiken en een goede opvoeding om het op prijs te stellen.

Professor Edsger W. Dijkstra is Nederlands eerste programmeur. Met zijn systematische programmeermethode won hij in 1972 de Turing Award, de Nobelprijs van de Informatica. Momenteel woont hij samen met zijn vrouw in Austin, Texas waar hij in 1984 naartoe verhuisde.

1 C: variabelen en controlestructuren

1.1 Voorbeeld: berekenen van grootst gemene deler

Python programma:

```
# ggd.py: bereken grootste gemene deler van twee getallen
def main():
    invoer = raw_input('Geef eerste getal: ')
    a = int(invoer)
    invoer = raw_input('Geef tweede getal: ')
    b = int(invoer)
    if a > b :
        x = b
        y = a
    else :
        x = a
        y = b
    while x != 0 :
        r = y % x
        y = x
        x = r
    print 'ggd(', a, ',', b , ') is ', y
#startoproep
main()
```

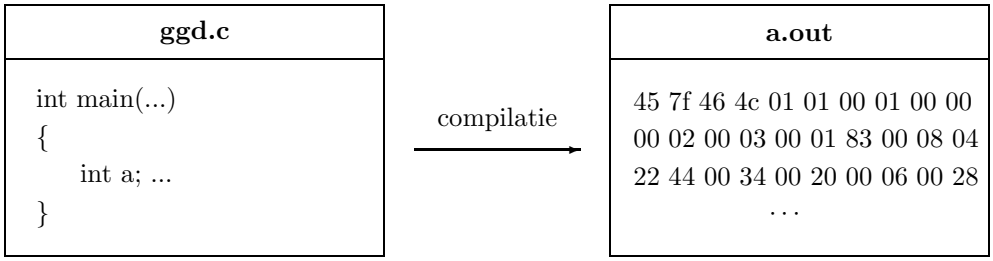
C programma:

```
/*
 *    ggd.c : berekening grootste gemene deler
 */
#include <stdio.h>
int main(int argc, char *argv[])
{
    int a;        /* eerste getal */
    int b;        /* tweede getal */
    int x, y, r;

    scanf("%d%d%*c", &a, &b);
    /*   bepaal x en y zodat x < y */
    if ( a > b )
    {
        x = b;
        y = a;
    }
    else
    {
        x = a;
        y = b;
    }
    while ( x != 0 )            /* herhaal tot rest gelijk aan nul */
    {
        r = y % x;              /* bereken rest */
        y = x;                  /* y krijgt de waarde van x */
        x = r;                  /* x krijgt de waarde van r */
```

```
        }
29      printf("ggd(%d, %d) = %d\n", a, b, y);
}
```

1. Een C programma heeft een `main` functie met parameters. Deze `main` functie moet niet expliciet opgeroepen worden; dit gebeurt in een startup functie uit de standaard C bibliotheek.
2. Een functie bestaat uit een opeenvolging van statements (een actie gevolgd door een puntkomma `;`). Er bestaan toekenningsstatements, controle statements en functieoproepen.
3. C is een imperatieve taal. Identifiers duiden geen objecten (met waarden) aan maar wel variabelen, d.i. geheugenplaatsen in het centraal geheugen. Zo'n variabele heeft een adres en kan een waarde hebben. Het uitvoeren van berekeningen heeft als effect dat variabelen nieuwe waarden krijgen.
4. Variabelen en parameters moeten voor gebruik gedeclareerd worden. Hiermee wordt een type vastgelegd. Dit is gerelateerd aan het feit dat een C programma eerst gecompileerd wordt (omgezet naar machine-code). Deze resulterende code kan dan uitgevoerd worden.
5. Wat tussen `/*` en `*/` staat is commentaar en wordt door de compiler genegeerd.
6. Constanten zijn ofwel getallen, ofwel tekens (tussen enkel quotes) ofwel strings (tussen dubbel quotes).
7. Er kan gebruik gemaakt worden van preprocessor directieven: met `include` kunnen header bestanden met definities en declaraties opgenomen worden.
8. Een C programma heeft een blok structuur: deze wordt syntactisch vastgelegd met behulp van een begin- en een eind symbool: de akkolades. Indentatie heeft geen syntactische betekenis, maar is sterk aanbevolen (*verplicht*) om de leesbaarheid te bevorderen.
9. In- en uitvoer gebeurt met functies uit de standaard C bibliotheek: `scanf` en `printf`. In deze functies gebeurt een omvorming tussen interne (binaire) en externe (tekstuele) voorstelling. Hoe deze conversie gebeurt, wordt aangegeven met de formaatspecificaties in het eerste argument.

Nadat het geheel in een bestand ingetikt is, bijvoorbeeld het bestand met naam `ggd.c`. kan het programma *gecompileerd* worden. De *compiler* zet het *bronbestand* (C-programma) om in een *doelbestand* (machine-code) voor een specifieke processor. Het doelbestand kan dan uitgevoerd worden op de computer. Het doelbestand wordt daarom ook dikwijls *executable* genoemd. De standaard naam in een unix/linux omgeving is **a.out**.



bron-bestand doel-bestand

1.2 Variabelen.

De eerste statements in het programma zijn *declaraties* van *variabelen*. Een *variabele* is een symbolische naam van een geheugenplaats in het werkgeheugen. Deze naam wordt gebruikt om de geheugenplaats aan te duiden (*lvalue*). Men had evengoed een nummer kunnen gebruiken: geheugenplaats nummer 1, geheugenplaats nummer 2, ... Maar in een programma met veel variabelen,

zou dit leiden tot een vrij moeilijk leesbare code. Men geeft dus een symbolische naam (bijv. `a`, `b`, `r`) aan zo'n geheugenplaats. De waarde van de variabele (*rvalue*) is de inhoud van zo'n geheugenplaats, dus een getalwaarde waarmee in het programma gerekend wordt.
Een declaratie bevat naast de naam van de variabele ook een *type*. Het type geeft aan de compiler aan hoe groot de geheugenplaats moet zijn, of hoeveel bytes voor de desbetreffende variabele moet voorzien worden. In dit voorbeeld wordt het type `int` gebruikt. Dit type geeft aan dat de variabele waarden kan hebben die behoren tot de verzameling van de gehele getallen (integers). Afhankelijk van het type van de processor wordt door de compiler twee of vier bytes voorzien voor een variabele van type `int`.
Het type van een variabele bepaalt ook welke operaties (bewerkingen) toegelaten zijn. Op variabelen van het type `int` kunnen de verschillende rekenkundige bewerkingen uitgevoerd worden: `+`, `-`, `*`, `/` en `%`.
Een *statement* waarin een berekening gebeurt (de restberekening bij deling van twee getallen) werkt nu met variabelen:

- neem de inhoud van variabele `y` (rvalue);
- bereken de rest bij deling door de inhoud van variabele `x` (rvalue);
- plaats het resultaat in de variabele `r` (lvalue).

Het `=` teken is het symbool voor de operatie *toekenning*. Langs de rechterkant van dit symbool worden de rvalues van variabelen (de numerieke waarden) gebruikt; langs de linkerkant de lvalue van een variabele (aanduiding van een geheugenplaats).
Een variabele wordt gekenmerkt door zes attributen: naam, adres, type, waarde, bereik en levensduur. Deze attributen worden aan een variabele *gebonden*.
Het **binden** is eigenlijk een belangrijk concept in een taal. Het is een associatie, bijv. tussen een attribuut en een entiteit of tussen een operatie en een symbool. De binding kan plaats hebben tijdens de ontwerpfaze van de taal, de implementatiefaze van de taal, de compilatie, de linking, het laden of tijdens de uitvoering zelf. Een binding is **statisch** wanneer deze gebeurt voor runtime en onveranderd blijft gedurende de uitvoering van het programma. Anders heeft men een **dynamische** binding.

Naam. Een naam is een string van tekens om een identiteit in een programma aan te duiden. De ideale vorm is een string met een (eventueel) redelijk grote lengtelimiet en een verbindingsteken ('_') om namen bestaande uit verschillende woorden te creëren. In C wordt een onderscheid gemaakt tussen hoofd- en kleine letters.
Om syntactische eenheden van elkaar te onderscheiden worden speciale woorden (*keyword*) gebruikt. In C kunnen deze niet als naam van een variabele of functie gebruikt worden.

Adres. Dit is het adres van het geheugen waarmee de variabele geassocieerd is. Het is mogelijk dat dezelfde naam met verschillende adressen geassocieerd is op verschillende plaatsen in het programma. Bijv. twee functies waarin in elk een variabele `i` gedefinieerd is. Analoog kan dezelfde naam met verschillende adressen geassocieerd zijn op verschillende momenten tijdens de uitvoering van het programma. Bijv. de lokale variabelen van een recursieve procedure.
Het is ook mogelijk dat verschillende namen naar hetzelfde adres, en dus dezelfde geheugenplaats, refereren. Deze namen zijn dan **aliases**. Deze vormen een belemmering voor de leesbaarheid en voor de verificatie van de juistheid van een programma.
De waarde van een variabele kan veranderen door een toekenning vanuit een andere variabele.

Type. Dit bepaalt het domein van waarden die een variabele kan hebben en ook de operaties die op de variabele mogelijk zijn. De primitieve types zijn: `int` (unsigned, short, long), `float` (double), en `char`. Tekens (char) worden gestockeerd d.m.v. een numerieke code. De meest gebruikte codering is ASCII. In C echter kan dit type (`char`) ook gebruikt worden voor kleine integer data.

In C bestaat het `boolean` type niet. Alle numerieke types kunnen gebruikt worden als logisch type: een niet-nul waarde komt overeen met **true**, terwijl nul de betekenis van **false** heeft.
De manier waarop een type aan een variabele gebonden wordt, is met behulp van **deklaraties**. In ANSI C zijn deze expliciet. Bij impliciete deklaraties (bijv. FORTRAN) zijn er regels in de taal voorzien om aan te geven welke type een bepaalde variabele heeft. De binding gebeurt bij het eerste voorkomen van de naam in het programma. Zowel expliciete als impliciete deklaratie zijn statische bindingen. Bij dynamische type-binding is er geen deklaratie. De binding met een type gebeurt wanneer een waarde toegekend wordt aan de variabele (of het object, bijv. Python). De variabele langs de linkerkant krijgt het type van de waarde, variabele of expressie van de rechterkant.

Waarde. Normaal is de waarde van een variabele de inhoud van de geheugencel (of -cellen) die met de variabele geassocieerd zijn. Eigenlijk zijn er twee waarden: de **l-value** is de aanduiding van de plaats van een variabele; de **r-value** is de waarde van een variabele. Deze namen zijn afkomstig van het feit dat langs de linkerkant van een toekenning een $l-$value van een variabele nodig is, terwijl een $r-$value nodig is wanneer een variabele gebruikt wordt langs de rechterkant. Om toegang te krijgen tot de $r-$value, moet eerst de $l-$value bepaald worden. Dit is niet altijd even eenvoudig omwille van de *bereik* regels.

Bereik en levensduur: zie verder.

De communicatie met de gebruiker is een vrij moeilijke taak. Maar hiervoor zijn standaard functies voorzien, `scanf` voor invoer van het toetsenbord en `printf` voor uitvoer naar het beeldscherm. De declaratie van deze functies kan teruggevonden worden in het *header*bestand `<stdio.h>`. De definitie van de functies wordt bewaard in de *standaard C-bibliotheek*. De compiler (linker) zal deze bibliotheek raadplegen en de definities van de gebruikte (opgeroepen) functies opnemen in de *executable*.
Tussen de haakjes na de functienaam staan een aantal *argumenten*. Het eerste argument (de formaatstring) geeft aan welke conversies moeten gebeuren. De volgende argumenten geven de variabelen aan die bij de in/uitvoer betrokken zijn.
Het formaat `ggd(%d, %d) = %d\n`
geeft aan dat `printf` drie getalwaarden moet weergeven op het beeldscherm. Deze weergave moet de decimale voorstelling zijn. Tussen de drie getallen worden een aantal tekens weergegeven en na de drie getallen wordt naar een nieuwe lijn (`\n`) gegaan. Het tweede argument geeft de naam van de variabele die als eerste waarde zal weergegeven worden. Door de naam van de variabele als argument te gebruiken, wordt de inhoud (rvalue) van de geheugenplaats doorgegeven aan de functie. In de functie `printf` wordt de waarde (interne voorstelling) geconverteerd naar een decimale vorm (externe voorstelling) en dan weergegeven op het scherm. Analoog voor het derde en het vierde argument.
Het formaat `%d%d%*c`
geeft aan dat `scanf` twee decimale getallen van het toetsenbord moet lezen, deze getallen moet omvormen naar een interne voorstelling en een extra teken.
Het tweede argument is niet de naam van een variabele, maar een expressie `&a`. Wanneer gewoon de naam zou gebruikt worden, zou aan `scanf` de inhoud (rvalue) van de variabele doorgegeven worden. Deze inhoud is echter nog niet gekend. Het is juist de bedoeling dat `scanf` voor een inhoud (een waarde) gaat zorgen door een decimaal getal van het toetsenbord in te lezen en dat te converteren naar een inwendige voorstelling. De functie `scanf` moet dus een aanduiding krijgen waar de waarde in het werkgeheugen moet gestockeerd worden. De expressie `&a` berekent deze aanduiding (of het *adres* van de geheugenplaats) en dit adres wordt doorgegeven aan `scanf`.
Analoog geeft het derde argument `&b` het adres van de variabele `b` door aan de functie `scanf`.
Het laatste deel in de formaatstring geeft aan dat nog een extra teken moet ingelezen worden (`%c`) maar dat dit teken niet moet toegekend worden aan een variabele (`*`).

1.3 Een overzicht van de operatoren

De operatoren gerangschikt van hoge naar lage prioriteit; in de laatste kolom is de associativiteit aangegeven.

`()`	`[]`	`->`	`.`								links
`!`	`~`	`++`	`--`	`-`	(type)		`*`	`&`	sizeof		rechts
`*`	`/`	`%`									links
`+`	`-`										links
`<<`	`>>`										links
`<`	`<=`	`>`	`>=`								links
`==`	`!=`										links
`&`											links
`^`											links
`|`											links
`&&`											links
`||`											links
`?:`											rechts
`=`	`+=`	`-=`	`*=`	`/=`	`%=`	`<<=`	`>>=`	`&=`	`^=`	`|=`	rechts
`,`											links

Merk op dat elk van de tekens `-`, `*` en `&` tweemaal voorkomt. De context van het desbetreffende teken in het programma bepaalt om welke operator het gaat.
Voor de logische operatoren (en/of) worden symbolen gebruikt in plaats van namen: `&&` (and), `||` (of) en `!` (not).
Naast de toekenningsoperatoren (`+=`,...) zijn er *unaire* operatoren voorzien waarmee tellers compact met 1 verhoogd of met 1 verlaagd kunnen worden: de *increment* operator `++` en de *decrement* operator `--`. Deze operatoren bestaan in *pre* en *post* versie:

```
int h = 4;  int k = 4;  int m = 4;  int n = 4;
int i, j;

i = ++h;  /* verhoging h voordat waarde toegekend wordt aan i
             na uitvoering: i en h  dezelfde waarde, nl. 5    */
j = k++;  /* verhoging k  nadat  waarde toegekend is aan j
             na uitvoering: j waarde 4 en  k waarde 5         */
i = --m;  /* verlaging m voordat waarde toegekend wordt aan i */
j = n--;  /* verlaging n nadat de waarde toegekend is aan j   */
```

1.4 De keywords van de taal

Sommige woorden zijn gereserveerd als *keyword*; zij hebben een vaste betekenis voor de compiler en mogen daarom niet als naam voor iets anders (bijvoorbeeld variabele) gebruikt worden.

Variabele declaratie :	`auto`	`char`	`const`	`double`	`extern`
	`float`	`int`	`long`	`register`	`short`
	`signed`	`static`	`unsigned`	`void`	`volatile`

Data type declaratie :	`enum`	`struct`	`typedef`	`union`

Controle statement :	`break`	`case`	`continue`	`default`
	`do`	`else`	`for`	`goto`
	`if`	`return`	`switch`	`while`

Operator : `sizeof`

1.5 Controle statements

1.5.1 Het `if` statement

Om te kunnen laten beslissen of een statement al dan niet moet worden uitgevoerd, kan gebruik gemaakt worden van een keuze- of conditioneel statement.

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

Als de `expressie` een waarde verschillend van 0 (d.i. *true*) heeft, dan wordt `statement1` uitgevoerd. Wanneer de waarde van de `expressie` gelijk is aan 0 (d.i. *false*), dan wordt `statement2` uitgevoerd en `statement1` niet.
De expressie waarvan de waarde getest wordt, is meestal een *logische expressie*. Eventueel kan het `else` gedeelte weggelaten worden. Dikwijls moet de verwerking opgeslitst worden in *meerdere* onderling exclusieve acties. Dit kan gebeuren met de `else if` constructie.

```
/*
 * tijd.c :  twee tijden in seconden inlezen en de som afdrukken
 */
#include <stdio.h>

int main(int argc, char *argv[])
{
    int tijd1,tijd2,som, factor;

    printf("Geef tijd 1: ");
    scanf("%d%*c",&tijd1);
    printf("Geef tijd 2: ");
    scanf("%d%*c",&tijd2);
    som = tijd1+tijd2;
    if (som < 60)
    {
        printf("De som is %d seconden\n",som);
    }
    else
    {
        factor = som/60;
        if ( factor < 60)
            printf("De som is %d minuten en %d seconden\n",
                                    factor, som%60);
        else
            printf("De som is %d uren %d minuten en %d seconden\n",
                                    factor/60, factor%60, som%60);
    }
}
```

1.5.2 Het `switch` statement

Wanneer een keuze moet gemaakt worden tussen meer dan twee mogelijkheden, kan dit gebeuren met een aantal `if` statements. Soms is het mogelijk een meer overzichtelijke structuur te gebruiken.

```
switch ( integer expressie )
{
    case waarde_1:
        nul of meerdere statements;
    case waarde_2:
        nul of meerdere statements;
    ...
    default:
        nul of meerdere statements;
}
volgend_statement;
```

De `integer expressie` wordt geëvalueerd en de resulterende waarde wordt vergeleken met de mogelijke constante waarden bij elke `case`. Bij de eerste gelijke waarde die gevonden wordt, worden de bijhorende statements uitgevoerd. Wanneer het laatste statement in zo'n `case` het `break` statement is, wordt de `switch` verlaten en wordt de uitvoering verder gezet bij `volgend_statement`. Wanneer geen enkele gelijke waarde gevonden wordt, worden de statements behorend bij `default` uitgevoerd. Indien de `default` case weggelaten is, wordt helemaal niets gedaan.
Voorbeeld:

```
1  /*
    * cijfer.c : inlezen en afdrukken van een cijfer
3   */
   #include <stdio.h>
5
   int main(int argc, char *argv[])
7  {
       int symbool;
9
       printf("Geef een cijfer: ");
11     scanf("%d%*c", &symbool);
       switch( symbool )
13     {
           case 1 :
15             printf("een\n");
               break;
17         case 2 :
               printf("twee\n");
19             break;
           case 3 :
21             printf("drie\n");
               break;
23         case 0 :
           case 4 :
25         case 5 :
           case 6 :
27         case 7 :
           case 8 :
29         case 9 :
               printf("cijfer\n");
```

```
31              break;
            default   :
33              printf("Dit is geen cijfer %d\n", symbool);
                break;
35      }
    }
```

1.5.3 Het `while` statement

De meeste programmeertalen hebben een aantal controle structuren voorzien waarmee het mogelijk is een herhaling (een iteratie) compact neer te schrijven. In deze contekst spreekt men dikwijls over een *programma-lus*.

```
while ( expressie)
{
        statement1;
        statement2;
}
volgend_statement;
```

Zolang de `expressie` een waarde *true* oplevert, wordt `statement` uitgevoerd. In één van deze statements gebeuren normaal aanpassingen aan verschillende variabelen zodat na een tijd de `expressie` de waarde *false* oplevert. Op dat moment wordt de lus verlaten en gaat het programma verder met de uitvoering van `volgend_statement`.

In de `while` constructie ligt alleen de "test op einde" syntactisch vast. Omdat deze test gebeurt voordat enig statement in de lus uitgevoerd wordt, kan het zijn dat deze test reeds de eerste maal `false` oplevert, zodat de statements in de lus nooit zullen uitgevoerd worden.
Voorbeeld:

```
/*
2    *   wsom2.c : berekenen van de som van 10 getallen
     */
4   # include <stdio.h>
    # define AANTAL 10
6
    int main(int argc, char *argv[])
8   {
        int getal, teller, som;
10
        som = 0;
12      teller = 0;                     /* initialisatie */
        while ( teller < AANTAL )       /* test op einde */
14      {
            printf("Geef een geheel getal: ");
16          scanf("%d%*c",&getal);
            som += getal;
18          ++teller;                   /* de stap */
        }
20      printf("De som van deze getallen is %d\n", som);
        printf("en het gemiddelde is %d\n", som/AANTAL);
22  }
```

1.5.4 Het `for` statement

In een tweede iteratie-structuur is syntactisch vastgelegd hoe de drie basiselementen die in een herhalingsstructuur voorkomen (initialisatie, test op einde en de stap), moeten neergeschreven

worden.

```
for (init_expressie; test_expressie; stap_expressie)
{
     statement;
}
```

- de initialisatie: de `init_expressie`; beginwaarde;
- de test op het beëindigen van de lus: `test_expressie`;
- de stap: de `stap_expressie`.

Voorbeeld:

```
/*
 * fsom2.c : berekenen van de som van 10 getallen
 */
# include <stdio.h>
# define AANTAL 10

int main(int argc, char *argv[])
{
    int getal, teller, som;

    som = 0;
    for (teller=0; teller < AANTAL; teller++)
    {
        printf("Geef een geheel getal: ");
        scanf("%d%*c", &getal);
        som += getal;
    }
    printf("De som van deze getallen is %d\n",som);
}
```

1.5.5 Het `do while` statement

```
do
{
     statement1;
     statement2;
}
while(expressie);
```

De statements in het blok worden uitgevoerd. Zolang de `expressie` de waarde *true* oplevert, worden de statements in het blok herhaald. Omdat hier de test op het einde van de lus gebeurt, zullen de statements in het blok minstens eenmaal uitgevoerd worden.

Voorbeeld:

```
/*
 *   dsom2.c : berekenen van de som van 10 getallen
 */
# include <stdio.h>
# define AANTAL 10

int main(int argc, char *argv[])
{
    int getal, teller, som;

```

```
    som = 0;
12  teller =0;
    do
14  {
        printf("Geef een geheel getal: ");
16      scanf("%d%*c", &getal);
        som += getal;
18      ++teller;
    }
20  while(teller < AANTAL);
    printf("De som van deze getallen is %d\n",som);
22 }
```

1.5.6 Het `break` en `continue` statement

Bij het `switch` statement wordt `break` gebruikt om de statements bij een bepaalde `case` af te sluiten zodat verder gegaan wordt met het statement na het `switch` statement.
In de drie lus-constructies is ook een `break` statement mogelijk. Het effect is dat de lus (voortijdig) verlaten wordt en dat de uitvoering verder gezet wordt na het lus statement. Hiermee kan een lus beëindigd worden met behulp van een test die niet in het begin of het einde van de lus staat maar ergens tussenin.
Daarnaast is er ook het `continue` statement. Er wordt dan meteen de test op beëindiging die bij de `while`, of `do while` hoort, uitgevoerd. Bij `for` wordt eerst de `stap_expressie` uitgevoerd en dan de test op beëindiging.
Voorbeeld:

```
/*
2  * sompos.c : sommeren van positieve getallen
 */
4 #include <stdio.h>
#define AANTAL 100
6
int main(int argc, char *argv[])
8 {
      int i, getal, som;
10
      som = 0;
12    for ( i=1; i<=AANTAL; i++ )
      {
14        scanf("%d%*c", &getal);
          if ( getal < 0 )
16            continue;
          if ( getal == 0 )
18            break;
          som += getal;
20    }
      printf("het aantal ingelezen getallen is %d\n", i);
22    printf("de som van de positieve getallen is %d\n", som);
}
```

1.6 Voorbeeld: zeef van Erathosthenes

Python programma:

```
# zeef.py   zoeken van priemgetallen (Erathostenes)
from numpy import *

def schrap(zeef,p,n) :
    i = 2 * p
    while i <n :
        zeef[i] = 0
        i += p

def main():
    invoer = raw_input('Geef maximum voor priemgetallen: ')
    n = int(invoer)
    zeef = ones([n],int)
    zeef[0] = 0
    zeef[1] = 0
    for i in range(2,n) :
        if zeef[i] == 1 :
            print i
            schrap(zeef, i, n)

#startoproep
main()
```

C programma:

```
/*
 *  zeef.c : alle priemgetallen < MAXAANTAL
 */
#include <stdio.h>
#define MAXAANTAL 1000

void vulzeef(int zeef[]);
void schrap(int i, int zeef[]);

int main(int argc, char *argv[])
{
    int zeef[MAXAANTAL];
    int i;

    vulzeef(zeef);
    printf("Priemgetallen:\n");
    for (i=2; i<MAXAANTAL; i++)
    {
        if ( zeef[i] == 1 )
        {
            printf("%4d",i);
            schrap(i,zeef);
        }
    }
    printf("\n");
}
```

```
28  void vulzeef(int zeef[])
    {
30      int j;

32      zeef[0]=zeef[1]=0;
        for (j=2; j<MAXAANTAL; j++)
34      {
            zeef[j] = 1;
36      }
    }
38
    void schrap(int getal, int zeef[])
40  {
        int h;
42
        for (h=2*getal; h<MAXAANTAL; h+=getal)
44      {
            zeef[h] = 0;
46      }
    }
```

1. Naast enkelvoudige variabelen bestaat er ook samengestelde variabelen. Wanneer de samenstellende elementen van hetzelfde type zijn, spreekt men van *arrays*. Net zoals enkelvoudige variabelen moet een array voor gebruik gedefinieerd en gedeclareerd worden. Hiermee wordt de grootte van de array en het type van de elementen vastgelegd.
2. Om de grootte van een array vast te leggen wordt gebruikt gemaakt van een naamconstante. Door middel van het preprocessor directief `define` wordt aan een constante een naam gegeven.
3. Een naam van een array duidt het begin van een rij geheugenplaatsen aan. Zo'n naam kan niet langs rechts of langs links in een berekening voorkomen. Een arraynaam kan wel gebruikt worden als argument van een functie.
4. Een C programma bestaat uit functies. Elke functie wordt gedefineerd met een hoofding en een lichaam. De hoofding geeft de naam van een functie aan en bevat het type van het resultaat en de opsomming van de verschillende parameters, telkens met bijhorend type. Het lichaam is een blok met statements, die moeten uitgevoerd worden wanneer de functie opgeroepen wordt.
5. Een functie kan een resultaat hebben. Dit resultaat wordt teruggegeven aan de oproepende functie via het **return** statement. Het is ook mogelijk dat een functie geen resultaat heeft; men spreekt in dat geval nogal eens van een procedure. Het type van de functie is dan **void**.
6. Een oproep van een functie gebeurt door middel van de naam met opsomming van de actuele argumenten tussen haakjes. Omdat de compiler controleert of de types van de actuele argumenten wel overeenkomen met de types van de formele parameters, moet de functie eerst gedeclareerd worden. Dit kan gebeuren door middel van de volledige definitie of door middel van een protoype (hoofding gevolgd door een `;`).
7. De waarden van de actuele argumenten worden gebruikt om de formele parameters te initialiseren. Wanneer deze parmeters in de functie van waarde veranderen, worden deze nieuwe waarden niet terug gecopieerd naar de actuele parameters.

Wanneer een programma opgedeeld is in functies komen de twee laatste attributen van een variabele tot uiting:

Bereik. Het bereik van een programma variabele is het domein van statements waarin een variabele **zichtbaar** is. Een variabele is zichtbaar wanneer er naar gerefereerd kan worden.
Bij *static scoping* kan het bereik *statisch*, d.i. voor de uitvoering, bepaald worden. Een variabele is *lokaal* in een programmadeel wanneer ze daar gedeklareerd is. **Niet-lokale** variabelen zijn deze die zichtbaar zijn in een programmadeel, maar niet gedeklareerd zijn in deze eenheid. Deze *globale* variabelen worden gedeklareerd buiten elke functie. In C kan elk compound statement zijn eigen deklaraties hebben. Dus bepaalde secties code (een *block*) kunnen hun eigen lokale variabelen hebben, waarvan het bereik dus zeer beperkt is.

Levensduur. De levensduur van een programmavariabele is de tijd gedurende dewelke de variabele gebonden is aan een specifieke geheugenplaats. De levensduur start dus bij de **binding**, het moment waarop een cel uit het beschikbare geheugen geallokeerd wordt. De levensduur eindigt wanneer de cel terug gedeallokeerd wordt (de **ontbinding**). Afhankelijk van de levensduur kan men volgende kategoriën onderscheiden.
Bij *statische variabelen* gebeurt de binding met geheugen voor de uitvoering en blijft bestaan totdat de uitvoering beëindigd wordt. Voordelen zijn globaal toegankelijke variabelen en de mogelijkheid voor lokale variabelen om hun waarde te behouden tussen twee uitvoeringen van een subprogramma (*historische gevoeligheid*). Efficiëntie (directe adressering) is het grootste voordeel. Het nadeel is de beperking op de flexibiliteit, een taal met alleen statische variabelen laat geen recursie toe.
Bij *semidynamische variabelen* wordt de geheugenbinding gecreëerd op het moment dat de uitvoering het codedeel bereikt waaraan de deklaratie van de variabele gekoppeld is. Het type van deze variabelen is statisch gebonden. Voorbeeld: bij een oproep van een procedure, net voor de uitvoering ervan begint, wordt er geheugen geallokeerd voor de **lokale** variabelen van de procedure. Dit geheugen wordt weer vrijgegeven wanneer de procedure verlaten wordt en teruggegaan wordt naar de oproepende code. Voordelen: hetzelfde geheugen kan voor verschillende procedures gebruikt worden; recursieve procedures zijn mogelijk. Nadelen: run-time overhead en geen mogelijkheid tot historische gevoeligheid.
In een volgend hoofdstuk worden *expliciet dynamische variabelen* gedefineerd.
Er is een verschil tussen levensduur en bereik. Het **bereik** van de variabele `j` zit volledig vervat in de `vulzeef` procedure. De procedure `schrap` wordt uitgevoerd terwijl `main` aktief is. Dus de **levensduur** van `i` loopt verder tijdens de uitvoering van `schrap`. De geheugenplaats die aan `i` gebonden is bij de start van de functie `main`, blijft behouden tijdens de uitvoering van `schrap` (is op dat moment wel niet te *bereiken*) en geldt ook nog na de uitvoering tijdens de verdere uitvoering van `main`.

1.7 Elementaire types

De C-taal kent de volgende elementaire types:

		signed		unsigned	
char	1 byte	$-2^7 ... 2^7 - 1$	127	$0, 1, 2, ... 2^8 - 1$	255
short int	2 bytes	$-2^{15} ..., -1, 0, 1, ... 2^{15} - 1$	32767	$0, 1, 2, ... 2^{16} - 1$	65535
int					
long int	4 bytes	$-2^{31} ..., -1, 0, 1, ... 2^{31} - 1$	2147483647	$0, 1, 2, ... 2^{32} - 1$	4294967295
float	4 bytes				
double	8 bytes				

Het aantal bytes dat voor een `int` gebruikt wordt, is afhankelijk van de processor: 2 of 4 bytes. Om *overdraagbare* programma's te schrijven is het dus beter te werken met `short` en `long`.
Bij deze binaire voorstellingen wordt telkens één bit gebruikt als *tekenbit*. Bij sommige toepassingen wordt alleen met positieve getallen gewerkt. In die gevallen is geen tekenbit nodig en kunnen grotere getallen voorgesteld worden.
Merk op dat de deling van twee gehele getallen (variabele `j`) het geheeltallig quotiënt oplevert. Noteer ook het verschil tussen *signed* en *unsigned* getallen. In een declaratie wordt `unsigned`

expliciet vermeld. De default waarde is signed; dit kan eventueel vermeld worden maar hoeft niet en wordt dus niet veel gedaan.

```
unsigned int   aantal;
signed int     waarde;
```

Het aantal bytes dat een bepaald data type gebruikt in het werkgeheugen kan opgevraagd worden met de `sizeof` operator. De operand van deze operator is de naam van het type tussen haakjes. Daarnaast is het ook mogelijk deze operator toe te passen op een variabele, deze hoeft niet tussen haakjes geplaatst te worden.

```
unsigned    len;
double      ff;
len = sizeof(double);
len = sizeof ff;
len = sizeof(ff);
```

1.7.1 Type conversie

Als operanden in een expressie van verschillend type zijn, dan worden deze operands naar een gemeenschappelijk type geconverteerd.

operand 1	operand 2	gemeenschappelijk type
char	int	int
short	int	int
int	long	long
int	float	float
int	double	double
float	double	double

Er wordt dus telkens impliciet geconverteerd naar het "grotere" type. Daardoor zal zo weinig mogelijk informatie omwille van afkapping verloren gaan.
Bij een toekenningsexpressie kan echter een conversie van een "groter" type naar een "kleiner" type nodig zijn.

```
int     i, res;
float   f;
i = 31;         /*  geen conversie  */
f = 4;          /*  impliciete conversie van 4 naar 4.0  */
res = i/f;      /*  deling geeft 7.75; na de toekenning: 7  */
```

In de tweede toekenning gebeurt een impliciete conversie van de integer constante 4 naar een float (4.0). Bij de deling wordt de inhoud van de variabele `i` omgezet naar een float, dan wordt de deling uitgevoerd met als resultaat `7.75`. Dit float resultaat moet nu toegekend worden aan een integer variabele. Omdat het type van de variabele `res` niet kan aangepast worden (deze variabele is als `int` gedeclareerd), moet het resultaat omgevormd worden naar een integer. Dit gebeurt door afkappen, er is dus informatie verlies.

linkerkant	rechterkant	opmerking
char	int	
short	int	
int	long	
int	float	
float	double	afronding: alleen precisie verlies

1.7.2 Type casting

Soms is het nodig expliciet een conversie op te leggen. Dit kan door het vermelden van het gewenste type tussen haakjes voor de operand die moet geconverteerd worden. Een type tussen haakjes voor een operand kan gezien worden als een unaire operator: de *cast-operator*.

```
int i = 1, j = 2;
float x, y;
x = i/j ;          /* geheeltallige deling, dus 0.0  */
y = (float)i/j ;   /* reele deling, dus 0.5  */
```

De eerste deling is een geheeltallige deling met als resultaat 0. Maar deze waarde wordt toegewezen aan variabele `x` van type float. Dus het gehele getal wordt impliciet omgevormd naar een reëel getal en dan toegekend aan de variabele `x`.
Bij de tweede deling wordt eerst de gehele waarde van variabele `i` omgezet naar een waarde van type float. Bij de deling zelf wordt impliciet de waarde van variabele `j` ook omgezet naar een waarde van type float. Het resultaat is een reële waarde en deze kan zonder omvorming toegekend worden aan de variable `y` (is namelijk van type float).

1.8 Denktaak

```
#include <stdio.h>
float bereken(int, float);
int dubbel(int);

int main(int argc, char *argv[])
{
    float a = 4.0;
    int i = 1;

    a = bereken(i, a);
    printf("In main : i%d a%f\n", i, a);
}

float bereken(int h, float b)
{
    float r;
    r = dubbel( (int)b );
    return r + (float)h;
}

int dubbel(int c)
{
    int p = 2 * c;
    return  p;
}
```

Bespreek de attributen (naam, type, waarde, bereik en levensduur) van de verschillende variabelen (geen formele parameters van functies) in bovenstaand programma.
Merk op dat er *type-casting* gebruikt wordt om het type van een actueel argument om te vormen zodat het overeenkomt met het type van de formele parameter.

2 C: niet-numerieke data-types en datastructuren

Een computer kan naast numerieke gegevens ook niet-numerieke informatie verwerken. Tot nu toe is dat alleen gebleken bij de `printf`. Het eerste argument is een opeenvolging van letters tussen dubbel quotes (").

2.1 Het type char

Het elementaire type is een `char`, een *teken*, ook *symbool* of *karakter* genoemd. Declaratie, definitie en initialisatie van een variabele:

```
char c;

c = 'a';            /* c = a;  dubbelzinnig */
```

Door de toekenning heeft de variabele `c` de teken-waarde van de letter `a` gekregen. Door middel van de enkel quotes (') wordt aangegeven dat het over een teken-constante gaat. Zo'n constante of variabele neemt in het werkgeheugen één byte in beslag. In één byte (acht bits) kunnen $2^8 = 256$ mogelijke waarden gestopt worden. Dit is ruim voldoende voor alle letters van het alfabet en nog heel wat andere tekens.
Intern wordt dus een teken voorgesteld als een rij van acht bits, bijvoorbeeld de letter 'a':

```
01100001          0110 0001       0x61
```

Deze waarde kan ook als een getal geïnterpreteerd worden, namelijk (decimaal) de waarde 97. De programmeertaal C laat toe dat de inhoud van teken variabelen ook als gehele getallen beschouwd kunnen worden. Bij interpretatie van de inhoud van een byte als een geheel getal, geldt natuurlijk dat de hoogste bit het teken van het getal aangeeft. Het interval van mogelijke waarden is dus $[-128, 127]$.
Oorspronkelijk werd beslist dat 127 tekens voldoende waren. Er werd een tabel opgesteld die voor elk teken de bijhorende numerieke waarde weergeeft en dus de interne (binaire) voorstelling definieert. Zoals gewoonlijk zijn er verschillende tabellen opgesteld geweest. Het meest gebruikte schema is ASCII (American Standard Code for Information Interchange).
Omdat de waarde van een teken variabele dus ook als geheel getal beschouwd kan worden, zijn hierop rekenkundige operaties, zoals optelling en aftrekking, en vergelijkingsoperatoren, zoals `<` en `==`, mogelijk.
In `<ctype.h>` vindt men een aantal macro's om het soort van teken te testen. In sommige C-implementaties wordt gebruikt gemaakt van functies:

int isalnum(int ch)	alfanumeriek
int isalpha(int ch)	alfabetisch
int isdigit(int ch)	numeriek
int isupper(int ch)	hoofdletter
int islower(int ch)	kleine letter
int isprint(int ch)	afdrukbaar-teken
int isspace(int ch)	spatie-teken
int tolower(int ch)	omzetting naar kleine letter
int toupper(int ch)	omzetting naar hoofdletter

Voorbeeld. Schrijf een programma dat een tekst symbool per symbool leest (niet stockeert) en het aantal woorden telt. De tekst wordt afgesloten met het EOF teken. Bij klassieke unix-besturingssystemen is Ctrl-D het EOF (End Of File) symbool. Bij PC-besturingssystemen is Ctrl-Z het EOF (End Of File) symbool.

Hulp. Gebruik de functie `getchar()` die één symbool van het toetsenbord leest en de ASCII-waarde van dat symbool als functiewaarde heeft. Indien Ctrl-Z / Ctrl-D ingegeven wordt, dan geeft getchar() EOF als functiewaarde. EOF is in `<stdio.h>` gedefinieerd als -1.

```
/*
 *   woorden.c : tellen van het aantal woorden in een tekst
 */
#include <stdio.h>

int main(int argc, char *argv[])
{
    int c;
    int v = ' ';
    int woorden = 0;

    printf("Geef een tekst, eindig met <Ctrl-Z><Enter>\n\n");
    while ((c=getchar()) != EOF)
    {
        if ( !isalpha(c) && isalpha(v) )
            ++woorden;
        v = c;
    }
    if ( c == EOF && isalpha(v) )
        ++woorden;
    printf("\nHet aantal woorden in de tekst is %d\n", woorden);
}
```

Omdat de toekenningsoperator (`=`) een lagere prioriteit heeft dan de vergelijkingsoperator (`!=`), staan er haakjes rond `c=getchar()`.

2.2 Operaties met bits

Een byte is het kleinste adresseerbare element in het werkgeheugen. Zo'n element kan wel geinterpreteerd worden als een rij van 8 bits. In sommige toepassingen zijn manipulaties op deze kleinere elementen, de bits, nodig. In C zijn hiervoor een aantal operatoren voorzien. Deze operatoren kunnen toegepast worden op variabelen van het type `char`, `short`, `int` en `long`.

`~`	bitsgewijze inversie of *1-complement*
`&`	bitsgewijze *en*
`^`	bitsgewijze exclusieve *of*
`|`	bitsgewijze inclusieve *of*
`<<`	schuiven naar links
`>>`	schuiven naar rechts

Definitie van de verschillende operatoren:

`~`	
0	1
1	0

`&`	0	1
0	0	0
1	0	1

`|`	0	1
0	0	1
1	1	1

`^`	0	1
0	0	1
1	1	0

Ook de corresponderende *toekenningsoperatoren* (`&=`, `^=`, `|=`, `<<=` en `>>=`) bestaan.

```
/*
 * bitop.c : bitoperaties
 */
```

```
4   #include <stdio.h>
    int main(int argc, char *argv[])
6   {
        char a, b, c;
8
        a = 0x01;
10      b = 0x02;
        c = 0x03;
12      printf("      ~ %2x  -> %2x\n",     b,  ~b );
        printf("  %2x & %2x  -> %2x\n", a, b, a&b );
14      printf("  %2x & %2x  -> %2x\n", a, c, a&c );
        printf("  %2x ^ %2x  -> %2x\n", a, b, a^b );
16      printf("  %2x ^ %2x  -> %2x\n", a, c, a^c );
        printf("  %2x | %2x  -> %2x\n", a, b, a|b );
18      printf("  %2x | %2x  -> %2x\n", a, c, a|c );
        printf("  %2x << 1  -> %2x\n", b, b<<1  );
20      printf("  %2x >> 1  -> %2x\n", b, b>>1  );
        c = 0xf0;
22      printf(" %3d >> 1  -> %2d\n", c, c>>1  );
        a = 'G';
24      b = a | 0x20;
        printf("        %c  ->  %c\n", a, b  );
26  }
```

De resultaten:

```
    ~  2  -> fffffffd
  1 &  2  ->  0
  1 &  3  ->  1
  1 ^  2  ->  3
  1 ^  3  ->  2
  1 |  2  ->  3
  1 |  3  ->  3
  2 << 1  ->  4
  2 >> 1  ->  1
-16 >> 1  -> -8
      G  ->  g
```

Bij de `<<` worden de bits naar links geschoven; vooraan verdwijnen bits, achteraan worden nul-bits toegevoegd. Verschuiven naar links over n posities komt overeen met een vermenigvuldiging met 2^n.
Bij de `>>` gebeurt het omgekeerde: er worden bits naar rechts geschoven en achteraan verdwijnen bits. Wat er vooraan toegevoegd wordt, is afhankelijk van het type van de eerste operand. Wanneer dit type `unsigned` is, worden nul-bits toegevoegd. Verschuiven naar rechts over n posities komt overeen met een deling door 2^n.
Bij een `signed` type echter, wordt naar de hoogste bit (dit is de tekenbit) gekeken. Wanneer deze tekenbit gelijk is aan 0, worden nul-bits toegevoegd. Wanneer deze tekenbit gelijk is aan 1, worden één-bits toegevoegd. Dit noemt men *sign-extension*.
De shift operatoren kunnen gebruikt worden om bepaalde waarden in de hoogste of laatste bytes van een `int` te plaatsen:

```
/*
2  * bitsht.c : bit shift operaties
   */
```

```
4   #include <stdio.h>
    #define MASK8  0xff
6   #define MASK16 0xffff

8   int main( int argc, char *argv[])
    {
10      char     c;
        short    s;
12      int      i, j;
        long     l;
14      unsigned char    uc;
        unsigned short   us;
16      unsigned int     ui;
        unsigned long    ul;
18
        c = 24;          s = c<<8;          i = s<<8;          l = i<<8;
20      printf("%.8x %.8x %.8x %.8x\n", c, s, i, l);
        c = -1;          s = c>>8;          i = s>>8;          l = i>>8;
22      printf("%.8x %.8x %.8x %.8x\n", c&MASK8, s&MASK16, i, l);
        uc = 255;        us = uc>>8;        ui = us>>8;        ul = ui>>8;
24      printf("%.8x %.8x %.8x %.8x\n", uc&MASK8, us&MASK16, ui, ul);
        c = 24;
26      s = c<<8  | 0xff;
        i = s<<8  | 0xff;
28      j = i>>16 & MASK16;
        printf("%.8x %.8x %.8x %.8x\n", c, s, i, j);
30  }
```

De uitvoer van dit programma (bij een "32 bit" compilatie):

```
00000018 00001800 00180000 18000000
000000ff 0000ffff ffffffff ffffffff
000000ff 00000000 00000000 00000000
00000018 000018ff 0018ffff 00000018
```

2.3 Strings

2.3.1 Constanten

Enkelvoudige tekenconstanten worden aangeduid met behulp van enkel quotes, bijvoorbeeld `'a'`, `'5'`, `'\n'`.
Een rij van tekens tussen dubbel quotes is een string:

```
"hoeveel getallen"
"a5\n"
"a"
"naam: Jos\nwoonplaats: Duffel\n"
```

In een stringconstante zijn een aantal tekenconstanten *samengenomen*. Het type is dus een *array van tekens*. De interne voorstelling is een opeenvolging van een aantal ASCII codes. Om het einde van de rij codes aan te geven wordt het teken `'\0'` gebruikt. Dus `"a5"` wordt intern voorgesteld als | 0x61 | 0x35 | 0x00 |.

2.3.2 Variabelen

Om met strings bewerkingen uit te voeren, zijn variabelen nodig die geconstrueerd worden met behulp van het `array` type, d.i. een samengesteld type van tekenvariabelen.

```
char woord[20];
```

De gedeclareerde variabele `woord` is een array waarin maximaal 20 tekens kunnen gestopt worden. In het werkgeheugen worden dus 20 bytes voorzien voor deze variabele. Een array van tekens wordt dikwijls aangeduid met de naam *string*.
De initialisatie moet zoals bij elke variabele van het array-type gebeuren, namelijk element per element.

```
woord[0] = 's';
woord[1] = 'a';
woord[2] = 'p';
woord[3] = '\0';        /* of (char)0   */
```

Handiger zou zijn: `woord = "sap";` maar dit is niet mogelijk. De reden hiervoor is dat de naam `woord` geen atomaire variabele aanduidt maar iets van het type `array`, en dus geen lvalue heeft. Dit wil zeggen dat bij compilatie de variabele `woord` omgezet wordt naar een constant adres, wijzend naar het eerste element van de array. Door de toekenning zou dit constant adres moeten wijzigen en dit kan niet.
Nochtans is het mogelijk string-variabelen te initialiseren bij de definitie-deklaratie:

```
char woord[20] = "sap";
```

Tijdens compilatie zal een array van 20 bytes voorzien worden. In de eerste drie elementen worden letters 's', 'a' en 'p' ingevuld. Het vierde element krijgt de waarde `(char)0`. De rest wordt ook opgevuld met de waarden `(char)0`.
String variabelen op zich kunnen alleen gebruikt worden als argumenten bij functies. Net zoals bij arrays van ints of floats, krijgt de formele parameter in de opgeroepen routine de waarde van het adres dat wijst naar het eerste element van de string. Wanneer via deze formele parameter een element in de string gewijzigd wordt, zal deze wijziging na het verlaten van de functie behouden blijven.

Opgave. Lees een string van toetsenbord. Schrijf een functie die alle alfanumerieke tekens omzet naar hoofdletters. Schrijf de originele en de gewijzigde string uit.

```
1   /*
     * k2g.c : omzetting kleine naar grote letters
3    */
    #include <stdio.h>
5   #include <ctype.h>
    #define WLEN 32
7   void doen(char in[], char uit[]);

9   int main(int argc, char *argv[])
    {
11      char    in[WLEN];
        char    uit[WLEN];
13      int     ch;

15      scanf("%s", in);
        ch = getchar();
17      doen(in, uit);
```

```
    printf("ch = %d\n%s : %s\n", ch, in, uit);
}

void doen(char in[], char uit[])
{
    int i;

    i = 0;
    while ( (uit[i] = in[i]) != '\0' )
    {
        if ( islower(uit[i]) )
            uit[i] += 'A' - 'a';
        i++;
    }
}
```

Omdat `in` de naam van een array is, moet bij `scanf` geen `&` voor de variabele geschreven worden. De naam `in` is zelf al een verwijzing naar de geheugenplaatsen waar de string moet geplaatst worden.
Merk op dat nadat `scanf` de string ingelezen heeft, er nog één teken kan ingelezen worden, namelijk de ENTER toets.
In plaats van `scanf("%s", in); ch=getchar();` is ook `scanf("%s%c", in, &ch);` mogelijk.

2.3.3 Bewerkingen door middel van functies

Om toch op een efficiënte manier met strings te kunnen werken zijn een heleboel stringfuncties opgenomen in de standaard C-bibliotheek. Een aantal voorbeelden:

int strlen(char *s)	lengte van de string `s`
int strcmp(char *s, char *t)	vergelijking van string `s` met `t`
int strncmp(char *s, char *t, int n)	vergelijking (maximaal n tekens)
char *strcpy(char *s, char *t)	copiëren van string `t` in `s`
char *strncpy(char *s, char *t, int n)	copiëren (maximaal n tekens)
char *strcat(char *s, char *t)	toevoegen van string `t` aan `s`
char *strncat(char *s, char *t, int n)	toevoegen (maximaal n tekens)

Om deze te gebruiken, moet een include van `<string.h>` gedaan worden.

Opgave. Lees twee strings van toetsenbord. Schrijf deze twee strings in alfabetische volgorde uit.

```
/*
 * alfa2.c : alfabetisch afdrukken van twee strings
 */
#include <stdio.h>
#include <string.h>
#define WLEN 32

int main(int argc, char *argv[])
{
    char naam1[WLEN];
    char naam2[WLEN];
    int vgl;
```

```
14      printf("Tik een naam in: ");
        scanf("%s%*c", naam1);
16      printf("Tik een tweede naam in: ");
        scanf("%s%*c", naam2);
18      vgl = strcmp(naam1,naam2);
        if ( vgl == 0 )
20      {
            printf("%s is gelijk aan %s\n", naam1, naam2);
22      }
        else
24      {
            if ( vgl < 0 )
26              printf("%s %s\n", naam1, naam2);
            else
28              printf("%s %s\n", naam2, naam1);
        }
30  }
```

2.4 Arrays van strings

Opgave. Schrijf een programma dat een naam inleest en daarna een lijst van namen inleest en deze in een array stockeert. De lijst wordt afgesloten met het EOF teken. Daarna wordt nagegaan of de eerst ingelezen naam in de elementen van de lijst voorkomt.

```
    /*
2    * linzoek.c : lineair zoeken in een lijst van namen
     */
4   #include <stdio.h>
    #include <string.h>
6   #define MAX 50
    #define LEN 20
8
    int leeslijst(char a[][LEN]);
10  int zoek(char a[][LEN], int n, char b[]);

12  int main(int argv, char *argv[])
    {
14      char tabel[MAX][LEN];
        char naam[LEN];
16      int  i, n;

18      printf("Te zoeken naam : ");
        scanf("%s%*c", naam);
20      n = leeslijst(tabel);
        i = zoek(tabel, n, naam);
22      if ( i >= 0 )
            printf("De naam %s is gevonden op plaats %d\n", naam, i);
24      else
            printf("De naam %s komt niet voor in de lijst \n", naam);
26  }

28  int leeslijst(char a[][LEN])
    {
```

```
30      int i = 0;

32      while ( 1 )
        {
34          printf("Naam  %2d : ", i);
            if ( scanf("%s%*c", a[i]) == EOF )
36              break;
            i++;
38      }
        return i;
40  }

42  int zoek(char a[][LEN], int n, char b[])
    {
44      int i;

46      for (i=0; i<n; i++)
        {
48          if ( strncmp(a[i], b, LEN) == 0 )
                return i;
50      }
        return -1;
52  }
```

2.5 Structures

2.5.1 Voorbeeld

Python programma:

```
    #mens.py
2   import copy

4   class Mens:
        def __init__(zelf, naam, getal, zwaar):
6           zelf.naam = naam
            zelf.leeftijd = getal
8           zelf.gewicht = zwaar

10      def __str__(zelf):
            return ' %s %d %f   ' % (zelf.naam, zelf.leeftijd, zelf.gewicht)
12
        def gelijk(zelf, x):
14          if zelf.leeftijd != x.leeftijd : return False
            if zelf.gewicht != x.gewicht : return False
16          return zelf.naam == x.naam

18      def aanpass(zelf):
            zelf.leeftijd -= 1
20          zelf.gewicht += 0.4

22
    # hoofdprogramma
```

```
b = list()
a = Mens('marieke', 27, 58.9)
b.append( copy.deepcopy(a) )
print a, b[0], ' : ', a.gelijk(b[0])
b.append( copy.deepcopy(b[0]) )
b[1].aanpass()
print b[0], b[1], ' : ', b[0].gelijk(b[1])
```

C programma:

```
/*
 * mens.c : bewerkingen op een structure
 */
#include <stdio.h>
#include <string.h>
#define NMLEN  20
#define AANTAL 10
typedef struct mens
        {
            char  naam[NMLEN];
            int   leeftijd;
            float gewicht;
        } Mens;
Mens aanpass(Mens z);
int gelijk(Mens x, Mens y);
void druk( Mens z );

int main(int argc, char *argv[])
{
    Mens  a;
    Mens  b[AANTAL];
    int   len1, len2;

    strcpy(a.naam, "marieke");
    a.leeftijd = 27;
    a.gewicht = 58.9;
    b[0] = a;
    b[1] = aanpass(b[0]);
    druk( a );      druk( b[0] );
    printf(" : %d\n", gelijk(a, b[0]) );
    druk( b[0] );      druk( b[1] );
    printf(" : %d\n", gelijk(b[0], b[1]) );
    len1 = sizeof a;
    len2 = sizeof(Mens);
    printf("Grootte van structure: %d, %d\n", len1, len2);
}
int gelijk( Mens x,  Mens y)
{
    if ( x.leeftijd != y.leeftijd )
        return 0;
    if ( x.gewicht != y.gewicht )
        return 0;
    return !strcmp(x.naam, y.naam);
}
```

```
Mens aanpass( Mens z)
{
    z.leeftijd--;
    z.gewicht += 0.4;
    return z;
}
void druk( Mens z )
{
    printf("%-9.9s %2d %5.2f   ", z.naam, z.leeftijd, z.gewicht);
}
```

```
Resultaat:      marieke   27 58.90   marieke   27 58.90    : 1
                marieke   27 58.90   marieke   26 58.50    : 0
                Grootte van structure: 28, 28
```

1. C is een klassieke imperatieve taal en kent geen klassen. Om toch een aantal variabelen van verschillend type te groeperen kan de voorloper van een klasse, namelijk een **struct** gebruikt worden. Hierin wordt wel data beschreven maar niet de mogelijke operaties.
2. Met behulp van **typedef** kan een nieuw type gedefineerd worden. In dit geval wordt een struct gedefineerd met drie velden. Elk van deze velden heeft een naam en een type. De struct zelf krijgt ook een naam in de vorm van een *tag*. Eventueel kan dit tag veld weggelaten worden.
3. Variabelen van dit type worden gedefinieerd zoals variabelen van een klassiek type. Een specifiek veld in deze variabele aanspreken kan met de *punt* . operator.
4. Aangezien er geen methodes bestaan, worden operaties op deze variabelen gedefineerd door middel van klassieke functies.

Een variabele van het nieuwe type `Mens` definiëren gebeurt op de gewone manier. De variabele `a` bestaat uit 28 bytes, maar die zijn netjes opgedeeld in drie velden. De variabele `b` bevat een array van 100 elementen, elk element is een structure, bestaande uit drie velden.
Bij de oproep van `aanpassen` wordt de waarde van het actuele argument (alle velden van de structure `b[0]`) gecopieerd in de formele parameter `z`. Hetzelfde gebeurt bij de `return`: de velden van de structure `b[1]` krijgen een nieuwe waarde.
Het vergelijken van twee structures kan echter niet met de eenvoudige vergelijkingsoperatoren (bijv. `==`). Hiervoor kan bijvoorbeeld een functie geschreven worden. Merk op dat het negatieteken (`!`) voor `strcmp` nodig is om een `0` terug te geven als de namen verschillend zijn.
Om de grootte van een structure te berekenen, kan men de `sizeof` operator gebruiken: Het resultaat is het aantal bytes dat de structure in beslag neemt.

2.5.2 Alternatieve syntax

In plaats van een nieuw type te definiëren, kan ook een struct met een *tag*-veld gebruikt worden.

```
struct mens
{
    char  naam[20];
    int   leeftijd;
    float gewicht;
};
struct mens a;
```

Declaratie en definitie kunnen samen gebeuren:

```
struct mens
```

```
{
        char  naam[20];
        int   leeftijd;
        float gewicht;
} a;
struct mens b[100];
```

Omdat in de eerste declaratie de structure een naam gekregen heeft, moet in de tweede declaratie niet meer de volledige beschrijving gegeven worden, de naam volstaat.
Een structure hoeft geen naam te krijgen; in dit geval is het *tag*-veld leeg:

```
struct
{
        char  naam[20];
        int   leeftijd;
        float gewicht;
} a;
```

In dit geval moet natuurlijk de definitie van de variabele `a` bij de declaratie gebeuren. Het nadeel is dat bij verder gebruik van deze structure telkens de volledige omschrijving moet gegeven worden.
Naast elementaire data-types of arrays (bijv. strings) kunnen ook structures zelf gebruikt worden om een veld in een (overkoepelende) structure te beschrijven:

```
typedef struct
        {
             Mens  bio;
             float punten[10];
        } Student;
Student x;
```

De variabele `x` bestaat uit twee velden, het eerste veld is zelf een structure, het tweede veld is een array van 10 floats.
De initialisatie van een structure kan ook bij de definitie gebeuren:

```
Mens aa = { "flup", 27, 83.0 };
Student xx = { {"marie", 21, 62.1 }, { 74.2, 69.3. 77.7 } };
```

Structures kunnen als geheel gebruikt worden bij toekenningen en als actuele argumenten bij functie oproepen. Een specifiek veld van een structure gebruiken in een expressie. kan met de *punt* `.` operator:

```
strcpy(a.naam, "jos");
a.leeftijd = 17;
a.gewicht = 54.4;
b[3].gewicht = a.gewicht + 1.0;
x.bio.leeftijd = 22;
x.punten[3] = 64.3;
b[2] = a;
x.bio = a;
```

Plaats in het bronbestand. Net zoals andere variabelen kunnen structure variabelen lokaal of globaal gedefinieerd worden. Het is wel de gewoonte om de declaratie van een structure of de definitie van een nieuw type vooraan in een bestand te doen. Wanneer het programma uit meerdere bronbestanden bestaat, is het aangewezen deze type-definities te verzamelen in een apart *header bestand*. Met de `include` constructie kan dit header bestand opgenomen worden in de verschillende bronbestanden.

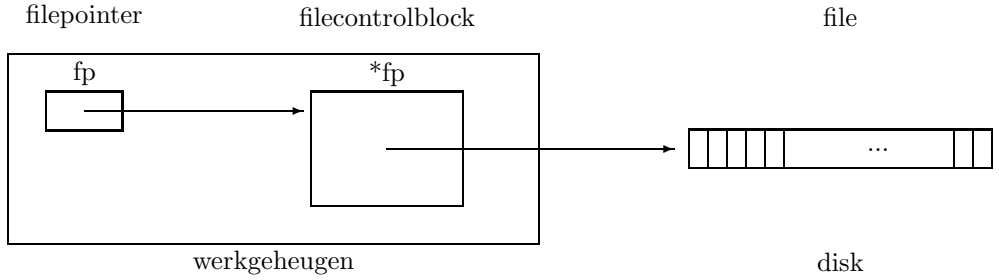
3 Operaties op bestanden

3.1 Inleiding

C bevat geen speciale taalconstructies voor in- en uitvoer. Men heeft hiervoor een verzameling van functies voorzien die samengebracht zijn in de standaard C-bibliotheek. Om de in- en uitvoer functies te kunnen gebruiken (d.w.z. oproepen) moet een include gedaan worden van de header file `stdio.h`.

3.2 Openen van een bestand

Een *filepointer* is een variabele die een pointer bevat naar een structure waarin allerlei administratieve gegevens zitten omtrent het bestand dat gebruikt wordt. De inhoud van het bestand zelf is ergens op een extern medium (bijv. disk) gelokaliseerd maar om in- of uitvoer naar het bestand te doen, moeten deze administratieve gegevens snel toegankelijk zijn en moeten deze dus in het werkgeheugen aanwezig zijn.
Deze administratieve gegevens zijn verzameld in het *file-control-block*. Hierin zit o.a. waar ergens op de disk de data van het bestand kan gevonden worden. De precieze inhoud is gedefinieerd door middel van het type `FILE`. De definitie hiervan kan teruggevonden worden in `stdio.h`. De filepointer zelf is dus van type `FILE *`.



Om in het file-control-block de verschillende velden te initialiseren met de administratieve gegevens van het bestand waarop in- of uitvoer gedaan gaat worden, moet het bestand *geopend* worden. Dit gebeurt met de functie `fopen`. Wanneer het lezen van of het schrijven naar het bestand afgelopen is, kan het file-control-block terug opgeruimd worden door het bestand te *sluiten* mbv de functie `fclose`.

```
FILE *fopen(char *naam, char *mode);
fclose(FILE *fp);
```

De functie `fopen` heeft twee argumenten. Het eerste argument, `naam`, specificeert welk bestand van disk moet geopend worden. Dit gebeurt met behulp van de naam van het bestand zoals die op de disk gekend is (*absoluut* tov. de rootdirectory of *relatief* tov. de working directory). Wanneer het om een nieuw te creëren bestand gaat, duidt dit argument de naam aan, die vanaf dan kan gebruikt worden om het bestand te benoemen. Het tweede argument, `mode`, geeft aan hoe het bestand moet geopend worden. Er zijn verschillende mogelijkheden:

`"r"`	lezen van bestaand bestand
`"w"`	schrijven op een nieuw bestand, eventueel reeds bestaande inhoud is verloren
`"a"`	achteraan toevoegen aan een bestaand bestand
`"r+"`	lezen van en schrijven op bestaand bestand
`"w+"`	schrijven op een nieuw bestand, ook lezen is mogelijk
`"a+"`	achteraan toevoegen aan een bestaand bestand, ook lezen is mogelijk

De return-waarde van `fopen` is een pointer naar een nieuw gecreëerd filecontrolblock. Indien er toch iets misloopt bij het openen van het bestand, wordt de waarde `NULL` teruggegeven.
Voor invoer van toetsenbord kan men de filepointer `stdin` gebruiken en voor uitvoer naar beeldscherm `stdout` en `stderr`. De bijhorende filecontrolblocks worden door het run-time systeem geïnitialiseerd wanneer het programma gestart wordt.

3.3 Geformatteerd lezen en schrijven

Eens een bestand geopend, kan ervan gelezen worden en kan erop geschreven worden met bijvoorbeeld de `fscanf` en `fprintf` functies, wanneer het over geformatteerde in- en uitvoer gaat:

```
fprintf(filepointer, formaat, arg1, arg2, ... );
fscanf(filepointer, formaat, arg1, arg2, ... );
```

Afhankelijk van de implementatie geeft de functie `fprintf` ook een waarde terug. Deze is dan gelijk aan het aantal overgedragen karakters of negatief in het geval van een fout.
De *formaat* string bevat twee soorten objecten: gewone af te drukken karakters en conversiespecificaties. Voor elk van de argumenten `arg1`, `arg2`, ... is er een conversiespecificatie in de string. Zo'n specificatie begint met een `%` en eindigt op een conversieteken.

type	conversieteken	het corresponderende argument wordt afgedrukt als
int	`%d`	een geheel getal in decimale notatie
int	`%x`	een geheel getal in hexadecimale notatie
float	`%f`	een reëel getal in vaste komma notatie
float	`%e`	een reëel getal in drijvende komma notatie
char	`%c`	een karakter
char[]	`%s`	een string

Tussen `%` en het conversieteken kan staan:

`-`	links in de beschikbare ruimte
getal	de veldbreedte
`.`	scheiding tussen veldbreedte en precisie
getal	de precisie - getal: het aantal cijfers rechts van het punt - string: het aantal afgedrukte karakters
`l`	argument is van type `long int` (`double`) in plaats van `int` (`float`)
`h`	argument is van type `short int` in plaats van `int`

De veldbreedte en precisie kan ook met een `*` aangegeven worden. In dat geval worden de waarden van de overeenkomstige argumenten genomen voor respektievelijk de veldbreedte en precisie.

```
printf("%*.*s : %*d\n", 8, 5, "appelsien", 4, 123 );
```

Wanneer na de `%` een karakter volgt dat niet in bovenstaande lijsten opgenomen is, wordt dat karakter gewoon afgedrukt. Dit is handig voor het procent-teken zelf af te drukken, door middel van `%%`. Voorbeeld:

3456		314.159265358979323846	
%10d	`3456`	%20.10f	`314.1592653590`
		%20.10e	`3.1415926536e+02`
		%-20.10f	`314.1592653590`
%-10d	`3456`	%-20.10e	`3.1415926536e+02`
	`1234567890`		`12345678901234567890`

Bij `fscanf` zijn de argumenten `arg1`, `arg2`, ... adressen (pointers), die aangeven waar in het centraal geheugen de ingelezen gegevens moeten opgeborgen worden.
De conversietekens die in *formaat* kunnen voorkomen, zijn `d`, `x`, `f`, `c`, `s`. De tekens `d` en `x` moeten voorafgegaan worden door een `l` wanneer het corresponderende argument een pointer is naar een `long` in plaats van een `int`; analoog wordt `h` gebruikt om aan te geven dat het over een `short` gaat. Wanneer het corresponderende argument een pointer naar een `double` is, wordt `f` voorafgegaan door een `l`.
Tussen `%` en het conversieteken kan een `*` staan. Hiermee wordt aangegeven dat wel iets moet gelezen worden, maar dat geen toekenning aan een variabele moet plaatsvinden. Daarnaast kan met behulp van een getal ook de maximale veldbreedte aangegeven worden.
Buiten de conversiespecificaties kunnen in het eerste argument nog voorkomen:

- spaties, tabs en newlines: deze worden genegeerd;
- gewone karakters (behalve `%`): deze moeten dan ook in de invoer op de corresponderende plaats voorkomen.

De functie `fscanf` geeft een waarde terug. Deze is dan gelijk aan het aantal gelezen en aan variabelen toegekende waarden. Als bij het lezen iets fout loopt, is de return waarde gelijk aan 0 of `EOF` (in `stdio.h` gedefinieerd als -1). Het resultaat 0 geeft aan dat geen enkele variabele een waarde gekregen heeft, omdat de gespecificeerde conversie in de formaatstring niet mogelijk is op basis van de ingelezen tekens.
Naast deze twee functies zijn er nog andere functies, bijvoorbeeld om karakter per karakter of lijn per lijn te werken:

```
int fgetc(FILE *fp);
int fputc(int ch, FILE *fp);
char *fgets(char *str, int maxlen, FILE *fp);
int fputs(char *str, FILE *fp);
```

Met `fgets` worden maximaal `maxlen-1` tekens ingelezen, zodat in de laatste positie plaats is voor een `'\0'`.
Het copiëren van een bestand in python:

```
# teken.py : copieert een bestand teken per teken
def main():
    fin = open('teken.py', 'r')
    fuit = open('weg.txt', 'w')
    while True :
        teken=fin.read(1)
        if not teken :
            break
        fuit.write(teken)
    fin.close()
    fuit.close()

#startoproep
main()
```

De volgende twee programma's kunnen gebruikt worden om een copie van een bestand te maken.

```
/*
 * teken.c : copieert een bestand teken per teken
 */
#include <stdio.h>
int main(int argc, char *argv[])
```

```
6   {
        FILE *fin;
8       FILE *fuit;
        int  teken;
10
        fin = fopen(argv[1], "r");
12      fuit = fopen(argv[2], "w");
        while ( (teken=fgetc(fin)) != EOF )
14          fputc(teken, fuit);
        fclose(fin);
16      fclose(fuit);
    }
```

Wanneer het einde van een bestand bereikt is, geeft de functie `fgetc` de waarde `EOF` terug. De naamconstante `EOF` is gedefinieerd in het headerbestand `stdio.h`.

```
1   /*
     * lijn.c : copieert een bestand lijn per lijn
3    */
    #include <stdio.h>
5   #define LLEN    80
    int main(int argc, char *argv[])
7   {
        FILE *fin,  *fuit;
9       char lijn[LLEN];

11      fin = fopen(argv[1], "r");
        fuit = fopen(argv[2], "w");
13      while ( fgets(lijn, LLEN, fin) != NULL )
            fputs(lijn, fuit);
15      fclose(fin);
        fclose(fuit);
17  }
```

Wanneer het einde van een bestand bereikt is, geeft de functie `fgets` de waarde `NULL` terug.
Soms kan het nuttig zijn om een rij tekens te lezen tot aan een bepaald teken, maar zonder dat teken zelf. Dit wordt opgelost door het teken eerst te lezen, maar dan wordt dit lezen ongedaan gemaakt. Het is alsof het teken weer in de invoer gezet wordt.

```
1   /*
     * woord.c : tel het aantal woorden in een bestand
3    */
    #include <stdio.h>
5   #include <ctype.h>
    #define WLEN 80
7   void leesspaties( FILE *fp, int *pi );

9   int main(int argc, char *argv[])
    {
11      FILE  *fin;
        int   spaties = 0;
13      int   tel = 0;
        char  woord[WLEN];
15
```

```
    fin = fopen(argv[1], "r");
17  leesspaties(fin, &spaties);
    while ( fscanf(fin, "%s", woord) != EOF )
19  {
        tel++;
21      printf("%4d : %s\n", tel, woord);
        leesspaties(fin, &spaties);
23  }
    fclose(fin);
25  printf("%4d woorden gelezen\n", tel);
    printf("%4d spaties weggelaten\n", spaties);
27 }

29 void leesspaties( FILE *fp, int *pi )
   {
31     int ch;

33     do
       {
35         ch = fgetc(fp);
           (*pi)++;
37     }
       while ( isspace(ch) );
39     (*pi)--;
       ungetc(ch,fp);
41 }
```

De functie `isspace` test of het argument een spatie, tab-teken, new-line, carriage return of form feed is.

3.4 Binaire informatie

Geformatteerde data is alleen nodig om de informatie toonbaar te maken voor de gebruiker zelf. Een voorbeeld hiervan is het opdelen van de informatie in lijnen, zodat er iets netjes op paper kan afgedrukt worden. Vele bestanden in een computersysteem bevatten informatie die alleen door het systeem zelf gelezen of aangepast wordt. Deze informatie hoeft dan niet geformatteerd te zijn. De externe representatie van de informatie (op een bestand) is dan dezelfde als de interne voorstelling (in de variabelen in het werkgeheugen). Er wordt geen conversie uitgevoerd.
Bestanden die zo'n informatie bevatten worden soms *binaire bestanden* genoemd in tegenstelling tot *tekst-bestanden*. In sommige C-omgevingen moet bij de functie `fopen` aangegeven worden dat het over een binair bestand gaat. Dit gebeurt door in het `mode` argument de letter **b** bij te voegen. Om bewerkingen op binaire bestanden uit te voeren, zijn volgende functies beschikbaar:

```
size_t fread(void *ptr, size_t len, size_t nobj, FILE *fp);
size_t fwrite(void *ptr, size_t len, size_t nobj, FILE *fp);
int fseek(FILE *fp, long offset, int code);
long ftell(FILE *fp);
```

De functies `fread` en `fwrite` geven als resultaat het werkelijk aantal getransporteerde elementen. Wanneer een fout opgetreden is of het einde van het bestand bereikt is, wordt de waarde 0 teruggegeven. De eerste parameter is van type `void *`: geeft aan dat het een adres in het werkgeheugen is naar een plaats waar zowel iets van type `int`, `float`, ... of een zelf gedefinieerde structure zit. Het type `size_t` is in het headerbestand `stdlib.h` met behulp van `typedef` gedefinieerd als een

`unsigned int`.
Naast lezen en schrijven is ook de functie `fseek` voorzien om de positie in het bestand waar de operatie zal uitgevoerd worden, te wijzigen. Deze functie heeft normaal 0 als resultaat, tenzij er iets misgegaan is. In dat geval is de terugkeerwaarde gelijk aan -1. De functie `ftell` geeft de actuele offset in een bestand, relatief ten opzichte van het begin en uitgedrukt in aantal bytes.

ptr	het adres van een gebied in het werkgeheugen	fread, fwrite
len	de lengte van een element in dit gebied	fread, fwrite
nobj	het aantal elementen in de operatie betrokken	fread, fwrite
fp	de filepointer	fread, fwrite, fseek, ftell
offset	relatieve positie (in bytes) tov een startpunt	fseek
code	indicatie voor het startpunt	fseek
	`SEEK_SET` : tov het begin van het bestand	
	`SEEK_CUR` : tov de actuele positie in het bestand	
	`SEEK_END` : tov het einde van het bestand	

Omdat met behulp van de `fseek` functie naar een willekeurige record in het bestand kan gegaan worden, zonder alle voorgaande records te lezen, wordt zo'n binair bestand ook een bestand met *directe toegankelijkheid* genoemd. Op een binair bestand worden elementen of objecten weggeschreven. Meestal spreekt men van *records*. Dus een binair bestand is een opeenvolging van records, beginnend met record 0. De lengte van de verschillende records in een binair bestand kan constant zijn. Men spreekt dan van een *bestand met vaste record lengte*. In andere gevallen kan een bestand records van verschillende lengte bevatten (*bestand met veranderlijke record lengte*).

3.5 Voorbeeld 1

```
/*
 * fil1.c : tekstbestanden en binaire bestanden
 */
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#define INNM  "koppel.txt"
#define RESNM "koppel.dat"
#define PMAX 25

typedef struct
        {
            float   x;
            float   y;
        } Punt;

int main( int argc, char    *argv[])
{
    FILE    *finp, *fuitp, *fdirp;
    Punt    p[PMAX], pnt;
    int     i, m;

    memset(p, '\0', PMAX*sizeof(Punt) );
    fuitp = stdout;
    finp = fopen(INNM, "r");
    if ( finp == NULL )
    {
        fprintf(stderr, "invoerbestand %s niet gevonden\n", INNM);
```

```
            exit(1);
30      }
        for ( m=1; m<PMAX; m++ )
32      {
            fscanf(finp, "%f %f%*c", &p[m].x, &p[m].y);
34          if ( feof(finp) )
                break;
36          fprintf(fuitp, "\t%f %f\n", p[m].x, p[m].y );
        }
38      fclose(finp);
        if ( m == PMAX )
40      {
            fprintf(fuitp, "maximum aantal punten (%d) ingelezen\n", PMAX);
42      }
        for (i=1; i<m; i++)
44          fprintf(fuitp, "%3d %10.3f %10.3f\n", i, p[i].x, p[i].y);
        fdirp = fopen(RESNM, "wb");
46      for (i=1; i<m; i++)
        {
48          fseek(fdirp, (long)i*sizeof(Punt), SEEK_SET);
            fwrite(&p[i], sizeof(Punt), 1, fdirp);
50      }
        fclose(fdirp);
52      fdirp = fopen(RESNM, "rb");
        for (i=1; i<m; i+=2)
54      {
            fseek(fdirp, (long)i*sizeof(Punt), SEEK_SET);
56          fread(&pnt, sizeof(Punt), 1, fdirp);
            fprintf(fuitp, "%3d %10.3f %10.3f\n", i, pnt.x, pnt.y);
58      }
        fseek(fdirp, 2L*sizeof(Punt), SEEK_SET);
60      for (i=2; i<m; i+=2)
        {
62          fread(&pnt, sizeof(Punt), 1, fdirp);
            fprintf(fuitp, "%3d %10.3f %10.3f\n", i, pnt.x, pnt.y);
64          fseek(fdirp, (long)sizeof(Punt), SEEK_CUR);
        }
66      fclose(fdirp);
    }
```

```
Invoerbestand:
  0.0    1.0
  2.0    4.0
  0.5   10.0
  4.1  100.0
  1.5 1000.0
```

```
Uitvoer:
  0.000000 1.000000
  2.000000 4.000000
  0.500000 10.000000
  4.100000 100.000000
  1.500000 1000.000000
1      0.000      1.000
2      2.000      4.000
3      0.500     10.000
4      4.100    100.000
5      1.500   1000.000
1      0.000      1.000
3      0.500     10.000
```

```
5       1.500   1000.000
2       2.000      4.000
4       4.100    100.000
```

De inhoud van het binaire bestand “koppel.dat” kan getoond worden met het UNIX-bevel

```
od -xf koppel.dat

0000000     0000    0000    0000    0000    0000    0000    3f80    0000
           0.0000000e+00   0.0000000e+00   0.0000000e+00   1.0000000e+00
0000020     4000    0000    4080    0000    3f00    0000    4120    0000
           2.0000000e+00   4.0000000e+00   5.0000000e-01   1.0000000e+01
0000040     4083    3333    42c8    0000    3fc0    0000    447a    0000
           4.0999999e+00   1.0000000e+02   1.5000000e+00   1.0000000e+03
0000060
```

Merk op. Bij elke `fseek` is er voor gezorgd dat het tweede argument van het type **long** is, bijvoorbeeld door middel van een cast operator (bijv. lijn 53). Op lijn 57 is het tweede argument het resultaat van een expressie van type **long** omdat één van de operands een long is, namelijk **2L**. Dit is belangrijk om het programma overdraagbaar te maken tussen 16 bit en 32 bit processoren.

3.6 Voorbeeld 2

```
/*
 *  fil2.c : bewerkingen op een binair bestand
 */
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

#define NAAM "koppel.dat"
#define PMAX 25

typedef struct
        {
            float   x;
            float   y;
        } Punt;

int main( int argc, char    *argv[])
{
    FILE    *fdirp;
    Punt    pnt, p[PMAX];
    int     i, m;

    memset(p, '\0', PMAX*sizeof(Punt) );
    fdirp = fopen(NAAM, "rb");
    m = fread(p, sizeof(Punt), PMAX, fdirp);
    printf("Aantal records %d\n", m);
    for (i=0; i<m ; i++)
    {
        printf("%3d %10.3f %10.3f\n", i, p[i].x, p[i].y);
    }
    fclose(fdirp);
```

```
32      fdirp = fopen(NAAM, "r+b");                    /* lezen en schrijven */
        for (i=1; i<m    ; i++)
34      {
            fseek(fdirp, (long)i*sizeof(Punt), SEEK_SET);
36          fread(&pnt, (long)sizeof(Punt), 1, fdirp);
            pnt.y += pnt.x;
38          pnt.x += 100.0;
            fseek(fdirp, -(long)sizeof(Punt), SEEK_CUR); /* een record terug */
40          fwrite(&pnt, sizeof(Punt), 1, fdirp);          /* overschrijven */
        }
42      fseek(fdirp, 0L, SEEK_SET);                              /* vooraan */
        for (m=0;    ; m++)
44      {
            fread(&pnt, sizeof(Punt), 1, fdirp);
46          if ( feof(fdirp) )
                break;
48          printf("%3d %10.3f %10.3f : %10.3f %10.3f\n",
                    m, p[m].x, p[m].y,   pnt.x, pnt.y);
50      }
        fseek(fdirp, 0L, SEEK_END);                            /* achteraan */
52      printf("offset %4d\n", ftell(fdirp) );
        for (i=1; i<m-1    ; i++)
54      {
            fwrite(&p[i], sizeof(Punt), 2, fdirp);             /* toevoegen */
56          printf("offset %4d : %4d\n", ftell(fdirp), i );
        }
58      fseek(fdirp, -(long)sizeof(Punt), SEEK_CUR);       /* vanaf laatste */
        for (i=1;   ; i++)
60      {
            fread(&pnt, sizeof(Punt), 1, fdirp);
62          printf("%3d %10.3f %10.3f \n", i, pnt.x, pnt.y);
            fseek(fdirp, -(2L*sizeof(Punt)), SEEK_CUR);        /* naar voor */
64          if ( ftell(fdirp) <= 0 )
                break;
66      }
        fclose(fdirp);
68  }
```

Uitvoer van dit programma:

```
  0      0.000      0.000
  1      0.000      1.000
  2      2.000      4.000
  3      0.500     10.000
  4      4.100    100.000
  5      1.500   1000.000
  0      0.000      0.000 :      0.000      0.000
  1      0.000      1.000 :    100.000      1.000
  2      2.000      4.000 :    102.000      6.000
  3      0.500     10.000 :    100.500     10.500
  4      4.100    100.000 :    104.100    104.100
  5      1.500   1000.000 :    101.500   1001.500
offset   48
offset   64 :    1
```

```
offset   80 :    2
offset   96 :    3
offset  112 :    4
  1      1.500   1000.000
  2      4.100    100.000
  3      4.100    100.000
  4      0.500     10.000
  5      0.500     10.000
  6      2.000      4.000
  7      2.000      4.000
  8      0.000      1.000
  9    101.500   1001.500
 10    104.100    104.100
 11    100.500     10.500
 12    102.000      6.000
 13    100.000      1.000
```

Merk op. Wanneer een bestand geopend is om te lezen en te schrijven, kan een `fread` gevolgd door een `fwrite` of omgekeerd eigenaardige (onjuiste) resultaten geven. Plaats tussen de `fread` en de `fwrite` een `fseek(...)` of een `ftell(...)`.

Verkleinen van een bestand. Wanneer achteraan een aantal records moeten verwijderd worden, dan kan het bestand afgekapt worden op de nieuwe lengte. Dit kan gebeuren met de functie `truncate`:

```
int truncate(char *padnaam, off_t lengte);
```

Het bestand met de naam in het eerste argument wordt afgekapt tot ten hoogste het aantal bytes gegeven in het tweede argument. Indien het oorspronkelijk bestand langer was, is de afgekapte informatie verloren. De terugkeerwaarde is `0` als het afkappen succesvol verlopen is. Bij een fout (bijvoorbeeld protectie: niet schrijfbaar) is het resultaat `-1`.
In het vorige voorbeeld worden vanaf lijn 49 een aantal records toegvoegd. Om deze terug te verwijderen kan op het einde van het programma het bestand afgekapt worden: `truncate(NAAM, 48);` Het resulterend bestand is niet het origineel bestand omdat in de eerste records ook nieuwe waarden zijn geschreven (lijnen 37-38).

3.7 Denktaak

```
#include <stdio.h>
#include <string.h>
#define LEN 6
typedef struct kleurfruit
        {
            char woord[LEN];
            short getal;
        } Tkf;

int main(int argc, char *argv[])
{
    FILE *fpt, *fpd;
    int i, n;
    Tkf korf, mand;
    long reclen = sizeof(Tkf);

```

```
    memset(&korf, '\0', sizeof(Tkf) );
18  fpt = fopen("fruit.txt", "r");
    fpd = fopen("fruit.dat", "wb");
20  for (n=0;    ; n++)
    {
22      fscanf(fpt, "%s%hd%*c", korf.woord, &korf.getal);
        if ( feof(fpt) )
24          break;
        fwrite(&korf, sizeof(Tkf), 1, fpd);
26  }
    printf("aantal elementen %d\n", n);
28  fclose(fpt);
    fclose(fpd);
30  fpd = fopen("fruit.dat", "rb");
    for (i=0; i<n ; i++)
32  {
        if ( (i%2) == 0 )
34          fseek(fpd, reclen, SEEK_CUR);
        fread(&mand, reclen, 1, fpd);
36      printf("%*.*s ", LEN, LEN, mand.woord);
        if ( (i%2) == 1 )
38      {
            printf("\n");
40          fseek(fpd, reclen, SEEK_CUR);
        }
42      else
            fseek(fpd, i*reclen, SEEK_SET);
44  }
    fclose(fpd);
46  }
```

tekstbestand
`fruit.txt`:

```
appel 2
rood -13
sinaas 3
oranje -17
peer 5
groen -19
banaan 7
geel -23
pruim 11
paars -29
```

Het resulterende data bestand `fruit.dat`:

```
0000000  a  p  p  e  l \0   0002    r  o  o  d \0 \0   fff3
0000020  s  i  n  a  a  s   0003    o  r  a  n  j  e   ffef
0000040  p  e  e  r \0  e   0005    g  r  o  e  n \0   ffed
0000060  b  a  n  a  a  n   0007    g  e  e  l \0  n   ffe9
0000100  p  r  u  i  m \0   000b    p  a  a  r  s \0   ffe3
0000120
```

Wat is de uitvoer van het programma; geef hierbij wat verklarende uitleg.
Wat kan er aan het programma verbeterd worden?

4 Ontwerpen van programma's

4.1 Ontwerp

Voor het ontwerpen en schrijven van computerprogramma's zijn twee zaken nodig:

- een goed begrip van de elementen van een programmeertaal;
- het begrijpen van de definitie en structuur van het probleem dat moet opgelost worden of de taak die moet uitgevoerd worden.

In vorige hoofdstukken werd een overzicht gegeven van de basiselementen van de programmeertaal C. Daarnaast zijn dus technieken nodig om een probleem te analyseren en daaruit een oplossingsmethode af te leiden welke kan omgevormd worden tot een programma.
Volgens G. Polya (eind jaren veertig) zijn er 4 stappen nodig om een algoritme te ontwerpen:

1. Begrijp het probleem.
2. Tracht een idee te vormen over hoe een algoritmische procedure het probleem zou kunnen oplossen.
3. Formuleer het algoritme en schrijf het neer als een programma.
4. Evalueer het programma op nauwkeurigheid en op de mogelijkheden om het als middel te gebruiken bij het oplossen van andere problemen.

Voor de meeste problemen moet de ontwerper een iteratieve oplossingsbenadering toepassen. Beginnen met een vereenvoudigd probleem en hiervoor een betrouwbare oplossing construeren. Daardoor krijgt men meer vat op het probleem. Dit beter begrijpen kan dan gebruikt worden om meer details in te vullen en een verfijnde oplossingsprocedure uit te denken, totdat men komt tot een bruikbare oplossing voor het realistische originele probleem.
De eerste twee stappen kunnen in praktijk zeer moeilijk zijn en in het geval van enkele wiskundige problemen, zal de oplossing equivalent zijn met het ontdekken en bewijzen van nieuwe stellingen voordat een computerprogramma kan geschreven worden. Het is bijvoorbeeld tot op dit moment niet geweten of volgende procedure altijd zal stoppen voor een willekeurige initiële waarde.

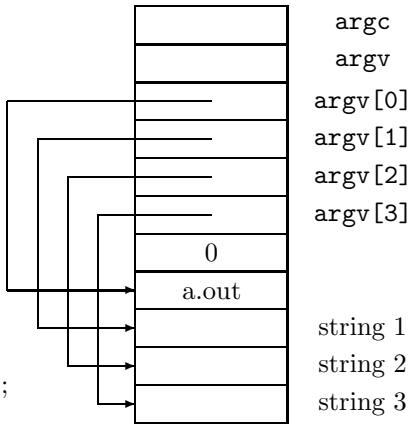
```
/*
 *  stop.c :  stopt dit programma ?
 */
#include <stdio.h>
#include <stdlib.h>
int main(int argc, char *argv[])
{
        int getal;
        int stap = 0;

        getal = atoi(argv[1]);
        while ( getal != 1 )
        {
             if ( getal%2 == 0 )              /* even ? */
                  getal = getal/2;
             else
                  getal = 3*getal+1;
             stap++;
        }
        printf("programma stopt na %d stappen\n", stap);
}
```

De functie *ascii-to-int* **int** atoi(**char** ∗) is een voorgedefinieerde functie met één parameter: een pointer naar een array van characters. Deze functie berekent de overeenkomstige geheeltallige waarde en deze waarde wordt als resultaat teruggegeven. Een gelijkaardige functie is *ascii-to-float* **double** atof(**char** ∗): het omzetten van een string naar de overeenkomstige reële waarde.
Niet alle problemen kunnen opgelost worden door middel van computerprogramma's. En zelfs wanneer een algoritme bestaat, is het niet altijd mogelijk dit algoritme te implementeren in een programma omwille van beperkingen van de grootte van het werkgeheugen en de beschikbare rekentijd.

4.2 De `main` functie

Een programma bestaat uit een aantal functies en een hoofdfunctie `main` waarvan de terugkeerwaarde van type `int` is. Deze hoofdfunctie kan ook parameters hebben.

```
/*    mainarg.c :   programmaparameters */
#include <stdio.h>

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

    printf("argc     = %d\n", argc);
    for (i=0; i<argc; i++)
       printf("argv[%d] = %s\n", i, argv[i]);
    argv++; argc--;
    printf("argc     = %d\n", argc);
    for (i=0; i<argc; i++)
       printf("\targv[%d] = %s\n", i, argv[i]);
    return 0;
}
```



Veronderstel na compilatie de uitvoerbare versie in het bestand `vbmp` zit. Wanneer het programma `vbmp` gestart wordt, door de naam ervan in te tikken (of door op een icoon te klikken), kunnen een willekeurig aantal argumenten bijgevoegd worden:

```
vbmp appel peer genoeg
```

De uitvoer is dan

```
argc    = 4
argv[0] = vbmp
argv[1] = appel
argv[2] = peer
argv[3] = genoeg
```

Tijdens het starten van het programma, heeft het systeem ervoor gezorgd dat er actuele argumenten gemaakt worden die doorgegeven worden aan de functie `main`. Er zijn twee argumenten:

argc : het aantal ingetikte woorden, inclusief de naam van het programma;

argv : een array van `argc` elementen, waarbij elk element een pointer is naar een string, en een bijkomend element met waarde `(char *)NULL`. Element i in deze array bevat het beginadres van het i-de ingetikte woord.

Dus `argv[0]` wijst naar een string met waarde de naam van het programma; `argv[1]` naar het eerste argument, ..., `argv[argc-1]` naar het laatste argument. Het laatste (NULL) element is nodig om het einde van de array aan te geven. Deze lengte kan niet op voorhand vastgelegd worden, omdat een willekeurig aantal argumenten ingetikt kunnen worden.

4.3 Structuur van een programma

Men kan weinig algemene richtlijnen geven om een programma te structureren, omdat de structuur afhankelijk is van het origineel probleem en het algoritme dat gebruikt wordt. Toch is het nuttig om een programma te verdelen in drie fazen:

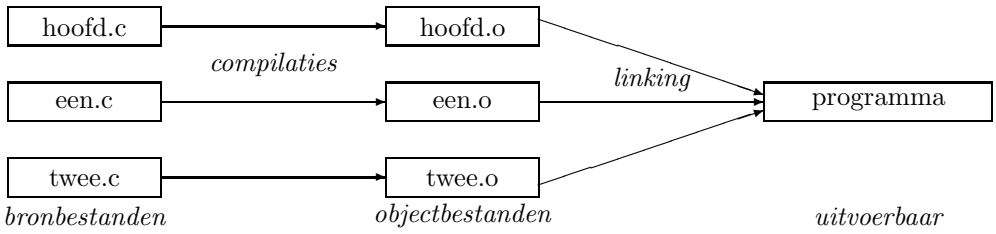
Initialisatie : declaratie en initialisatie van de variabelen. Het is nuttig om alle beschikbare informatie in te voeren *voordat* de berekeningen starten, zodat controles op juistheid van gegevens kan gebeuren. Een complexe simulatie uitvoeren kan uren duren en het zou spijtig zijn dat het programma halverwege vastloopt omdat de gebruiker een foutieve invoer doet.

Data verwerking : het grootste gedeelte van het programma. De implementatie van de oplossingsprocedure met behulp van functies, iteratie-lussen en keuze-statements.

Stockeren van de resultaten : op het einde van het programma. Het uitschrijven naar het beeldscherm en/of naar een bestand.

4.3.1 Opsplitsen in verschillende bronbestanden

Wanneer er verschillende functies nodig zijn en het programma een zekere omvang krijgt, kan het nuttig zijn het programma te verdelen over verschillende bronbestanden. Een voordeel hierbij is dat de verschillende bronbestanden afzonderlijk kunnen gecompileerd worden. Zo'n compilatie resulteert in een *object*bestand. De verschillende objectbestanden worden samen ge*link*t met als resultaat een uitvoerbaar programma.



In het voorbeeld op de volgende bladzijde is langs de linkerkant een programma weergegeven dat bestaat uit één bronbestand, `bindec.c`. Het bevat een `main()` en twee functies. Met de eerste functie kan een decimaal getal omgezet worden in zijn binair equivalent en de tweede functie kan voor de omgekeerde bewerking gebruikt worden.
Langs de rechterkant is het programma opgedeeld over drie bronbestanden: `hoofd.c`, `een.c` en `twee.c`. Hiernaast zijn de bijhorende headerbestanden `een.h` en `twee.h` weergegeven.

```
/* een.h */
#define LEN 100
EXT int rbits[LEN];
```

```
/*  twee.h */
#define LEN 100
EXT int gbits[LEN];
EXT int l;
```

4.3.2 Een `header` bestand

Wanneer in de verschillende bronbestanden dezelfde constanten gebruikt worden, kunnen de definities van deze constanten verzameld worden in een *header*bestand. Ook de functie-prototypes

```c
/*
 * bindec.c : omzetting
 */
#include <stdio.h>
#define LEN 100

int gbits[LEN];
int rbits[LEN];
int l;
int b2d(void);
int d2b(int);

int main(int argc, char *argv[])
{
    int i, r, getal;

    printf("decimaal getal : ");
    scanf("%d%*c", &getal);
    l = d2b(getal);
    for (i=0; i<l; i++)
    {
        printf("%d", rbits[i]);
        gbits[i] = rbits[i];
    }
    r = b2d();
    printf(" : %d\n", r);
}

int d2b(int n)
{
    int i=0, l=0, a[LEN];
    while ( n > 0 )
    {
        a[l++] = n%2;
        n /= 2;
    }
    for (i=0; i<l; i++)
        rbits[i] = a[l-1-i];
    return l;
}

int b2d(void)
{
    int i=0, r=1;
    for (i=1; i<l; i++)
        r = r * 2 + gbits[i];
    return r;
}
```

```c
/* hoofd.c */
#include <stdio.h>
#define EXT
#include "een.h"
#include "twee.h"
int b2d(void), d2b(int);
int main(int argc, char *argv[])
{
    int i, r, getal;
    printf("decimaal getal : ");
    scanf("%d%*c", &getal);
    l = d2b(getal);
    for (i=0; i<l; i++)
    {
        printf("%d", rbits[i]);
        gbits[i] = rbits[i];
    }
    r = b2d();
    printf(" : %d\n", r);
}
```

```c
/* een.c */
#include <stdio.h>
#define EXT extern
#include "een.h"
int d2b(int n)
{
    int i=0, l=0, a[LEN];
    while ( n > 0 )
    {
        a[l++] = n%2;
        n /= 2;
    }
    for (i=0; i<l; i++)
        rbits[i] = a[l-1-i];
    return l;
}
```

```c
/*  twee.c */
#include <stdio.h>
#define EXT extern
#include "twee.h"
int b2d(void)
{
    int i=0, r=1;
    for (i=1; i<l; i++)
        r = r * 2 + gbits[i];
    return r;
}
```

Figuur 4.1: Een programma met meerdere bronbestanden

kunnen hierin gedeclareerd worden. Zo'n headerbestand kan met een `#include` ingevoegd worden in een bronbestand.
Door middel van zo'n headerbestand wordt de *publieke interface* van een functie gescheiden van de actuele definitie van de functie. Om de functie te gebruiken in een programma is enkel de prototype-informatie nodig; de gebruiker hoeft de specifieke details van de implementatie van de functie niet te kennen. Dit is algemeen aanvaard voor standaardfuncties, bijvoorbeeld `sqrt`, maar kan dus ook toegepast worden op zelf gedefinieerde functies of functies die door een collega in het project gedefinieerd zijn. Het scheiden van functie-declaratie en functie-definitie in twee aparte bestanden geeft de ontwerper de mogelijkheid van *complexity hiding*. Dit is een belangrijk middel om grote en complexe programma's te ontwerpen.
Naast de definitie van constanten en de declaratie van functies kunnen in een headerbestand ook globale variabelen gedeclareerd worden. Wanneer het headerbestand in verschillende bronbestanden ingevoegd wordt, moet er voor gezorgd worden dat de declaratie alleen de naam en het type van de variabele vastlegt. Wanneer er ook plaats in het werkgeheugen gereserveerd zou worden, zou dit verschillende keren gebeuren, wat niet de bedoeling is. Zo'n declaratie wordt aangeduid met `extern`. Extern geeft aan dat de definitie (d.i. geheugenreservering) in een ander bronbestand gebeurt.

Static. Naast globale variabelen die bereikbaar zijn in gans het programma en lokale variabelen die alleen binnen een functie gekend zijn, kan men variabelen definiëren die alleen binnen één bronbestand gekend zijn. Dit zijn globale variabelen die `static` gedeclareerd zijn.
Lokale variabelen die `static` gedeclareerd zijn, zijn alleen binnen de functie toegankelijk. Het verschil met `auto` lokale variabelen is de levensduur. Een lokale automatische variabele wordt gecreëerd op het moment dat een functie opgeroepen wordt en verdwijnt wanneer de functie uitgevoerd is. Een lokale statische variabele wordt gecreëerd op het moment dat het programma start en verdwijnt pas wanneer het programma stopt. Tussen twee oproepen van de functie, behoudt zo'n statische variabele zijn waarde, maar deze is niet bereikbaar.

Voorbeeld:

```
1   /*
     *  vbstat.c :  voorbeeld met een static-variabele
3    */
    #include <stdio.h>
5   void f(void);

7   int main(int argv, char *argv[])
    {
9       int i = 14;

11      f();
        printf("main     : i = %d\n", i);
13      f();
        printf("main     : i = %d\n", i);
15      f();
    }
17
    void f(void)
19  {
        static int i = 5;       /* statische variabele  */
21
        printf("functie : i = %d\n", i);
23      i++;
    }
```

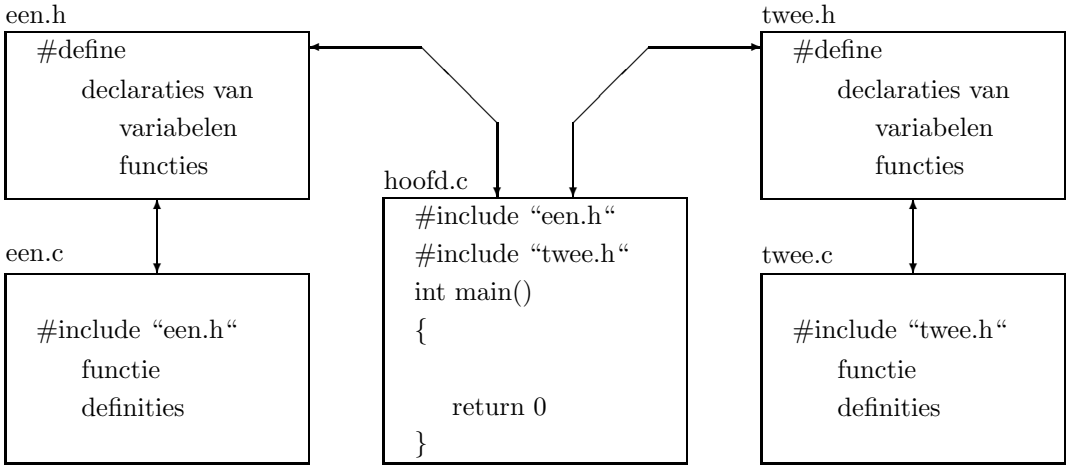
Dit programma geeft als output:

```
functie : i = 5
main    : i = 14
functie : i = 6
main    : i = 14
functie : i = 7
```

4.3.3 Beheer met `make`

Wanneer het aantal header-bestanden en bronbestanden groot wordt, is het niet eenvoudig meer om juist te weten welke bestanden moeten hercompileerd worden nadat ergens één kleine aanpassing gebeurd is aan het programma. Ook de *volgorde* waarin de bestanden moet gecompileerd worden kan belangrijk zijn. Met behulp van `make` kan gemakkelijk bijgehouden worden welke bronbestanden moeten hercompileerd worden en welke objectbestanden moeten herlinkt worden nadat sommige bronbestanden aangepast zijn.

Veronderstel volgend project:



Om het compilatie en linking proces te automatiseren, kan een `makefile` gecreëerd worden:

```
# voorbeeld van een makefile
all: programma

programma: hoofd.o een.o twee.o
     cc -o programma hoofd.o een.o twee.o

hoofd.o: hoofd.c een.h twee.h
     cc -c hoofd.c

een.o: een.c een.h
     cc -c een.c

twee.o: twee.c twee.h
     cc -c twee.c
```

Het `makefile` bestand bestaat uit een aantal blokken. Voor de leesbaarheid zijn de blokken van elkaar gescheiden met een lege lijn. Een blok bestaat uit één of meerdere lijnen. De eerste lijn moet in de eerste kolom beginnen. Deze lijn bevat een naam (een *doel*), een dubbele punt en een lijst van namen (*afhankelijkheden*). De betekenis is dat het doel afhankelijk is van elk van de volgende afhankelijkheden. Wanneer één van deze afhankelijkheden van recentere datum is dan het doel (omwille van een wijziging in het desbetreffende bestand), moet het doel hermaakt worden.

Hoe dit hermaken gebeurt, wordt op de volgende lijn gespecificeerd. De tweede lijn MOET met een TAB teken beginnen. Daarna volgt het bevel dat moet uitgevoerd worden om vertrekkende van de verschillende afhankelijkheden, het doel te maken. Voordat dit bevel gestart wordt, wordt eerst nagegaan of niet één van de afhankelijkheden zelf geen doel is. Indien dit het geval is, wordt nagegaan of dit doel misschien niet eerst moet hermaakt worden.
In het voorbeeld wordt het default doel `all` gecontroleerd. Dit doel is afhankelijk van `programma`, wat op zijn beurt afhankelijk is van 3 objectbestanden. Wanneer één of meer van deze objectbestanden van recentere datum is dan `programma`, wordt het link statement van de volgende lijn uitgevoerd. Maar eerst wordt vastgesteld dat de objectbestanden zelf een doel zijn. Dus van elk van de objectbestanden worden de afhankelijkheden gecontroleerd. Bijvoorbeeld bij `hoofd.o`, wanneer `hoofd.c`, `een.h` of `twee.h` van recentere datum is dan `hoofd.o`, wordt `hoofd.c` hercompileerd.

4.4 Denktaak

```
/* gvar.h */
#include <stdio.h>
float functie(float);
double reken(double);
int routine();

EXT int g;
```

```
/* gvar.c */
#include <stdio.h>
#define EXT
#include "gvar.h"

int main(int argc, char *argv[])
{
   float a = 4.0;

   a = functie(a);
   printf("main: g%d a%f\n", g,a);
   a = functie(a);
   printf("main: g%d a%f\n", g,a);
}

float functie(float b)
{
   double z;
   g = routine();
   z = reken( (double)b );
   return (float)g + b + z;
}
```

```
/* svar.c */
#include <stdio.h>
#include <math.h>
#define EXT extern
#include "gvar.h"

static int p = 0;

int routine()
{
   static int q = 1;
   int i ;
   i =  g + p++ + q++;
   printf("\t p%d q%d\n", p,q);
   return  i;
}

double reken(double c)
{
   double f;
   f =  (++p) + sqrt(c);
   printf("\t p%d ", p);
   printf(" c%lf f%lf\n", c,f);
   return  f;
}
```

Bespreek de attributen (naam, type, bereik en levensduur) van de verschillende variabelen (maar niet de formele parameters) in bovenstaand programma, bestaande uit drie bronbestanden `gvar.h`, `svar.c` en `gvar.c`.
Maak een bijhorende `makefile` (vergeet de mathematical library niet).

5 Arrays en pointers

5.1 Een-dimensionale array

Gerelateerde variabelen van hetzelfde data-type kunnen samengevoegd worden in een *array*. Een array is een *samengesteld data-type* en kan gezien worden als een rij waarvan de elementen allemaal met dezelfde naam maar elk met een eigen volgnummer worden aangeduid.
Een programma om een rij van getallen in omgekeerde volgorde af te drukken:

```
1  /*
    * keerom3.c : (met functies)
3   */
   #include <stdio.h>
5  #define AANTAL 100

7  void leesrij(int x[], int n);                    /* prototype */
   void drukrij(int x[], int n);
9
   int main(int argc, char *argv[])
11 {
       int m;
13     int a[AANTAL];

15     scanf("%d%*c", &m);
       leesrij(a, m);                          /* functie oproep */
17     drukrij(a, m);
   }
19
   void leesrij(int x[], int n)              /* functie definitie */
21 {
       int i;
23
       printf("Geef %d gehele getallen: ", n);
25     for(i=0; i<n; i++)
       {
27         scanf("%d", &x[i]);                    /* naast elkaar */
       }
29     scanf("%*c");
   }
31
   void drukrij(int r[], int m)
33 {
       int i;
35
       printf("In omgekeerde volgorde:\n");
37     for (i=0; i<m; i++)
       {
39         printf("%6d", r[m-1-i]);
       }
41     printf("\n");
   }
```

De definitie (miv. declaratie) `int a[AANTAL]` geeft aan dat `a` de naam is van een rij van honderd gehele (int) getallen. Een declaratie van een array bevat dus drie elementen:

data type : het type (int, double, ...) van elk van de elementen in de array;

naam : zoals bij elke variabele;

grootte : het aantal elementen in de array; dit moet een *constante-expressie* zijn, zodat de compiler de juiste hoeveelheid geheugenplaatsen in het werkgeheugen kan reserveren.

In plaats van een *naamconstante* (symbolische naam) had ook een getal voor de grootte van de array kunnen gebruikt worden (`int a[100]`). Wanneer het programma moet aangepast worden om bijvoorbeeld rijen met 1000 elementen om te keren, moet dit bij gebruik van een naamconstante slechts op één plaats gebeuren, bij de `define`.
De lengte of grootte van een array wordt ook wel *dimensie* genoemd. Een rij is een een-dimensionale array en wordt soms ook *vector* genoemd.
Bij de declaratie kunnen de elementen van de array geïnitialiseerd worden:

```
int a[5] = { 17, 19, 23, 29, 31 };
```

Bij tegelijk declareren en initialiseren, hoeft de lengte niet gespecificeerd te worden:

```
int a[] = { 17, 19, 23, 29, 31 };
```

Dit is nu echter geen goede manier van werken meer. Eén reden hiervoor is het niet gebruiken van de `define` constructie om de lengte een symbolische naam te geven, die dan overal in het programma kan gebruikt worden. Wanneer er minder initialiserende waarden zijn dan opgegeven in de lengte, worden de eerste elementen geïnitialiseerd met de gegeven waarden en de volgende elementen krijgen de waarde nul (afhankelijk van de gebruikte compiler).

```
int a[12] = { 17, 19, 23, 29, 31 };
```

De verschillende elementen in de array `a` kunnen aangesproken worden met de `[]` operator:

```
a[0],  a[1],  a[2],  ..., a[98],  a[99]
```

Honderd geeft het aantal elementen aan, maar de rangnummer begint bij 0 en eindigt dus bij 99. Voor het element `a[100]` is dus GEEN plaats voorzien.
Bij het gebruik van een array-element `a[i]` wordt i de *index* of *subscript* genoemd. Deze index kan een constante, een variabele of een expressie zijn. De waarde van de index moet wel een geheel getal zijn, groter of gelijk aan nul en kleiner dan het aantal elementen dat bij de declaratie is opgegeven.
De naam van de array zelf kan niet zomaar in om het even welke expressie gebruikt worden. Deze naam duidt namelijk geen enkelvoudige geheugenplaats aan maar een rij van geheugenplaatsen. De rekenkundige operator `+` bijvoorbeeld kan wel gebruikt worden om de inhoud van twee geheugenplaatsen op te tellen. Het resultaat kan dan met de toekenningsoperator (`=`) toegewezen worden aan een derde variabele (geheugenplaats).

```
double a, b, c;

a = b + c;
```

Wanneer `a`, `b` en `c` arrays zijn, elk bijvoorbeeld van lengte 10, is bovenstaande rekenkundige operatie NIET mogelijk. Een variabele van het type array heeft geen *lvalue*. De optelling moet element per element gebeuren:

```
/*
 * arsom.c : optellen van twee arrays
 */
#include <stdio.h>
#define AANTAL 5
```

```
void main(void)
{
    double a[AANTAL], b[AANTAL], c[AANTAL];
    int   i;

    /* inlezen van vectoren b en c */
    for (i=0; i<AANTAL; i++)
        a[i] = b[i] + c[i];
    /* afdrukken van vector a  */
}
```

De expressie `a == b` is syntactisch wel correct maar wordt semantisch anders geïnterpreteerd dan waarschijnlijk bedoeld is. Deze expressie gaat niet na of de twee arrays dezelfde elementen bevatten. De gelijkheid nagaan moet weer element per element gebeuren:

```
/*
 * arvgl.c : vergelijken van twee arrays
 */
#include <stdio.h>
#define AANTAL 5

void main(void)
{
    int a[AANTAL], b[AANTAL];
    int   i;
    int   gelijk = 1;

    /* inlezen van vectoren a en b */
    for (i=0; i<AANTAL; i++)
        if ( a[i] != b[i] )
        {
            gelijk = 0;
            break;
        }
}
```

Na de lus kan de variabele `gelijk` getest worden. Wanneer deze gelijk is aan 1, zijn de twee arrays gelijk, in de betekenis dat de waarden in de corresponderende elementen gelijk zijn.
Merk op dat in het programma een *kortsluitingsprincipe* kan ingebouwd worden. Na het vinden van een eerste index waarbij de twee elementen niet gelijk zijn, hoeven de volgende elementen niet meer vergeleken te worden.
De naam van een array duidt dus een rij van geheugenplaatsen aan. De *rvalue* van deze variabele is het *beginadres* van de rij. Wanneer deze naam als actuele parameter in een functie-oproep gebruikt wordt, dan wordt de waarde van de formele parameter in de functie-definitie gelijk aan dit beginadres. Resultaat: de formele parameter duidt dezelfde rij van geheugenplaatsen aan! Dit wil dus zeggen dat wanneer een arraynaam als actueel argument gebruikt wordt, deze array niet element per element in de formele parameter gecopieerd wordt (zoals wel gebeurt bij een enkelvoudig data-type).
Omdat de formele parameter wijst naar de originele rij, zal elke wijziging die in de functie door middel van de formele parameter gebeurt, uitgevoerd worden op deze originele rij.
Bij de definitie van de functie wordt bij de formele parameter de lengte van de array niet gegeven. Deze lengte is van geen belang omdat in de formele parameter alleen een beginadres zal komen bij een oproep en niet de volledige rij.

5.2 Meer-dimensionale arrays

In veel toepassingen heeft men te maken met een tabel, soms ook *matrix* genoemd. Dit kan in C neergeschreven worden als een array met twee dimensies:

```
int  a[MAXRIJ][MAXKOLOM];
```

a[0][0]	a[0][1]
a[1][0]	a[1][1]
a[2][0]	a[2][1]

De eerste dimensie geeft aan hoeveel rijen er zijn en de tweede dimensie het aantal kolommen. Wanneer MAXRIJ gelijk is aan 3 en MAXKOLOM gelijk aan 2 geeft dit de volgende matrix
Deze matrix wordt in het werkgeheugen echter gestockeerd als één lange rij van opeenvolgende elementen:

a[0][0]	a[0][1]	a[1][0]	a[1][1]	a[2][0]	a[2][1]

Dit kan gezien worden als een rij van elementen waarin elk element op zich een rij is.
Bij de definitie van de variabele kan deze geïnitialiseerd worden:

```
int a[3][2] = { {1, 2}, {2, 3}, {3, 4} };
```

Rekening houdend met het feit dat deze matrix als een rij gestockeerd wordt, kan men ook schrijven

```
int a[3][2] = { 1, 2, 2, 3, 3, 4 };
```

Dit is echter veel minder duidelijk.

Opgave. Construeer een matrix waarin elk element gelijk is aan de som van de rij-index en de kolom-index en druk deze matrix af.

```
/*
 *  matrix.c :  met mat[i][j] = i+j
 */
#include <stdio.h>
#define MAXRIJ   3
#define MAXKOLOM 5

void drukmat(int mat[][MAXKOLOM], int m, int n);

int main(int argc, char *argv[])
{
    int  mat[MAXRIJ][MAXKOLOM];
    int  rij, kolom;

    for (rij=0; rij<MAXRIJ; rij++)
        for (kolom=0; kolom<MAXKOLOM; kolom++)
            mat[rij][kolom] = rij + kolom;
    drukmat(mat, MAXRIJ, MAXKOLOM);
}

void drukmat(int mat[][MAXKOLOM], int m, int n)
{
    int i, j;

    for (i=0; i<m; i++)
    {
        for (j=0; j<n; j++)
        {
```

```
            printf("%4d", mat[i][j]);
        }
        printf("\n");
    }
}
```

De geconstrueerde matrix heeft volgende inhoud:

0	1	2	3	4
1	2	3	4	5
2	3	4	5	6

en wordt in het werkgeheugen als één lange rij gestockeerd:

0	1	2	3	4	1	2	3	4	5	2	3	4	5	6

In de formele parameter zal bij een oproep alleen het beginadres komen net zoals bij een eendimensionale array. Toch wordt in de definitie van de functie bij de formele parameter de waarde van de tweede dimensie van de array opgegeven. Deze lengte is in de functie nodig om de positie te kunnen berekenen van het element in de lange rij waarin de matrix in het werkgeheugen gestockeerd is. Het element `mat[i][j]` kan gevonden worden op positie $i * \text{MAXKOLOM} + j$. De grootte van de eerste dimensie is hiervoor niet nodig en moet dus ook niet gespecificeerd worden in de declaratie van de formele parameter.

5.3 Pointers: operatoren

De unaire operator `&` toegepast op een variabele heeft als resultaat het adres van deze variabele in het werkgeheugen. Zo'n variabele wordt een *pointer* variabele genoemd, omdat de inhoud een verwijzing is naar een andere variabele.

```
p = &a
```

De omgekeerde operator bestaat ook: In dit geval wordt de verwijzing die in de pointer `p` zit, *gevolgd*. Men komt dus uit bij een andere geheugenplaats en de inhoud van die geheugenplaats is het resultaat.

```
b = *p
```

Het is ook mogelijk `*p` als lvalue te gebruiken: Op de plaats waar de pointer `p` naar wijst, wordt de waarde 8 gestockeerd.

```
*p = 8
```

Bij de declaratie van pointervariabelen wordt aangegeven naar wat voor data-type de pointer wijst:

```
int *p;
```

Het object waar `p` naar wijst (`*p`) is van het type `int`. Of, `p` is dus een pointer naar een integer. Als je de verwijzing in `p` volgt, dan kom je op een `int` uit. Noteer wel dat na deze declaratie, alleen maar plaats gereserveerd is in het werkgeheugen. De inhoud van `p` is nog niet geïnitialiseerd, dus `p` wijst nog naar *nergens*.

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

int main(int argc, char *argv[])
{
    int a, b;
    int *p;

    p = &a;
```

```
    *p = 5;

    p = &b;
    printf("Geef een geheel getal: ");
    scanf("%d%*c", p);
    printf("a = %d,     b = %d (p=%p)\n", a, b, p);
}
```

Soms wordt een iets andere notatie gebruikt: Men zou dit dan kunnen lezen als de variabele `p` is een "pointer naar een int".

```
int* p;
```

Deze notatie is niet zo duidelijk wanneer verschillende variabelen op dezelfde lijn gedeclareerd worden. Niettegenstaande de intentie van de schrijver, is de variabele `i` geen pointer maar gewoon een `int`.

```
int* p, i;
```

5.4 Toepassing: het wijzigen van variabelen via parameters

Wanneer in C een functie met parameters opgeroepen wordt, worden de waarden van de actuele argumenten doorgegeven naar de formele parameters: de waarden van de actuele argumenten worden gebruikt als initialisatie voor de formele parameters.
In de functie zelf worden de formele parameters gebruikt als operands in de expressies. Deze formele parameters kunnen daarbij van waarde veranderen. Maar wanneer de functie uitgevoerd is en verlaten wordt (via `return`), worden deze eventuele aangepaste waarden van de formele parameters NIET teruggecopieerd naar de actuele argumenten in de oproep.
Om wijzigingen die in de functie gebeuren, toch effect te laten hebben in de oproepende routine, moet iets analoogs gebeuren als met het doorgeven van een array naar een functie. In plaats van de waarde zelf door te geven naar de functie, kan het adres naar de variabele doorgegeven worden. De formele parameter is dan van het pointer-type. In de functie wijst de waarde van de pointer naar de variabele zelf, zodat de waarde ervan kan gewijzigd worden.

```
/*
 * wissel.c :   wijzigen van variabelen via parameters
 */
#include <stdio.h>
void wissel(int *pi, int *pj);

int main(int argc, char *argv[])
{
    int a, b;

    a = 5;
    b = 8;

    printf("Voor de functieoproep is a = %d   b = %d\n", a, b);
    wissel(&a, &b);
    printf("  Na de functieoproep is a = %d   b = %d\n", a, b);
}

void wissel(int *x, int *y)
{
    int hulp;

    hulp = *x;
    *x = *y;
```

```
25          *y = hulp;
        }
```

Dit programma drukt af:

```
Voor de functieoproep is a = 5   b = 8
  Na de functieoproep is a = 8   b = 5
```

5.5 Arrays en pointers

5.5.1 De naam van een array

Door middel van de declaratie `int x[5]` wordt in het werkgeheugen plaats voor 5 gehele getallen gereserveerd. Deze plaatsen kunnen aangesproken worden als `x[0], x[1], ... x[4]`. De naam van de array `x` is een aanduiding voor het beginadres van deze gereserveerde zone. De grootte van deze gereserveerde zone wordt vastgelegd tijdens compilatie. De naam van de array is een expressie met een waarde gelijk aan het adres van het eerste element van de array: de *rvalue* van `x` is het beginadres van de array, `x` heeft geen *lvalue*. De variabele `x` kan dus NIET langs links in een toekenningsexpressie gebruikt worden.

```
int    x[5];
int    *p;
int    *q;

p = &x[0] ;
q = x;
p++;
/* x++  : dit kan niet: x heeft geen lvalue */
```

Het resultaat van deze twee toekenningen is identiek: zowel `p` als `q` zijn pointers die wijzen naar het begin van de plaats waar de array `x` gestockeerd is. Ook de naam `x` wijst naar dit begin, maar hier is een klein onderscheid. `p` en `q` zijn variabelen die door de toekenningen een waarde gekregen hebben; deze variabelen kunnen een andere waarde krijgen. `x` is de naam van een array en kan niet van waarde veranderen! Of `p` en `q` hebben een *lvalue*: er is plaats in het werkgeheugen voorzien; terwijl voor `x` zelf geen plaats voorzien is: `x` heeft geen *lvalue*.

5.5.2 De grenzen van een array

Door middel van een declaratie `int x[5]` wordt een gebied van 5 integers gereserveerd in het werkgeheugen. Deze elementen kunnen aangesproken worden met `x[0], x[1], x[2], x[3]` en `x[4]`. Maar een standaard C compiler zal geen problemen maken wanneer `x[5]` of `x[-1]` gespecificeerd wordt. Het is zo dat elke index mag gebruikt worden, tijdens run-time wordt geen controle gedaan of de index wel binnen de gedefinieerde grenzen ligt.
Om een element met index **i** uit een array van integers op te halen, wordt gekeken naar de integer die gestockeerd is op een offset van $i \times$ **sizeof(int)** bytes vanaf de start van de array. Deze index i kan positief of negatief zijn. Wanneer deze index buiten de gedefinieerde grenzen van de array valt, zal een element in het werkgeheugen aangesproken worden dat waarschijnlijk voor iets anders in gebruik is. Wanneer zo'n element in een expressie gebruikt wordt, zal dit element een waarde geven die waarschijnlijk niet erg zinvol is. Wanneer een toekenning aan zo'n element gebeurt, zal andere nuttige informatie verloren zijn. Het kan ook zijn dat de toekenning leidt tot een run-time fout omdat een gedeelte van het werkgeheugen wordt aangesproken waar door het programma niet mag gestockeerd worden. Dit kan tot zeer moeilijk te vinden fouten leiden, die eventueel pas opduiken nadat het programma al jaren in gebruik is.

Deze flexibele omgang met arrays is één van de grote ergernissen bij mensen die van een andere taal overstappen naar C. Maar een echte C-programmeur weet wat hij aan het doen is en heeft daarbij geen run-time controles nodig.

5.5.3 Rekenen met pointers

Een pointer is een data type gelijkaardig aan een integer. Er kunnen een beperkte set van bewerkingen met pointers gebeuren:

- De optelling of de aftrekking van een integer bij/van een pointer geeft een nieuw adres.
- Pointers kunnen met elkaar vergeleken worden (`==` en `!=`) om na te gaan of ze naar hetzelfde element in het geheugen wijzen; of ze dus hetzelfde adres bevatten.
- De aftrekking van twee pointers geeft het aantal elementen tussen de twee adressen.

```
/*
 * arp.c : rekenen met pointers
 */
#include <stdio.h>
#define AANTAL 5

int main(int argc, char *argv[])
{
    int     a[AANTAL];
    int     *pi;
    int     *qi;
    int     i;

    for (i=0; i<AANTAL; i++)
        a[i] = i*i;
    pi = &a[1];
    qi = &a[3];
    printf(" pi %8p   qi %8p   pi+2 %8p   *(pi+2) %d   qi-pi %d\n",
            pi, qi, pi+2, *(pi+2), qi-pi);
    pi = &a[0];
    qi = &a[AANTAL-1];
    for (   ; pi<=qi; pi++)
    {
        i = pi - a;
        printf("      a[%1d] = %4d %4d  (%8p)\n", i, *pi, a[i], pi);
    }
}
```

De uitvoer van dit programma:

```
pi 7f7f112c   qi 7f7f1134   pi+2 7f7f1134   *(pi+2) 9   qi-pi 2
     a[0] =    0    0  (7f7f1128)
     a[1] =    1    1  (7f7f112c)
     a[2] =    4    4  (7f7f1130)
     a[3] =    9    9  (7f7f1134)
     a[4] =   16   16  (7f7f1138)
```

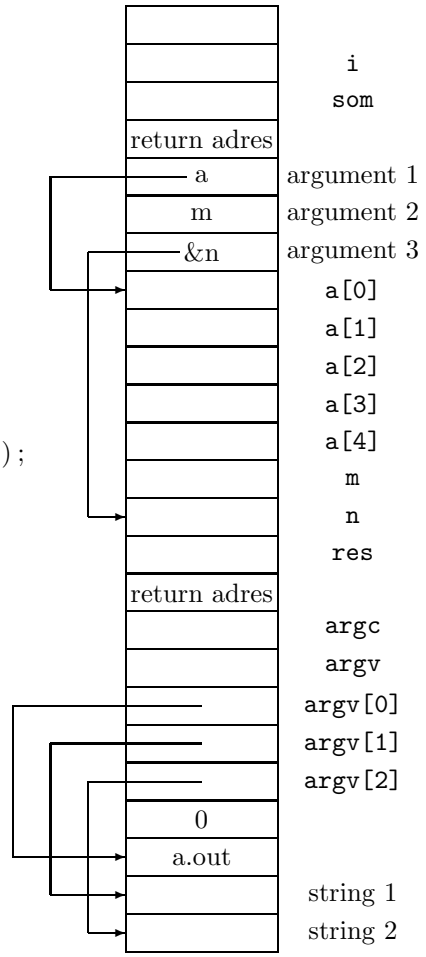
Een element in een array `a` kan aangesproken worden met `a[i]` (de array voorstelling) en met `*(a+i)` (de pointer voorstelling).

Wanneer een array als actueel argument gebruikt wordt in een functie dan wordt in de functiedefinitie de formele parameter getypeerd als `int []`. In deze formele parameter wordt bij een oproep een adres naar een gebied (de array) gestockeerd. Het type kan dus evengoed gespecificeerd worden als `int *`. Nochtans krijgt `int []` de voorkeur omdat hiermee beter aangegeven wordt dat het over een array gaat. Voor de compiler is er echter geen enkel verschil. Omdat bij een array het adres naar het begin van de array doorgegeven wordt, kunnen via de formele parameter de elementen in de array gewijzigd worden in de functie.

5.6 De stack van een programma.

In figuur 5.1 wordt langs de linkerkant een voorbeeld met een hoofdprogramma *main* en een functie *func* gegeven. Langs rechts wordt de inhoud van de stack weergegeven op het moment dat de routine *func* verlaten wordt bij `a.out 2 4`.

```
#include <stdio.h>
#define MAX  5

int main( int  argc, char  *argv[])
{
    int  a[MAX];
    int  m;
    int  n;
    int  res;

    if ( argc == 3 )
    {
        m = atoi(argv[1]);
        n = atoi(argv[2]);
        printf("m=%d n=%d\n", m, n);
        res = func(a, m, &n);
        printf("m=%d n=%d\n", m, n);
        printf("res=%d =%d\n", res, a[0]);
    }
}

int func( int  a[], int  x, int  *py)
{
    int i;
    int som;

    som = x + *py;
    for (i=0; i<=x; i++)
        a[i] = i;
    x += 5;
    *py += 10;
    a[0] = x + *py;
    return som;
}
```



Figuur 5.1: Stack op het einde van routine func

Merk op. Wat gebeurt er bij een oproep `a.out 9 4`?

5.7 Pointers naar structures

Net zoals pointers naar integers of floats kunnen gedeclareerd worden, kan dit ook voor structures:

```
Mens aa;
Mens *pa;

pa = &aa;
```

In dit voorbeeld wordt een variabele van type Mens gedefinieerd. Hiervoor worden 28 bytes voorzien. Daarnaast is er ook pointer `pa`, die 4 bytes in beslag neemt. De toekenning heeft als effect dat de inhoud van de variabele `pa` het adres bevat van de structure `aa` en dus naar die structure *wijst*.
De afzonderlijke velden van de structure kunnen nu aangesproken worden via de pointer variabele:

```
(*pa).leeftijd = 34;
```

Door `*pa` vindt men de structure zelf terug. Hiervan heeft men een bepaald veld nodig. Dit wordt gevonden door de *punt* operator te gebruiken. Omdat `.` een hogere prioriteit heeft dan `*`, moeten haakjes gebruikt worden.
In de meer gebruikelijke manier maakt men gebruik van het feit dat de variabele `pa` *wijst* naar het begin van de structure:

```
pa->leeftijd = 34;
```

Pointers naar structures waren oorspronkelijk vrij belangrijk omwille van efficiëntie. Naast structures met punt operatoren, wordt zoveel mogelijk gebruik gemaakt van pointers. De eerste C-compilers konden voor zo'n programma efficiëntere code genereren, omdat de manier van adresseren beter aansloot bij de beschikbare adresseermechanismen van de processor.

5.8 Pointers naar functies

Omdat functies, net als gegevens, worden opgeslagen in het werkgeheugen, hebben zij een beginadres. Het is in C mogelijk pointervariabelen te declareren waaraan het beginadres van een functie kan toegekend worden. Na deze toekenning kan de functie opgeroepen worden via de oorspronkelijke naam maar ook via de pointervariabele.

```
1  /*
    * funpoin.c : pointers naar functies
3   */
   #include <stdio.h>
5  #include <math.h>

7  float omtrek(float straal)
   {
9      return 2.0 * M_PI * straal;
   }
11
   float opp(float straal)
13 {
       return M_PI * straal * straal;
15 }

17 int main(int argc, char *argv[])
   {
19      float (*pf)(float x);
        float r;
21      int   k;
```

```
23      printf("Geef de straal : ");
        scanf("%f%*c", &r);
25      printf("Omtrek=1  Oppervlakte=2 : ");
        scanf("%d%*c", &k);
27      pf = (k==1 ? omtrek : opp);
        printf("Het resultaat is %f\n", (*pf)(r) );
29  }
```

Noteer dat er een verschil is tussen `float (*pf)(float x)` en `float *pf(float x)`.
In het eerste geval is `pf` een pointervariabele die naar een functie wijst met één formele parameter van type `float` en een terugkeerwaarde van type `float`.
In het tweede geval is `pf` de naam van een functie met één formele parameter van type `float` en een terugkeerwaarde van type **pointer naar** een `float`.

In dit voorbeeld is de pointer naar de functie niet echt nodig:

```
if ( k==1 )
     printf("Het resultaat is %f\n", omtrek(r) );
else
     printf("Het resultaat is %f\n", opp(r) );
```

Pointers naar functies hebben hun nut in grote programma's waar in het begin van het programma een keuze moet gemaakt worden uit een aantal functies en waar de gekozen functie op verschillende plaatsen in het programma moet opgeroepen worden. Door gebruik te maken van een pointer die naar de gekozen functie wijst, kan de functie via deze pointer telkens opgeroepen worden.
Daarnaast maakt een pointer naar een functie het ook mogelijk een functie als actuele parameter door te geven naar een andere functie.

Toepassing. Schrijf een functie die de trapeziumregel voor de berekening van een benaderende waarde voor een bepaalde integraal implementeert. De functie heeft drie argumenten: de functie, de ondergrens en de bovengrens.

```
        /*
2        * funtrap.c : functie trapeziumregel
         */
4       #include <math.h>
        #define  EPS  1.0e-4
6
        double trapezium(double (*pf)(double), double a, double b)
8       {
            int    i, n;
10          double stap, x;
            double integraal, vorig;
12
            n = 1;
14          vorig = 0.0;
            integraal = ( (*pf)(a) + (*pf)(b) )/2.0;
16          while ( fabs((vorig-integraal)/integraal) > EPS )
            {
18              vorig = integraal;
                n *= 2;
20              stap = (b-a)/n;
                x = a;
```

```
22                  integraal = ( (*pf)(a) + (*pf)(b) )/2.0;
                    for (i=1; i<n; i++)
24                  {
                        x += stap;
26                      integraal += (*pf)(x) ;
                    }
28                  integraal *= stap;
                }
30              return integraal;
        }
```

De waarde van de expressie `trapezium(sin, 0.0, 1.0)` wordt berekend door de functie `trapezium` op te roepen met drie argumenten: het beginadres van de functie `sin` en twee getallen. Het resultaat is gelijk aan 0.4597.
Merk op. In plaats van een voorgedefineerde functie (`sin`) als argument te gebruiken, kan ook een zelf-gedefinieerde functie gebruikt worden:

```
        double fun(double x)
        {
            double y;

            if ( x <= 0.0 )
                  return 0.0;
            y = sqrt(x);
            return y * exp(-x*x);
        }
```

De oproep is dan bijvoorbeeld `trapezium(fun, 0.5, 0.9);`.

Voorbeeld. In volgend programma wordt een array van structuren gebruikt, waarbij sommige velden van deze structuur pointers naar functies zijn.

```
1   /*
     * funarp.c : een array van structuren met pointers naar functies
3    */
    #include <stdio.h>
5   #include <math.h>
    #define AANTAL 4
7   #define LEN 12

9   float cirkel_omtrek(float straal)
    {       return 2.0 * M_PI * straal;       }
11
    float cirkel_opp(float straal)
13  {       return M_PI * straal * straal;    }

15  float vierk_omtrek(float lengte)
    {       return 4.0 * lengte;       }
17
    float vierk_opp(float lengte)
19  {       return lengte * lengte;    }

21  struct functies
```

```
{
23      char  afm[LEN];
        float (*omtrek)(float);
25      float (*opper)(float);
} far[AANTAL] = { { "straal", cirkel_omtrek, cirkel_opp},
27                { "lengte", vierk_omtrek, vierk_opp} };

29 int main(int argc, char *argv[])
{
31      float r, z;
        int   k, t;
33
        printf("Geef de figuurtype: 0(cirkel) of 1(vierkant): ");
35      scanf("%d%*c", &t);
        printf("Geef de %s : ", far[t].afm);
37      scanf("%f%*c", &r);
        printf("Omtrek=1  Oppervlakte=2 : ");
39      scanf("%d%*c", &k);
        z = (k==1) ? (*far[t].omtrek)(r) : (*far[t].opper)(r) ;
41      printf("Het resultaat is %f\n", z );
}
```

5.9 Denktaak

```
#include <stdio.h>
int main( int  argc, char  *argv[])
{
    float a[3] = { 0.0, 0.0, 0.0 };
    int  m = 100, n = 101, res;
    res = rout(a, n, &m );
    printf("res = %d  m %d  n %d : %f\n", res, m, n, a[1] );
}
int rout(float b[], int x, int *m )
{
    b[x%2] = 1.0;
    *m = ++x;
    x = func(&b[1], x);
    return x;
}
int func(float c[], int y )
{
    c[y%2] = 2.0;
    y++;
    return y;
}
```

Teken de stack van bovenstaand programma op het moment dat het statement "`return y`" uitgevoerd wordt. Wat is het resultaat van het programma?

6 Stapsgewijze verfijning

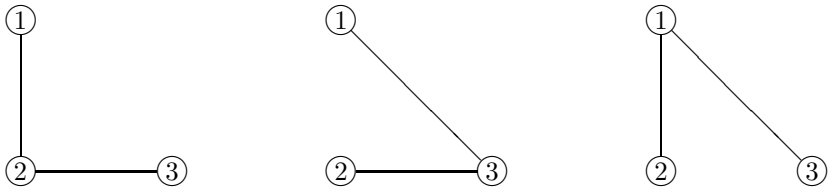
6.1 Techniek

In dit hoofdstuk wordt geïllustreerd hoe een programma kan ontworpen worden:

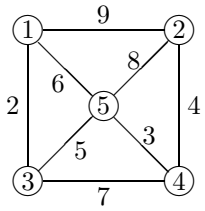
- van boven naar beneden werken;
- alles zo lang mogelijk uitstellen, bijvoorbeeld de keuze van de informatiestructuren;
- eerst de moeilijke stukken verfijnen.

Opgave. Gegeven een graaf met N knooppunten die met elkaar verbonden zijn. Met elk van deze verbindingen is een gewicht, bijvoorbeeld lengte, geassocieerd. Gevraagd: de kortste spanboom (*shortest spanning tree*).

Twee punten kunnen door middel van één verbindingslijn met elkaar verbonden worden. Drie punten kunnen door middel van twee lijnstukken met elkaar verbonden worden; er zijn drie mogelijkheden:



In het algemeen kunnen N punten door middel van $N-1$ punt-tot-punt verbindingen met elkaar verbonden worden. Zo'n verzameling van interconnecties wordt een *boom* genoemd. De punt-tot-punt verbindingen worden *takken* genoemd. Cayley heeft aangetoond dat het aantal mogelijke bomen tussen N punten gelijk is aan N^{N-2}.



Bij een graaf met $N = 5$ knooppunten is het aantal bomen dus gelijk aan 5^3 of 125.
Wanneer voor elke mogelijke tak een lengte gegeven is, kan de lengte van de boom berekend worden als de som van de lengtes van de takken van de boom. Om de kortste boom te vinden, kan men alle bomen tussen de N punten genereren, daarvan telkens de lengte berekenen en tot slot de kortste kiezen. Omdat er tamelijk veel mogelijke bomen zijn, is dit geen efficiënte oplossing.

Gelukkig is er een stelling:
Gegeven een deelboom van de kortste spanboom. De kortste tak die kan gevonden worden tussen één van de punten van deze deelboom en één van de punten niet behorend tot deze deelboom, is een element van de kortste spanboom tussen de N punten.
Deze stelling verdeelt de knooppunten en takken in:

rood : de knooppunten van de deelboom van de kortste spanboom en ook de takken van deze deelboom;

blauw : de andere knoopunten;

violet : de verbindingen tussen een rood en een blauw knooppunt.

Een rode deelboom van de kortste spanboom kan dus uitgebreid wordt met één knoopunt en één tak die naar dat knooppunt leidt: namelijk de kortste violette tak en het bijhorende blauwe knooppunt. Deze kunnen dan rood gekleurd worden. Dus, wannneer we een rode deelboom kunnen vinden om mee te starten, kunnen we deze laten groeien met telkens één tak. Zo'n rode deelboom kan gemakkelijk gevonden worden: een deelboom bestaande uit één knooppunt (om het even welk) en geen takken. Deze deelboom kan in $N-1$ stappen uitgroeien tot de kortste spanboom; tijdens elke stap wordt één nieuwe rode tak en één rood knooppunt toegevoegd.

```
kleur 1 punt rood en alle andere blauw;
while ( aantal rode punten < N )
{
        kies kortste violette tak;
        kleur die tak en haar blauwe eindpunt rood;
}
```

In dit *ruw* algoritme is de moeilijkste stap: `kies kortste violette tak`, omdat het aantal violette takken vrij groot kan zijn: $k \times (N - k)$ met k het aantal rode punten. Deze operatie moet $N - 1$ maal uitgevoerd worden en de opeenvolgende verzamelingen violette takken zijn sterk gerelateerd: het zijn takken tussen rode en blauwe punten en telkens verandert er slechts één punt van kleur.
Misschien is het mogelijk de verzameling takken waarvan telkens de kortste moet gekozen worden te beperken: we zoeken dus naar een nuttige deelverzameling van violette takken. We weten nog niet of zo'n deelverzameling wel bestaat, maar laat ons veronderstellen dat ze kan gevonden worden en we noemen ze "ultraviolet". Deze deelverzameling is alleen nuttig wanneer ze gemakkelijk kan geconstrueerd worden door bijvoorbeeld gebruik te maken van de verzameling ultraviolette takken uit de vorige stap.

```
kleur 1 punt rood en alle andere blauw;
construeer de verzameling ultraviolette takken;
while ( aantal rode punten < N )
{
        kies kortste ultraviolette tak;
        kleur die tak en haar blauwe eindpunt rood;
        pas de verzameling ultraviolette takken aan;
}
```

De verzameling ultraviolette takken moet voldoen aan:

1. de deelverzameling bevat gegarandeerd de kortste violette tak;
2. de verzameling ultraviolette takken is kleiner dan de verzameling violette takken;
3. de operatie `pas de verzameling ultraviolette takken aan` is eenvoudig.

Er zijn twee voor de hand liggende verzamelingen ultraviolette takken:

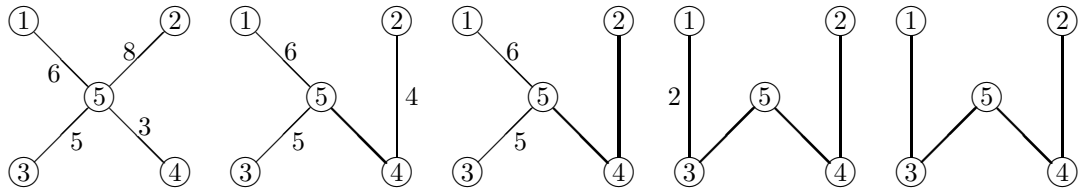
1. Voor elk rood punt zijn er $N - k$ violette takken; kies hiervan de kortste als ultraviolet: dus k ultraviolette takken;
2. Voor elk blauw punt zijn er k violette takken; kies hiervan de kortste als ultraviolet: dus $N - k$ ultraviolette takken;

De verzameling ultraviolette takken moet klein zijn. Maar hiermee kan geen keuze tussen de twee alternatieven gemaakt worden. Bij de eerste keuze groeit het aantal van 1 naar $N-1$; bij de tweede keuze is het net andersom. Maar de operatie `pas de verzameling ultraviolette takken aan` is eenvoudiger met het tweede alternatief. Met deze definitie van ultraviolet is elk blauw punt slechts op één manier met de rode deelboom verbonden: de som van het aantal rode en ultraviolette takken is steeds gelijk aan $N - 1$. Initieel bevat de verzameling $N - 1$ ultraviolette takken: de verbindingen van het ene rode knooppunt naar de overige $N - 1$ blauwe punten.
Beschouw een rode deelboom R. Laat van de corresponderende verzameling ultraviolette takken de kortste tak leidend naar het blauwe punt P en het punt P zelf net rood gekleurd zijn. Het aantal ultraviolette takken is dus met 1 verminderd. Zijn de overblijvende ultraviolette takken de juiste? Ze geven voor elk blauw punt de kortste verbinding naar de rode deelboom R. Maar nu moeten ze de kortste verbinding geven met de nieuwe rode deelboom $R+P$. Dit kan gecontroleerd worden door middel van een eenvoudige vergelijking voor elk blauw punt B: indien de tak BP

korter is dan de ultraviolette tak die B met R verbindt, dan moet deze ultraviolette tak vervangen worden door de tak BP.
Verfijning van de operatie `pas de verzameling ultraviolette takken aan` :

```
kleur 1 punt rood en alle andere blauw;
construeer de verzameling ultraviolette takken;
while ( aantal rode punten < N )
{
        kies kortste ultraviolette tak, noem haar blauwe eindpunt P
        kleur die tak en punt P rood;
        pas de verzameling ultraviolette takken aan door voor elk blauwpunt B
                de oude ultraviolette tak te vergelijken met BP;
}
```

De verschillende stappen voor het voorbeeld met $N = 5$ knooppunten:



Nu we weten WAT we met de gegevens gaan doen, kunnen we vastleggen HOE we de gegevens gaan voorstellen.
We hebben *punten* en *takken*; elke tak heeft een *lengte*. Deze moeten we kunnen *rood* en *blauw* kleuren. Daarnaast moeten we takken ook *ultraviolet* kunnen kleuren.
De lengte van de verbindingen wordt bijgehouden in een matrix:

```
afstand[i][j] == afstand[j][i] : de lengte tussen punt i en j
```

Voor de oplossing, een boom van $N-1$ takken, wordt een array van structures gebruikt, met twee velden voor de identificatie van de eindpunten van elke tak:

```
opl[i].van en opl[i].naar : i-e tak tussen twee punten
```

Omdat de som van rode en ultraviolette takken constant is, kunnen ze in dezelfde array van structures gestockeerd worden:

```
indien k het aantal rode knooppunten is
        opl[i].van, opl[i].naar is rood met 1 ≤ i < k
        opl[i].van, opl[i].naar is ultraviolet met k ≤ i < N
```

Een ultraviolette tak leidt *van* een rood punt *naar* een blauw punt. Omwille van de efficiëntie wordt ook een veld `lengte` in de `Tak` structure toegevoegd:

```
opl[i].lengte = afstand[ opl[i].van ][ opl[i].naar ] : lengte van i-e rode of UV tak
```

Het knooppunt dat als eerste rood gekleurd wordt, is knooppunt N.

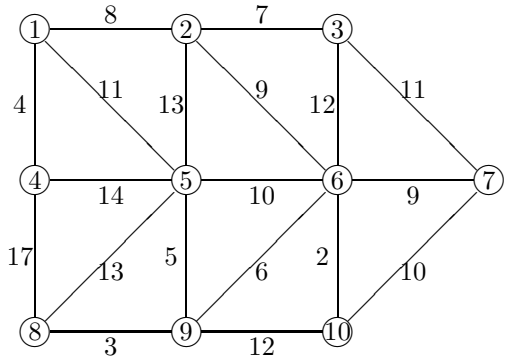
6.2 Programma

```
1   /*
     * mst.c : minimum spanning tree
3    *           symmetrische afstandsmatrix
     */
5   #include <stdio.h>
    #define NMAX 20
7   #define ZEERGROOT 30000

9   typedef struct tak
            {
11              int van;
                int naar;
13              int lengte;
            } Tak;
15
    int leesgraaf(int a[][NMAX] );
17  void drukgraaf( int a[][NMAX], int n);
    int mst( int n, int afstand[][NMAX], Tak opl[]);
19
    int main(int argc, char *argv[])
21  {
        int  afstand[NMAX][NMAX];
23      Tak  opl[NMAX];
        int  n;
25      int  len = 0;

27      n = leesgraaf(afstand);
        drukgraaf(afstand, n);
29      len = mst(n, afstand, opl);
    }
31
    int leesgraaf(int a[][NMAX] )
33  {
        int  i, j, w;
35      int  n = 0;

37      for (i=0; i<NMAX; i++)
            for (j=0; j<NMAX; j++)
39              a[i][j] = ZEERGROOT;
        while ( scanf("%d %d %d%*c", &i, &j, &w) != EOF )
41      {
            if ( (i<NMAX) && (j<NMAX) )
43              a[i][j] = a[j][i] = w;
            if ( n < i )
45              n = i;                          /* bepaling van aantal knooppunten */
            if ( n < j )
47              n = j;
        }
49      if ( n >= NMAX )
        {
51          fprintf(stderr, "De graaf is te groot : n=%d\n", n);
            exit(1);
```

```
53      }
        return n;
55  }

57  int mst( int n, int afstand[][NMAX], Tak opl[])
    {
59      int  i, j, h;
        int  minj, minlen;
61      int  len;
        Tak  temp;
63
        for (i=1; i<n; i++)
65      {                                                   /* ultraviolette takken */
            opl[i].van = n;                          /* punt N is eerste rode punt */
67          opl[i].naar = i;
            opl[i].lengte = afstand[n][i];
69      }
        for (j=1; j<n;     )
71      {
            minj = j; minlen = opl[j].lengte;
73          for (i=j+1; i<n; i++)
            {                                                /* kies kortste UV tak */
75              len = opl[i].lengte;
                if ( len < minlen )
77              {
                    minlen = len;  minj = i;
79              }
            }
81          h = opl[minj].naar;              /* blauw punt h wordt rood gekleurd */
            if ( minj != j )                        /* verschuif kortste UV tak */
83          {                                                    /* kleur rood */
                temp = opl[j];      opl[j] = opl[minj];       opl[minj] = temp;
85          }
            j++;                                          /* er is een rode tak meer */
87          for (i=j; i<n; i++)
            {                                              /* pas verzameling UV aan */
89              len = afstand[h][opl[i].naar];
                if ( len < opl[i].lengte )
91              {
                    opl[i].lengte = len;  opl[i].van = h;
93              }
            }
95      }
        len = 0;
97      for (i=1; i<n; i++)
        {                                                   /* resultaat afdrukken */
99          len += opl[i].lengte;
            printf(" van %3d naar %3d lengte %6d : %6d\n",
101                    opl[i].van, opl[i].naar, opl[i].lengte, len);
        }
103     return len;
    }
105
    void drukgraaf( int a[][NMAX], int n)
```

```
107  {
         int  i, j;
109
         printf("       ");
111      for (j=1; j<=n; j++)
             printf("%5d", j );
113      printf("\n");
         for (i=1; i<=n; i++)
115      {
             printf("%4d : ", i );
117          for (j=1; j<=n; j++)
             {
119              if ( a[i][j] == ZEERGROOT )
                     printf("     ");
121              else
                     printf("%5d", a[i][j] );
123          }
             printf("\n");
125      }
     }
```

Een voorbeeld: een graaf met $N = 10$ knooppunten:



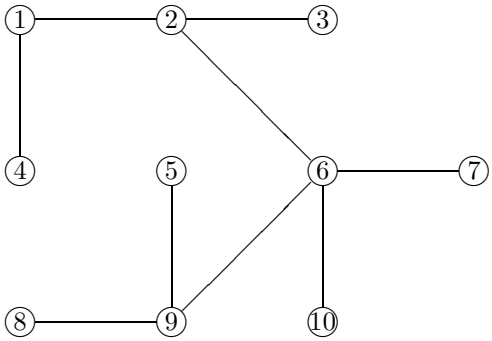
Merk op dat in deze graaf niet alle mogelijke verbindingen aanwezig zijn. In de afstandsmatrix wordt voor zo'n verbinding een heel groot getal ingevuld. Daardoor zal het algoritme zo'n verbinding nooit als ultraviolet beschouwen.

```
        1    2    3    4    5    6    7    8    9   10
 1 :         8         4   11
 2 :    8         7        13    9
 3 :         7                   12   11
 4 :    4                   14              17
 5 :   11   13         14         10         13    5
 6 :         9   12         10         9         6    2
 7 :             11                   9              10
 8 :                   17   13                   3
 9 :                        5    6         3         12
10 :                             2   10         12
```

Na de eerste 4 rode takken (bovenaan) zijn er nog 5 ultraviolette takken (onderaan):

10	6	2
6	9	6
9	8	3
9	5	5
5	4	14
5	1	11
6	7	9
6	3	12
6	2	9

De oplossing met een lengte gelijk aan 53:



6.3 Denktaak

Gegeven: een ontwerp van de `alloc` functie in pseudo-code:

i is de index van het eerste vrije blok;
zolang er nog vrije blokken zijn;
    l is het aantal vrije plaatsen in het blok op index i
    indien $l >=$ gevraagd aantal
        pas index in vorig vrij blok aan;
        pas beschikbaar aantal in dit vrij blok aan;
        geef terug i;
    anders
        pas i aan (index van het volgende vrije blok);
geef terug -1 (er is geen vrij blok met voldoende plaatsen);

Gevraagd: een gelijkaardig ontwerp van de `dealloc` functie.

7 Recursie

7.1 Definitie

Een object wordt *recursief* genoemd wanneer het partieel bestaat uit of partieel gedefinieerd is in termen van zichzelf. Recursie wordt gebruikt bij wiskundige definities, bijvoorbeeld:

1. Natuurlijke getallen:
   - 1 is een natuurlijk getal;
   - de opvolger van een natuurlijk getal is een natuurlijk getal.
2. de faculteitsfunctie $n!$:
   - $0! = 1$;
   - indien $n > 0$ dan is $n! = n \times (n-1)!$.
3. De Ackermann Functie A voor alle niet-negatieve gehele getallen m en n:

$$\begin{array}{lll} A(0,n) & = n+1 & \\ A(m,0) & = A(m-1,1) & (m>0) \\ A(m,n) & = A(m-1,A(m,n-1)) & (m,n>0) \end{array}$$

Recursie biedt de mogelijkheid een oneindige verzameling van objecten te definiëren met een eindig statement. Analoog kan een oneindig aantal berekeningen beschreven worden met een eindig recursief programma dat zelf geen expliciete herhaling bevat.
Om een functioneel programma te hebben moet de recursie natuurlijk wel ergens stoppen. Een belangrijk probleem bij het ontwerp van een recursief algoritme is deze stopconditie.

Voorbeeld. Bereken $S = \sum_{i=m}^{n} i^2$ met $m \leq n$.
Iteratief:

```
int somkwad(int m, int n)
{
    int i;
    int som = 0;

    for (i=m; i<=n; i++)
        som += i*i;
    return som;
}
```

De som van de kwadraten van de getallen tussen m en n kan als volgt gedefinieerd worden:

$$\sum_{i=m}^{n} i^2 \quad = \quad \begin{cases} m^2 & \text{indien} \quad m = n \\ m^2 + \sum_{i=m+1}^{n} i^2 & \text{indien} \quad m < n \end{cases}$$

In woorden:

> Om de som van de kwadraten van de getallen tussen m en n te berekenen
> Indien er meer dan één getal zit tussen m en n
> wordt de oplossing berekend als de som van het kwadraat van m
> en de som van de kwadraten van de getallen tussen $m+1$ en n
> anders is de oplossing het kwadraat van m

In feite wordt het probleem opgedeeld (*decompositie*):

- identificeer een deelprobleem dat eenvoudig op te lossen is;
- de rest is het originele probleem, maar met een kleinere grootte.

Zet deze techniek verder tot het restprobleem herleid is tot een eenvoudig op te lossen probleem.

```
int somkwad(int m, int n)
{
    if ( m < n )
        return m*m + somkwad(m+1,n);
    else
        return m*m;
}
```

Men kan ook vanaf n beginnen:

$$\sum_{i=m}^{n} i^2 \quad = \quad \begin{cases} n^2 & \text{indien} \quad m = n \\ n^2 + \sum_{i=m}^{n-1} i^2 & \text{indien} \quad m < n \end{cases}$$

```
int somkwad(int m, int n)
{
    if ( m < n )
        return somkwad(m,n-1) + n*n;
    else
        return n*n;
}
```

Er is geen reden om in het restprobleem slechts 1 getal buiten beschouwing te laten:

$$\sum_{i=m}^{n} i^2 \quad = \quad \begin{cases} m^2 & \text{indien} \quad m = n \\ \sum_{i=m}^{\mu} i^2 + \sum_{i=\mu+1}^{n} i^2 \quad \text{met} \quad \mu = \frac{m+n}{2} & \text{indien} \quad m < n \end{cases}$$

```
int somkwad(int m, int n)
{
    int midden;

    if ( m == n )
        return m*m;
    else
    {
        midden = (m+n)/2;
        return somkwad(m,midden) + somkwad(midden+1,n);
    }
}
```

De verschillende functieoproepen:

```
somkwad(5,10) = somkwad(5,7) + somkwad(8,10)
              = (somkwad(5,6) + somkwad(7,7)) +
                (somkwad(8,9) + somkwad(10,10))
              = ((somkwad(5,5) + somkwad(6,6)) + somkwad(7,7)) +
                ((somkwad(8,8) + somkwad(9,9)) + somkwad(10,10))
              = ((25 + 36) + 49) + ((64 + 81) + 100)
              = (61 + 49) + (145 + 100)
              = 110 + 245
              = 355
```

Het resultaat is dat we voor een eenvoudig probleem een complex algoritme gerealiseerd hebben. Nochtans is deze techniek bijzonder effectief bij moeilijkere problemen zoals bijvoorbeeld sorteren en beeldcompressie.

7.2 Staartrecursie

De faculteitsfunctie:

$$n! = \begin{cases} 1 & \text{indien} \quad n = 1 \\ n \times (n-1)! & \text{indien} \quad n > 1 \end{cases}$$

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

Wanneer de recursieve oproep van de functie op het einde van de functie staat, spreekt men van *staartrecursie*. Zo'n recursief algoritme is eenvoudig om te vormen naar een iteratieve functie:

```
int faculteit(int n)
{
     int i;
     int f = 1;

     for (i=2; i<=n; i++)
          f *= i;
     return f;
}
```

7.3 De trap

Gegeven een trap met N treden. Op hoeveel verschillende manieren kan men deze trap oplopen wannneer men 1 of 2 treden tegelijk kan doen en ook alle mogelijke combinaties hiervan.
Beschrijving van de oplossing:

1. Onderaan neem je 1 of 2 treden.
2. Wanneer je 1 trede neemt, blijft er een trap van $(N-1)$ treden over.
3. Wanneer je 2 treden neemt, blijft er een trap van $(N-2)$ treden over.

Bij een trap met 1 trede, is het aantal mogelijkheden gelijk aan 1. Bij een trap met 2 treden, is het aantal mogelijkheden gelijk aan 2 (de twee treden apart of ineens de twee treden). Bij een trap met meer dan twee treden, is het aantal mogelijkheden gelijk aan de som van het aantal mogelijkheden om een trap met $(N-1)$ treden op te lopen en het aantal mogelijkheden om een trap van $(N-2)$ treden op te lopen.

```
/*
 * trap1.c : aantal mogelijkheden : eenvoudig
 */
#include <stdio.h>

int main(int argc, char *argv[])
{
    int     n;
    int     res;

    printf("Aantal treden: ");          scanf("%d%*c", &n);
    res = treden(n);
    printf("Trap met %d treden: %d mogelijkheden\n", n, res);
}

int treden(int n)
{
    if ( n == 1 )
        return 1;
    if ( n == 2 )
        return 2;
    return treden(n-1) + treden(n-2);
}
```

Het aantal manieren bij een trap met N treden is gelijk aan het N-e Fibonnacci getal. Bij groter wordende N gaat voorgaand programma enorm veel rekentijd vragen. De reden hiervoor is dat heel veel combinaties herhaalde malen opnieuw uitgerekend worden.

N	mogelijkheden	rekentijd
10	89	
20	10946	0.1
25	121393	0.5
29	832040	3.4
30	1346269	5.5
31	2178309	8.8

Dit kan verholpen worden door tussenresultaten te stockeren in een array:

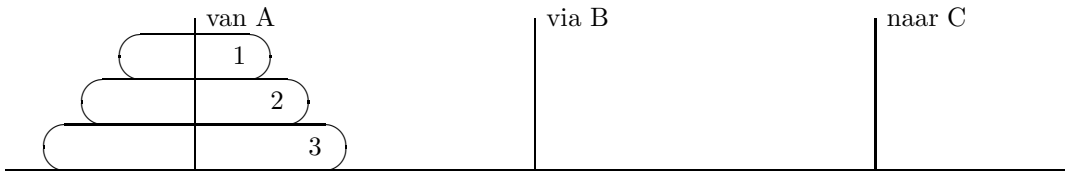
```
/*
 * trap2.c : aantal mogelijkheden : met geheugen
 */
#include <stdio.h>
#define MAXAANT     50

int main(int argc, char *argv[])
{
    int     n;
    int     res;

```

```
        printf("Aantal treden: ");          scanf("%d%*c", &n);
13      if ( n >= MAXAANT )
        {
15          fprintf(stderr, "Aantal treden %d teveel!\n", n);
            exit(1);
17      }
        res = treden(n);
19      printf("Trap met %d treden: %d mogelijkheden\n", n, res);
    }
21
    int treden(int n)
23  {
        static int geheugen[MAXAANT] = { 0, 1, 2 };
25      int     t;

27      if ( geheugen[n] )
            return geheugen[n];
29      t = treden(n-1) + treden(n-2);
        geheugen[n] = t;
31      return t;
    }
```

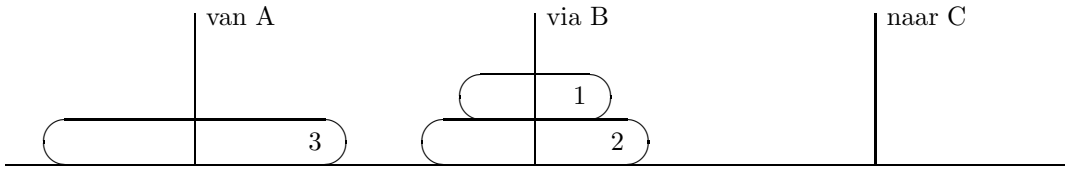
Met deze versie is de rekentijd bij een trap met 30 treden slechts 0.1 seconden.

7.4 Klassieker: de torens van Hanoi

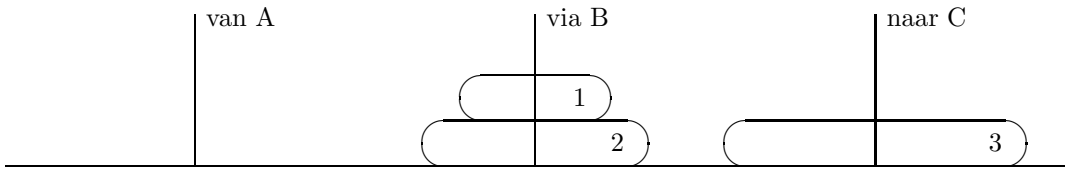
Gegeven zijn 3 palen (torens) die we voor de eenvoud 'A', 'B' en 'C' noemen. Daarnaast hebben we ook ook N schijven ($N \geq 1$) met afnemende diameters en met in het centrum een gat zodat ze over een paal kunnen geschoven worden. Bij aanvang liggen deze schijven rond paal 'A', de schijf met de grootste diameter onderaan, de rest is in afnemende diameter hierop gestapeld. De bedoeling is deze schijven te verplaatsen naar paal 'C'. Hierbij mag telkens slechts 1 schijf verplaatst worden en op geen enkel moment mag een grotere schijf op een kleinere schijf gelegd worden. De paal 'B' kan als intermediaire stockeringsruimte gestockeerd worden.
(Volgens de legende gaat het om 64 gouden schijven die door monniken verplaatst moeten worden. Deze schijven zijn zo zwaar dat een grotere en dus zwaardere schijf nooit boven op een kleinere schijf mag gelegd worden. Wanneer deze 64 schijven verplaatst zijn, zal de volgende *Maha Pralaya* beginnen.)



We moeten eerst de twee bovenste schijven van de *van* toren op de *via* toren kunnen plaatsen.



Dan kunnen we de derde schijf naar de *naar* toren brengen.



Wanneer dit gebeurd is, moeten we de schijven van de *via* toren nog op de *naar* toren brengen. Het probleem van het verplaatsen van n schijven herleidt zich dus tot het verplaatsen van $n-1$ schijven. Dit moet wel tweemaal gebeuren: eerst van de *van* toren naar de *via* toren en dan van de *via* toren naar de *naar* toren. Tussen deze twee operaties kunnen we de onderste schijf van de *van* toren naar de *naar* toren brengen.

```
/*
 * hanor.c : de torens van hanoi : recursief
 */
#include <stdio.h>

void toren(int n, char van, char via, char naar);
void beweeg(int n, char van, char naar);

int main(int argc, char *argv[])
{
    int     n = 1;

    if ( argc == 2 )
        n = atoi(argv[1]);
    toren(n, 'A', 'B', 'C');
}

void toren(int n, char van, char via, char naar)
{
    if ( n >= 1 )
    {
        toren(n-1, van, naar, via);
        beweeg(n, van, naar);
        toren(n-1, via, van, naar);
    }
}

void beweeg(int n, char van, char naar)
{
    printf("Beweeg %2d van .%c. naar .%c.\n", n, van, naar);
}
```

Het resultaat van dit programma bij $N = 3$:

```
Beweeg  1 van .A. naar .C.
Beweeg  2 van .A. naar .B.
Beweeg  1 van .C. naar .B.
Beweeg  3 van .A. naar .C.
Beweeg  1 van .B. naar .A.
```

```
Beweeg  2 van .B. naar .C.
Beweeg  1 van .A. naar .C.
```

Merk op. Het is niet erg realistisch dit programma uit te voeren voor grote N omdat het totaal aantal benodigde bewegingen gelijk is aan $2^N - 1$.
De iteratieve oplossing is minder stijlvol. Het bijhouden van welke bewegingen nog moeten gebeuren, moet nu in het programma neergeschreven worden. Elke beweging van $N > 1$ schijven wordt vervangen door drie "kleinere" bewegingen.

```
1   /*
     * hanoi.c : de torens van hanoi : iteratief
3    */
    #include <stdio.h>
5   #define MAXAANT 10

7   void toren(int n, char tvan, char tvia, char tnaar);
    void beweeg(char van, char naar);
9
    int main(int argc, char *argv[])
11  {
        int     n = 1;
13
        if ( argc == 2 )
15          n = atoi(argv[1]);
        toren(n, 'A', 'B', 'C');
17  }

19  void toren(int n, char tvan, char tvia, char tnaar)
    {
21      int     k;
        int     a[MAXAANT];
23      char    van[MAXAANT];
        char    via[MAXAANT];
25      char    naar[MAXAANT];

27      a[1] = n;
        van[1] = tvan;    via[1] = tvia;    naar[1] = tnaar;
29      k = 1;
        do
31      {
            if ( a[k] == 1 )
33          {
                beweeg(van[k], naar[k]);
35              k--;
            }
37          else
            {
39              a[k+2] = a[k] - 1;
                van[k+2] = van[k];    via[k+2] = naar[k];    naar[k+2] = via[k];
41              a[k+1] = 1;
                van[k+1] = van[k];    naar[k+1] = naar[k];
43              a[k] = a[k+2];
                van[k] = naar[k+2];    via[k] = van[k+2];  naar[k] = via[k+2];
45              k += 2;
```

```
            }
47      } while ( k > 0 );
    }
49
    void beweeg(char van, char naar)
51  {
        printf("Beweeg schijf van .%c. naar .%c.\n", van, naar);
53  }
```

Bij $N = 3$ schijven:

k	a	van	via	naar
1	3	A	B	C
1	2	B	A	C
2	1	A		C
3	2	A	C	B
1	2	B	A	C
2	1	A		C
3	1	C		B
4	1	A		B
5	1	A		C
1	1	A		C
2	1	B		C
3	1	B		A
0				

7.5 Denktaak

```
#include <stdio.h>
#define LEN 7
void doen(int a[], int m, int n)
{
    int temp;
    if ( m < n )
    {
        temp = a[m];
        a[m] = a[n];
        a[n] = temp;
        doen(a, m+1, n-1);
    }
}
void functie(int a[], int n)
{
    doen(a, 0, n-1);
}
```

```
int main(int argc, char *argv[])
{
    int a[LEN] = { 3,7,5,1,4,2,6};
    int i;

    functie(a, LEN);
    for (i=0; i<LEN; i++)
        printf("%3d", a[i]);
    printf("\n");
}
```

Wat doet bovenstaande functie? Geef aan wat de verschillende oproepen van doen zijn.

8 Dynamisch geheugen beheer

8.1 De functies `malloc` en `free`

Normaal wordt plaats in het werkgeheugen gereserveerd tijdens de compilatie aan de hand van de declaraties van de variabelen. Deze geheugenreservering is *statisch*: in het bronbestand van het programma wordt reeds vastgelegd hoe groot bijvoorbeeld een array zal zijn. Hieraan is tijdens de uitvoering van het programma niets meer te veranderen.
In plaats daarvan kan geheugen meer *dynamisch* gereserveerd worden. Hiervoor zijn functies ter beschikking om geheugen tijdens run-time aan te vragen en eventueel later weer vrij te geven. Om deze functies te gebruiken moet een include van het `malloc.h` headerbestand gebeuren. Hierin worden de twee functies gedeclareerd:

```
void *malloc( size_t grootte );
void free( void *p );
```

De eerste functie wordt gebruik voor *memory allocatie*. Het argument is de grootte van de ruimte die men wenst te alloceren, uitgedrukt in aantal bytes. Indien de uitvoering van de functie succesvol verloopt, wordt het adres teruggegeven van het begin van de toegekende ruimte. Deze ruimte bestaat uit een opeenvolging van een aantal bytes gelijk aan `grootte`. Indien er iets misgaat, wordt een `NULL` waarde teruggeven.
De returnwaarde van `malloc` is een adres van een geheugenruimte dat zelf geen naam heeft. Wanneer deze returnwaarde toegewezen wordt aan een pointer variabele, kan de *anonieme variabele* via die pointer toch gebruikt worden.
De tweede functie heeft als argument zo'n pointer variabele die via `malloc` een waarde toegewezen gekregen heeft. Het effect van de functie is dat de ruimte waarnaar de pointer wijst, vrij gegeven wordt. Deze ruimte kan dan achteraf door een oproep van `malloc` terug gealloceerd worden (om voor iets anders gebruikt te worden). Deze functie heeft geen return-waarde en kan dus als een *procedure* bestempeld worden.
Afhankelijk van de toepassing zal de gealloceerde ruimte data bevatten van een bepaald type. Omdat tijdens de uitvoering van `malloc` geen informatie ter beschikking is omtrent het type van de anonieme variabele, wordt door `malloc` een pointer teruggegeven die naar data van een willekeurig type wijst. Dit wordt aangegeven door het type `void *`. Dit type moet omgezet worden naar het juiste type, door middel van de *casting* operator.

```
/*
 *      dynvar.c : dynamische geheugen allocatie
 */
#include <stdio.h>
#include <stdlib.h>
#include <malloc.h>

int main(int argc, char *argv[])
{
    int     *p;
    double  *q;

    p = (int *) malloc(sizeof(int));
    *p = 5;
    printf("De toegekende waarde is %d\n", *p);
    free(p);

    q = (double *) malloc(sizeof(double));
    *q = 5.25;
    printf("De toegekende waarde is %f\n", *q);
```

```
    free(q);
}
```

In plaats van string pointers te initialiseren met adressen van vooraf gedefinieerde karakters en arrays van karakters, kan ook de functie `malloc` gebruikt worden om de geheugenplaatsen voor de string pas tijdens run-time te voorzien.

```
char   *cp;
cp = (char *)malloc(32);
strcpy(cp, "dit is een nieuwe string");
```

Merk op dat er een verschil is met een gewone array van karakters: **char** w[32];.

Het gebruik van pointers in programma's leidt vrij dikwijls tot fouten. Twee bekende fenomenen:

dangling pointer : een variabele die een adres bevat naar een reeds gedeallocceerd object:

```
p = malloc(20);
free(p);
```

lost object : een gealloceerd dynamische object dat niet langer toegankelijk is vanuit het programma, maar toch nog bruikbare data bevat:

```
p = malloc(40);
p = malloc(4);
```

8.2 Dynamische arrays

```
/*
 *     dynar.c : array met dynamische lengte
 */
#include <stdio.h>
#include <stdlib.h>
#include <malloc.h>
#define VIJF  5

int main(int argc, char *argv[])
{
    double *p;
    int i;

    p = (double *) malloc( VIJF * sizeof(double) );
    memset(p, 0, VIJF*sizeof(double) );
    *p = 1.11;
    *(p+1) = 2.22;
    *(p+2) = 3.33;
    *(p+3) = 4.44;
    *(p+4) = 5.55;
    printf("De toegekende waarden zijn: ");
    for (i=0; i<VIJF; i++)
        printf("%5.2f ",p[i]);
    printf("\n");
}
```

Om een element in de dynamisch gealloceerde array aan te spreken, kan men zowel de pointer notatie (`*(p+i)`) als de array notatie (`p[i]`) gebruiken.

```
1  /*
    *    dynmat.c : matrix met dynamisch aantal rijen en kolommen
3   */
   #include <stdio.h>
5  #include <stdlib.h>
   #include <malloc.h>
7
   int main(int argc, char *argv[])
9  {
       float  **q;
11     int    n, m;
       int    i, j;
13
       printf("Geef aantal rijen en aantal kolommen : ");
15     scanf("%d%d%*c", &n, &m);
       /* reservering geheugen */
17     q = (float **) malloc( n*sizeof(float *) );
       for (i=0; i<n; i++)
19         q[i] = (float *) malloc( m*sizeof(float) );
       printf("Geef de elementen in  rij per rij \n");
21     for (i=0; i<n; i++)
       {
23         for (j=0; j<m; j++)
               scanf("%f", &q[i][j] );
25         scanf("%*c");
       }
27     /* berekeningen ... */
       for (i=0; i<n; i++)
29     {
      printf("%8p : ", q[i] );
31         for (j=0; j<m; j++)
               printf("%8.3f ", *(*(q+i)+j) );
33         printf("\n");
       }
35     /* vrijgeven geheugen */
       for (i=0; i<n; i++)
37         free(q[i]);
       free(q);
39 }
```

Bij deze dynamische 2-dimensionale array is `q` een pointer naar een pointer; vandaar de twee `*` in de declaratie.

8.3 Dynamische structures

Na definitie (lijn 20) bevat de variabele `p` geen zinnige informatie. Er is alleen plaats voorzien om het resultaat van een `malloc` op te slaan. Wanneer deze toekenning gebeurd is, wijst `p` naar een ruimte van 32 bytes (`sizeof(Koppel)`). Door middel van de casting operator wordt deze ruimte geïnterpreteerd als iets van type `Koppel`. Bij de tweede allocatie wordt ineens ruimte voorzien voor $N = 6$ structures en `q` wijst naar het eerste element hiervan.

Men kan `q` ook interpreteren als een array van N elementen waarbij elk element van type `Koppel` is (`for`-lus vanaf lijn 32). Dit kan natuurlijk ook op een pointer-achtige manier geschreven worden, zie `for`-lus vanaf lijn 41. Met behulp van de `r++` expressie wordt telkens naar het volgende element in de array gewezen.

```
1  /*
    * dynstru.c : dynamische structuren
3   */
   #include <stdio.h>
5  #include <stdlib.h>
   #include <string.h>
7  #include <malloc.h>
   #define N 6
9  typedef struct
           {
11              char  naam[14];
                short leeftijd;
13         } Mens;
   typedef struct
15         {
                Mens links;
17              Mens rechts;
           } Koppel;
19 int main(int argc, char *argv[])
   {
21     Koppel *p, *q, *r;
       Mens   *m;
23     int     i;

25     p = (Koppel *)malloc( sizeof(Koppel) );
       p->links.leeftijd = 21;
27     p->rechts.leeftijd = 19;
       printf(" links %d rechts %d\n",
29                 p->links.leeftijd , p->rechts.leeftijd);

31     q = (Koppel *)malloc( N * sizeof(Koppel) );
       memset(q, 0, N*sizeof(Koppel) );
33     for (i=0; i<N; i++)
       {
35          q[i].links.leeftijd = 30+i;
            q[i].rechts.leeftijd = 30-i;
37     }
       for (i=0, r=q; i<N; i++, r++)
39          printf(" links %d rechts %d\n",
                   r->links.leeftijd , r->rechts.leeftijd);
41
       for (i=0, r=q; i<N; i++, r++)
43     {
            (*r).links.leeftijd = 40+i;
45          (*r).rechts.leeftijd = 40-i;
       }
47     for (i=0, r=q; i<N; i++, r++)
            printf(" links %d rechts %d\n",
49                 r->links.leeftijd , r->rechts.leeftijd);
```

```
51      m = (Mens *)q;
        for (i=0; i<2*N; i++, m++)
53          m->leeftijd = 50 + (i%2 ? i/2 : -i/2);
        for (i=0, r=q; i<N; i++, r++)
55          printf(" links %d rechts %d\n",
                    r->links.leeftijd , r->rechts.leeftijd);
57  }
```

In het laatste gedeelte (vanaf lijn 50) wordt de array van N `Koppel` structures geïnterpreteerd als een array van $2N$ `Mens` structures. Hiervoor wordt de `q` pointer van type `Koppel *` via de casting operator omgezet naar de `m` pointer van type `Mens *`.

8.4 Een voorbeeld.

In het programma-deel wordt een *vector* object beschreven: wat de elementen van het object zijn en welke operaties op dit object mogelijk zijn.

```
1   /*
     * vector.h : definities
3    */
    typedef struct vector
5           {
                int lengte;
7               double *parr;
            } Vector;
9   Vector creatie(int lengte, double waarde);
    void   teniet(Vector *pv);
11  void   drukaf(Vector v);
    Vector plus(Vector v1, Vector v2);
13  double inwpro(Vector v1, Vector v2);
```

```
1   /*
     * vector.c : implementatie
3    */
    #include <stdio.h>
5   #include <stdlib.h>
    #include <malloc.h>
7   #include "vector.h"

9   Vector creatie(int lengte, double waarde )
    {
11      Vector vec = { 0, NULL };
        int    i;
13
        if ( lengte > 0 )
15      {
            vec.lengte = lengte;
17          vec.parr = (double *)malloc(lengte*sizeof(double) );
            for (i=0; i<lengte; i++)
19              vec.parr[i] = waarde;
        }
21      return vec;
```

```
}
void teniet(Vector *pv)
{
    if ( pv->lengte == 0 && pv->parr == NULL )
        return;
    else
    {
        free(pv->parr);
        pv->lengte = 0;
        pv->parr = NULL;
    }
    return;
}
void drukaf(Vector v)
{
    int i;

    for (i=0; i<v.lengte; i++)
        printf("%10.3f", v.parr[i] );
    printf("\n");
}
Vector plus(Vector v1, Vector v2)
{
    Vector vec;
    int    i;

    if ( v1.lengte != v2.lengte )
        return creatie(0, 0.0);
    vec = creatie(v1.lengte, 0.0);
    for (i=0; i<vec.lengte; i++)
        vec.parr[i] = v1.parr[i] + v2.parr[i];
    return vec;
}
double inwpro(Vector v1, Vector v2)
{
    double   res = 0.0;
    int      i;

    if ( v1.lengte != v2.lengte )
        return res;
    for (i=0; i<v1.lengte; i++)
        res += v1.parr[i] * v2.parr[i];
    return res;
}
```

Om deze routines te testen kan een testprogramma gemaakt worden:

```
/*
 * testvec.c : testprogramma
 */
#include <stdio.h>
#include <stdlib.h>
#include "vector.h"

```

```
int main(int argc, char *argv[])
{
      Vector a, b, c;

      a = creatie(5, 2.5);          b = creatie(5, 6.9);
      c = plus(a,b);                drukaf(c);
      printf("Inwendig product = %f\n", inwpro(a,c) );
      teniet(&a);        teniet(&b);         teniet(&c);
}
```

8.5 Denktaak

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

#define LEN 16

int main(int argc, char *argv[])
{
   char *cp;
   char car[LEN];

   if ( argc != 3 )
      exit(1);
   cp = (char *)malloc( LEN*sizeof(char) );
   strcpy(cp, argv[1]);
   strcpy(car, argv[2]);
   cp++;          printf("via pointer %s\n", cp);
   car++;         printf("via array %s\n", car);
   fpoin(cp);     printf("\t\t pointer %s\n", cp);
   farray(car);   printf("\t\t array %s\n", car);
}

fpoin(char *tp)
{
   tp++;       printf("\tfun pointer %s\n", tp);
}

farray(char tar[])
{
   tar++;      printf("\tfun array %s\n", tar);
}
```

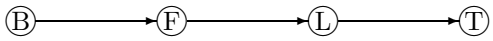
Bespreek aan de hand van bovenstaand programma het verschil tussen een array van characters en een pointer naar een character. Leg uit wat er gebeurt bij starten van `a.out joske flup`. Merk op dat lijn 18 een syntax fout bevat; wat is er verkeerd?

9 Lineaire data structuren

9.1 Lijsten

9.1.1 Definitie

Een *array* bevat een vast aantal data items die aaneensluitend gestockeerd zijn; de elementen zijn bereikbaar via een index. Een *lijst* daarentegen bestaat uit een aantal individuele elementen die met elkaar verbonden (gelinkt) zijn. Deze links geven de volgorde in de lijst aan. Een bepaald element in de lijst kan alleen aangesproken worden vanuit het voorgaande element.



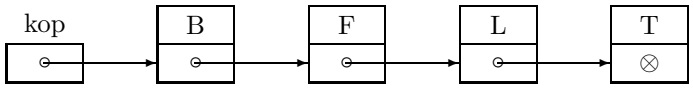
Er zijn twee voordelen:

- de lengte van de lijst kan veranderen tijdens de levensduur: men kan gemakkelijk elementen toevoegen en/of verwijderen;
- de flexibiliteit om de elementen te herschikken in een andere volgorde.

Een knooppunt van een lijst bevat twee delen: de informatie (van type `Info`) en een "link" naar de volgende node.

```
typedef struct snode
        {
            Info          data;
            struct snode *volgend;
        } Node;
```

In bovenstaand voorbeeld is de data van het type `char` (`typedef char Info;`). Het veld `volgend` kan een pointer bevatten naar het volgende element op de lijst. Daarnaast moet ook aangegeven worden hoe het eerste element van de lijst kan bereikt worden en moet ook aangegeven worden dat een bepaald element het laatste element is. Om het eerste element aan te geven, wordt gebruik gemaakt van een extra variabele, `kop` (type is `Node *`). Het `volgend` veld van het laatste element wordt gelijkgesteld aan `(Node *)NULL`.



9.1.2 Doorlopen van een lijst

De lijst sequentieel doorlopen kan met een eenvoudige `while` lus:

```
void doorloop(Node *p)
{
        while ( p != (Node *)NULL )
        {
                printf("%c  ", p->data);
                p = p->volgend;
        }
        printf("\n");
}
```

Een mogelijke oproep is doorloop(kop); .
Een lijst *construeren* kan eenvoudig gebeuren door telkens een element vooraan toe te voegen:

```
Node *voegvooraantoe(Node *kop, Info gegeven)
{
        Node *nieuw;
```

```
        nieuw = (Node *)malloc(sizeof(Node));
        nieuw->data = gegeven;
        nieuw->volgend = kop;
        return nieuw;
}
```

Bovenstaande lijst wordt geconstrueerd door volgende oproepen:

```
Node *kop = (Node *)NULL;
kop = voegvooraantoe(kop, 'T')
kop = voegvooraantoe(kop, 'L')
kop = voegvooraantoe(kop, 'F')
kop = voegvooraantoe(kop, 'B')
```

Het *zoeken* van een element met waarde **gegeven** in de gelinkte lijst vanaf knooppunt **kop**:

Iteratief:

```
Node *zoek(Node *kop, Info gegeven)
{
    Node *p;

    p = kop;
    while ( p != (Node *)NULL )
    {
        if ( p->data == gegeven )
            return(p);
        p = p->volgend;
    }
    return (Node *)NULL;
}
```

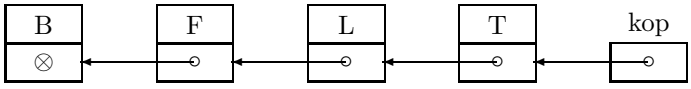
Recursief:

```
Node *rzoek(Node *p, Info gegeven)
{
    if ( p == (Node *)NULL )
        return(p);
    if ( p->data == gegeven )
        return(p);
    return rzoek(p->volgend, gegeven);
}
```

Oproep in beide gevallen:

```
Node *v = (r)zoek(kop, 'T');
```

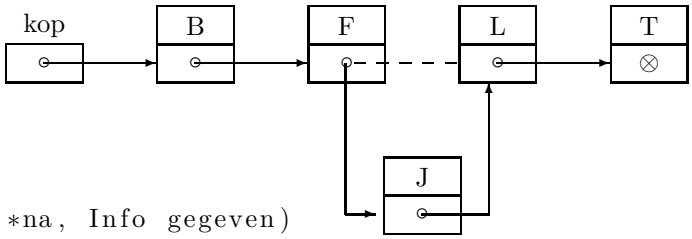
Het *omkeren* van de volgorde in de gelinkte lijst:




```
Node *omkeren( Node *kop )
{
    Node *p, *q, *r;

    p = kop;
    r = (Node *)NULL;
    while ( p != (Node *)NULL )
    {
        q = p;
        p = p->volgend;
        q->volgend = r;
        r = q;
    }
    return r;
}
```

9.1.3 Operaties

Toevoegen. Het toevoegen van een nieuw element na knooppunt **na**:

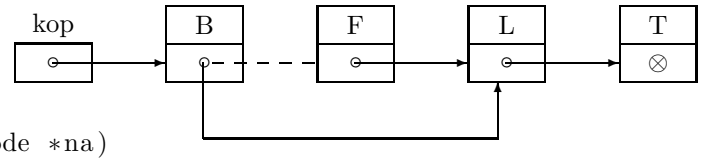



```
Node *voegtoe(Node *na, Info gegeven)
{
    Node *nieuw;

    nieuw = (Node *)malloc(sizeof(Node));
    nieuw->data = gegeven;
    if ( na != (Node *)NULL )
    {
        nieuw->volgend = na->volgend;
        na->volgend = nieuw
    }
    else
        nieuw->volgend = (Node *)NULL;
    return nieuw;
}
```

Mogelijke oproep:

```
if (kop==NULL)
    kop=voegtoe(NULL,g)
else
    na->volgend = voegtoe(na,g);
```

Verwijderen. Het verwijderen van een element na knooppunt **na**:



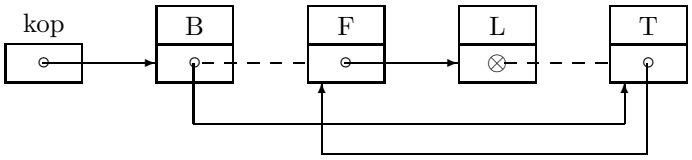

```
Node *verwijder(Node *na)
{
    Node *q = na->volgend;

    if ( q == (Node *)NULL )
    {
        /* er is geen volgend element */
        /* een mogelijkheid is het element zelf verwjderen */
        free(na);
        /* pointer die "na" aanwijst is NIET aangepast in NULL !! */
        return (Node *)NULL;
    }
    else
    {
        na->volgend = q->volgend;
        free(q);
```

```
            return na;
        }
    }
```

Een oproep zou kunnen zijn: `x->volgend = verwijder(x->volgend);`.
Het verwijderen van het laatste element:

```
Node *verwijderachteraan( Node *kop )
{
    Node *vorig, *actueel;

    if ( kop != (Node *)NULL )
    {
        if ( kop->volgend == (Node *)NULL )
        {
            free(kop);
            return (Node *)NULL;
        }
        else
        {
            vorig = kop;
            actueel = kop->volgend;
            while ( actueel->volgend != (Node *)NULL )
            {
                vorig = actueel;
                actueel = actueel->volgend;
            }
            free(actueel);
            vorig->volgend = (Node *)NULL;
            return kop;
        }
    }
    return kop;
}
```

Herschikken. Het herschikken van de lijst: element **twee** komt vlak achter element **een**:




```
Node *herschik( Node *kop, Node *een, Node *twee)
{
    Node *p  = kop;

    if ( p == (Node *)NULL )
        return (Node *)NULL;
    while ( p->volgend != (Node *)NULL )
    {
        if ( p->volgend == twee )
            break;
        p = p->volgend;
    }
```

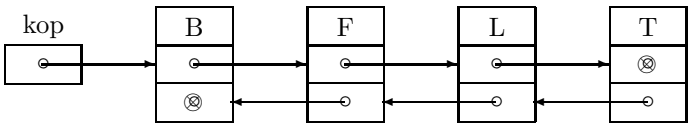
```
    if ( p->volgend != twee )
        return (Node *)NULL;     /* er moet een element voor "twee" zitten */
    p->volgend = twee->volgend;
    twee->volgend = een->volgend;
    een->volgend = twee;
}
```

Oefening. Wat gebeurt er als element `twee` reeds vlak na element `een` zit?

9.2 Dubbel gelinkte lijsten

Met een gelinkte lijst kan gemakkelijk het volgend element gevonden worden. Om het vorige element te vinden moet gans de lijst tot aan het element doorzocht worden. Om dit efficiënter te doen kan gebruik gemaakt worden van een dubbel gelinkte lijst. Naast een verwijzing naar het volgende element op de lijst is ook een verwijzing naar het vorige element aanwezig.

```
typedef struct dub
{
    Info
data;
    struct dub *volgend;
    struct dub *vorig;
} Dnode;
```



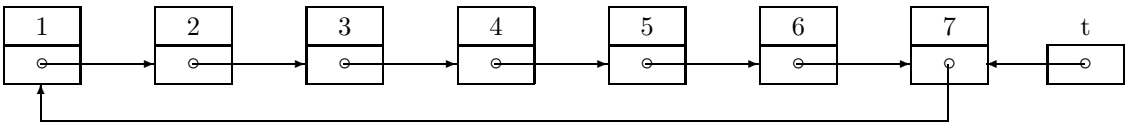
In de herschik functie moet nu niet naar het voorgaande element van **twee** gezocht worden:

```
Dnode *herschik( Dnode *een, Dnode *twee)
{
    twee->vorig->volgend = twee->volgend;
    twee->volgend->vorig = twee->vorig;
    twee->volgend = een->volgend;
    twee->vorig = een;
    een->volgend->vorig = twee;
    een->volgend = twee;
}
```

Merk op. Voorgaande functie is correct wanneer na **een** nog minstens één element aanwezig is en wanneer voor en na **twee** ook minstens één element aanwezig is. Om deze voorwaarden niet te hoeven controleren kunnen we een circulaire lijst gebruiken.

9.3 Circulaire lijsten

Wanneer het laatste element in de lijst wijst naar het eerste element, heeft men een circulaire lijst:



Het probleem van Josephus: Veronderstel dat N mensen besloten hebben om collektief zelfmoord te plegen door in een cirkel te gaan staan en telkens de M-e persoon te elimineren. Wie blijft er over of wat is de volgorde van afvallen?
Bijvoorbeeld bij $N = 7$ personen met $M = 4$ is de volgorde: 4 1 6 5 7 3 2.

```c
/*
 * telaf.c : het probleem van Josephus : met een lijst
 */
#include <stdio.h>
#include <malloc.h>
#define MAXP  25

int aftellen(int n, int m);

int main(int argc, char *argv[])
{
    int     n, m, res;

    printf("Aantal mensen: ");          scanf("%d%*c", &n);
    printf("Hoeveel overslaan : ");     scanf("%d%*c", &m);
    res = aftellen(n, m);
    printf("De laatste van %d met %d overslaan : %d\n", n, m, res);
}

typedef  struct node
         {
             int          nr;
             struct node  *next;
         } Elem;
void drukaf(Elem *x);

int aftellen(int n, int m)
{
    Elem    *v;
    Elem    *t;   /* wijst naar persoon die revolver net heeft doorgegeven */
    int     i;

    v = t = (Elem *)malloc( sizeof(Elem) );
    v->nr = 1;
    for(i=2; i<=n; i++)
    {
        v->next = (Elem *)malloc( sizeof(Elem) );
        v = v->next;
        v->nr = i;
    }
    v->next = t;
    t = v;
    drukaf(t);
    while ( t != t->next )
    {
        for (i=1; i<m; i++)
            t = t->next;
        v = t->next;
        t->next = t->next->next;
        free(v);
        drukaf(t);
    }
    return t->nr;
```

```
}

void drukaf(Elem *x)
{
    Elem  *t;

    t = x;
    do
    {
        printf("%3d", t->nr);
        t = t->next;
    }
    while (t != x);
    printf("\n");
}
```

9.4 Stacks

In veel toepassingen is het niet nodig om op een willekeurige plaats in de lijst elementen te kunnen toevoegen of verwijderen. Ook het herschikken van de elementen in de lijst is niet altijd vereist. Een *stack* is een data-structuur waarop slechts twee mogelijke operaties gedefinieerd zijn:

push : het toevoegen van een element in het begin van de lijst;

pop : het verwijderen van het eerste element van de lijst.

Een klassiek voorbeeld is het berekenen van de waarde van een eenvoudige rekenkundige uitdrukking. De intermediaire resultaten kunnen gemakkelijk op een stack bewaard worden. De waarde van $5 * (((9 + 8) * (4 * 6)) + 7)$ kan berekend worden met

```
push(5);
push(9);
push(8);
push( pop() + pop() );
push(4);
push(6);
push( pop() * pop() );
push( pop() * pop() );
push(7);
push( pop() + pop() );
push( pop() * pop() );
printf( "%d\n", pop() );
```

Wanneer de maximale lengte van de stack op voorhand gekend is, kan bij de implementatie gebruik gemaakt worden van een *array*.

```
#define MAX 100
static int stack[MAX];
static int top;

void stackinit(void)
{
    top = 0;      /* top wijst naar eerstvolgende vrije element */
}
```

```
10      void push(int w)
        {
12          stack[top++] = w;
        }
14
        int pop(void)
16      {
            return stack[--top];
18      }

20      int isstackleeg(void)
        {
22          return !top;
        }
24
        int isstackvol(void)
26      {
            return top==MAX;
28      }
```

De variabele `top` is de index van de eerste vrije plaats, de plaats waar een volgend element kan gepusht worden. Merk op dat in deze eenvoudige functies NIET nagegaan wordt of de array volzet is (bij een `push`) of de array leeg is (bij een `pop`).

Oefening. Implementeer deze stack functies met behulp van een gelinkte lijst.

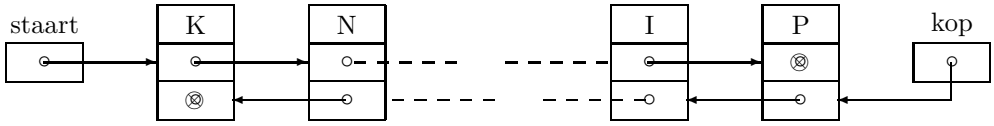
9.5 Queues

Bij een queue zijn er ook maar twee basisoperaties mogelijk: het *toevoegen* van een element aan de ene kant van de lijst, en het *wegnemen* van een element aan de andere kant.
Een implementatie met behulp van een *array*:

```
        #define MAX 100
2       static int queue[MAX];
        static int kop;
4       static int staart;

6       void queueinit(void)
        {
8           staart = kop = 0;
        }
10
        void put(int w)
12      {
            queue[staart++] = w;        /* staart wijst naar lege plaats */
14          if ( staart == MAX )        /* waar eerstvolgende element    */
                staart = 0;             /* moet toegevoegd worden        */
16      }

18      int get(void)
        {
20          int w  = queue[kop++];      /* kop wijst naar eerste element */
            if ( kop == MAX )           /* dat kan weggenomen worden     */
22              kop = 0;
```

```
        return w;
24  }

26  int isqueueleeg(void)
    {
28      return kop==staart;
    }
```

Oefening. Implementeer deze queue functies met behulp van een dubbel gelinkte lijst.



9.6 Denktaak

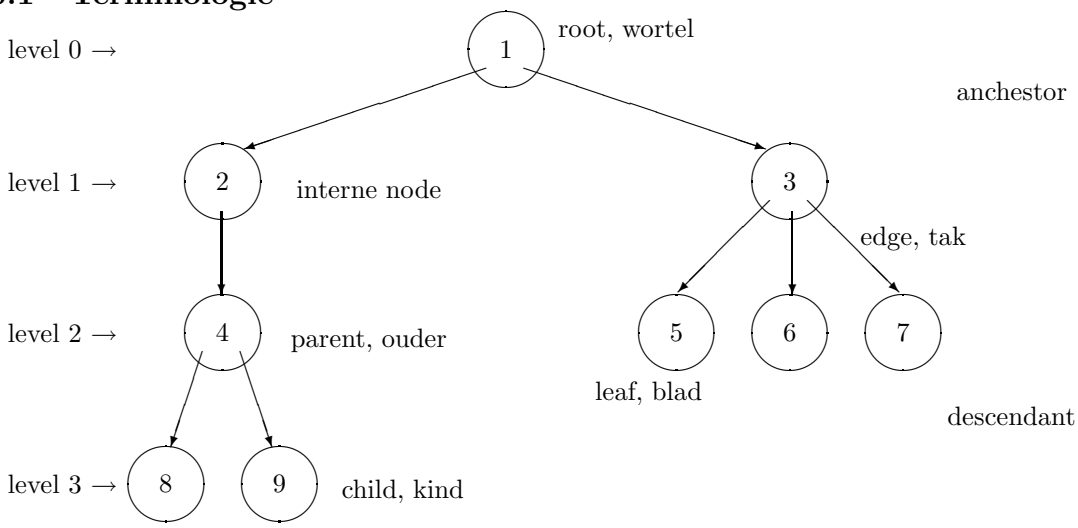
Veronderstel dat top een pointer is naar het eerste element van een dubbel gelinkte lijst. Zo'n element is van type Stack.
Wat is het effect van de functieoproep top = functie(top); ? Verklaar uw antwoord met een tekening.

```
typedef struct node
        {
            int waarde;
            struct node *volgend;
            struct node *vorig;
        } Stack;
```

```
Stack *functie(Stack *top)
{
    Stack *p = top;
    Stack *t;
    Stack *v = top;
    while ( p != NULL )
    {
        t = p->volgend;
        p->volgend = p->vorig;
        p->vorig = t;
        v = p;
        p = t;
    }
    return v;
}
```

10 Binaire bomen

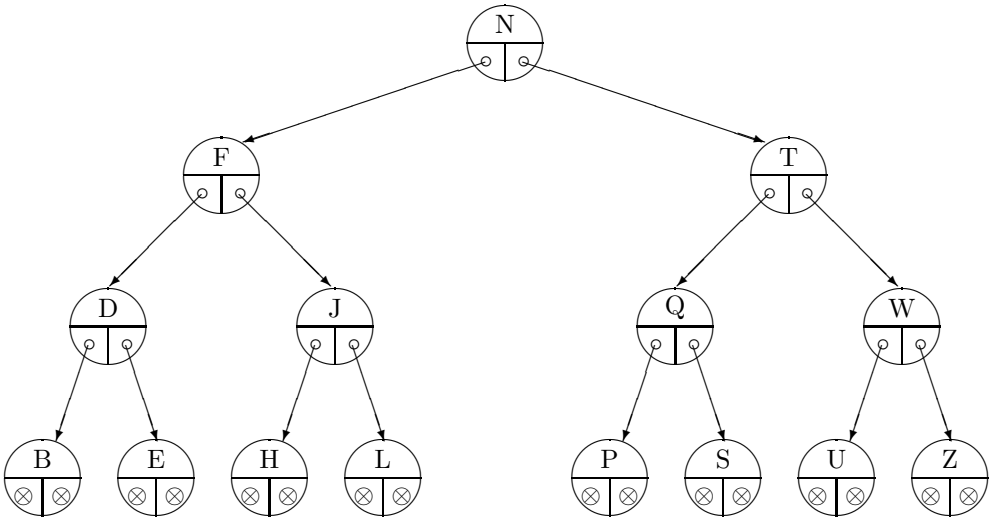
10.1 Terminologie



10.2 Definitie

Een *binaire boom* is

- ofwel een lege boom (geen node, geen takken);
- ofwel een node dat een linkse en een rechtse deelboom heeft en deze deelbomen zijn zelf binaire bomen (elke node heeft een linker- en rechternode, sommige daarvan zijn leeg).

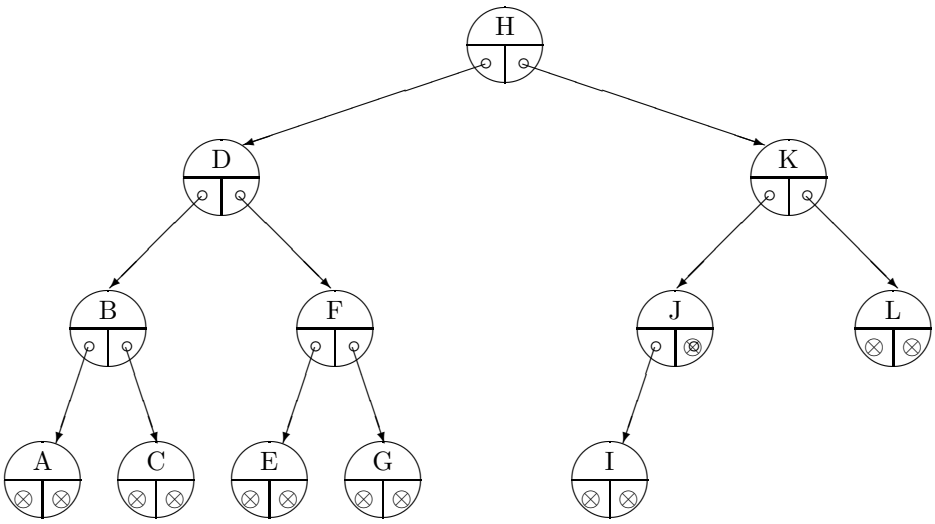


Het aantal knooppunten is gelijk aan n, het aantal levels l. Elke interne node heeft twee kinderen: een linker- en een rechter deelboom. Ook de leaves hebben twee kinderen, maar zo'n kind is de lege boom. In de figuur is een ideale boom (met $l = 4$ levels in $n = 15$ nodes) weergegeven: elk level is volledig opgevuld. Meestal is dat niet mogelijk, omdat bijvoorbeeld het aantal gegevens dat gestockeerd moet worden, niet overeenkomt met een ideaal aantal nodes $(2^l - 1)$.
Een *complete binaire* boom benadert zoveel mogelijk de ideale boom:

- slechts bladen op één level of bladen op twee naburige levels;
- in een partieel opgevuld level zitten de bladen in het meest linkse gedeelte.

Zo'n boom kan sequentieel met behulp van een array voorgesteld worden:

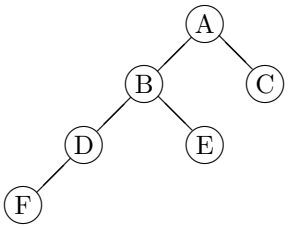
om te vinden:	gebruik	indien
linkerkind van $A[i]$	$A[2i]$	$2i \leq n$
rechterkind van $A[i]$	$A[2i+1]$	$2i+1 \leq n$
ouder van $A[i]$	$A[i/2]$	$i \geq 1$
de wortel	$A[1]$	A niet leeg
of $A[i]$ een blad is	$2i > n$	



A:		H	D	K	B	F	J	L	A	C	E	G	I
	0	1	2	3	4	5	6	7	8	9	10	11	12

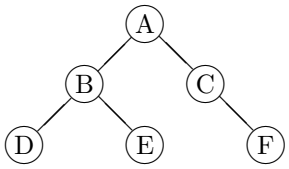
Voorbeelden van niet-complete bomen:

Een blad op een derde level terwijl level 2 nog niet opgevuld is:



A:		A	B	C	D	E	?	?	F
	0	1	2	3	4	5	6	7	8

Een blad op level 2 niet links aangevuld:



A:		A	B	C	D	E	?	F
	0	1	2	3	4	5	6	7

10.3 Een gebalanceerde boom

Opgave. Gegeven een getal n (het aantal knopen in de boom) en de bijhorende gegevens, in dit voorbeeld een reeks van n gehele getallen. Gevraagd een boom met minimale hoogte.
Om voor een gegeven aantal knopen een boom met minimale hoogte te construeren, moet op elk niveau het maximum aantal knopen gebruikt worden, behalve op het laagste niveau. Dit kan gerealiseerd worden door de nieuwe knopen gelijk te verdelen over de linkse en de rechtse kant van elke knoop. De regel om n knopen gelijkmatig links en rechts te verdelen kan gemakkelijk recursief geformuleerd worden:

1. Gebruik 1 knoop voor de wortel.
2. Genereer een linkse deelboom met $nl = n/2$ knopen op de hier gedefinieerde manier.
3. Genereer een rechtse deelboom met $nr = n - nl - 1$ knopen op de hier gedefinieerde manier.

Het resultaat is een *gebalanceerde* boom: voor elke knoop is het verschil in het aantal knopen tussen linkse en rechtse deelboom ten hoogste gelijk aan 1.

```
1   /*   balansboom.c  : een gebalanceerde boom   */
    #include <stdio.h>
3   #include <malloc.h>
    typedef int     Info;
5   typedef struct bnode
            {
7                   Info data;
                    struct bnode *links;
9                   struct bnode *rechts;
            } Node;
11
    Node *maken(int n);
13  void drukaf(Node *p, int spatie);
    int main(int argc, char *argv[] )
15  {
        Node *wortel;
17      int  n;

19      printf("Aantal knooppunten : "); scanf("%d%*c", &n);
        printf("\n");
21      wortel = maken(n);     scanf("%*c");
        drukaf(wortel, 0);
23  }

25  Node *maken(int n)
    {
27      Node *p;
        int  x, nl, nr;
29
        if ( n == 0 )
31              return (Node *)NULL;
        nl = n / 2;       nr = n - nl - 1;
33      scanf("%d", &x);
        p = (Node *)malloc(sizeof(Node));
35      p->data = x;
        p->links = maken(nl);
37      p->rechts = maken(nr);
        return(p);
39  }

41  void drukaf(Node *p, int spatie)
    {
43      int i;

45      if ( p != (Node *)NULL )
        {
47              drukaf(p->rechts, spatie+1);
```

```
            for (i=0; i<spatie; i++)
49              printf("      ");
            printf("%6d\n", p->data);
51          drukaf(p->links, spatie+1);
        }
53  }
```

Een invoer van de volgende data geeft een boom met 21 knopen.

```
21
 8  9 11 15 19 20 21  7  3  2  1  5  6  4 13 14 10 12 17 16 18
```

Het programma drukt de boom 90 graden gedraaid af:

```
                    18
               12
                    17
                         16
          5
                    14
                         10
                6
                     4
                         13
     8
                     1
                7
                     3
                          2
          9
                    20
                         21
               11
                    15
                         19
```

10.4 Wandelen doorheen een boom

In voorgaand programma werd de resulterende boom afgedrukt. Tijdens dit afdrukken werd telkens eerst een deelboom bekeken, dan de wortel van de deelboom en tenslotte de andere deelboom. Wanneer eerst de linkerdeelboom, dan de wortel en tenslotte de rechterdeelboom bekeken wordt, spreekt men van *inorder*. Daarnaast bestaat er ook *preorder* en *postorder*. Deze drie procedures kunnen gemakkelijk recursief gedefinieerd worden:

```
/* eerst parent en dan linkse en rechtse kind */
void preorder(Node *p)
{
    if ( p != (Node *)NULL )
    {
        printf("%3d", p->data);
        preorder(p->links);
        preorder(p->rechts);
    }
}
```

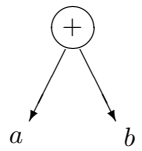
```
/* parent tussen linkse en rechtse kind */
void inorder(Node *p)
{
      if ( p != (Node *)NULL )
      {
            inorder(p->links);
            printf("%3d", p->data);
            inorder(p->rechts);
      }
}

/* eerst linkse en rechtse kind en dan de parent */
void postorder(Node *p)
{
      if ( p != (Node *)NULL )
      {
            postorder(p->links);
            postorder(p->rechts);
            printf("%3d", p->data);
      }
}
```

Voor voorgaande gegevens krijgt men volgende output:

```
Preorder:    8  9 11 15 19 20 21  7  3  2  1  5  6  4 13 14 10 12 17 16 18
Inorder:    19 15 11 21 20  9  2  3  7  1  8 13  4  6 10 14  5 16 17 12 18
Postorder:  19 15 21 20 11  2  3  1  7  9 13  4 10 14  6 16 17 18 12  5  8
```

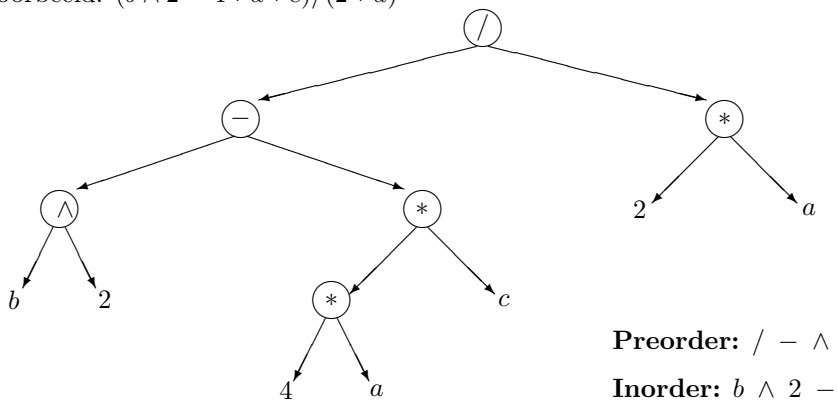
10.5 Expressiebomen

Een *expressieboom* is een binaire boom die een voorstelling geeft van een algebraïsche expressie met binaire operatoren. Bijvoorbeeld: $a + b$ is een algebraïsche expressie met de binaire operator $+$. De operands a en b zijn de bladen van de binaire boom; de operator $+$ is een interne node.



Wanneer de algebraïsche expressie meerdere operatoren bevat, moet rekening gehouden worden met de *prioriteit* en de *associativiteit* van de operatoren. Machtsverheffing ($\wedge$) heeft een hogere prioriteit dan vermenigvuldiging en deling ($*, /$) die op hun beurt een hogere prioriteit hebben dan optelling en aftrekking ($+, -$). De $\wedge$ operator is rechts associatief, terwijl de andere operatoren links associatief zijn.

Voorbeeld: $(b \wedge 2 - 4 * a * c)/(2 * a)$



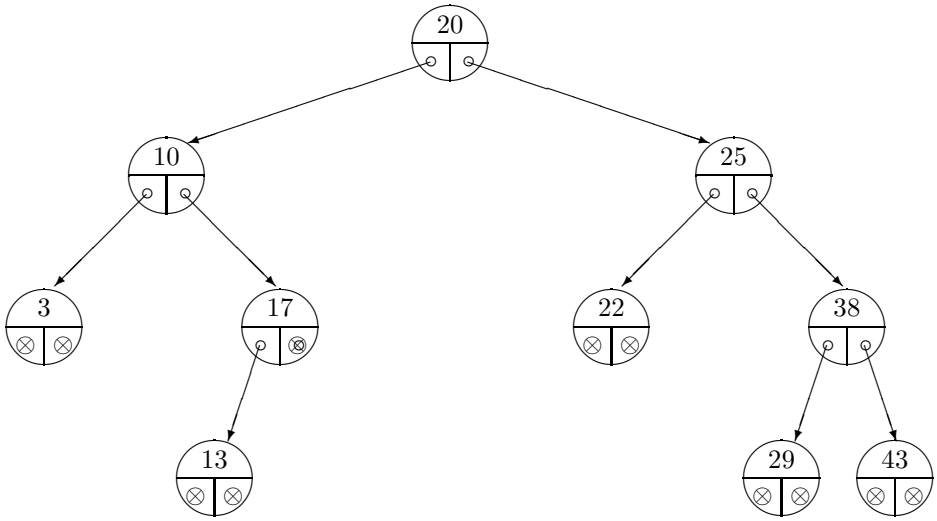
Preorder: / − ∧ b 2 ∗ ∗ 4 a c ∗ 2 a

Inorder: b ∧ 2 − 4 ∗ a ∗ c / 2 ∗ a

Postorder: b 2 ∧ 4 a ∗ c ∗ − 2 a ∗ /

10.6 Zoekbomen

Binaire bomen worden dikwijls gebruikt om een verzameling gegevens te stockeren waarvan de elementen via een *sleutel* moeten kunnen opgezocht worden. Indien een boom georganiseerd wordt zodat voor elke knoop t_i alle sleutels in de linkerdeelboom van t_i kleiner zijn dan de sleutel van t_i en alle sleutels in de rechterdeelboom van t_i groter zijn dan de sleutel van t_i, dan wordt zo'n boom een *zoekboom* genoemd.
In een zoekboom is het mogelijk een bepaald element te vinden door vanaf de wortel te beginnen en dan verder te gaan langs een pad door de linkse of rechtse knoop te kiezen op basis van de sleutel van de knoop. Omdat een verzameling van n elementen in een boom met hoogte $\log n$ kan gestockeerd worden, heeft het zoeken van een element dus slechts $\log n$ vergelijkingen nodig. Dit is heel wat minder dan wanneer de elementen in een lineaire lijst gestockeerd zijn.
Een voorbeeld van een zoekboom:



Hierin een element zoeken kan met de volgende functie gebeuren:

Iteratief:

```
Node *zoek(Node *t, Info x)
{
    while ( t != (Node *)NULL )
    {
        if ( x == t->data )
            return t;
        else if ( x < t->data )
            t = t->links;
        else
/* if ( x > t->data ) */
            t = t->rechts;
    }
    return (Node *)NULL;
}
```

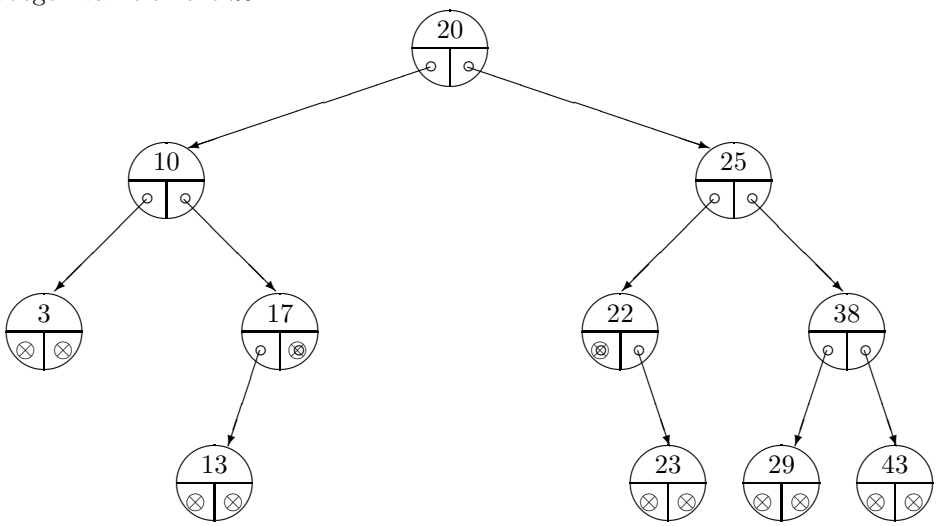
Recursief:

```
Node *zoek(Node *t, Info x)
{
    if ( t == (Node *)NULL )
        return (Node *)NULL;
    if ( x == t->data )
        return t;
    if ( x < t->data )
        return zoek(t->links, x);
    else
/* if (x > t->data) */
        return zoek(t->rechts, x);
}
```

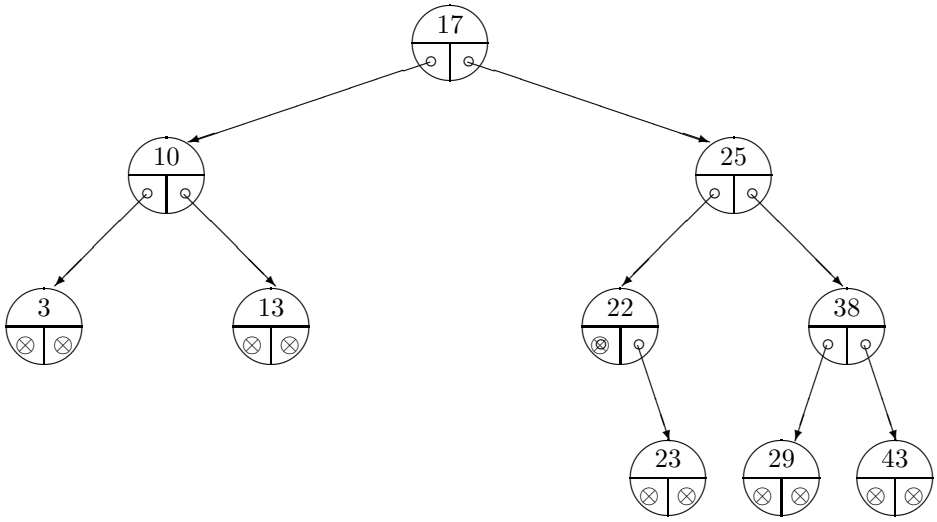
Wanneer het element niet aanwezig is in de zoekboom, zal de functie `zoek` wel de plaats gelokaliseerd hebben waar het element eventueel kan toegevoegd worden zodat de zoekboom een zoekboom blijft.
De functie `voegtoe` is een uitbreiding van de recursieve functie `zoek`:

```
1       Node *voegtoe(Node *t, Info x)
        {
3           Node *p;

5           if ( t == (Node *)NULL )
            {
7               t = (Node *)malloc(sizeof(Node));
                t->data = x;
9               t->links = t->rechts = (Node *)NULL;
                return t;
11          }
            if ( x == t->data )
13              return t;
            else if ( x < t->data )
15              t->links = voegtoe(t->links, x);
            else            /* if ( x > t->data )    */
17              t->rechts = voegtoe(t->rechts, x);
            return t;
19      }
```

Het toevoegen van element *23*:



Het verwijderen van element *20*:



Uit de figuur blijkt dat het verwijderen van een element niet altijd eenvoudig is wanneer de resulterende boom nog steeds een zoekboom moet zijn. Wanneer het te verwijderen element twee kinderen heeft, kan de ouder hiervan niet naar deze beide elementen wijzen, in de ouder is slechts één pointer beschikbaar. In dit geval moet het verwijderde element vervangen worden door ofwel het meest rechtse element in zijn linkerdeelboom ofwel door het meest linkse element in zijn rechterdeelboom.
In de volgende functie worden vier gevallen onderscheiden:

1. er is geen element met sleutel x aanwezig;
2. de knoop met sleutel x heeft alleen een linkerdeelboom;
3. de knoop met sleutel x heeft alleen een rechterdeelboom;
4. de knoop met sleutel x heeft zowel een linker- als een rechterdeelboom: de knoop wordt vervangen door het meest rechtse element van zijn linkerdeelboom.

```
1     Node *verwijder(Node *t, Info x)
      {
3           Node *p, *q, *s;

5           if ( t == (Node *)NULL )
            {
7                 return (Node *)NULL;
            }
9           if ( x < t->data )
                  t->links = verwijder(t->links, x);
11          else  if ( x > t->data )
                  t->rechts = verwijder(t->rechts, x);
13          else
            {
15                p = t->links;
                  q = t->rechts;
17                if ( p == (Node *)NULL )
                  {
19                      free(t);
                        return q;
21                }
                  else if ( q == (Node *)NULL )
23                {
                        free(t);
```

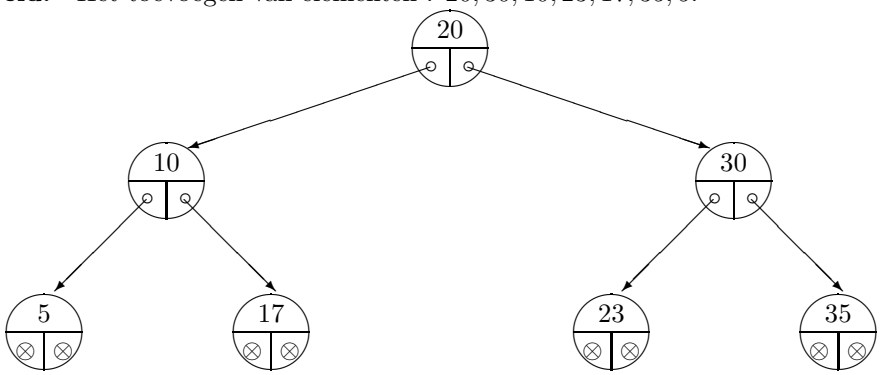
```
25                      return p;
                }
27              else
                {
29                      q = t;
                        s = t->links;
31                      while ( s->rechts != (Node *)NULL )
                        {
33                              q = s;
                                s = s->rechts;
35                      }
                        printf("moeilijk geval %d\n", s->data);
37                      if ( q == t )
                                q->links = s->links;
39                      else
                                q->rechts = s->links;
41                      t->data = s->data;
                        free(s);
43              }
        }
45      return t;
    }
```

Deze routines kunnen getest worden met volgend testprogramma:

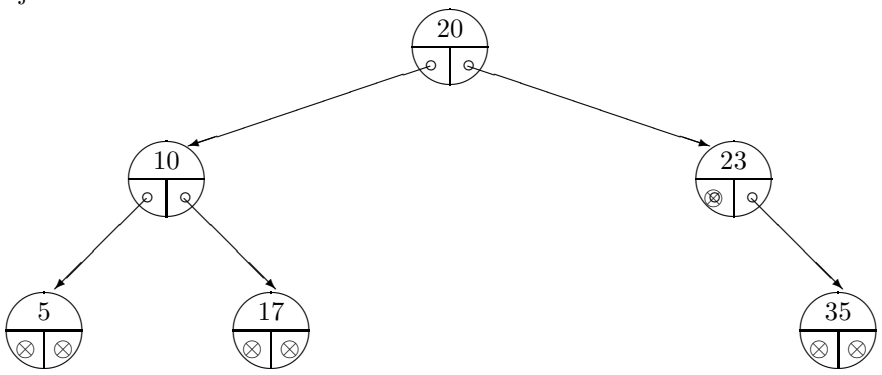
```
    /*
2    *  bid.c : zoekboom : toevoegen (zoeken) en verwijderen
     */
4   #include <stdio.h>
    typedef int     Info;
6   typedef struct bnode
        {
8           Info data;
            struct bnode *links;
10          struct bnode *rechts;
        } Node;
12
    Node *zoek(Node *t, Info x);
14  Node *voegtoe(Node *t, Info x);
    Node *verwijder(Node *t, Info x);
16  void inorder(Node *p);
    int main(int argc, char *argv[])
18  {
      Node *wortel = (Node *)NULL;
20    int  n;
      Info  x;
22
      while ( scanf("%d", &x) != EOF )
24    {
          wortel = voegtoe(wortel, x);
26    }
      printf("\nInorder : "); inorder(wortel); printf("\n");
```

```
28          freopen("/dev/tty", "r", stdin);
            scanf("%d%*c", &x);
30          printf("%x\n", zoek(wortel, x) );
            scanf("%d%*c", &x);
32          printf("%x\n", zoek(wortel, x) );
            while ( scanf("%d", &x) != EOF )
34          {
                    wortel = verwijder(wortel, x);
36                  if ( wortel == (Node *)NULL )
                    {
38                          printf("\nAlles is weg\n");
                            break;
40                  }
            }
42          printf("\nInorder : "); inorder(wortel); printf("\n");
    }
```

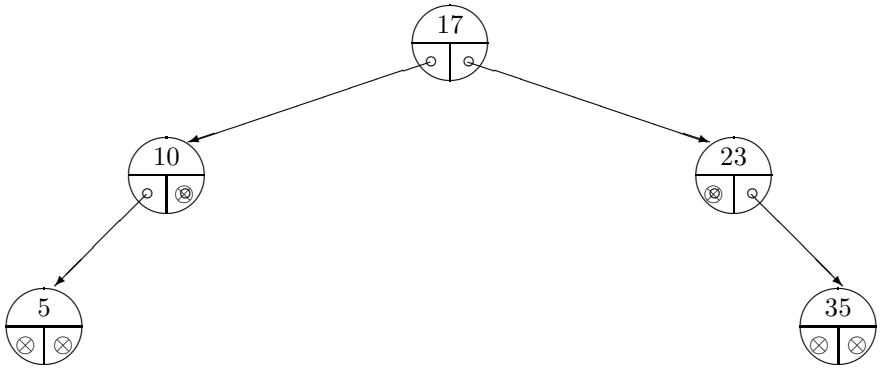
Voorbeeld. Het toevoegen van elementen : $20, 30, 10, 23, 17, 35, 5$:



Het opzoeken van elementen : $17, 26$.
Het verwijderen van element 30:



Het verwijderen van element 20:



Merk op.

1. Het maken van een gesorteerde lijst kan gebeuren door de boom in `inorder` te doorlopen.
2. Belang van een gebalanceerde boom: wanneer de elementen in een "verkeerde" volgorde toegevoegd worden, kan een lineaire lijst ontstaan in plaats van een min of meer gebalanceerde boom. Bijvoorbeeld bij de rij van getallen: 53, 38, 46, 41, 43, 60.

10.7 Denktaak

```
typedef int Info;
typedef struct node
        {
             Info          data;
             struct node *links;
             struct node *rechts;
        } Bnode;

void functie (Bnode *p, int d, int *md)
{
   if ( p != (Bnode *)NULL )
   {
      d++;
      if ( *md < d )
      {
         *md = d;
      }
      functie(p->links, d, md);
      functie(p->rechts, d, md);
   }
}
```

Veronderstel dat wortel een pointer is naar de wortel van een binaire boom.
Wat is het resultaat van de oproep functie(wortel, 0, &g); waarbij de variabele g op 0 geïnitialiseerd is?
Verklaar uw antwoord aan de hand van de expressieboom van de expressie $a + b \times c/d - e$.

11 Sorteren

11.1 Inleiding

Sorteren van gegevens is een veel voorkomende operatie. Men kan *sorteren* definiëren als het herschikken van een gegeven verzameling objecten zodat ze in een specifieke volgorde geplaatst worden. Een voorbeeld is een aantal getallen van klein naar groot sorteren. Meestal bestaat het object echter uit verschillende delen of velden. Het veld waarop de sortering gebeurt, wordt *sleutel* genoemd.
De verzameling objecten kan in een array zitten. We hebben het dan over *interne* sortering, omdat de gegevens in intern (snel toegankelijk) RAM geheugen zitten. Wanneer de verzameling objecten als structures in een bestand zitten, spreekt men van *externe* sortering.
In de volgende paragrafen worden enkele routines voor het sorteren van een array gegevens beschreven. Voor de eenvoud wordt een array van n integers genomen (het object bevat alleen de sleutel, namelijk een geheel getal). Bij zo'n interne sortering is de methode meestal gebaseerd op *in situ* sortering omwille van het zuinig omgaan met benodigde geheugenruimte. Dit wil zeggen dat geen tweede array mag gebruikt worden om het resultaat in te plaatsen. De bedoeling is dat de elementen in de gegeven array herschikt worden.

11.2 Sorteren door selectie

Het principe: *kies* het element met de kleinste sleutel en wissel dit met het eerste element. Deze operatie wordt dan herhaald voor de overblijvende $n-1$ elementen, dan met $n-2$ elementen, enz.
Voorbeeld:

	44	55	12	42	94	18	06	67
$i=1$	06	55	12	42	94	18	44	67
$i=2$	06	12	55	42	94	18	44	67
$i=3$	06	12	18	42	94	55	44	67
$i=4$	06	12	18	42	94	55	44	67
$i=5$	06	12	18	42	44	55	94	67
$i=6$	06	12	18	42	44	55	94	67
$i=7$	06	12	18	42	44	55	67	94

Voor $i = 1, 2, 3, \ldots, n-1$ doe
{
    k is de index van het kleinste element uit $\{a_i, \ldots, a_n\}$;
    wissel a_i en a_k;
}

De functie:

```
void selectie(int a[], int n)
{
    int     i, j, k;
    int     x;

    for (i=1; i<=n-1; i++)
    {
        x = a[i];     k = i;
        for (j=i+1; j<=n; j++)
        {
            if ( a[j] < x )
            {
```

```
                x = a[j];       k = j;
            }
        }
        a[k] = a[i];        a[i] = x;
    }
}
```

11.3 Sorteren door invoegen

Het principe: de objecten zijn verdeeld in een doelreeks $a_1, \ldots, a_{i-1}$ en een bronreeks $a_i, \ldots, a_n$; in opeenvolgende stappen wordt het volgende element a_i van de bronreeks *ingevoegd* in de doelreeks. Voorbeeld:

	44	55	12	42	94	18	06	67
$i = 2$	44	55	12	42	94	18	06	67
$i = 3$	12	44	55	42	94	18	06	67
$i = 4$	12	42	44	55	94	18	06	67
$i = 5$	12	42	44	55	94	18	06	67
$i = 6$	12	18	42	44	55	94	06	67
$i = 7$	06	12	18	42	44	55	94	67
$i = 8$	06	12	18	42	44	55	67	94

Voor $i = 2, 3, \ldots, n$ doe
{
    $x = a_i$;
    voeg x in op de juiste plaats tussen $\{a_1, \ldots, a_i\}$;
}

De functie:

```
void invoegen(int a[], int n)
{
    int     i, j;
    int     x;

    for (i=2; i<=n; i++)
    {
        a[0] = x = a[i];
        j = i - 1;
        while ( x < a[j] )
        {
            a[j+1] = a[j];    j--;
        }
        a[j+1] = x;
    }
}
```

Merk op. Om de extra test $j > 0$ niet telkens te moeten doen, wordt gebruikt gemaakt van een *sentinel*, door in `a[0]` de waarde van x te stoppen. Dit nulde element van de array bevat toch geen element van de rij die moet gesorteerd worden. De rij met de te sorteren getallen begint vanaf `a[1]`.

11.4 Sorteren door wisselen

Het principe van *bubble sort*: vergelijk twee naast elkaar gelegen elementen; indien ze niet in volgorde staan, wissel dan deze de elementen. Dit wordt gedaan voor elk paar buren in de rij en het geheel wordt herhaald tot geen enkel paar buren nog gewisseld wordt.
Voorbeeld:

	44	55	12	42	94	18	06	67
$i = 2$	06	44	55	12	42	94	18	67
$i = 3$	06	12	44	55	18	42	94	67
$i = 4$	06	12	18	44	55	42	67	94
$i = 5$	06	12	18	42	44	55	67	94
$i = 6$	06	12	18	42	44	55	67	94

De functie:

```
void bubble(int a[], int n)
{
    int     i, j;
    int     x;

    for (i=2; i<=n; i++)
    {
        for (j=n; j>=i; j--)
        {
            if ( a[j-1] > a[j] )
            {
                x = a[j-1];    a[j-1] = a[j];    a[j] = x;
            }
        }
    }
}
```

11.5 Een verdeel-en-heers techniek

Het principe: *verdeel* de verzameling objecten in twee delen en sorteer deze twee delen onafhankelijk van elkaar.

```
Quicksort ( a, l, r)
{
      indien meerdere elementen in {a_l, ..., a_r}
      {
            verdeel {a_l, ..., a_r} in een links en een rechts deel;
                  (gebruik makend van een pivot element)
            Quicksort het linkse deel;
            Quicksort het rechtse deel;
      }
}
```

De verdeling moet gebeuren zodat

- het pivot element a_i bevindt zich op zijn uiteindelijke plaats i;
- al de elementen in $\{a_l, \ldots, a_{i-1}\}$ zijn kleiner dan of gelijk aan a_i;

- al de elementen in $\{a_{i+1}, \ldots, a_r\}$ zijn groter dan of gelijk aan a_i.

Voorbeeld:

l	$(l+r)/2$	r								
1	4	8	44	55	12	*42*	94	18	6	67
1	2	3	6	*18*	12	42	94	55	44	67
1	1	2	*6*	12	18	42	94	55	44	67
5	6	8	6	12	18	42	94	*55*	44	67
7	7	8	6	12	18	42	44	55	*94*	67
			6	12	18	42	44	55	67	94

De functie:

```
void quicksort(int a[], int l, int r )
{
    int     i, j;
    int     x, w;

    i = l ;
    j = r ;
    x = a[ (l+r)/2 ] ;
    do
    {
        while ( a[i] < x )
            i++ ;
        while ( a[j] > x )
            j-- ;
        if ( i <= j )
        {
            w = a[i] ;
            a[i++] = a[j] ;
            a[j--] = w ;
        }
    }
    while ( i <= j ) ;
    if ( l < j )
        quicksort(a,l,j) ;
    if ( i < r )
        quicksort(a,i,r) ;
}
```

Oefening. Probeer quicksort uit op de rij $44, 55, 12, 94, 42, 18, 6, 67$.

11.6 Vergelijking

Om de hierboven beschreven sorteerroutines te vergelijken op gebruikte rekentijd, is volgend hoofdprogramma gebruikt.

```
1   #include <sys/types.h>
    #include <sys/param.h>
3   #include <sys/times.h>
    #include <time.h>
5   #include <stdio.h>
```

```
7       #define AANTAL 8260
        #define GROOT 2147483648.0
9
        int main(int argc, char *argv[])
11      {
            int   a[AANTAL];
13          int   b[AANTAL];
            int   i, n;
15          struct tms   actueel ;
            int   tijd;
17          int   qtijd;

19          if ( argc > 1 )
                n = atoi(argv[1]);
21          else
                n = 10;
23          srand(1063);
            for (i=1; i<=n; i++)
25              b[i] = a[i] = 1 + (int) ( (double)n * rand()/GROOT);
            druk(a,b, n);
27          times(&actueel) ;
            tijd = actueel.tms_utime;
29          quicksort(a, 1, n );
            druk(a,b, n);
31          times(&actueel) ;
            qtijd = actueel.tms_utime-tijd;
33          printf("n %5d:   q %5d\n", n, qtijd);
        }
35
        void druk( int a[], int b[], int n)
37      {
            int  i;
39
            for (i=1; i<=n; i++)
41          {
                printf("%3d%c", a[i], (i%19 ? ' ':'\n') );
43              a[i] = b[i];
            }
45          printf("\n");
        }
```

In volgende tabel is de gebruikte rekentijd weergegeven voor een aantal verschillende n waarden:

n	selectie	invoegen	bubble	quicksort
128	3	2	3	2
256	8	6	12	3
512	25	17	40	6
1024	85	59	158	14
2048	316	214	580	29
4096	1234	797	2240	59
8192	4828	3122	8890	120

Uit deze tabel blijkt duidelijk dat quicksort de snelste methode is. Bubblesort, wat een klassiek voorbeeld is in veel inleidende cursussen informatica, is de minst goede methode.

11.7 Een generieke sorteerfunctie

De functie die de rangorde/volgorde van de elementen die moeten gesorteerd worden bepaalt, wordt als actueel argument doorgegeven aan de sorteerfunctie. In de sorteerfunctie is deze parameter een pointer naar een functie. Om de gepaste *kleiner dan* relatie te bepalen, wordt met behulp van deze pointer de juiste functie opgeroepen.
Deze functie heeft twee parameters, namelijk pointers naar de twee elementen die met elkaar moeten vergeleken worden. De functie heeft als terugkeerwaarde:

- -1 : als het eerste element *kleiner* is dan het tweede element;
- 0 : als het eerste element *gelijk* is aan het tweede element;
- 1 : als het eerste element *groter* is dan het tweede element.

Bijvoorbeeld wanneer de twee elementen pointers naar strings zijn, dan kan de standaard bibliotheekfunctie **int** strcmp(**const char** *, **const char** *) als functie gebruikt worden.

```
/*
 * gensort : een generieke sorteerfunctie
 */
#include <stdlib.h>
#include <stdio.h>
#define AANTAL 8260
#define GROOT 65535.0

typedef struct
        {
            int x;
            int y;
        } Blok;

void selectie(Blok a[], int b, int n, int (*vgl)(Blok *, Blok *));
void druk( Blok a[], int n);
int vglx(Blok *s, Blok *t);
int vglf(Blok *s, Blok *t);

int main(int argc, char *argv[])
{
    Blok    a[AANTAL];
    int     i, n = 10;

    if ( argc > 1 )
        n = atoi(argv[1]);
    srand(n*1063);
    printf(" %d %d\n",  rand(), RAND_MAX );
    for (i=1; i<=n; i++)
    {
        a[i].x = 1 + (int) ( (rand()/(float)RAND_MAX) * 4.0*n );
        a[i].y = 1 + (int) ( (rand()/(float)RAND_MAX) * 4.0*n );
    }
    druk(a, n);
    selectie(a, 1, n, vglx );
    druk(a, n);
    selectie(a, 1, n, vglf );
    druk(a, n);
}
```

```
40
    int vglx(Blok *s, Blok *t)
42  {
        if ( s->x < t->x )
44          return -1;
        else if ( s->x > t->x )
46          return 1;
        else
48          return 0;
    }
50  int vglf(Blok *s, Blok *t)
    {
52      if (  (float)s->x/s->y  <  (float)t->x/t->y  )
            return -1;
54      else if (  (float)s->x/s->y  >  (float)t->x/t->y  )
            return 1;
56      else
            return 0;
58  }
    void selectie(Blok a[], int b, int n, int (*vgl)(Blok *, Blok *))
60  {
        int    i, j, k;
62      Blok    x;

64      if ( b == n )
            return;
66      x = a[b];    k = b;
        for (j=b+1; j<=n; j++)
68      {
            if ( vgl(&a[j], &x) < 0 )
70          {
                x = a[j];    k = j;
72          }
        }
74      a[k] = a[b];     a[b] = x;
        selectie(a, b+1,n, vgl);
76      return;
    }
78  void druk( Blok a[], int n)
    {
80      int i;

82      for (i=1; i<=n; i++)
        {
84          printf("%3d%c", a[i].x, (i%19 ? ' ':'\n') );
        }
86      printf("\n");
        for (i=1; i<=n; i++)
88      {
            printf("%3d%c", a[i].y, (i%19 ? ' ':'\n') );
90      }
        printf("\n\n");
92  }
```

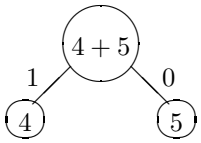
12 Toepassing: Huffman codes

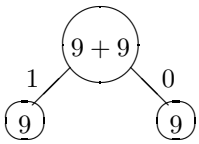
Binaire bomen kunnen gebruikt worden om een codering met minimale lengte te bouwen wanneer de frequentie van de gebruikte letters in de boodschap gekend is. De verschillende letters worden gecodeerd met bit-sequenties van verschillende lengte. Korte bit-sequenties worden gebruikt voor veel-voorkomende letters, terwijl minder gebruikte letters worden voorgesteld door middel van langere bit-sequenties.

12.1 Huffman coding boom.

Gegeven 5 letters (E, T, N, I, S) en hun frequenties van voorkomen in een boodschap, namelijk 29, 10, 9, 5 en 4.
We vertrekken van de waarden van de frequenties en we sorteren deze in oplopende volgorde: (4, 5, 9, 10, 29).

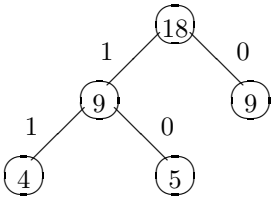
Hiervan worden de twee kleinste waarden genomen: 4 en 5. Deze worden gebruikt om een binaire boom te bouwen. De waarden worden in de bladeren geplaatst en de wortel bevat de som van de waarden in de bladeren. De linkse tak krijgt label "1" en de rechtse tak label "0".

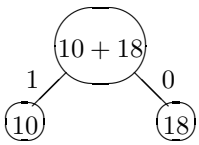




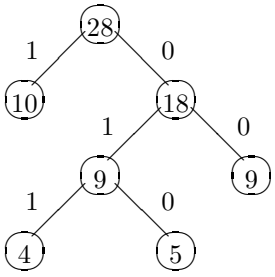
In de rij van getallen worden de 4 en 5 vervangen door hun som: (9, 9, 10, 29). Deze rij wordt terug gesorteerd van klein naar groot (in dit geval is de volgorde al ok). De twee kleinste waarden wordt terug gebruikt om een binaire boom te vormen.

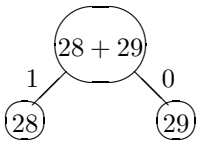
De twee kleine bomen worden nu samengebracht in een samengestelde boom door in de tweede boom het linkse blad dat een som bevat, te vervangen door de volledige eerste boom.





In de rij worden de twee negens vervangen door hun som: (18, 10, 29) en gesorteerd: (10, 18, 29). Met de twee kleinste waarden wordt een boom gevormd en het blad met de som 18 wordt vervangen door de hierboven gebouwde boom.

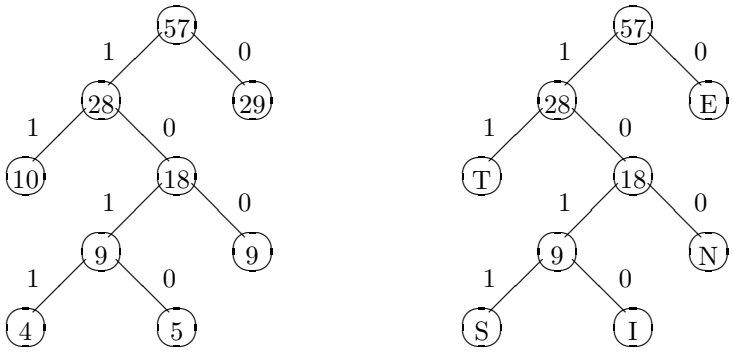




De rij van getallen is nu (28, 29) en er moet dus nog één stap uitgevoerd worden.

12.2 Coderen en decoderen.

In de resulterende boom kunnen de frequentiewaarden in de bladeren vervangen worden door de respektievelijke letters.



Deze boom kan gebruikt worden om een rij van tekens met de letters (E, T, N, I, S) te coderen in een bitrij en omgekeerd om een bitrij te decoderen in een rij van tekens.
De code die aan een letter toegekend wordt, kan gevonden worden door vanaf de wortel het pad te volgen naar het blad dat die letter bevat. De opeenvolgende 1 en 0 bits langs dit pad vormen de code. Het resultaat is in de tabel weergegeven. Merk op dat de letters met de hoogste frequentiewaarden de kortste codes hebben.
Om een woord te coderen wordt elke letter vervangen door haar corresponderende bitrij.

letter	bit code	frequen
E	0	29
T	11	10
N	100	9
I	1010	5
S	1011	4

woord	geëncodeerde bitrij
SENT	1011010011
TENNIS	11010010010101011
NEST	1000101111
SIT	1011101011

Om een bit-string S te decoderen, worden de opeenvolgende bits van S gebruikt om een pad doorheen de coding boom te vinden vertrekkend vanaf de wortel. Een "1" betekent naar links en een "0" naar rechts. Telkens een blad bereikt wordt, wordt de bijhorende letter genoteerd. Er wordt terug vanaf de wortel vertrokken om op basis van de volgende bits een volgend pad te zoeken.
Bijvoorbeeld, bij de bit-string `1 1 0 1 0 0 1 1 1 0 1 1` wordt vanaf de wortel op basis van de eerste twee 1-bits een "T" gevonden. De volgende 0-bit geeft een "E". Dan heeft met een "N" op basis van de `100`-rij. Er volgt nog een "T" (`11`) en een "S" (`1011`). Het woord is dus "TENTS".
Men kan aantonen dat een Huffman code de lengte van de gecodeerde boodschap minimaliseert wanneer in de boodschap letters gebruikt worden met dezelfde frequentie als in de steekproef waarmee de coding boom geconstrueerd is.

13 Backtracking algoritmes

13.1 Techniek

Voor vele problemen kan een oplossing gevonden worden door een bepaalde methode stap voor stap (een algoritme) te volgen. Er bestaan echter nog veel meer problemen waarvoor zo'n algoritme niet voor handen is. Men heeft hier te maken met "general problem solving" en men kan zo'n probleem aanpakken met *trial-and-error*.
Gewoonlijk wordt een *trial-and-error* proces opgedeeld in partiële taken. Deze taken kunnen op een natuurlijke manier in recursieve termen uitgedrukt worden en bestaan uit het onderzoeken van een eindig aantal deeltaken. Het gehele proces kan gezien worden als een zoekproces dat stelselmatig opgebouwd wordt waarbij telkens een waaier van deeltaken moet verder onderzocht worden. Men spreekt soms van een *zoekboom*. Deze boom kan zeer snel groeien, meestal exponentieel snel.
Een karakteristiek element van de methode is dat er een poging ondernomen wordt in de richting van een volledige oplossing. Eens deze poging aanvaard is, doet men een volgende poging zodat men telkens dichter en dichter bij de volledige oplossing komt. Het is natuurlijk mogelijk dat men op een bepaald moment vaststelt dat men de volledige oplossing niet kan bereiken. Op zo'n moment kunnen aanvaarde pogingen echter ongedaan gemaakt worden: men keert dus terug op zijn stappen (*backtracking*) om dan met een alternatieve poging terug in de richting van de oplossing te werken.

```
probeer()
{
        initialiseer keuze van mogelijke kandidaten ;
        doe
        {
                genereer volgende kandidaat;
                indien aanvaadbaar
                {
                        voeg kandidaat aan oplossing toe;
                        indien oplossing onvolledig
                        {
                                probeer();
                                verwijder kandidaat van oplossing;
                        }
                        anders
                                er is een oplossing gevonden;
                }
        }
        totdat er geen kandidaten meer zijn;
}
```

13.2 Voorbeeld: het koninginnen-probleem

Plaats op een 8×8 schaakbord een koningin op elke rij zodat er geen enkele koningin op dezelfde kolom of dezelfde diagonaal staat.
Merk op dat deze oefening kan opgelost worden voor een $N \times$N bord met N koninginnen waarbij $N >= 4$.

```
/*   nqueen.c : N-koningingen probleem  */
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <setjmp.h>
```

```
6   #include <memory.h>
    #define LEN 21
8
    int geldig( char a[][LEN], int  m, int i, int j );
10  void probeer( char a[][LEN], int  m, int i );
    void drukbord( char a[][LEN], int m );
12  int verbose = 0;
    int aantal_pogingen = 0;
14  int aantal_oplossing = 0;
    int maxtal_oplossing = 1;
16  jmp_buf omgeving;     /* stack context/omgeving  */

18  int main( int argc, char *argv[])
    {
20      char    b[LEN][LEN];
        int     n = 4;
22      char    c;

24      while ( (c=getopt(argc,argv,"n:a:v")) != EOF )
        {
26          switch ( c )
            {
28              case 'n': n = atoi(optarg);
                          break;
30              case 'a': maxtal_oplossing = atoi(optarg);
                          break;
32              case 'v': verbose++;
                          break;
34          }
        }
36      if ( optind < argc )
        {
38          fclose(stdout);
            stdout = fopen(argv[optind], "w");
40      }
        memset(b, '\0', LEN*LEN);
42      if ( setjmp(omgeving) == 0 )
            probeer(b,n,1);
44      printf("Aantal pogingen : %d\n", aantal_pogingen);
    }
46  void drukbord( char a[][LEN], int m )
    {
48      int i, j, l=2*m;

50      for(i=1; i<=m; i++)
        {
52          for(j=0; j<=l; j++)
                printf("-");
54          printf("\n");
            for(j=1; j<=m; j++)
56              if ( a[i][j] != '\0' )
                    printf("|*");
58              else
                    printf("| ");
```

```
60          printf("|\n");
        }
62      for(j=1; j<=l; j++)
            printf("-");
64      printf("\n");
    }
66  int geldig( char b[][LEN], int  n, int rij, int kol )
    {
68      int i, k;

70      for (i=1; i<rij; i++)
        {
72          if ( b[i][kol] == 1 )
                return 0;
74          k = kol-i;
            if ( k >= 1 && b[rij-i][k] == 1 )
76              return 0;
            k = kol+i;
78          if ( k <= n && b[rij-i][k] == 1 )
                return 0;
80      }
        return 1;
82  }
    void probeer( char b[][LEN], int  n, int rij )
84  {
        int j;
86
        aantal_pogingen++ ;
88      for (j=1; j<=n; j++)
        {
90          if ( geldig(b, n, rij, j) == 1 )
            {
92              if ( verbose )
                    printf("rij %2d : kolom %2d \n", rij, j );
94              b[rij][j] = 1;
                if ( rij == n )
96              {
                    aantal_oplossing++;
98                  printf("Oplossing %4d/%4d na %d pogingen\n",
                    aantal_oplossing, maxtal_oplossing, aantal_pogingen);
100                 drukbord( b, n);
                    if ( aantal_oplossing == maxtal_oplossing)
102                     longjmp(omgeving,1);
                }
104             else
                    probeer(b, n, rij+1);
106             b[rij][j] = 0;
            }
108     }
    }
```

Merk op dat dit een programma is met *opties* (`a.out -v -n8`).
De verschillende opties kunnen in een **while** lus verwerkt worden met de getopt functie. Eerste

en tweede argument van deze functie zijn de parameters van de main functie; het derde argument is een string met de mogelijke opties. De terugkeerwaarde is telkens één van de mogelijke opties. Door middel van een **switch** kan de desbetreffende optie geselecteerd worden.
In het derde argument na de optieletter een dubbelpunt (:) vermelden, geeft aan dat er bij deze optie een bijkomende parameter is. Deze parameter wordt door getopt beschikbaar gesteld via de variabele optarg, van type **char** $*$. Indien de parameter een getal is, moet de bijhorende numerieke waarde berekend worden, bijv. met de functie atoi of atof.
Nadat alle opties behandeld zijn, geeft getopt de waarde EOF terug. De variabele optind heeft dan als waarde de index van het eerstvolgende element in de argv array (dat niet met een minteken begint).
Indien slechts een beperkt aantal oplossingen moeten gezocht worden, wordt met behulp van `longjmp` teruggegaan naar de plaats in het programma waar de zoektocht naar oplossingen begonnen is. Hierdoor worden alle terugkeren uit alle tussenliggende recursieve oproepen vermeden. De plaats waar de zoektocht begonnen is, wordt onthouden in een globale variabele `omgeving` met behulp van de functie `setjmp`.

Een aantal oplossingen:
$N = 4$

```
Oplossing  1/2
---------
| |*| | |
---------
| | | |*|
---------
|*| | | |
---------
| | |*| |
--------
Aantal pogingen : 8

Oplossing    2/2
---------
| | |*| |
---------
|*| | | |
---------
| | | |*|
---------
| |*| | |
--------
Aantal pogingen : 15
```

$N = 8$

```
Oplossing   1/100
-----------------
|*| | | | | | | |
-----------------
| | | | |*| | | |
-----------------
| | | | | | | |*|
-----------------
| | | | | |*| | |
-----------------
| | |*| | | | | |
-----------------
| | | | | | |*| |
-----------------
| |*| | | | | | |
-----------------
| | | |*| | | | |
----------------
Aantal pogingen : 113
```

```
Oplossing   92/100
-----------------
| | | | | | | |*|
-----------------
| | | |*| | | | |
-----------------
|*| | | | | | | |
-----------------
| | |*| | | | | |
-----------------
| | | | | |*| | |
-----------------
| |*| | | | | | |
-----------------
| | | | | | |*| |
-----------------
| | | | |*| | | |
----------------
Aantal pogingen : 1965
```

Bij $N = 4$ zijn er 2 verschillende oplossingen: de eerste wordt gevonden na 8 pogingen, de tweede na 11 pogingen. Daarna zijn er nog 4 pogingen waaruit geconcludeerd wordt dat er geen bijkomende oplossingen meer zijn.

Bij $N = 8$ zijn er 92 verschillende oplossingen (waarvan een heleboel een spiegelbeeld van een andere oplossing zijn). In weze zijn er 12 echt verschillende oplossingen. Een uitbreiding in het programma zou kunnen zijn: het genereren van deze echt verschillende oplossingen.
Bij $N = 20$ zijn er 199635 pogingen nodig om een eerste oplossing te vinden. Een klassieke PC gebruikt hiervoor een drietal CPU seconden rekentijd.

13.3 Oefening 1

Oefening 1: (4pt)

Gegeven een matrix met in elk element (rij,kolom) een positief getal dat het aantal bommen weergeeft dat op die plaats ligt.
Gevraagd : een functie die een weg bepaalt doorheen dit bommenveld van start naar einde met het minste aantal bommen.

Mogelijke stappen:
- naar vorige rij in dezelfde kolom,
- naar volgende kolom in dezelfde rij,
- naar volgende rij in dezelfde kolom.

Startpositie is element (1,1); eindpositie is een element in een rij in de laatste kolom.
Geef aan welke deel van je functie voor een hogere performantie kan zorgen.
Voorbeeld:

```
1  8  1  2  9  9
1  4  3  9  7  8
9  2  1  9  1  2
9  3  1  8  1  9
7  3  1  7  1  8
6  5  1  1  1  7
```

```
( 1, 1) ( 2, 1) ( 2, 2) ( 3, 2) ( 3, 3)
( 4, 3) ( 5, 3) ( 6, 3) ( 6, 4) ( 6, 5)
( 5, 5) ( 4, 5) ( 3, 5) ( 3, 6)  :    19
```

13.4 Oefening 2

Gegeven een $n \times n$ bord met n^2 velden. Plaats in deze velden de getallen van 1 tot n^2 in oplopende volgorde waarbij een volgend getal slechts op bepaalde posities ten opzichte van het vorige getal geplaatst kan worden: naar links, rechts, onder of boven: twee velden tussen laten en in de diagonalen één veld tussen laten.

			7			
	6				8	
5			X			1
	4				2	
			3			

Een aantal oplossingen:

$n = 5$				
1	24	14	2	25
16	21	5	8	20
13	10	18	23	11
4	7	15	3	6
17	22	12	9	19

$n = 6$					
1	21	30	2	22	29
35	26	18	36	25	17
14	32	23	28	31	3
6	20	10	7	19	11
34	27	15	33	24	16
13	8	5	12	9	4

14 Hutscodering

14.1 Inleiding

Een *tabel* T is een abstract stockage middel met *plaatsen* die leeg kunnen zijn of die elementen van de vorm (K, I) bevatten. K is de sleutel (of key) en I is de bijhorende informatie. Op een tabel zijn verschillende operaties mogelijk:

- creatie van een lege tabel;
- een nieuw element aan een tabel toevoegen;
- een element verwijderen uit een tabel;
- een element opzoeken in de tabel (en eventueel de bijhorende I wijzigen);
- een lijst maken van alle elementen in de tabel, eventueel gesorteerd.

Er zijn verschillende manieren om zo'n tabel te organiseren: een array, een lineaire lijst of een boom. Welke keuze moet gemaakt worden is afhankelijk van welke van de bovenstaande operaties het meeste uitgevoerd zal worden.
In dit hoofdstuk wordt nog een bijkomende manier voorgesteld, *hashing*. Deze techniek is bijzonder geschikt wanneer toevoegen en opzoeken de meest voorkomende operaties zijn. Het maken van een gesorteerde lijst van alle elementen in de tabel is echter heel wat moeilijker.

14.2 De techniek

Om de techniek te illustreren, hebben de elementen een sleutel K gelijk aan de letters van het alfabet met een index. Deze index geeft de positie van de letter in het alfabet aan: bijvoorbeeld A_1, B_2, C_3, R_{18} en Z_{26}. Voor de eenvoud is er geen bijhorende informatie I in een element opgenomen.
Om deze elementen op te slaan, kan een array van 26 plaatsen voorzien worden, voor elk mogelijk element één plaats. Het is echter mogelijk dat het aantal reëel voorkomende elementen veel kleiner is dan het aantal mogelijke waarden voor de sleutel. In dat geval kan er sprake zijn van een enorme geheugenverspilling. Het zou interessant zijn om een tabel te kunnen voorzien met slechts een beperkt aantal plaatsen, d.w.z. kleiner dan het totaal aantal mogelijke waarden voor K.
Om de techniek te illustreren, wordt een tabel met 7 plaatsen gebruikt. Deze plaatsen zijn genummerd van 0 tot en met 6.

0	1	2	3	4	5	6

een lege tabel met zeven plaatsen

De plaats waar een element moet toegevoegd worden, moet nu berekend worden uit de sleutel. In dit voorbeeld kan dit gebeuren door de sleutelwaarde (= de index bij de letter) te delen door de lengte van de tabel (7) en de restwaarde te gebruiken als nummer van de plaats in de tabel.
Dus voor het element J_{10} wordt een plaats 3 berekend. Indien plaats 3 nog leeg is, kan op die plaats het element J_{10} toegevoegd worden. Op dezelfde manier wordt B_2 op plaats 2 toegevoegd en het element S_{19} op plaats 5.

0	1	2	3	4	5	6
		B_2	J_{10}		S_{19}	

na toevoegen van drie elementen

In het algemeen geval wordt de plaats berekend uit de sleutelwaarde (L_n) met behulp van een *hash functie*:

$$h(L_n) = n \,\%\, 7.$$

Een goede hash functie zal de sleutels zo uniform mogelijk verdelen over alle mogelijke plaatsen van de tabel.
Indien in het voorbeeld nu de elementen N_{14}, W_{23} en X_{24} toegevoegd worden, ontstaan er problemen:

0	1	2	3	4	5	6
N_{14}		B_2	J_{10}		S_{19}	

Toevoegen van element N_{14}
geen probleem: plaats 0 is leeg

Het volgende element met sleutel W_{23}, resulteert in een plaats $h(W_{23}) = 2$. Maar op plaats 2 is reeds een element aanwezig, namelijk B_2. Element W_{23} kan dus op deze plaats niet meer toegevoegd worden. Men spreekt van een *botsing*, omdat de twee sleutels B_2 en W_{23} op hetzelfde hash adres botsen: $2 = h(B_2) = h(W_{23})$.
Een mogelijke oplossing is naar lege plaatsen met lagere nummers te gaan zoeken, te vertrekken vanaf de plaats waar de botsing plaats vond. Deze methode, waarbij elementen op lege plaatsen van de tabel toegevoegd worden, wordt *open addressing* genoemd. Voor W_{23} wordt de lege plaats 1 gevonden, waar het element kan toegevoegd worden.

0	1	2	3	4	5	6
N_{14}	W_{23}	B_2	J_{10}		S_{19}	

element W_{23}: niet op plaats 2
plaats 1 is nog leeg

Het element met sleutel X_{24} resulteert in het hash adres 3. Op deze plaats is reeds een element aanwezig. Maar dit is ook het geval in alle plaatsen met lagere nummers. Indien dit het geval is, wordt verder gezocht vanaf het laatste element in de tabel. Dus, X_{24} kan toegevoegd worden op plaats 6.

0	1	2	3	4	5	6
N_{14}	W_{23}	B_2	J_{10}		S_{19}	X_{24}

X_{24}: niet op 3, 2, 1, 0
uiteindelijk op plaats 6

De plaatsen die onderzocht worden wanneer een nieuw element met sleutel L_n toegevoegd wordt, vormen de *probe* sequentie. Elke van deze plaatsen wordt *gepeild* om te bepalen of ze nog leeg is. De eerste plaats in deze sequentie is het hash adres, $h(L_n)$. De volgende plaatsen worden bepaald door de *collision resolution policy*. In bovenstaand voorbeeld start de peil-sequentie voor de sleutel W_{23} op plaats 2, gevolgd door de plaatsen 1, 0, 6, 5, 4 en 3. Een peil-sequentie is zodanig georganiseerd dat elke plaats in de tabel precies eenmaal onderzocht wordt.
Om er zeker van te zijn dat er altijd een lege plaats gevonden wordt bij elke peil-sequentie, wordt een *volle* tabel gedefinieerd als een tabel met exact één vrije plaats. Op die manier zal dus steeds tijdens het aflopen van de peil-sequentie een lege plaats gevonden worden. Het is niet nodig om tijdens het aflopen het aantal bezochte elementen te tellen om te kunnen bepalen wanneer er kan gestopt worden met zoeken.
Er zijn verschillende politieken om een botsing op te lossen. In bovenstaand voorbeeld werd de peil sequentie van sleutel L_n gevormd door af te tellen vanaf het hash adres $h(L_n)$ en na 0 werd verdergegaan met het laatste element van de tabel. Men spreekt van een *probe decrement* van 1, omdat voor het vinden van de volgende peillocatie 1 afgetrokken wordt van de actuele peillocatie. Dit proces wordt ook *linear probing* genoemd.
Omdat de elementen op lege plaatsen toegevoegd worden, spreekt men van *open addressing with linear probing*. Wanneer de tabel bijna vol is, resulteert deze methode in een slechte performantie.

Men kan ook een niet-linaire peilmethode gebruiken om voor verschillende sleutels L_n verschillende probe decrements te berekenen. Omdat de probe decrement hier berekend wordt met behulp van een hash functie ($p(L_n)$), spreekt men ook van *dubbel hashing*.
Een mogelijke methode om de decrement te berekenen is het quotiënt te nemen van de deling van n door 7. Wanneer dit quotiënt echter gelijk is aan nul, wordt een decrement gelijk aan 1 genomen. Wiskundig kan deze functie geschreven worden als:

$$p(L_n) = \max(1, n/7).$$

De probe decrement van W_{23} is gelijk aan $\max(1, 23/7)$ of 3. Voor B_2 bekomt men een probe decrement van 1, omdat het quotiënt gelijk is aan nul.
Nadat de elementen J_{10}, B_2, S_{19} en N_{14} toegevoegd zijn aan een lege tabel treedt een eerste botsing op bij het toevoegen van element W_{23}. Omdat $h(W_{23}) = 2$, zou dit element toegevoegd moeten worden op plaats 2. Deze plaats is echter reeds bezet. De probe decrement wordt berekend: $p(W_{23}) = 3$, dus wordt naar plaats (2-3) 6 gekeken. Deze plaats is nog vrij, zodat het element kan toegevoegd worden.

0	1	2	3	4	5	6
N_{14}		B_2	J_{10}		S_{19}	W_{23}

hash adres: $h(W_{23}) = 2$
probe decrement $p(W_{23}) = 3$

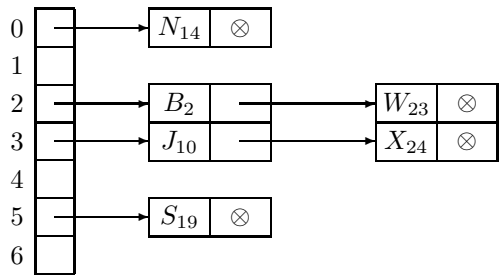
Ten slotte moet element X_{24} nog toegevoegd worden. Omdat plaats 3 ($h(X_{24})$) bezet is wordt de probe decrement berekend, $p(X_{24}) = 3$. De plaats (3-3) 0 is ook bezet, dus wordt verder gekeken naar plaats (0-3) 4. Hier kan het element toegevoegd worden.

0	1	2	3	4	5	6
N_{14}		B_2	J_{10}	X_{24}	S_{19}	W_{23}

$h(X_{24}) = 3$: niet leeg
$p(X_{24}) = 3$, dus 0 en dan 4

Het voordeel van dubbel hashing is dat bij een botsing van twee sleutels op een initieel hash adres, de peil sequenties voor deze twee sleutels meestal verschillend zijn, omdat een verschillende probe decrement gebruikt wordt. Bijvoorbeeld de sleutels J_{10} en X_{24} geven het zelfde hash adres, namelijk 3. Maar omdat $p(J_{10}) = 1$ en $p(X_{24}) = 3$ zijn de twee peil sequenties verschillend. Het effect is dat meestal sneller een lege plaats zal gevonden worden dan wanneer bij een botsing dezelfde peil sequenties gebruikt worden.

Een andere manier om botsingen op te lossen is gebruik te maken van gelinkte lijsten: *collision resolution by separate chaining*. Het resultaat van het toevoegen van de elementen J_{10}, B_2, S_{19}, N_{14}, W_{23} en X_{24} is weergegeven in volgende figuur.



De tabel bevat pointers naar gelinkte lijsten. Op zo'n gelinkte lijst zitten de elementen waarbij de hashfunctie toegepast op de sleutel een zelfde waarde geeft.

14.3 Parameters

Een botsing treedt op wanneer bij het toevoegen van twee verschillende sleutels, K en L, aan de tabel, deze beide sleutels hetzelfde hash adres hebben: $h(K) = h(L)$. Om een goede performantie te hebben, dit wil zeggen niet teveel botsingen, moeten bij een implementatie van deze hash techniek enkele parameters goed gekozen worden.
Een eerste parameter is de lengte van de tabel. De tabel moet zeker groter zijn dan het te verwachte aantal op te slaan elementen. De *load factor*, λ, van een hash tabel met lengte M waarvan N plaatsen bezet zijn is $\lambda = N/M$. Hoe kleiner de load factor, hoe minder kans op botsingen. Voor de lengte van de tabel, wordt normaal een priemgetal gekozen. Dit heeft ook te maken met een goede hashfunctie.
Een tweede keuze die moet gemaakt worden is de aard van de hash functie. Een goede hash functie zal de verschillende sleutels op een randomachtige maar uniforme manier over de volledige tabel verdelen. Daarnaast moet de hash functie snel te berekenen zijn.
De *deling* methode is in bovenstaande voorbeelden gebruikt. Het hash adres wordt berekend als de rest bij deling van de sleutel door de lengte van de tabel.

$$h(K) = K \,\%\, M \qquad \text{bij tabellengte } M$$

Indien deze lengte een macht van 2 zou zijn, komt dit neer op het afzonderen van de laagste bits van de sleutel. Het gevaar bestaat dat bij sommige sleuteltypes vrij dikwijls dezelfde rest gevonden wordt, wat dus leidt naar botsingen. Om een random achtige hash functie te hebben is het beter de deling uit te voeren met een priemgetal.
Er bestaan nog verschillende andere vormen voor de hash functie, bijvoorbeeld *folding*, *middle-squaring* en *truncation*.
Bij *folding* (vouwen) wordt de sleutel verdeeld en de verschillende delen worden opgeteld. Bij een 9-cijfer sleutel kunnen drie delen gemaakt worden en deze kunnen opgeteld worden.

$$K = 013|402|122 \qquad \rightarrow \qquad 013 + 402 + 122 \quad \rightarrow \quad 537$$

In plaats van optellen kan ook een vermenigvuldiging, een aftrekking of een deling gebruikt worden.
Bij *middle-squaring* worden de middelste cijfers uit de sleutel gehaald en gekwadratteerd. Indien het resultaat nog groter is dan de lengte van de tabel, kan de operatie herhaald worden.

$$K = 013|402|122 \qquad \rightarrow \qquad 402^2 \quad \rightarrow \quad 161604 \qquad \rightarrow \qquad 16^2 \quad \rightarrow \quad 256$$

Bij *truncation* wordt een deel van de sleutel verwijderd en wordt de rest als adres overgehouden,

$$K = 013402|122 \qquad \text{drie laatste cijfers} \quad \rightarrow \quad 122$$

Deze methode vraagt heel weinig rekentijd maar de sleutels worden meestal niet op een uniforme manier over de tabel verspreid.
Een volgende parameter is de probe sequentie. Lineair peilen is eenvoudig, maar niet goed. Bij dubbel hashing moet een tweede hash functie gekozen worden. Bij de *deling* methode kan het quotiënt bij deling van de sleutel door de lengte van de tabel gebruikt worden, tenzij dit quotiënt nul is.

$$p(K) = \max(1, K/M) \qquad \text{bij tabellengte } M$$

Vanuit praktisch standpunt is het nuttig om de waarde van $p(K)$ te reduceren tot het bereik $1 : M - 1$. Dit kan gebeuren door een waarde $p(K)$ die groter is dan M te vervangen door $\max(1, p(K)\%M)$.
Een peil sequentie moet alle plaatsen van de tabel nagaan. Voor lineair peilen is dit het geval. Bij een tabellengte van 7 en een begin hashadres gelijk aan 4, worden achtereenvolgens de plaatsen 4, 3, 2, 1, 0 en dan 6 en 5 bekeken totdat een vrije plaats gevonden wordt. Bij dubbel hashing is dit echter niet zo duidelijk. Maar wanneer de tabellengte M en de probe decrement $p(K)$ *relatief priem* zijn, zal de peil sequentie de volledige tabel aandoen. Twee getallen zijn relatief

priem wanneer ze alleen 1 als gemeenschappelijke deler hebben. Dus wanneer de tabellengte een priemgetal is, kan voor de probe decrement gelijk welk getal groter dan of gelijk aan 1 gebruikt worden. Een andere combinatie dat een goede peil sequentie geeft is een tabellengte gelijk aan een macht van twee en een oneven getal voor de probe decrement.
Een *cluster* is een rij van aaneensluitende bezette plaatsen in een hash tabel. In zo'n rij zit dus geen enkele vrije plaats meer. De lineaire peil methode geeft aanleiding tot *primary clustering*. Wanneer een aantal sleutels op dezelfde plaats botsen en wanneer lineair peilen gebruikt wordt, worden deze botsende sleutels toegevoegd op plaatsen vlak voor de botsplaats. Op die manier kunnen op een aantal plaatsen in de tabel clusters ontstaan. Zo'n cluster groeit steeds sneller en sneller omdat de kans op een botsing bij een nieuw toe te voegen element steeds groter wordt. Tijdens die groei komen sommige kleine clusters samen wat aanleiding geeft tot grote clusters. Het proces versterkt dus zichzelf.
Dubbel hashing daarentegen geeft geen aanleiding tot *primary clustering*. Bij een botsing wordt het element niet noodzakelijk toegevoegd vlak naast de botsplaats. Er ontstaat dus niet altijd een cluster van twee elementen.

14.4 Een voorbeeld

Gegeven een aantal elementen waarvan de sleutel (K) telkens bestaat uit drie letters, bijv. "MEC". Deze elementen moeten in een hash tabel met lengte $M = 11$ geplaatst worden. De hash functie $h(K)$ wordt berekend door de sleutel te interpreteren als een "basis 26" geheel getal. Bij een sleutel $K = X_2X_1X_0$, krijgt elk alfabetisch teken X_i een geheeltallige waarde tussen 0 en 25. Voor 'A' wordt de waarde 0 gebruikt, 'B' de waarde 1 en zo verder tot 'Z' met een waarde 25. Hiermee kan een drie-letter code omgezet worden van een basis-26 getal naar een decimaal getal:

$$\text{Basis26Waarde}(K) = X_2 \times 26^2 + X_1 \times 26^1 + X_0 \times 26^0$$

Bijvoorbeeld:

$$\begin{aligned} \text{Basis26Waarde}(MEC) &= 12 \times 26^2 + 4 \times 26^1 + 2 \times 26^0 \\ &= 8112 + 104 + 2 \\ &= 8218 \end{aligned}$$

Op basis van deze basis-26 waarde van de sleutel kunnen de hash functies gedefinieerd worden:

$$\begin{aligned} h(K) &= \text{Basis26Waarde}(K) \ \% \ 11 \\ p(K) &= \max(1, (\text{Basis26Waarde}(K)/11) \ \% \ 11) \end{aligned}$$

Dus het hash adres van "MEC" is $h(MEC) = 8218\%11 = 1$. Bij lineair peilen wordt een $p(MEC)$ gelijk aan 1 genomen, zoals voor alle andere sleutels. Bij dubbel hashing is de probe decrement $p(MEC) = \max(1, (8218/11)\%11) = 10$.
In volgende tabel is de volgorde waarin de elementen moeten toegevoegd worden weergegeven.

K	basis26(K)	$h(K)$	$p(K)$ (lineair)	$p(K)$ (dubbel)
MEC	8218	1	1	10
ANT	357	5	1	10
KOR	7141	2	1	1
BRU	1138	5	1	4
ZOL	17275	5	1	8
GEN	4173	4	1	5
LEU	7560	3	1	5
GEE	4164	6	1	4
TUR	13381	5	1	6

In de volgende figuren wordt de hash tabel weergegeven na het toevoegen van de eerste drie, de eerste zes en tenslotte alle elementen. Bij lineair peilen ontstaat reeds een vrij grote cluster na zes toevoegingen. Bij dubbel hashing worden de elementen meer uniform over de tabel verdeeld. Na zes toevoegingen zijn er drie clusters.

Lineair:

	na 3 elementen	na 6 elementen	na 9 elementen
0		GEN	GEN
1	MEC	MEC	MEC
2	KOR	KOR	KOR
3		ZOL	ZOL
4		BRU	BRU
5	ANT	ANT	ANT
6			GEE
7			
8			
9			TUR
10			LEU

Dubbel hashing:

	na 3 elementen	na 6 elementen	na 9 elementen
0		ZOL	ZOL
1	MEC	MEC	MEC
2	KOR	KOR	KOR
3			LEU
4		GEN	GEN
5	ANT	ANT	ANT
6			GEE
7			
8		BRU	BRU
9			
10			TUR

Het programma:

```
/*   hash.c : sleutel op basis van basis26 waarde    */
#include <stdio.h>
#include <stdlib.h>

#define NRIJ     11                                  /* priemgetal ! */
#define LEN      4
char *invoer[] =
      { "MEC", "ANT", "KOR", "BRU", "ZOL", "GEN", "LEU", "GEE", "TUR", 0};

void voegtoe( char a[][LEN], int  m, char inv[], int (*decfun)(char [], int) );
int plinfun( char inv[], int m );
int pdubfun( char inv[], int m );
void druktab( char a[][LEN], int m );
int verbose = 0;
int aantal_botsingen = 0;

int main( int argc, char *argv[])
{
```

```
        char    tabel[NRIJ][LEN];
20      int     m = NRIJ;
        int     i = 0;
22      int     c = 0;
        int     (*decfun)(char inv[], int m);
24
        decfun = plinfun;
26      while ( (c=getopt(argc,argv,"ldv")) != EOF )
        {
28          switch ( c )
            {
30              case 'l': decfun = plinfun;
                          break;
32              case 'd': decfun = pdubfun;
                          break;
34              case 'v': verbose++;
                          break;
36          }
        }
38      memset(tabel, '\0', NRIJ*LEN);
        while ( invoer[i] )
40      {
            voegtoe(tabel, m, invoer[i], decfun);
42          druktab(tabel, m);
            i++;
44      }
        printf("Aantal botsingen : %d\n", aantal_botsingen);
46  }

48  int basis26( char inv[] )
    {
50      return ((inv[0]-'A')*26*26) + ((inv[1]-'A')*26) + inv[2]-'A';
    }
52  int hfun( char inv[], int   m )                 /* deling methode */
    {
54      int i;

56      i = basis26(inv);
        return i%m ;
58  }
    int plinfun( char inv[], int m )                /* linear probing */
60  {
        return 1;
62  }
    int pdubfun( char inv[], int m)                 /* dubbel hashing */
64  {
        int i;
66
        i = basis26(inv);
68      i = (i/m)%m;
        return i==0 ? 1 : i ;
70  }
    void druktab( char a[][LEN], int m )
72  {
```

```
        int i;
74
        for(i=0; i<m; i++)
76          printf("%3d : %-4.4s\n", i, a[i] );
    }
78  void voegtoe( char a[][LEN], int  m, char inv[], int (*decfun)() )
    {
80      int adr, i;
        int dec;
82
        i = adr = hfun(inv, m);
84      dec = (*decfun)(inv, m);
        if ( verbose )
86          printf("%-4.4s : %3d : %3d\n", inv, adr, dec );
        while ( *a[i] )                          /* zolang er botsing is */
88      {
            if ( strncmp( a[i], inv, LEN) == 0 )
90          {
                /* element reeds aanwezig */
92              return;
            }
94          aantal_botsingen++ ;
            i -= dec;                          /* volg probe sequentie */
96          if ( i < 0 )
                i += m;
98      }
        strncpy( a[i], inv, LEN);                /* vul lege plaats in */
100 }
```

Merk op dat dit een programma is met *opties* en *argumenten* (`a.out -l bestand`).

14.5 Denktaak

Veronderstel dat hashfunctie in het voorbeeld vervangen wordt door nevenstaande functie. Bereken de index in de hashtabel voor de eerste drie elementen (MEC, ANT, KOR), met m gelijk aan 11. Aangezien i van type **char** is, gebeuren de berekeningen in 8 bits. Treedt er bij deze drie elementen reeds een botsing op?

```
int hfun( char inv[], int m )
{
    char i;
    i = (inv[0]<<2) ^ (inv[1]>>1);
    return i%m ;
}
```

15 C: diversen

15.1 Operatoren

15.1.1 Een ternaire operator

Aan een variabele een waarde toekennen die afhankelijk is van een conditie kan met behulp van het `if` statement:

```
if ( a < b )
    z = a + 1;
else
    z = b - 1;
```

Dit kan op een kortere manier met behulp van de `?:` operator:

```
z =  a<b ? a+1 : b-1;
```

Indien a kleiner is dan b, dan is de waarde van de expressie $a+1$, anders $b-1$. Deze waarde wordt toegekend aan de variabele z. De operator heeft drie operands : `a<b`, `a+1` en `b-1`.
In plaats van de waarde van de expressie toe te kennen, kan ze natuurlijk ook in een andere expressie gebruikt worden.

```
/*
 * tabel.c : 1-dim rij afdrukken in tabel vorm
 */
void rij2tab(int a[], int n, int m)
{
    int i;

    for (i=0; i<n; i++)
        printf("%4d%c", a[i], ((i+1)%m ? ' ' : '\n') );
    if ( n%m )
        printf("\n");
}
```

15.1.2 De komma operator

Expressies kunnen onderling gescheiden worden met een komma. Deze komma wordt als een operator opgevat: de expressies worden van links naar rechts uitgewerkt en de laatste, dus de meest rechtse bepaalt de waarde van het geheel.

```
/*
 * invers.c : omdraaien van de elementen in een array
 */
void draai(int a[], int b[], int n)
{
    int i, j;

    for (i=0, j=n-1; i<n; i++, j--)
        b[j] = a[i];
}
```

Met deze operator kan men er ook voor zorgen dat bij een iteratie-lus de beëindigingstest niet in het begin en ook niet op het einde, maar ergens in het midden van de lus gebeurt.

```c
/*
 * lustest.c : eindtest in het midden
 */
#include <stdio.h>
void main(void)
{
    int n;
    int som;

    som = 0;
    while ( scanf("%d%*c", &n), n>0 )
        som += n;
    printf("De som is %d\n", som);
}
```

De lus bestaat uit drie acties:

1. het lezen van een getal;
2. de test op het einde van de lus;
3. het verhogen van `som` met `n`.

Hetzelfde zou kunnen gerealiseerd worden met een `break`. Maar volgens sommigen is dit geen stijlvolle constructie.

Merk op dat de komma-operator een zeer lage prioriteit heeft:

```c
int a = 5, b=8 ;
int r;

r = a++, b++;
```

15.2 Data-types

15.2.1 Het enumeratie type

Een *enum*eratie is een groep symbolische constanten. Eventueel krijgt de groep als geheel ook een naam, door middel van het *tag* veld, net zoals bij structures.

```c
enum { maandag, woensdag, vrijdag };
enum tussen {dinsdag, donderdag, zaterdag, zondag };
```

De eerste declaratie geeft gewoon een opsomming van namen. Inwendig worden deze namen gecodeerd als natuurlijke getallen: 0, 1 en 2. Ook bij de tweede declaratie worden deze waarden (0,1 en 2) toegekend en voor zondag de waarde 3.
De definitie van een variabele:

```c
enum { maandag, woensdag, vrijdag } dag;
enum tussen ookdag;
```

Omwille van het *tag* veld bij de tweede declaratie, kan de definitie van `ookdag` korter gebeuren.
Het gebruik:

```c
dag = maandag;
dag++;
ookdag = zondag;
```

Omdat de onderliggende waarden van de variabelen van het type `int` zijn, kunnen minder fraaie toekenningen geschreven worden. De compiler zal hiervoor hoogstens een fout genereren.

```c
ookdag = dag;
dag = 2;
dag++;                    /*  ???  */
```

Aan de elementen van de enumeratie wordt geen adresseerbare geheugenruimte toegekend. Het is niet toegestaan de `&` operator (het adres van) toe te passen op een enumeratie-element.

Opmerking. Het gebruik van dit type is een beetje persoonsgebonden. Sommige programma-ontwerpers vinden dit een handige methode om integrale constanten te groeperen die hetzij een programmafunctie gemeen hebben hetzij een set van gelijkaardige waarden definiëren.

```
enum uitzonderingen { e_internal, e_external, e_nuldeling } fout;
enum boolean { onwaar, waar } gevonden;
```

Het gebruik van enumeratie-constanten is een handige manier om het programma te documenteren. Langs de andere kant is het niet aangewezen constanten van uiteenlopende aard onder te brengen in een enumeratie. In dat geval verdienen aparte `#define`s de voorkeur.

Voorbeeld. Gebruik van `enum` om de verschillende toestanden van een variabele te beschrijven.

```
/*
 * dagop.c : voorbeeld van enum
 */
#include <stdio.h>
enum markt { maandag, woensdag, vrijdag } ;
enum tussen { dinsdag, donderdag, zaterdag, zondag } ;
void drukmar(enum markt d);
void druktus(enum tussen d);
int main(int argv, char *argv[])
{
    enum markt dag;
    enum tussen ookdag;

    for ( dag = maandag; dag <=4; dag++ )
    {
        drukmar(dag);
        ookdag = dag-1;
        druktus(ookdag);
    }
}
void drukmar(enum markt d)
{
    switch (d)
    {
        case maandag:
            printf(" maandag (%d)   ", d);
            break;
        case woensdag:
            printf("woensdag (%d)   ", d);
            break;
        case vrijdag:
            printf(" vrijdag (%d)   ", d);
            break;
        default:
            printf("     ..    (%d)   ", d);
            break;
    }
}
void druktus(enum tussen d)
{
    switch (d)
    {
```

```
        case dinsdag:
            printf("  dinsdag (%d)\n", d);
            break;
        case donderdag:
            printf("donderdag (%d)\n", d);
            break;
        case zaterdag:
            printf(" zaterdag (%d)\n", d);
            break;
        case zondag:
            printf("   zondag (%d)\n", d);
            break;
        default:
            printf("     ....  (%d)\n", d);
            break;
    }
}
```

```
Het resultaat:      maandag (0)       ....  (-1)
                   woensdag (1)     dinsdag (0)
                    vrijdag (2)   donderdag (1)
                       ..   (3)    zaterdag (2)
                       ..   (4)      zondag (3)
```

15.2.2 Structuren met bitvelden

Om de benodigde ruimte voor een structure zo klein mogelijk te houden, kan men in een structure een veld definiëren dat slechts één of enkele bits groot is:

```
typedef struct
        {
            char   naam[20];
            int    leeftijd;
            float  gewicht;
            unsigned vrouwelijk: 1;
            unsigned gehuwd: 1;
        } Mens;
```

Voor de velden `vrouwelijk` en `gehuwd` zijn er maar twee mogelijke waarden: waar of niet waar. Er is voor elk van deze velden slechts één bit nodig. De hoeveelheid bits wordt aangegeven door het getal achter het dubbelpunt na de veldnaam. Dit aantal ligt tussen 1 en de lengte van een machinewoord (dit is normaal gelijk aan 32).
In dit voorbeeld wordt aangenomen dat een bit gelijk aan 1 overeenkomt met de waarde waar.

```
Mens aa;
aa.vrouwelijk = 1;          /* vrouw */
aa.gehuwd = 0;              /* ongehuwd */
```

Wanneer in plaats van `gehuwd` ook andere toestanden mogelijk zijn, zoals gescheiden of weduwnaar (weduwe), zijn meerdere bits nodig. In de definitie van het type `Mens` wordt het veld `gehuwd` gewijzigd:

```
unsigned status: 2;
```

In dit geval zijn twee bits genoeg; deze hebben nu volgende betekenis:

```
Mens aa;
aa.status = 0;        /* ongehuwd */
aa.status = 1;        /* gehuwd */
aa.status = 2;        /* gescheiden */
aa.status = 3;        /* weduwnaar/weduwe */
```

Deze codering van de verschillende toestanden is niet handig en bevordert ook niet de leesbaarheid van het programma (tenzij men overal in commentaar de betekenis bijvoegt). Om de leesbaarheid te bevorderen, kan men gebruik maken van het `enum` data type:

```
enum state { ongehuwd, gehuwd, gescheiden, wedu };
aa.status = gescheiden;
```

Omdat bitvelden slechts een onderdeel van een machinewoord zijn, hebben zij geen adres: de `&` operator (het adres van) kan dus niet op een bitveld toegepast worden!

Voorbeeld van de combinatie van een bitveld en een enumeratie type.

```
/*
 * bitveld.c
 */
#include <stdio.h>
enum state { ongehuwd, gehuwd, gescheiden, wedu };
typedef struct
        {
          enum state status;
          unsigned rest: 30;
          int     w ;
        } S;
int main(int argc, char *argv[])
{
    S a;
    enum state jos = ongehuwd;

    a.w = 4;      a.status = wedu;        a.rest = 2;
    printf ("w  %d  s %d  r %d\n", a.w, a.status, a.rest );
    a.w = 8;      a.status--;             a.rest += 2;
    printf ("w  %d  s %d  r %d\n", a.w, a.status, a.rest );
    a.w = 16;     a.status = jos;         a.rest <<= 2;
    printf ("w %d  s %d  r %d\n", a.w, a.status, a.rest );
}
```

```
Resultaat:        w  4  s 3  r 2
                  w  8  s 2  r 4
                  w 16  s 0  r 16
```

15.2.3 Union

De geheugenruimte die door een *structure* in beslag genomen wordt, is tenminste gelijk aan de som van de ruimte van de componenten. Alle componenten van een structure zijn gelijktijdig fysiek aanwezig. Soms wenst men bepaalde alternatieven van een object te bewerken, die niet gelijktijdig (kunnen) voorkomen. In dat geval moet de benodigde geheugenruimte niet groter zijn dan de grootste van de verschillende varianten. Dit wordt gerealiseerd met een `union`.
Declareren gebeuren zoals bij een `structure`: met of zonder een *tag*-veld; declaratie en definitie van een variabele samen of apart; creatie van een nieuw type via `typedef`:

```
typedef union
        {
              char   naam[16];
              int    leeftijd;
              double gewicht;
        } Eigenschap;
```

Ook het gebruik is volledig analoog aan het gebruik van een structure:

```
Eigenschap u;
int  len;
u.leeftijd = 37;
u.gewicht = 103.5;
len = sizeof(Eigenschap);
```

Opmerking. Bij een union kan een zelfde geheugenplaats met behulp van verschillende namen aangesproken worden. Dit noemt men *aliasing*. Bij ondeskundig gebruik kan dit leiden tot programmeerfouten die in sommige gevallen zeer moeilijk te vinden zijn. Voor de meeste problemen is het gebruik van unions niet nodig en daarom is het aan te raden deze constructie niet te gebruiken. Zowel structures met bitvelden als unions stammen uit de tijd dat RAM geheugens nog zeer duur waren. Elke mogelijke besparing op het vlak van ruimte voor variabelen in een programma, was in die tijd bijzonder welkom.

15.3 Keywords

15.3.1 Type qualifiers

Register. De geheugenklasse `register` meldt aan de computer dat de bijbehorende lokale variabelen opgeslagen moeten worden in high-speed geheugenregisters, mits zulks fysisch en semantisch mogelijk is. Dit heeft alleen zin als snelheid een essentiële factor is en voor die variabelen waarop het meest frequent operaties uitgevoerd worden.
Voorbeeld:

```
/*
 *   vbreg.c : programma met een register-variable
 */
#include <stdio.h>
#define LIMIET 1000

int main(int argc, char *argv[])
{
    register int i;      /* register variabele  */

    for (i=0; i<LIMIET; i++)
    {
        printf("%8d\n", i);
    }
}
```

De variabele `i` is een lokale variabele die omwille van efficiëntie in een register gestopt wordt. Voor de rest is het een *lokale* variabele. De register toekenning gebeurt wanneer de functie start en wordt ongedaan gemaakt bij het einde van de functie.

Auto. Omdat het aantal registers beperkt is, worden de meeste lokale variabelen ergens in het werkgeheugen bewaard. Wannneer de functie start, wordt een plaats toegewezen aan de variabele. Dit gebeurt automatisch, vandaar dat men soms ook van *automatische* variabelen spreekt. Men kan dit expliciet in de declaratie aangeven:

```
auto int    i;
auto double f;
```

Dit wordt echter bijna nooit gedaan, meestal wordt de `auto` qualifier gewoon weggelaten.

Daarnaast biedt ANSI-C de mogelijkheid een extra *type-kwalificatie* te geven:

const : de variabele kan na initialisering nooit gewijzigd worden (een alternatief voor een naamconstante); wordt ook vaak gebruikt bij het specificeren van formele parameters;

```
const int weeklengte = 7;
```

volatile : (te beschouwen als het omgekeerde van `const`) deze variabelen kunnen in principe ook gewijzigd worden door bijvoorbeeld de hardware (het gebruik is systeemafhankelijk).

15.3.2 Het `goto` statement

In *oude* programma's kan men soms nog het `goto` statement terugvinden:

```
goto identifier;
```

waarbij de `identifier` *label* genoemd wordt. Er moet namelijk in dezelfde functie een statement voorkomen dat voorafgegaan wordt door deze *label*.

```
void main(void)
{
        register int i = 0;
        const int n = 10;

    opnieuw:
        i++;
        ...
        if ( i > n )
            goto gedaan;
        goto opnieuw;
    gedaan:
        printf("Het is gedaan\n");
        exit(0);
}
```

Sinds de tweede helft van de jaren zeventig (opkomst van *gestructureerd programmeren*) wordt het ten stelligste afgeraden dit statement te gebruiken. Er bestaan steeds andere, meer *leesbare* manieren om het hetzelfde effect te bekomen.
Een `goto` statement is ook vrij beperkt: er kan alleen een sprong binnen een functie uitgevoerd worden. Men kan bijvoorbeeld niet uit een een hulpfunctie springen naar het einde van de functie `main`. Om de uitvoering van een programma (voortijdig) te beëindigen, kan men gebruik maken van de functie `exit()`. Tijdens de uitvoering van `exit` worden alle geopende bestanden netjes gesloten.

15.4 De preprocessor

15.4.1 Directieven

Voordat de compiler het echte werk doet, wordt het bronbestand behandeld door de preprocessor. De C-preprocessor behandelt die lijnen die in de eerste kolom beginnen met het `#` symbool.

Volgende *compiler control lines* zijn onder meer beschikbaar.

define : hiermee kan aan een symbolische naam een constante toegekend worden; daarnaast kan hiermee ook een *macro* gedefinieerd worden.

undef : aangeven dat de compiler vanaf dan de definitie van de bijhorende symbolische naam moet vergeten.

include : geeft aan dat het bestand moet tussengevoegd worden in het bronbestand. Wanneer de bestandsnaam tussen dubbel quotes (") staat, wordt het bestand eerst gezocht in de huidige directory en daarna in de *algemeen toegankelijke* directories (systeemafhankelijk). Een naam tussen <> geeft aan dat het bestand alleen in de algemeen toegankelijke directories moet gezocht worden. Deze bestanden worden dikwijls *header* bestanden genoemd, vandaar dat ze meestal een extensie `.h` hebben.

ifdef : om het compileren van bepaalde gedeelten van de programmatekst te laten afhangen van een voorwaarde (*conditionele compilatie*).

line : met twee argumenten: het eerste argument is een getal vanaf waar de compiler de volgende programmaregels zal nummeren; het tweede argument is een naam die vanaf dan de compiler als naam van het bestand zal gebruiken.

15.4.2 Macro's

Een macro is te vergelijken met een functie. Er is een macro-definitie

```
#define max(a,b) ((a) > (b) ? (a) : (b))
```

en een macro-oproep `y = 2 * max(i+1, j-1) + 3;`

Door de preprocessor zal de macro-oproep (*in line*) vervangen worden door de macro-definitie (*macro-expansie*):

```
y = 2 * ((i+1) > (j-1) ? (i+1) : (j-1)) + 3;
```

De C-compiler krijgt dus een expressie te verwerken. Er wordt geen functie-oproep met doorgeven van argumenten (naar parameters) uitgevoerd.
Merk op dat er in de macro-definitie nogal wat haakjes gebruikt worden. Dit heeft te maken met de prioriteit van de operatoren. Met de definitie

```
#define kwadraat(x) x * x
```

en de oproep `x = kwadraat(a+b);`
wordt de macro-expansie `x = a+b * a+b;`
Dit is echter niet de bedoeling. Er zijn dus haakjes nodig

```
#define kwadraat(x) ((x) * (x))
```

De macro-expansie wordt dan `x = ((a+b) * (a+b));`

Het open-haakje dat het begin van de parameters aangeeft, moet direct volgen op de macro-naam (zonder spatie). Anders zou de parameterlijst gebruikt worden als het eerste deel van de definitie.

Wanneer een macro-definitie te lang wordt voor één lijn, dan kan de definitie gesplitst worden over meerdere lijnen. Dit gebeurt door de eerste lijn te eindigen met een backslash (\).

In het headerbestand `<ctype.h>` zijn een aantal macro's gedefinieerd om het type teken te testen. Omdat op tekenvariabelen bewerkingen kunnen uitgevoerd worden zijn deze macro's vrij eenvoudig. Bijvoorbeeld:

```
#define isdigit(c) ((c)>='0'&&(c)<='9')
#define tolower(c) ((unsigned char)(c)-'A'+'a')
```

15.4.3 Conditionele compilatie

```
#define VROEGER
void main(void)
{
        ...
#ifdef VROEGER
        /* programmadeel A */
#else
        /* programmadeel B */
#endif
        ...
}
```

Wanneer `VROEGER` gedefinieerd is (zoals in het voorbeeld), zal de compiler niet programmadeel B compileren; alleen programmadeel A wordt opgenomen in de uitvoerbare versie. Wanneer `VROEGER` niet gedefinieerd zou zijn, zou alleen programmadeel B gecompileerd worden.

16 Het testen van software

16.1 Doelstellingen

Testen is het uitvoeren van een programma met de intentie een fout te vinden. Een goede test heeft een hoge waarschijnlijkheid om een nog niet-ontdekte fout te vinden. De test is succesvol wanneer zo'n niet-gekende fout inderdaad ontdekt wordt.
Merk op dat testen alleen kan aantonen dat er software fouten aanwezig zijn. Men kan met testen niet bewijzen dat er geen fouten zijn.

Principes.

1. Alle testen moeten relateren naar de klantvereisten: de belangrijkste fouten zijn deze waardoor het programma niet voldoet aan de behoeften van de klant.
2. Testen moeten reeds gepland zijn voor dat het effectief testen begint (na design fase en voor de programmatie).
3. Het Pareto principe: 80 % van alle ontdekte fouten hebben waarschijnlijk te maken met slechts 20 % van alle programma modules. Het probleem is deze module te ontdekken en deze dan grondig uit te testen.
4. Testen moet met *kleine* dingen beginnen en evolueren naar *testen* in het groot.
5. Alles testen is onmogelijk. Bijvoorbeeld, aantal verschillenden combinaties doorheen een reeks van `if ... else` structuren, kan erg oplopen.
6. Om de meest effectieve testen (grootste kans om een fout te vinden) te realiseren, moeten deze uitgevoerd worden door een onafhankelijke derde partij.

Testbaarheid.
Bij het ontwerp van een programma, moet al gedacht worden aan het vergemakkelijken van de testen achteraf.

1. Werkbaarheid: *hoe beter het werkt, hoe efficienter het kan getest worden.*
2. Observeerbaarheid: *wat je ziet, is wat je test.* Specifieke input moet specifieke uitvoer genereren. De systeemtoestand is zichtbaar tijdens het uitvoeren.
3. Controleerbaarheid: *hoe beter we de software kunnen controleren, hoe meer testen geautomatiseerd en geoptimaliseerd kunnen worden.* Reproduceerbaarheid: genereren van alle mogelijke output door beperkte input.
4. Opsplitsbaarheid (moduleerbaarheid): *door het bereik van de testen te beheersen, kunnen problemen sneller geisoleerd worden en kan het hertesten intelligenter gedaan worden.* Software modules kunnen onafhankelijk van elkaar getest worden.
5. Eenvoud (zowel functioneel, structureel als de code): *hoe minder er is om te testen, hoe sneller het te testen is.*
6. Stabiliteit: *hoe minder veranderingen, hoe minder wijzigingen getest moeten worden.*
7. Begrijpbaarheid (een goed begrepen ontwerp, de afhankelijkheden tussen interne, externe en gemeenschappelijke componenten zijn begrepen, technische documentatie is beschikbaar): *hoe meer informatie we hebben, hoe intelligenter we kunnen testen.*

Eigenschappen van een "goede" test:

1. Een goede test heeft een grote kans om een fout te vinden. Hiervoor moet de tester een goed begrip van de software hebben en moet hij proberen een mentaal beeld te vormen van hoe de software zou kunnen falen.

2. Een goede test is niet redundant. Het heeft niet veel zin om een test uit te voeren die dezelfde bedoeling heeft als een andere test.

3. Een goede test heeft een groot "bereik": uit een verzameling van testen die gelijkaardige bedoelingen hebben, die test die de grootste kans heeft om een hele klas van fouten te ontdekken.

4. Een goede test mag niet te eenvoudig maar ook niet te complex zijn. Door een reeks testen te combineren in één grote test, kunnen een aantal fouten niet opvallen. Elke test moet afzonderlijk uitgevoerd worden.

Verificatie en validatie.
Verificatie is de verzameling activiteiten om te verzekeren dat de software een specifieke functie correct implementeert: *zijn we het product juist aan het maken?*
Validatie is de verzameling activiteiten, verschillend van de voorgaande, om te verzekeren dat de gemaakte software aan de behoeften van de klant zal tegemoet komen: *zijn we het juiste product aan het maken?*

Een probleempje.
Wanneer het testen van de software begint, ontstaat er een belangenconflict. Vanuit een psychologisch standpunt zijn software analyse en ontwerp *constructieve* taken. De software ingenieur creëert een computerprogramma (code en data structuren) en bijhorende documentatie. Normaal resulteert zo'n werk in een zekere fierheid en dus gaat de software-ontwerper nogal sceptisch staan ten opzichte van mensen die gaan proberen aan te tonen dat het geleverde werk niet goed of juist is. Testen kan als een (psychologisch) *destructieve* taak beschouwd worden. Wanneer de software-ontwerper zijn eigen software moet testen, kan dit leiden tot de ontwikkeling van testen die gaan aantonen dat de software inderdaad werkt. Hoe dan ook er zullen fouten in de software zitten, en als deze niet tijdens het testen gevonden worden, zal de klant ze vinden met alle nadelige gevolgen vandien.

16.2 Technieken

Elk product kan op twee manieren getest worden.

1. Omdat men weet voor welke specifieke functies een product ontworpen is, kan men testen ontwerpen die aantonen dat elke functie daadwerkelijk operationeel is en waarbij ook naar fouten in elke functie gezocht wordt.
2. Wanneer de interne werking van een product gekend is, kan men testen ontwerpen om te verzekeren dat de interne operaties functioneren zoals gespecificeerd is.

De eerste benadering wordt *black-box* testen genoemd en heeft bij software voor een groot stuk te maken met de interface. Alhoewel deze testen ontworpen zijn om fouten te ontdekken, worden ze voornamelijk gebruikt om aan te tonen dat de software functies operationeel zijn: de invoer wordt juist aanvaard, de uitvoer wordt correct geproduceerd en de integriteit van de externe informatie (o.a. de gegevensbestanden) is gegarandeerd. Een black-box test onderzoekt een fundamenteel aspect van een systeem zonder veel te kijken naar de interne logische structuur van de software.
De tweede benadering, *white-box* testen, situeert zich op het nauwkeurige onderzoek van het procedurele detail. Logische paden doorheen de software worden getest door middel van specifieke verzamelingen van condities. De "status van een programma" kan op verschillende plaatsen onderzocht worden om na te gaan of de verwachte of vooropgestelde status overeenkomt met de actuele status.

16.2.1 White-box testen

Deze methode gebruikt de controle structuur van het procedurele ontwerp om testgevallen af te leiden.

- Verzekeren dat alle onafhankelijke paden in een module tenminste eenmaal worden uitgevoerd.
- Uittesten van alle logische beslissingen op hun *waar* en *onwaar* kanten.
- Uitvoeren van alle lussen op hun grenswaarden en tussen hun grenswaarden.
- Uittesten van alle interne datastructuren om hun validiteit te verzekeren.

Op het eerste zicht lijkt grondige white-box testen te leiden naar honderd procent correcte programma's. Voor een iets of wat betekenisvol programma is een *volledige* white-box test in praktijk niet realiseerbaar.

Code coverage is een methode om bij te houden welke functies, statements en vertakkingen bij keuze statements uitgevoerd worden tijdens de test. Eenvoudige implementatie:

```
int procedure_is_opgeroepen = 0;
int procedure(int a, int b)
{
    procedure_is_opgeroepen++;
     // statements voor de functionaliteit
     if ( a > 0 && b > 0 )
       return a + b;
    return a - b;
}
```

Er zijn vier niveaus:

- *function coverage*: wordt elke functie uitgevoerd;
- *statement coverage*: wordt elk statement uitgevoerd;
- *branch coverage*: heeft elk controle statement alle mogelijke waarden aangedaan, bijv. is bij **if** zowel true als false tak uitgevoerd;
- *condition coverage*: heeft elk onderdeel van een logische expressie alle mogelijke waarden gehad.

Wanneer een programma wat omvang heeft met een aantal geneste lussen en keuzestatements, dan worden white box testen met volledige code coverage bijna onmogelijk omdat deze veel te veel tijd vragen.

Men kan zich dus de vraag stellen of het niet beter is de testinspanning te spenderen om te verzekeren of aan alle programma-vereisten voldaan is (d.i. black-box testen). White-box testen zijn echter nodig omwille van de aard van software gebreken.

1. Logische fouten en onjuiste veronderstellingen zijn omgekeerd evenredig met de kans dat een bepaald programmapad zal uitgevoerd worden.
2. Men denkt dikwijls dat een bepaald logisch pad niet dikwijls uitgevoerd wordt, terwijl het effectief het normale geval is.
3. Tikfouten zijn random.

16.2.2 Black-box testen

Hier gaat alle aandacht naar de functionele vereisten van de software. Er worden verzamelingen van inputvoorwaarden ontwikkeld waarmee alle functionele vereisten van een programma kunnen uitgetest worden. Black-box testen is geen alternatief voor white-box testen maar complementair: er wordt een ander soort fouten mee ontdekt. Bijvoorbeeld: onjuiste of ontbrekende functies; interface fouten; fouten in de datastructuren of in de externe databank-toegang; performantiefouten; initialisatiefouten en fouten bij het programma-einde.

Black-box testen worden uitgevoerd in de latere stadia van de software-engineering cyclus. Ze worden ontworpen om antwoord te geven op volgende vragen:

- Hoe wordt functionele validiteit getest?
- Welke *klassen* van invoer resulteren in goede testgevallen?
- Is het systeem gevoelig voor specifieke invoerwaarden?
- Kunnen de grenzen van een data-klas goed gedefinieerd worden?
- Welke data-frequentie en data-volume kan het systeem verdragen?
- Wat is het effect van specifieke data combinaties op de werking van het systeem?

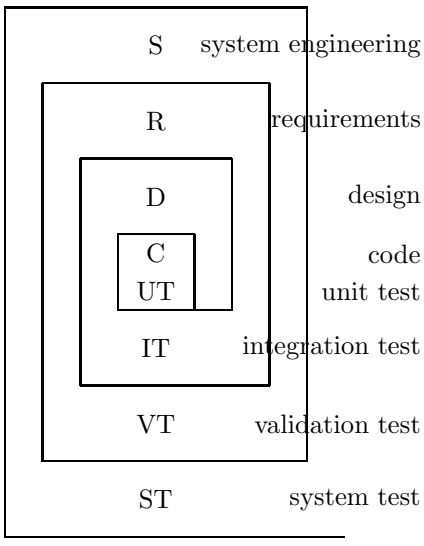
16.3 Strategie

Een strategie voor het testen van software integreert het ontwerp van de testgevallen in een goed-geplande reeks van stappen die resulteert in een succesvolle constructie van de software. Men krijgt dus een planning die bruikbaar is voor de software-ontwerper, de *quality assurance* dienst en de klant. Deze planning moet langs de ene kant flexibel genoeg zijn om de creativiteit niet in de weg te staan. Langs de andere kant kan er niet te veel van afgeweken worden zodat het management weet waar het aan toe is.
In het verleden zijn verschillende benaderingen voorgesteld. Deze hebben allen een aantal algemene karakteristieken:

- Testen begint op het module niveau en werkt naar buiten toe, naar de integratie van het volledige computergebaseerde systeem.
- Verschillende test-technieken zijn nodig op verschillende momenten.
- Het testen wordt uitgevoerd door de software-ontwerper en bij grote projecten door een onafhankelijke testgroep.
- Testen en debuggen zijn twee verschillende activiteiten.

16.3.1 De spiraal

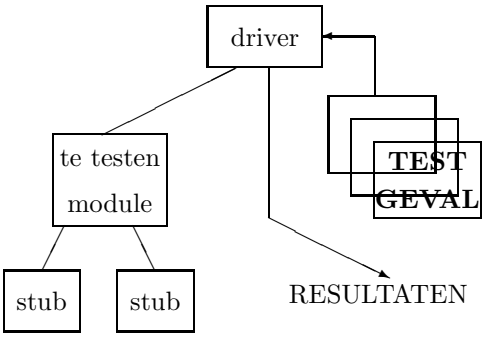
Het software engineering proces kan gezien worden als een spiraal.
Men begint met *system engineering* waarbij de rol van de software in het gehele systeem gedefinieerd wordt. In de *software behoefte analyse* wordt het informatie domein, de functies, het gedrag, de performantie en de beperkingen bepaald. Dan wordt het systeem *ontworpen* en tenslotte wordt de code geschreven (*programmatie*).
Software testen kan in deze contekst bekeken worden. *Unit testing* concentreert zich op elke eenheid van de software zoals die in de broncode geschreven is. Daarna komen de *integratie testen* waarbij het accent ligt op het ontwerp en de constructie van de software architectuur. Bij de *validatie testen* wordt nagegaan of de geschreven software voldoet aan de verwachtingen gedefinieerd in de behoefte analyse. Tenslotte moeten de *systeem testen* uitgevoerd worden waar de software met de andere systeemelementen als een geheel getest worden.



16.3.2 Unit test

Hier wordt gefocusseerd op de kleinste eenheid van software ontwerp, de module. Met de procedurele ontwerp beschrijving als gids, worden belangrijke controle paden getest om fouten te

ontdekken binnen de grenzen van de module. De unit test is normaal white-box georiënteerd en kan in parallel uitgevoerd worden voor verschillende modules.



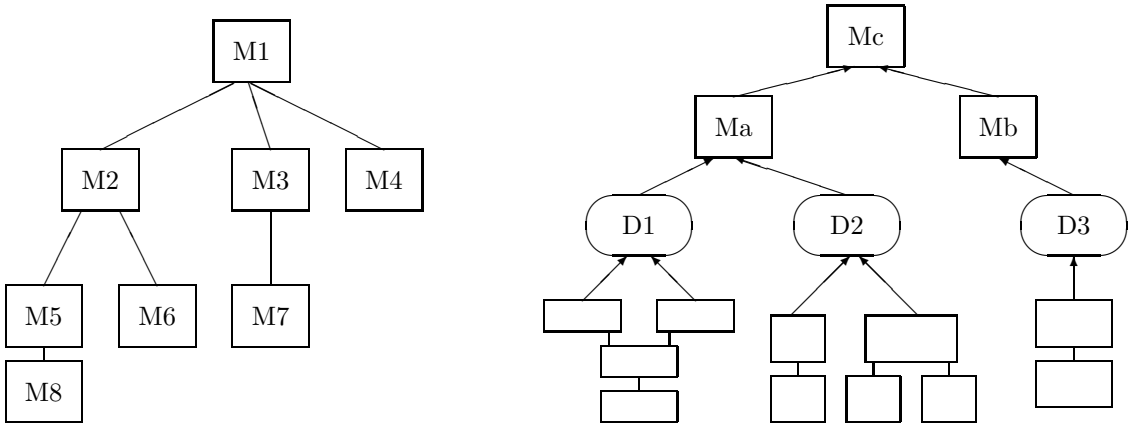
Unit testen worden meestal uitgevoerd onmiddellijk na het programmeren van de module. Omdat zo'n module meestal geen standalone programma is, moet *driver* en/of *stub* software ontwikkeld worden. Een *driver* is een hoofdprogramma dat test data inleest, deze doorgeeft aan de te testen module en de relevante resultaten afdrukt. *Stubs* dienen om ondergeschikte modules (die door de te testen module opgeroepen worden) te vervangen: ze doen een minimale data bewerking en keren terug naar de oproepende module.

Drivers en stubs betekenen overhead, deze software komt niet terecht in het uiteindelijke product. Door modules met een hoge cohesie te ontwerpen waarbinnen slechts één functie uitgevoerd wordt, kan deze test-ondersteunende software beperkt blijven. In vele gevallen echter kunnen modules niet op een eenvoudige manier als zelfstandige eenheid getest worden. In zo'n gevallen kan de test verschoven worden naar de integratie test fase waar een aantal modules samen getest worden.

16.3.3 Integratie test

Nadat elke module apart getest is door unit testen, is het niet noodzakelijk zo dat het gehele software systeem zal functioneren. Door de verschillende modules samen te voegen, kunnen er *interfacing* problemen ontstaan.
Data kan verloren gaan wanneer deze van een module naar een andere module moet doorgegeven worden. Bepaalde modules hebben ongewenste neveneffecten (bijvoorbeeld in de globale datastructuren) zodat andere modules niet meer functioneren. Door samenvoegen van procedures wordt toch niet de gewenste functie gerealiseerd. Onvolkomenheden die op zich aanvaardbaar zijn, worden door samenvoegen versterkt tot een onaanvaardbaar mankement.
Het integratie testen is een systematische techniek om de globale programmastructuur op te bouwen waarbij telkens testen uitgevoerd worden om fouten geassocieerd met interfacing te ontdekken.
Bij *non-incrementele integratie* gebruikt men de "big bang" benadering: alle unit-geteste modules worden in één keer bij elkaar gebracht in de definitieve structuur. Het volledige programma wordt als één geheel getest. Meestal resulteert dit in chaos. Een aantal fouten worden waargenomen maar verbeteren is moeilijk omdat het isoleren van de fout bemoeilijkt wordt door de omvang van het gehele programma. Eens bepaalde fouten verbeterd zijn, blijken er nieuwe fouten op te duiken en dit proces zet zich verder in een schijnbaar oneindige lus. Bij *incrementele integratie* wordt het programma opgebouwd en getest in kleine segmenten. Hierdoor zijn fouten gemakkelijker te isoleren en te verbeteren. Er is meer kans dat interfaces volledig uitgetest worden en een systematische test aanpak kan toegepast worden.
Een incrementele benadering kan *top-down* of *bottom-up* opgebouwd worden. Implementatie van een top-down strategie:

1. De hoofd controle module wordt gebruikt als een test driver en de onmiddellijk onderliggende modules worden vervangen door stubs.
2. Afhankelijk van de gekozen aanpak (in de diepte of in de breedte) worden ondergeschikte stubs afzonderlijk vervangen door de reële module.
3. Wanneer een module geïntegreerd is, worden testen uitgevoerd.
4. Bij het voltooien van een set van testen, wordt een volgende stub vervangen door de reële module.
5. Regressie testen (zie verder) kunnen uitgevoerd worden om zich er van te verzekeren dat geen nieuwe fouten geïntroduceerd zijn.



Figuur 16.1: Top-down en bottom-up testen

Bij deze strategie worden de belangrijkste controle- en beslissingspunten vrij vroeg getest. Daarnaast kan bij een "in-de-diepte-eerst" benadering een volledige software functie geïmplementeerd en gedemonstreerd worden. (Dit kan zowel voor de ontwerper als voor de klant leuk zijn.)
Op het eerste zicht lijkt een top-down strategie relatief eenvoudig, maar in de praktijk treden allerlei problemen op. Het meest voorkomende probleem is wanneer berekeningen op lagere niveaus nodig zijn om de functies op het hogere niveau nauwkeurig te kunnen testen. De modules op de lagere niveaus worden aanvankelijk vervangen door stubs waaruit geen significante data flow komt. Ofwel moeten bepaalde testen uitgesteld worden totdat de stubs vervangen zijn door de reële modules maar dit kan leiden tot een onoverzichtelijke aanpak. Ofwel moeten er stubs gebouwd worden die toch een beperkte functionaliteit van de echte module simuleren, maar dit betekent overhead. Men kan ook overschakelen op de bottom-up strategie.
Implementatie van een bottom-up strategie:

1. Low-level modules worden gecombineerd in *clusters* (of *builds*) die een specifieke software subfunctie uitvoeren.
2. Een driver (controle programma voor het testen) wordt geschreven om de test case invoer en uitvoer te controleren.
3. De cluster wordt getest.
4. De drivers worden verwijderd en de clusters worden gecombineerd. Op die manier beweegt men opwaarts in de programma structuur.

Omdat men van het laagste niveau begint, zijn er geen stubs nodig. Naarmate men hogerop komt, vermindert ook de behoefte aan testdrivers. Een nadeel is dat het programma als een volledige afgewerkte entiteit pas bestaat als de laatste module toegevoegd is.
Men maakt nogal dikwijls gebruik van een gecombineerde aanpak (*sandwich testing*): een top-down strategie voor de bovenste niveaus en een bottom-up strategie voor de ondergeschikte niveaus.

Regressie testen is het heruitvoeren van een deelverzameling van de testen die reeds zijn uitgevoerd om zich ervan te verzekeren dat bepaalde wijzigingen geen ongewenste neveneffecten introduceren. Het doorvoeren van wijzigingen kan nodig geweest zijn omdat tijdens de integratie testen na het integreren van een volgende module, bepaalde fouten ontdekt zijn. Door de fout te verbeteren kan het zijn dat er ongewenste effecten of bijkomende fouten geïntroduceerd worden. Regressie testen omvatten drie soorten:

- een representatieve deelverzameling van de testen waarbij alle hoofdfuncties uitgetest worden;
- bijkomende testen die op de software functies focusseren waarop de veranderingen waar-

schijnlijk effect hebben;

- testen die focusseren op de aangepaste softwarecomponenten.

Bij het vorderen van de integratie testen, kan het aantal regressie testen aanzienlijk toenemen. Het is daarom belangrijk om zich bij de regressie testen te concentreren op de hoofdfuncties. Men kan niet elke test telkens herhalen.

16.3.4 Validatie test

Eén van de definities is dat validatie slaagt wanneer de software functioneert op een manier die redelijkerwijs kan verwacht worden door de klant. Maar wie of wat is de scheidsrechter van *redelijke verwachtingen*? Deze zijn gedefinieerd in de "software vereisten specificatie", een document dat alle gebruikers-zichtbare attributen van de software beschrijft. Software validatie wordt gerealiseerd door een reeks van black-box testen die de conformiteit met de vereisten demonstreren.
Het resultaat van zo'n test kan zijn dat de functie- of performantie karakteristieken conform de specificaties zijn. Het is ook mogelijk dat een afwijking met de specificaties ontdekt wordt. Zo'n afwijking, ontdekt in dit stadium van het project, is meestal niet te corrigeren zonder de geplande voltooiingsdatum in het gedrang te brengen. Het is dus dikwijls nodig om met de klant te onderhandelen over de manier waarom het probleem moet weggewerkt worden.
Wanneer specifieke software voor één klant gebouwd is, worden een reeks van *acceptatie testen* uitgevoerd om de klant toe te laten alle vereisten te valideren. Zo'n acceptatie testen kunnen weken of zelfs maanden duren zodat ook cumulatieve fouten, die een degradatie van het systeem in de tijd veroorzaken, kunnen ontdekt worden.
Indien software ontwikkeld is als een product dat door vele klanten gaat gebruikt worden, is het onpraktisch om bij elke klant formele acceptatie testen uit te voeren. Meestal wordt een *alfa en beta testing* proces gebruikt om fouten te ontdekken die alleen door eindgebruikers blijken gevonden te worden.
Alfa testen gebeuren in een gecontroleerde omgeving bij de producent door een (potentiële) klant. De software wordt op een natuurlijke manier gebruikt terwijl de ontwerper over de schouders meekijkt en optredende fouten en gebruiksproblemen noteert.
De beta test wordt uitgevoerd bij één of meerdere klanten door de eindgebruikers waarbij de ontwerper normaal niet aanwezig is. Het is een "live" toepassen van de software in een omgeving niet gecontroleerd door de ontwerper. De klant noteert alle (reële en ingebeelde) problemen die tijdens de beta test opgetreden zijn en rapporteert deze aan de ontwerper.

16.3.5 Systeem test

Software is slechts één element in een computergebaseerd systeem. Dus ook het geheel moet getest worden, waarbij het klassieke probleem van *finger pointing* kan optreden. De software ingenieur kan zich hiertegen wapenen door te anticiperen op mogelijke interfacing problemen.
Er zijn een aantal systeemtesten die het volledige computergebaseerde systeem uittesten:

Recovery.
Veel computergebaseerde systemen moeten bij het optreden van fouten snel kunnen hersteld worden zodat binnen een bepaalde tijd de werking kan hernomen worden. In sommige gevallen moet het systeem *fault tolerant* zijn: het optreden van een fout mag niet leiden tot het niet verder functioneren van het gehele systeem. In andere gevallen moet een systeemfout binnen een gespecificeerde periode hersteld zijn.
Herstel testen is een systeemtest waardoor de software op verschillende manieren faalt zodat kan nagegaan worden of de recovery juist uitgevoerd wordt.

Security.
Systemen met interessante informatie zijn een doelwit voor indringers: *hackers* die het voor de sport doen; ontevreden werknemers uit wraak; en oneerlijke individuen om zichzelf te verrijken.

Veiligheidstesten dienen om te verifiëren of de ingebouwde beveiligingsmechanismen daadwerkelijk het systeem afschermen tegen oneigenlijk gebruik.

Stress.
Stress testen confronteren programma's met abnormale situaties: "hoe ver kunnen we gaan voor het kapot gaat?" Er worden dingen van het systeem gevraagd in abnormale hoeveelheden, frequenties, volumes. Wanneer het gemiddeld aantal interrupts twee per seconde is, wordt het effect nagegaan van bijvoorbeeld 10 interrupts per seconde. De data-input snelheid wordt verhoogd om te zien hoe de invoerroutines reageren.
Bij wiskundige algoritmes spreekt men in dit verband ook van *sensitivity testen*. Bepaalde data (nog binnen het domein van geldige data) kan resulteren in verkeerde verwerking en/of een sterke degradatie van de performantie.

Performance.
Bij real-time systemen moet de software binnen vooropgestelde tijden reageren, anders is ze onaanvaardbaar. *Performance testen* test het run-time gedrag in de contekst van een geïntegreerd systeem. Deze testen zijn dikwijls ook gekoppeld aan stress testen.

16.4 Debuggen

Het gevolg van een succesvolle test (er is een fout gevonden!) is *debugging*: het proces om de fout te verwijderen uit het systeem.
Debuggen is nog steeds voor een groot deel een kunst. Wanneer een software ingenieur de resultaten van een test bestudeert, wordt hij dikwijls geconfronteerd met een "symptomatische" aanduiding van een software probleem. Het externe voorkomen van de fout en de interne oorzaak van de fout vertonen dikwijls geen duidelijk verband met elkaar. Het slecht begrepen mentale proces dat een symptoom met een oorzaak relateert, is debugging.
Enkele karakteristieken van bugs kan helpen bij het vinden van de fout:

- Het symptoom kan veroorzaakt zijn door een menselijke fout, die moeilijk kan nagegaan worden.
- Het symptoom kan veroorzaakt zijn door niet-fouten (bijvoorbeeld afrondingsonnauwkeurigheden).
- Het symptoom kan (tijdelijk) verdwijnen wanneer een andere fout verbeterd is.
- Symptoom en oorzaak zijn ver van elkaar verwijderd: het symptoom treedt op in een bepaald deel van een programma, terwijl de fout in een totaal ander deel van het programma gebeurt. Strikt gekoppelde programma-structuren versterken zo'n situatie.
- Het symptoom kan het resultaat van timing problemen zijn, in plaats van processing problemen.
- Het kan moeilijk zijn om de input condities nauwgezet te reproduceren (bijv. in een real-time toepassing is de input volgorde onbepaald).
- De symptomen treden nu en dan op: dit is klassiek in embedded systemen waar hard- en software op een onontwarbare manier gekoppeld zijn.
- Het symptoom kan het gevolg zijn van oorzaken die gedistribueerd zijn over verschillende taken die op verschillende processoren draaien.

In het algemeen zijn er drie benaderingsmethodes:

brute force : memory dumps, run-time traces en programma overladen met `printf` statements: geeft enorm veel informatie maar het is zoeken naar een speld in een hooiberg. Meest gebruikte en minst efficiënte methode.

backtracking : vanaf de plaats waar het symptoom opgetreden is, teruggaan in de code totdat de oorzaak ontdekt wordt. Niet echt bruikbaar wanneer het aantal lijnen groot is.

cause elimination : gebruik makend van *binaire partitionering*. Stel een oorzaak-hypothese op en alle beschikbare data wordt gebruikt om deze hypothese aan te nemen of te verwerpen.

Waarom is debuggen zo moeilijk? Het antwoord blijkt niet bij software technologie te liggen, maar bij psychologie. Dit kan bijvoorbeeld afgeleid worden uit een commentaar van Schneiderman:

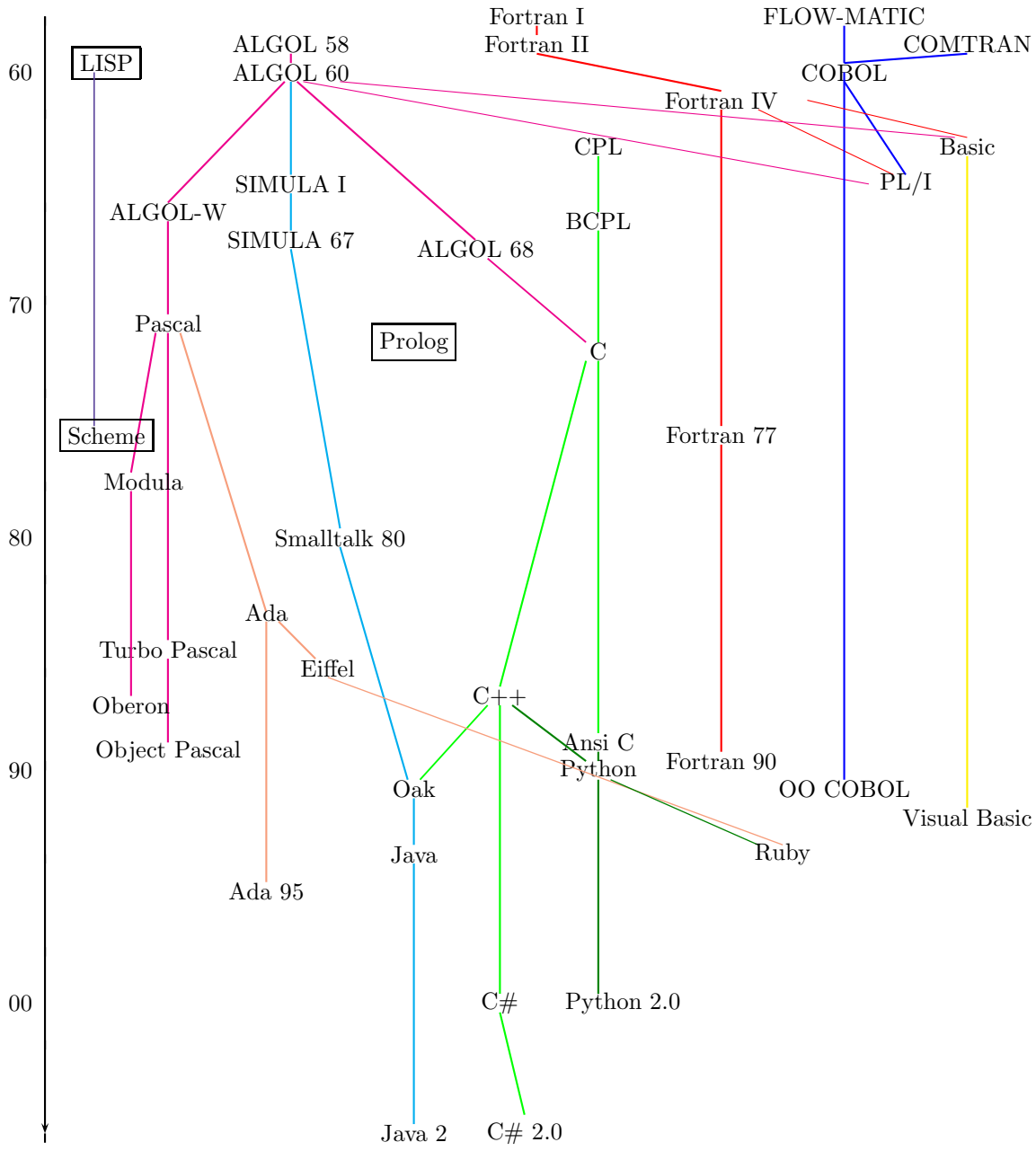
> Debugging is one of the more frustrating parts of programming. It has elements of problem solving or brain teasers, coupled with the annoying recognition that you have made a mistake. Heightened anxiety, and the unwillingness to accept the possibility of errors, increases the task difficulty. Fortunately there is a great sigh of relief and a lessening of tension when the bug is ultimately ... corrected.

Er zijn verschillende debugging tools ter beschikking, maar een vrij effectieve methode is het probleem voor te leggen aan andere mensen. Het valt dikwijls voor dat zo iemand (met een frisse kijk en zonder frustraties) zeer snel een fout vindt waar je zelf uren of dagen hebt zitten naar te zoeken. Dus, “when all else fails, get help!”

17 Programmeertalen: algemeenheden.

17.1 Genealogie.

In figuur 17.1 wordt de genealogie gegeven van een aantal programmeertalen.



Figuur 17.1: Genealogie van programmeertalen

Verschillende van deze talen zijn voor een specifiek domein ontworpen:

- wetenschappelijke toepassingen: FORTRAN, ALGOL 60;
- administratieve toepassingen: COBOL;
- artificiële intelligentie: LISP, Prolog;

- systeem software: C.

Naast deze algemene talen zijn er ook talen voor specifieke doeleinden ontwikkeld: RPG (het maken van administratieve rapporten), APT (taal voor NC machines), GPSS (simulatie).

17.2 Evaluatie criteria.

1. **Leesbaarheid.** Hieronder wordt verstaan de gemakkelijkheid waarmee een programma gelezen en begrepen wordt. Dit is een belangrijk aspect bij het onderhoud van programma's. Het onderhoud is sinds de jaren 70 steeds een groter deel gaan uitmaken van de software levenscyclus.

   Elementen hier zijn: eenvoud, ortogonaliteit, controle statements, data structuren en syntax.

2. **Schrijfbaarheid.** Aspecten van leesbaarheid komen hier terug aan bod, omdat bij het schrijven van programma's dikwijls grote stukken moeten herlezen worden. Schrijfbaarheid moet bekeken worden binnen de contekst van het toepassingsdomein. Het heeft niet veel zin COBOL en APL te vergelijken voor het schrijven van een programma voor matrix-operaties of voor het produceren van rapporten met ingewikkelde formaten.

   Elementen hier zijn: eenvoud en ortogonaliteit, mogelijkheden tot abstractie en de uitdrukkingskracht.

3. **Betrouwbaarheid.** Een programma is betrouwbaar wanneer de uitvoering gebeurt volgens de specificaties en dit onder alle omstandigheden.

   Mogelijkheden van type controle en afhandelen van excepties zijn hier belangrijk. Aliasing daarentegen is een vrij gevaarlijk concept.

4. **Kost.** De kost van een programmeertaal is een functie van verschillende elementen: de training van de programmeurs, het schrijven van een programma, programmeer-omgeving, het compileren van een programma, het uitvoeren van een programma, het onderhoud van een programma.

5. Andere criteria, zoals bijvoorbeeld: draagbaarheid, algemeenheid, goed-gedefinieerdheid.

17.3 Soorten.

17.3.1 Imperatief programmeren.

Met behulp van een imperatieve taal wordt een algoritme omgezet in een opeenvolging van akties die moeten uitgevoerd worden om de oplossing van een probleem te vinden.
Het ontwerp van deze talen is gebaseerd op de "von Neumann" architectuur van een computer. In deze architectuur zijn zowel data als programma gestockeerd in hetzelfde geheugen. De CPU die de instructies uitvoert, is hiervan gescheiden. Daarom moeten instructies en data getransfereerd worden van geheugen naar de CPU. Resultaten van operaties in de CPU moeten terug naar geheugen gebracht worden.
De belangrijste elementen in een imperatieve taal zijn daarom variabelen, toekenningen, conditionele opdrachten en de iteratieve vorm van herhaling (instructies zitten in naburige geheugencellen). Een *variabele* is niets anders dan een abstractie van een geheugenlocatie. Een toekenning is een overdracht van CPU naar geheugen. Controle opdrachten zijn gestruktureerde hoog-niveau varianten van voorwaardelijke en onvoorwaardelijke sprongopdrachten. Het ultieme resultaat wordt bekomen door het toekennen van waarden aan variabelen.

17.3.2 Functioneel programmeren.

Alle berekeningen in een functioneel programma worden gerealiseerd door het toepassen van functies op argumenten. Er is geen behoefte aan toekenningsstatements. Er zijn ook geen variabelen in de zin van die van imperatieve talen, namelijk om een waarde de stockeren die verandert tijdens

de uitvoering van het programma. Iteratieve processen worden gespecificeerd met behulp van recursieve funktie oproepen.
De exacte volgorde waarin functies geactiveerd worden heeft geen invloed op het uiteindelijke resultaat: iedere oproep van een functie met dezelfde argumenten zal steeds hetzelfde resultaat teruggeven. Hierdoor kunnen diverse functies in parallel geactiveerd worden op verschillende processors. Daarnaast krijgen lijststructuren een meer centrale positie, en is de notie van hogere-orde functies erg cruciaal. Dit zijn functies die andere functies als argument kunnen aanvaarden en/of als resultaat kunnen afleveren.
Een voorbeeld in Scheme:

```
(define ( macht n x)
    ;sig: Pos * Num -> Num
    ;effect: berekent de n-de macht van x
    (if (= n 1)
        x
        (* x (macht (- n 1) x))))
```

Toepassingen: één van de eerste functionele talen is LISP en werd ontwikkeld voor formule manipulatie en lijstverwerking. Op dit moment wordt LISP gebruikt in AI: expert systemen, kennisvoorstellling, natuurlijke taal verwerking, intelligente trainingssystemen, modellering van spraak en visie.

17.3.3 Logisch programmeren.

Dit kan beschouwd worden als een doelgerichte aanpak: het probleem (de doelstelling) wordt gespecificeerd, en er wordt aangegeven hoe het probleem kan opgelost worden in functie van een aantal deelproblemen. Omdat er meestal meerdere oplossingen zijn voor een probleem, zal de onderliggende machinerie gebruik maken van *backtracking* en *unification* bij het zoeken naar de oplossing. De waarheid van proposities hangt niet af van de volgorde waarin ze geëvalueerd worden. Er wordt gebruik gemaakt van een formele logische notatie om een bepaald berekeningsproces aan de computer duidelijk te maken. Het programmeren is niet-procedureel. In zo'n programma wordt niet exact gezegd *hoe* een resultaat moet berekend worden, maar de vorm van het resultaat wordt beschreven (declaratief).
Om dit te realiseren is er een compacte manier nodig om aan de computer zowel de relevante informatie als de inferentiemethode om de gewenste resultaten te berekenen, te geven. *Predikaten logica* is de basisvorm van communicatie. De bewijsmethode *resolutie* is de inferentietechniek.
Voorbeelden in Prolog:

```
ouder(piet, jan).
ouder(an, jan).
ouder(an, ria).
man(piet).
man(jan).
gelijk(X,X).

vader(X, Y) :- ouder(X, Y), man(X).
broer(X, Y) :- man(X), ouder(Z, X), ouder(Z, Y),
               not gelijk(X, Y).

?- vader(piet, X), broer(X, ria).
```

Toepassingen: relationele database management systemen en een aantal domeinen in AI: expert systemen, natuurlijke taal verwerking.

Experimenten met het aanleren van Prolog aan jonge kinderen hebben aangetoond dat dit mogelijk is. Op deze manier kan informatica heel snel in het onderwijs geintroduceerd worden. Als neveneffect wordt ook logica onderwezen wat resulteert in meer helder denken en zich uitdrukken. Het blijkt ook gemakkelijker te zijn logisch programmeren te onderwijzen aan jonge kinderen dan aan een programmeur met een grote ervaring met een imperatieve taal.
Functionele en logische talen zijn de belangrijkste onder de zogenaamde *declaratieve talen*, omdat men zich bij het programmeren eerder zal richten naar het **wat** dan naar het **hoe**.

17.3.4 Object-georiënteerd programmeren.

Problemen worden opgelost door de reële wereld objecten van het probleem en de bewerkingen op deze objecten te identificeren. De drijvende kracht bij de uitvoering van een programma zijn objecten, die onderling communiceren door het uitwisselen van boodschappen. Omdat dit een getrouwe reproductie is van de manier waarop wij mensen plegen om te gaan met allerlei dingen in onze omgeving, wordt OOP algemeen beschouwd als de meest intuïtieve stijl van programmeren. Ieder object in een object-georiënteerd programma heeft een lokaal geheugen, specifiek gedrag en de mogelijkheid om eigenschappen over te nemen van andere objecten. Dit concept (*overerving*) aangevuld met *abstracte data types*, *polymorfisme* en *dynamische binding* leidt tot een effectieve ondersteuning bij het aanpassen en het herbruiken van software.
Voorbeelden: zie cursus netwerken en objectoriëntatie.

17.4 Een vergelijking.

In "Het chaos computer boek", pp. 112–115 (vertaling van Pieter Janssens) wordt een verband gelegd tussen de favoriete programmeertaal van een gebruiker en zijn of haar voorkeur voor muziekstromingen in de echte wereld.

> In BASIC zingen ze hun kinderliedjes ... Ik word overvallen door een zachte weemoed als ik denk aan het verlies van mijn programmeeronschuld toen ik probeerde een zelfgemaakt BASIC-programma, dat ik al een paar weken niet meer had ingekeken en waarin het toeging als op een code-geworden kinderverjaardagsfeestje, in een gestructureerd dialect te vertalen. Twee dagen lang huppelde ik met bonte viltstiftstrepen achter tientallen GOTO-sprongen aan en probeerde wanhopig me te herinneren wat ik met deze of gene variabele (`N=FF*PF-QT`) kon hebben bedoeld.
>
> ... Met LOGO werken alleen beginners ouder dan vijfendertig, die ergens gehoord hebben dat het een programmeertaal is voor kinderen en die daarom denken dat ze die in elk geval wel onder de knie zullen krijgen.
>
> FORTRAN is iets voor fatsoenlijke burgers - degelijk, geborneerd en saai als Duitse Schlagers, in het gunstige geval ongenaakbaar en gecompliceerd als het Derde Brandenburgse Concert van Bach, dat volgens mij klinkt als een draaiorgel dat in vertraagd tempo een keldertrap afdendert. Met FORTRAN kun je filmcamera's in Jupitersondes sturen, atoombomsimulaties afwikkelen en main-frames klein krijgen, kortom: FORTRAN is geen greintje hip. COBOL is haast nog erger. Een soort marsmuziek.
>
> Code in ASSEMBLER is al haast machinetaal en laat zich lezen als een gitaarstuk voor John McLaughlin, geschreven door een astrofysicus – zuinig, superpriegelig, ergens wel grappig en consequent onbegrijpelijk. Tempo: furioso. Daarenboven in muzikaal opzicht: Burundi-beat (tam tam tom tom), hardcore punk (vol gas) of Mozart, gespeeld op een oude 78-toerengrammafoon.
>
> Dan is er nog de MACHINETAAL, processor-klare tekst. MACHINETAAL is schrikbarend geciseleerd als een flamenco van Manitas de Plata, puur en direct als een tremolo-etude à la "Riquerdos de l'Alhambras". Als je voor het eerst de bitmonsters in de vorm van eindeloze hexadecimale kolommen ziet, bekruipt je het angstige vermoeden dat het om een systeemstoring gaat of om de derdemachtswortel uit Schillers

Lied von der Glocke. Er schijnen MACHINETAAL-freaks te zijn die af en toe hun computer openschroeven en proberen om met een microscoop en met van dat bestek zoals oogchirurgen meestal gebruiken, de bits met de hand in de processor rond te schuiven.

C (ontwikkeld uit een programmeertaal die B heette), is iets heel verfijnds, C-programmeurs luisteren meestal naar goede, elegante popmuziek, die minstens even "sophisticated" is als hun code. In C kunnen met een beetje talent op een virtuoze manier algoritmen worden geprogrammeerd die ritselen en huppelen, swingen en klinken. Veel C-compilers zijn, als ze in stemming komen, zo goed op dreef dat ze zelfs onjuiste foutmeldingen geven ... De mogelijkheid om met afkortingen, compressies en het overnemen van opdrachten een volkomen individueel programmaontwerp te kunnen maken, wordt door veel stijlbewuste C-programmeurs aangegrepen en kan – zoals bij radicale chic – tot onbegrip van minder gevoeligen leiden, dat wil zeggen: van mensen voor wie de C-code eruitziet alsof iemand een opgerold gordeldier over het toetsenbord heen en weer heeft gerold, om te zien wat voor tekencombinaties er op het scherm zouden verschijnen.

Veel overeenkomst met C vertoont Pascal, misschien wat meer "disco", en met een neiging tot opruimen die de ongeremde creativiteit in de weg staat...

LISP is een taal uit de sfeer van de zogenaamde "kunstmatige intelligentie" waarin, muzikaal gesproken, een poging wordt ondernomen om Tsjaikowski's eerste pianoconcert te spelen op een bongotrommel; ook PROLOG probeert aan dit streven tegemoet te komen. Waarbij ik er altijd weer van sta te kijken dat stukken KI-programma's er door een ongebreideld gebruik van teksthaken uitzien als visgraten – where is the meat.

... Leden van de FORTH-Indianenstam hebben meestal een even gecompliceerd karakter als hun programma's en houden van de natuur, in het bijzonder van binairbomen.

Het nieuwste voortbrengsel van verheven digitale uitdrukkingsvormen is OCCAM... In OCCAM worden transputers geprogrammeerd – computers waarin, anders dan tot nu toe, het hele werk niet stap voor stap wordt gedaan door één enkele microprocessor, maar door verscheidene processors naast elkaar... OCCAM-programmeurs zijn letterlijk gedwongen een cultuursprong te maken: ze moeten – indien dat menselijkerwijs gesproken mogelijk is – parallel leren denken. Zevenduizend jaar schrift, teken voor teken aaneengeregen op een regel, hebben de lineaire volgorde van letters, begrippen en gedachten in ons hoofd geprent. Het zal spannend zijn toe te kijken, of mee te experimenteren, tot welke resultaten de pogingen zullen leiden om polyfone processen – zoals we die bijvoorbeeld kennen van orkestpartituren of ritmegebonden improvisaties – op een voor een machine geschikte manier te formuleren en er misschien zelfs een basisvoorwaarde voor rationeel denken van te maken.

18 Prolog

18.1 Inleiding

PROLOG (PROgramming in LOGic) ontstaan in 1972. Zoals uit de naam enigszins blijkt is het een programmeertaal gebaseerd op logica.
1982: basis voor het "vijfde generatie" project in Japan.
Toepassingsgebied: concurrent voor LISP op het vlak van AI:

- relationele database: logisch, goed gestructureerd formalisme;
- software engineering: logische specifikatie van een systeem: → logisch programma;
- natuurlijke taal verwerking: o.a. automatische vertaling;
- expert systemen.

procedurele taal: met behulp van een algoritme: beschrijving van de manier waarop de oplossing kan gevonden worden;

deklaratieve taal: het probleem: de gegevens en de wetmatigheden er rond worden beschreven; Prolog zal met behulp van het ingebouwd inferentie-mechanisme één of alle mogelijke oplossingen deduceren.

18.2 Syntax

Er is slechts één datatype: de term waarvan de inhoud dynamisch bepaald wordt. Volgende vormen zijn mogelijk:

constante: naam van een specifiek object of een specifiek verband: ofwel een *atoom* (een naam beginnend met kleine letter) of een *integer* (een getal);

variabele: een object dat (nog) niet kan benoemd worden (een naam beginnend met een hoofdletter of met '_');

structuur: een object dat uit een verzameling andere objecten (componenten) bestaat: een *functor* met een aantal *argumenten*; de functor is een atoom, elk argument op zich (een term) kan een atoom, variabele of structuur zijn; het aantal argumenten is de *ariteit* van de functor; de functor van de primaire structuur wordt een *predikaat* genoemd.

Een programma is opgebouwd uit een aantal *clauses*. Een clause heeft syntactisch de vorm van een term, afgesloten met een punt (.). Er zijn drie types:

fact	feit	$< structuur > .$
rule	regel	$< structuur > :- < structuur > .$
query	vraag	$?- < structuur >$

Een operator is een structuur waarbij de meer klassieke *prefix*, *infix* of *postfix* schrijfwijze gebruikt wordt.

18.3 Semantiek

De verzameling van beschikbare feiten en regels bij de uitvoering van een programma wordt *database* genoemd.
Wanneer een vraag ingegeven wordt, zal Prolog trachten dit *doel* te laten slagen. Dit gebeurt door sequentieel de feiten en de regels van de database met het actieve doel te vergelijken. Een doel slaagt bij succesvolle unificatie met een feit, of bij succesvolle unificatie met het *hoofd* van een regel en slagen van het *lichaam* van die regel (bij succesvolle unificatie met het hoofd van een regel verschuift het actieve doel naar het lichaam van de regel).
Een voorbeeld:

```
   DATABASE                          DOEL

lust(marie,wortel).
lust(marie,patat).               ?- lust(marie,X), lust(jef,X).
lust(jef,patat).
lust(jef,knol).                                 wortel       wortel
          1. eerste doel slaagt
          2. X krijgt de waarde wortel
          3. probeer nu tweede doel, waarbij X de waarde wortel heeft
lust(marie,wortel).
lust(marie,patat).               ?- lust(marie,X), lust(jef,X).
lust(jef,patat).
lust(jef,knol).                                 wortel       wortel
          4. het tweede doel slaagt niet
          5. dus backtrack: vergeet toegekende waarde aan X

lust(marie,wortel).
lust(marie,patat).               ?- lust(marie,X), lust(jef,X).
lust(jef,patat).
lust(jef,knol).                                  patat        patat
          6. het eerste doel slaagt opnieuw
          7. X krijgt de waarde patat
          8. probeer nu tweede doel, waarbij X de waarde patat heeft

lust(marie,wortel).
lust(marie,patat).               ?- lust(marie,X), lust(jef,X).
lust(jef,patat).
lust(jef,knol).                                  patat        patat
          9. het tweede doel slaagt
         10. de vraag krijgt een positief antwoord

Eventueel doet prolog verder om te zien of er nog andere
mogelijkheden zijn door eerst voor het tweede doel nog de
volgende feiten te bekijken en dan voor het eerste doel
verder te zoeken naar alternatieven.
```

Bij elke succesvolle unificatie van een doel wordt een *backtrackingspunt* ingesteld. Bij het globaal falen van een doel, wordt het laatst geplaatste backtrackingspunt verwijderd, terwijl de erbij horende toestand hersteld wordt. De poging tot slagen van dit herstelde doel wordt nu herhaald, maar enkel door consultatie van het resterende gedeelte van de database.
Door middel van het predicaat '!' (*cut*) kunnen backtrackingspunten vernietigd worden: alle keuzes die op het moment van de uitvoering van de cut gemaakt zijn, worden bevroren; bij faling worden hiervoor geen nieuwe alternatieven meer bekeken.
De volgende code definieert een predikaat waarbij het derde argument gelijk is aan het maximum van de twee eerste argumenten:

```
max(A,B,B) :- A < B.
max(A,B,A).
```

Omwille van backtracking kunnen onjuiste antwoorden gegeven worden:

```
?- max(5,10,X).
X=10;
X=5
```

Om backtracking naar de tweede regel te voorkomen, kan in de eerste regel het cut symbool toegevoegd worden:

```
max(A,B,B) :- A < B, !.
max(A,B,A).
```

Er zal nu geen verkeerd tweede antwoord gegeven worden. Merk op dat cuts gelijkaardig zijn aan gotos: ze hebben de neiging de complexiteit van de code te verhogen in plaats van te vereenvoudigen. Cuts zouden zoveel mogelijk moeten vermeden worden.
Cut is een voorbeeld van een *ingebouwd predikaat*. Deze predikaten geven enerzijds de mogelijkheid om bepaalde handelingen te verrichten die niet binnen Prolog te definiëren zijn. Anderzijds vormen zij een basisbibliotheek aan nuttige predikaten.

Een procedure is een verzameling clauses met hetzelfde primaire predikaat (zelfde naam en ariteit). Een regel bestaat uit een *hoofd* en een *lichaam*:

```
hoofd :- lichaam.          head :- body.
```

Een lichaam bestaat uit een aantal subgoals. Deze subgoals zijn met elkaar verbonden d.m.v. de , (AND) operator en/of de ; (OR) operator.

Interpretatie van : moeder(X,Y) :– echtgenote(X,Z), vader(Z,Y).

deklaratief: benadrukt het statisch bestaan van relaties, volgorde van de subgoals is van geen belang:

```
X is de moeder van Y      wanneer X de echtgenote is van Z
                               en Z de vader is van Y
```

procedureel: benadrukt de volgorde van de stappen die de interpreter moet nemen om de vraag te evalueren en te beantwoorden:

```
Om de kinderen (Y) van moeder X  te vinden
    1. zoek naar de persoon Z waarvan X de echtgenote is
    2. zoek naar de personen Y waarvan  Z de vader is
```

Er wordt veel gebruik gemaakt van **recursieve procedures**. Zo'n procedure heeft tenminste twee componenten:

1. een niet-recursieve clause, die het basisgeval beschrijft (waarbij de recursie stopt);
2. een recursieve regel: in het lichaam van de regel worden eerst nieuwe argument waarden gegenereerd, om dan de procedure met deze nieuwe argumenten terug op te roepen.

18.4 Variabelen

Een variabele X in een query is een existentiële kwantor:

```
?- echtgenote(marie,X), write(X), nl.
```

```
Bestaat er tenminste 1 waarde voor de variabele X zodat de query waar is.
```

Een variabele X in het hoofd van een regel is een universele kwantor:

```
moeder(X,Y) :- echtgenote(X,Z), vader(Z,Y).
```

```
Voor alle mensen X en Y geldt:   X is de moeder van Y indien
                                 er bestaat een persoon Z zodanig dat
                                      X de echtgenote is van Z
                                   en Z de vader is van Y
```

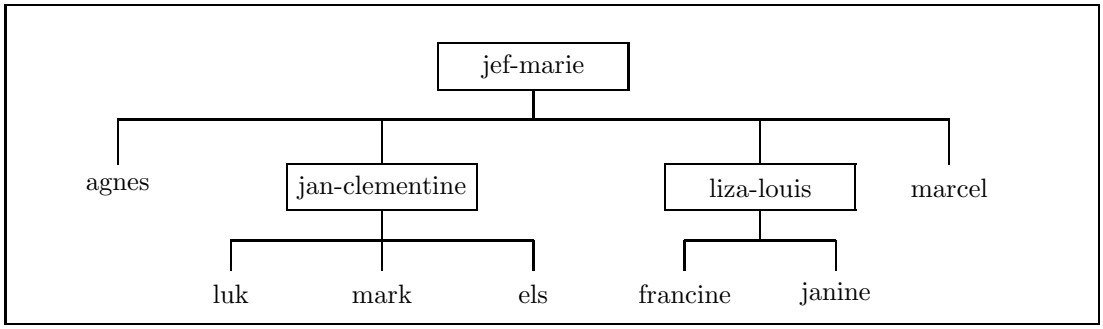
Het bereik van een variabele is beperkt tot de regel waarin hij gebruikt wordt: er bestaan geen globale variabelen.
Naargelang van de toestand van een variabele, kan deze al of niet een waarde hebben:

- initieel: inhoudloos: niet-geïnstantieerd;
- een geïnstantieerde variabele vertegenwoordigt een niet-variabele term;
- gelijknamige variabelen binnen dezelfde clause zijn gelinkt;
- gelinkte variabelen hebben dezelfde inhoud.

Unificatie van twee termen (bijv. subgoal met een feit of een hoofd van een regel) is succesvol tussen:

- twee constanten die gelijk zijn: atoom: zelfde naam, integer: zelfde waarde;
- niet-geïnstantieerde variabele met geïnstantieerde variabele waarbij de eerste geïnstantieerd wordt;
- niet-geïnstantieerde variabele met niet-geïnstantieerde variabele: worden gelinkt;
- twee structuren met hetzelfde primaire predikaat (naam en ariteit) en succesvolle unificatie tussen de overeenkomstige termen.

18.5 Voorbeeld




```
man(jef).        man(jan).              man(louis).
man(marcel).     man(luk).              man(mark).
vrouw(marie).    vrouw(clementine).     vrouw(liza).
vrouw(agnes).    vrouw(els).            vrouw(francine).   vrouw(janine).
echtgenote(marie,jef).                  echtgenote(clementine,jan).
echtgenote(liza,louis).

vader(jef,agnes).   vader(jef,jan).    vader(jef,liza).   vader(jef,marcel).
vader(jan,luk).     vader(jan,mark).   vader(jan,els).
vader(louis,francine).                 vader(louis,janine).

moeder(X,Y) :- echtgenote(X,Z), vader(Z,Y).
grootvader(X,Y) :- vader(X,Z), (vader(Z,Y) ; moeder(Z,Y)).
grootmoeder(X,Y) :- echtgenote(X,Z), grootvader(Z,Y).

broer(X,Y) :- man(X), vader(Z,X), vader(Z,Y), X \= Y.
zus(X,Y) :- vrouw(X), vader(Z,X), vader(Z,Y), X \= Y.
ouder(X,Y) :-  (vader(X,Y) ; moeder(X,Y)).
```

```
broer2(X,Y) :- broer(X,Y).
broer2(X,Y) :- echtgenote(Z,X) , zus(Z,Y).
zus2(X,Y) :-  zus(X,Y).
zus2(X,Y) :-  echtgenote(X,Z) , broer(Z,Y).

oom(X,Y) :- broer2(X,Z), ouder(Z,Y).
tante(X,Y) :- zus2(X,Z), ouder(Z,Y).
neef(X,Y) :- man(X), oom(Z,X), ouder(Z,Y).
nicht(X,Y) :- vrouw(X), oom(Z,X), ouder(Z,Y).

va(X) :- write(X), write(' vader van '),
            vader(X,Y), write(Y), write(' '), fail.
vb(X) :- write(X), write(' moeder van '),
            moeder(X,Y), write(Y), write(' '), fail.
vc(X) :- write(X), write(' grootvader van '),
            grootvader(X,Y), write(Y), write(' '), fail.
vd(X) :- write(X), write(' grootmoeder van '),
            grootmoeder(X,Y), write(Y), write(' '), fail.
ve(X) :- write(X), write(' broer van '),
            broer(X,Y), write(Y), write(' '), fail.
vf(X) :- write(X), write(' zus van '),
            zus(X,Y), write(Y), write(' '), fail.
vg(X) :- write(X), write(' oom van '),
            oom(X,Y), write(Y), write(' '), fail.
vh(X) :- write(X), write(' tante van '),
            tante(X,Y), write(Y), write(' '), fail.
vi(X) :- write(X), write(' neef van '),
            neef(X,Y), write(Y), write(' '), fail.
vj(X) :- write(X), write(' nicht van '),
            nicht(X,Y), write(Y), write(' '), fail.
vk(Y) :- findall(X, oom(X,Y), L), length(L, Aantal), write(Aantal),
            write(' ooms van '), write(Y), write(' : '), write(L), fail.
```

18.6 De torens van Hanoi

```
hanoi(N) :- move(a,b,c,N).
move(F, T, H, N) :- N < 1, !.
move(F, T, H, N) :- NN is N - 1, move(F,H,T,NN),
                    write(F), write(' -> '), write(T), nl,
                    move(H,T,F,NN).
```

In dit voorbeeld moet een rekenkundige uitdrukking geëvalueerd worden. Dit wordt aangegeven met de **is** operator. De uitdrukking langs rechts van **is** wordt uitgerekend, alle variabelen in die uitdrukking moeten geïnstantieerd zijn. Het resultaat wordt gebruikt om de variable langs links te instantiëren. De klassieke rekenkundige operatoren zijn in de meeste Prolog versies voorzien. Het = teken in bijvoorbeeld de term X = Y gaat na of de variabelen X en Y unificeerbaar zijn. Als één van de twee variabelen geïnstantieerd is en de andere nog niet, dan zal deze ook geïnstantieerd worden. Om te testen of twee numerieke (geïnstantieerde) variabelen gelijk aan elkaar zijn, moet de =:= operator gebruikt worden.

18.7 List processing

De operator om een lijst te construeren is de punt (.) operator: deze heeft twee operands: het eerste element van de lijst en de rest van de lijst, bijv. `.(a, .(b, .(c, []) ) )`. Omwille van het veelvuldig gebruik van lijsten is een vereenvoudige syntax voorzien:

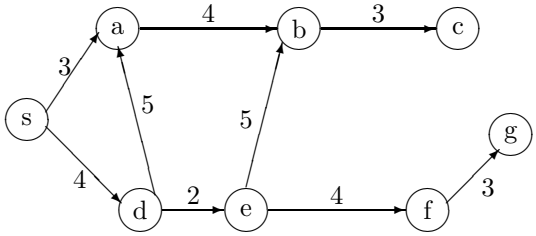
```
[a,b,c]  is de voorstelling voor  .(a,.(b,.(c,[])))
[k|l]    is de voorstelling voor  .(k,l)
```

```
/* het toevoegen van lijst L2 aan lijst L1 resulterend in lijst L3 */
append([],L,L).
append([X|L1],L2,[X|L3]) :- append(L1,L2,L3).

/* het omdraaien van een lijst */
reverse([],[]).
reverse([H|T],L) :- reverse(T,Z), append(Z, [H], L).

/* test of X (alfabetisch) kleiner is dan Y */
order(X,Y) :- integer(X), integer(Y), !, X < Y.
order(X,Y) :- atom(X), atom(Y), !,
              name(X,Lx), name(Y,Ly), aless(Lx,Ly).
aless([],[_|_]).
aless([X|_],[Y|_]):- X < Y.
aless([X|P],[X|Q]):- aless(P,Q).

/* insertion sort */
insort([],[]).
insort([X|L], M) :- insort(L,N), insortx(X,N,M).
insortx(X, [A|L], [A|M]) :- order(A,X), !, insortx(X, L, M).
insortx(X, L, [X|L]).
```

18.8 Een doolhof

```
member(X,[X|_]).
member(X,[_|R]) :- member(X,R).

n(s,a,3).   n(s,d,4).   n(a,b,4).   n(b,c,3).   n(d,a,5).
n(d,e,2).   n(e,b,5).   n(e,f,4).   n(f,g,3).

ga(X,X,T,T).
ga(X,Y,T,W) :- n(X,Z,_), write(X), tab(1), write(Z), tab(3),
               not(member(Z,T)), write([Z|T]), nl, ga(Z,Y, [Z|T],W).

weg(X,Y) :- ga(X,Y,[X],T), write(T), nl.
```

```
?- weg(s,g).
      s a   [a,s]
      a b   [b,a,s]
      b c   [c,b,a,s]
      s d   [d,s]
      d a   [a,d,s]
      a b   [b,a,d,s]
      b c   [c,b,a,d,s]
      d e   [e,d,s]
      e b   [b,e,d,s]
      b c   [c,b,e,d,s]
      e f   [f,e,d,s]
      f g   [g,f,e,d,s]
[g,f,e,d,s]
```



Maak een lijst L van alle mogelijke vertrekpunten X waarvoor een weg bestaat naar **g**:

```
?- findall(X, weg(X,g), L).
```

Resultaat is `L = [g,s,d,e,f]`.

18.9 Het acht-koninginnen probleem

Plaats op een 8×8 bord een koningin op elke rij zodat er geen enkele koningin op dezelfde kolom of dezelfde diagonaal staat.

```
queens :- queens1( f(0,[]), B), write(B), nl,
          print_board(B), nl, write('More? y/n: '), get(I), nl, I = 110.
queens1( f(8,B), B) :- !.
queens1( f(M,Q), B) :- add_queen( f(M,Q), f(M1,R)), queens1( f(M1,R), B).

add_queen(f(M,Q), f(M1,[p(M1,K)|Q])) :-
                M1 is M+1, gc(K), possible(p(M1,K), Q).

gc(1). gc(2). gc(3). gc(4). gc(5). gc(6). gc(7). gc(8).
possible(P, []).
possible(P, [Q|L] ) :- possible(P, L), ok(P, Q).

ok(p(R1,K), p(R2,K)) :- !, fail.                    /* zelfde kolom  */
ok(p(R1,K1), p(R2,K2)) :- X is R1-K1,
                          X is R2-K2, !, fail.      /* rechtse diagonaal */
ok(p(R1,K1), p(R2,K2)) :- X is R1+K1,
                          X is R2+K2, !, fail.      /* linkse diagonaal */
ok(P,Q).                                            /* OK */

print_board( [] ) :- !.
print_board( [p(_,K)|T] ) :- print_board(T),
                             print_line(K,8), nl, !.
print_line(_, 0):- !.
print_line(Q,K):- L is K-1, print_line(Q,L),
                  (Q == K, write('* '); write('. ')).
```

De eerste twee oplossingen:

```
[p(8,4),p(7,2),p(6,7),p(5,3),
       p(4,6),p(3,8),p(2,5),p(1,1)]
* . . . . . .
. . . . * . . .
. . . . . . . *
. . . . . * . .
. . * . . . . .
. . . . . . * .
. * . . . . . .
. . . * . . . .
```

```
[p(8,5),p(7,2),p(6,4),p(5,7),
       p(4,3),p(3,8),p(2,6),p(1,1)]
* . . . . . .
. . . . . * . .
. . . . . . . *
. . * . . . . .
. . . . . . * .
. . . * . . . .
. * . . . . . .
. . . . * . . .
```

18.10 Oefening

Het paardprobleem (*knight's tour*). Op een $n \times n$ schaakbord zijn n^2 velden. Een paard kan twee velden horizontaal en één veld vertikaal of twee velden vertikaal en één veld horizontaal bewegen, zolang het op het bord blijft.
Zoek een toer van $n^2 - 1$ bewegingen zodat het paard vertrekkend van een gegeven veld, elk ander veld van het bord juist eenmaal bezoekt.

23	10	15	4	25
16	5	24	9	14
11	22	1	18	3
6	17	20	13	8
21	12	7	2	19

Uitbreiding: zoek een gesloten toer waarbij het paard na het laatst veld bezocht te hebben terug op de startveld kan komen.

A Multi-User operating system: UNIX

A.1 Gebruikers en enkele eenvoudige commando's

Op een unix systeem kunnen verschillende gebruikers tegelijk werken. Om te kunnen werken heeft men een loginnaam nodig (publiek gekend) en een geheim paswoord. Met behulp hiervan kan men inloggen. Wanneer dit succesvol gebeurd is, draait er een shell die de commando's zal interpreteren en uitvoeren.

exit : om terug af te loggen;

passwd : wijzigen van het paswoord: een paswoord moet bestaan uit 3 letters en 3 cijfers; tijdens het intikken verschijnen deze tekens niet op het scherm; ter controle moet daarom het paswoord tweemaal ingegeven worden;

hostname : geeft de naam van de computer waarop men ingelogd is;

whoami : de eigen loginaam wordt getoond;

who : geeft een lijst van de ingelogde gebruikers op de lokale computer;

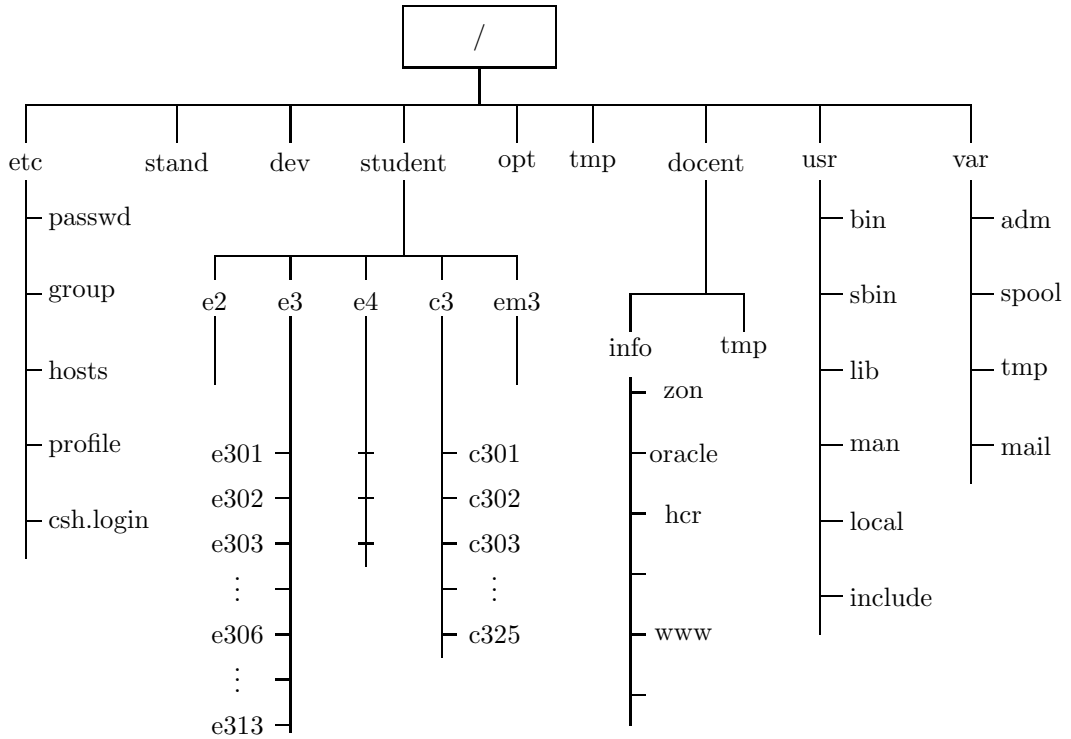
rwho : geeft een lijst van de ingelogde gebruikers op de verschillende computers van het netwerk;

mail : versturen van boodschappen en lezen van de binnengekomen berichten.

De loginnaam komt overeen met een userid. Deze relatie wordt gelegd in de `/etc/passwd` file. Hierin vindt men ook de default groep waartoe de gebruiker behoort, het geëncrypteerde paswoord, de home directory en de shell die moet opgestart worden bij inloggen. Deze user- en groupid kan met het commando *id* opgevraagd worden.

A.2 Bestanden structuur

De verschillende disks vormen samen één logisch bestandensysteem dat georganiseerd is in een boom:



Enkele commando's:

ll : toont de inhoud van een directory;

more : toont de inhoud van een bestand op het scherm;

cp : copiëren van een bestand naar een ander bestand;

rm : verwijdert een bestand uit een directory;

mv : geeft een nieuwe naam aan een bestand;

mkdir : maakt een nieuwe directory;

rmdir : verwijdert een directory, wanneer deze geen files of subdirectories meer bevat;

pwd : drukt het pad van de actuele directory af;

cd : de actuele directory wijzigt.

Het protectie systeem is gebaseerd op drie klassen van gebruikers:

- eigenaar: gewoonlijk de gebruiker die het bestand creëerde;
- groep: de groep waartoe de eigenaar behoort;
- anderen: al de rest van de gebruikers

Per klasse kunnen drie toegangsprivilegies gegeven worden:

r	het bestand mag gelezen worden
w	het bestand mag gewijzigd of verwijderd worden
x	het bestand mag uitgevoerd worden
	de directory mag doorzocht worden

Deze privileges kunnen opgevraagd worden met ll:

```
-rw-r-----      1      hcr   info     102       Oct 3 11:03    jefke
type+protec   links    eig.  grp.    grootte    laatste_upd    naam
```

De *eigenaar* (hcr) mag de file **jefke** lezen en schrijven, de gebruikers van de *groep* info kunnen deze file lezen; de *rest* van de gebruikers (o.a. studenten) hebben geen toegang tot deze file.

chmod : het wijzigen van de protectiebits van een bestand;

chown : het wijzigen van de eigenaar van een bestand;

chgrp : het wijzigen van de groep-eigenaar van een bestand.

Commando's om bewerkingen op bestanden te doen:

diff : vergelijken van twee bestanden;

sort : sorteren van een bestand;

grep : doorzoeken van een bestand naar een tekenpatroon;

find : doorzoeken van een directorystructuur naar een bestand;

wc : tellen van aantal lijnen, woorden, tekens van een bestand;

pr : formatteren van een bestand;

lp : doorsturen van een bestand naar de printer-spooler;

lpstat : opvragen van de status van de spooler.

A.3 De vi-editor

Tekstbestanden kunnen gecreëerd en gewijzigd worden met vi. Deze editor heeft twee basis modes:

- commando mode
- tekst invoer mode

Wanneer men vi opstart komt men in commando mode. Overgaan naar tekst invoer mode kan met volgende commando's gebeuren:

i	invoeren van tekst voor de cursor positie
a	invoeren van tekst na de cursor positie
o	invoeren van tekst op de volgende lijn
O	invoeren van tekst op de vorige lijn
R	overschrijven van tekst

Tijdens tekst invoer mode kan alleen met de backspace toets naar links bewogen worden. Terugkeren naar commando mode gebeurt m.b.v. de ESC toets. In deze mode kan met de pijltjes over de tekst bewogen worden. ook kunnen NEXT en PREV gebruikt worden om de cursor over 24 lijnen naar onder of naar boven te bewegen. Andere positioneringen:

*lijnnr*G	naar de lijn met lijnnummer *lijnr*
G	naar het einde van het bestand;
^	naar het begin van een lijn
$	naar het einde van een lijn
/*zoekarg*	naar het eerst volgende voorkomen van *zoekarg*
n	naar het volgende voorkomen van *zoekarg*
?	naar het vorige voorkomen van *zoekarg*

Tekst verwijderen kan op verschillende manieren gebeuren:

dd	de lijn waarop de cursor staat
dw	het woord waarop de cursor staat
D	vanaf de cursor tot op het einde van de lijn
x	het teken op de positie waar de cursor staat
r	overschrijven van één teken (geen verwijdering)

Tekst copiëren:

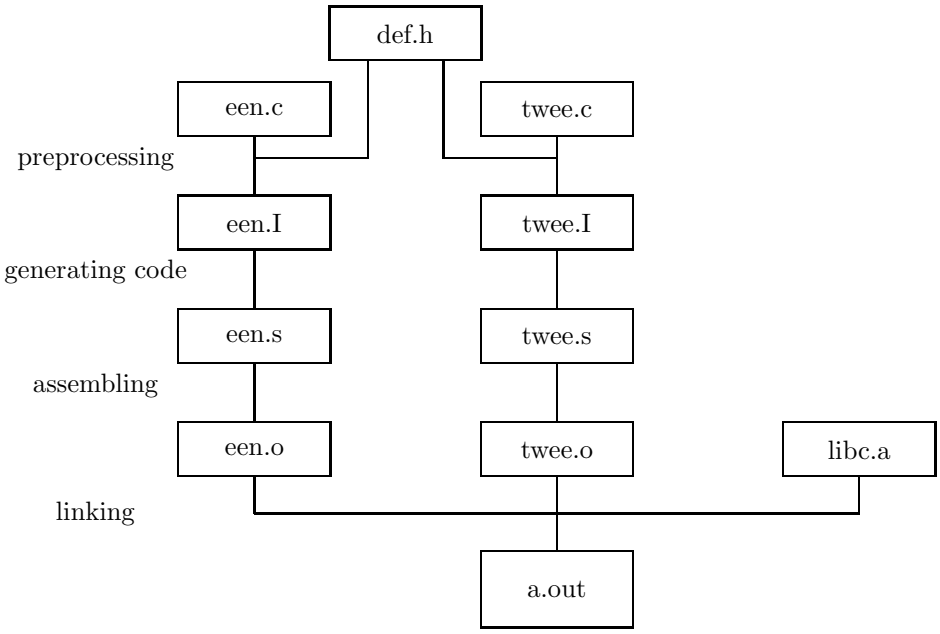
yy	de lijn waarop de cursor staat wordt in een buffer geplaatst
p	inhoud van de buffer wordt toegevoegd na de cursorlijn
P	inhoud van de buffer wordt toegevoegd voor de cursorlijn
J	samenvoegen van twee opeenvolgende lijnen

Het *yy* en *dd* kan voorafgegaan worden door een getal. In dat geval wordt de bewerking uitgevoerd op het gespecificeerde aantal lijnen vanaf de lijn waarop de cursor staat.
Algemene commando's:

ZZ	bewaar de veranderingen en verlaat de editor
:wq	bewaar de veranderingen en verlaat de editor
:q!	verlaat de editor zonder de veranderingen te bewaren
.	herhaal vorige actie
u	maak vorige actie ongedaan
U	herstel de lijn in haar vorige toestand

A.4 Programma ontwikkeling

De source van het programma moet eerst ingetikt worden m.b.v. bijvoorbeeld de vi-editor. Daarna moet het gecompileerd en gelinkt worden. Op de meeste Unix systemen zijn verschillende compilers beschikbaar: bijvoorbeeld *pc*: pascal, *cc*: c en *fc*: fortran. Een compiler werkt meestal in verschillende passen:



Vanuit de bronbestanden een.c en twee.c en een header file def.h wordt een executable gemaakt welke in het bestand *a.out* terecht komt.
Wanneer het programma niet foutloos werkt, kan beroep gedaan worden op de **GNU** debugger. Hiervoor moet het programma wel met de **-g** optie gecompileerd en gelinkt zijn. De debugger starten gebeurt met het bevel :

```
gdb [laadnaam]            (te vinden in /opt/langtools/bin)
```

De debugger geeft aan de gebruiker aan dat zij klaar is om iets te doen met een prompt teken, wat in dit geval `(gdb)` is.
Een beperkte lijst van commands:

q (van quit) debugger beëindigen;

l display van de volgende 10 lijnen

l - display van de vorige 10 lijnen

l van-lijnno,tot-lijnno display van enkele lijnen uit het bronprogramma,
met *lijnnummer* wordt de nummer van een lijn bedoeld in het bronprogramma;

b lijnnummer (van break) een breekpunt plaatsen op een lijn;

d breekpuntnummer (van delete) een breekpunt verwijderen;
de breekpuntnummers kan u vinden met behulp van **info break**;

r (van run) het programma starten;

c (van continue) het programma laten verder werken tot het volgend breekpunt;

s (van single step) het programma lijn voor lijn uitvoeren, eventueel kan dmv. een getal het aantal lijnen opgegeven worden dat moet uitgevoerd worden;

n *aantal* (van next) het programma een *aantal* lijnen laten verder werken (miv. functie-oproepen), nodig om niet in functies van de standaard C-library te sukkelen;

p [/*formaat*] naam van variabele : opvragen van de waarde van een variabele

formaat kan een van volgende letters zijn: d(ecimaal), f(loating point), x(hexadecimaal), c(haracter), a(dres);

x [/Nuf] expressie (naam van een variabele) : inhoud van opeenvolgende geheugenplaatsen

N is het aantal opeenvolgende eenheden; u geeft de eenheid: b(ytes), h(alfword), w(ord), g(iant) en f is zoals hierboven;

bt trace van alle frames van de stack (m.i.v. argumenten);

l bronbestandsnaam:lijnnummer veranderen van bronbestand:

dit bevel heb je alleen nodig indien het laadprogramma samengesteld is uit meerdere bronbestanden.

Wanneer een programma uit verschillende bronbestanden bestaat, kan de utility *make* gebruikt worden. Make doet beroep op een `makefile` waarin aangegeven wordt hoe een executable tot stand kan komen:

```
#makefile voor bovenstaand voorbeeld
all: a.out
CFLAGS=-g
LFLAGS=-g

een.o: een.c def.h
           cc -c $(CFLAGS) een.c

twee.o: twee.c def.h
           cc -c $(CFLAGS) twee.c

a.out: een.o twee.o
           cc -o a.out $(LFLAGS) een.o twee.o
```

Het doelbestand a.out wordt gemaakt op basis van twee object bestanden die zelf tot stand komen door een aantal bronbestanden te compileren.
De eerste lijn is een commentaarlijn (aangeduid door het # teken). De tweede lijn geeft het vooropgestelde (*default* voor de kenners) doel weer, wanneer geen argument weergegeven wordt bij het oproepen van make. Dus de bevelen *make* en *make all* hebben hetzelfde effect. Op de twee volgende lijnen worden de variabelen CFLAGS en LFLAGS gedefiniëerd. Deze worden in de volgende bevelen gebruikt om aan te geven hoe de compilatie en linking moet gebeuren.
Daarna volgen de afhankelijkheden de de bevelen nodig om van het bronbestand of -bestanden het doelbestand te maken:

```
<doel> : afhankelijk van {<bron>}
TAB-teken UNIX-bevel
[{TAB-teken UNIX-bevel}]
```

Deze UNIX-bevelen worden enkel uitgevoerd wanneer één van de < *bron* >bestanden jonger (recentere modificatietijd) is dan het < *doel* >bestand, of wanneer het < *doel* >bestand nog niet bestaat.

A.5 Het proces systeem

Enkele commando's:

ps: produceert een lijst van processen (opties: **-u** *loginnaam*, **-e**, **-f**, **-l**);

```
  PID TTY      TIME COMMAND
13849 pts/2    0:00 csh
14153 pts/2    0:00 ps
```

nice: verandert de prioriteit van een proces (bijv. `nice +4 ps -l`);

time: chronometreert de uitvoering van een proces (bijv. `time ps -ef`);

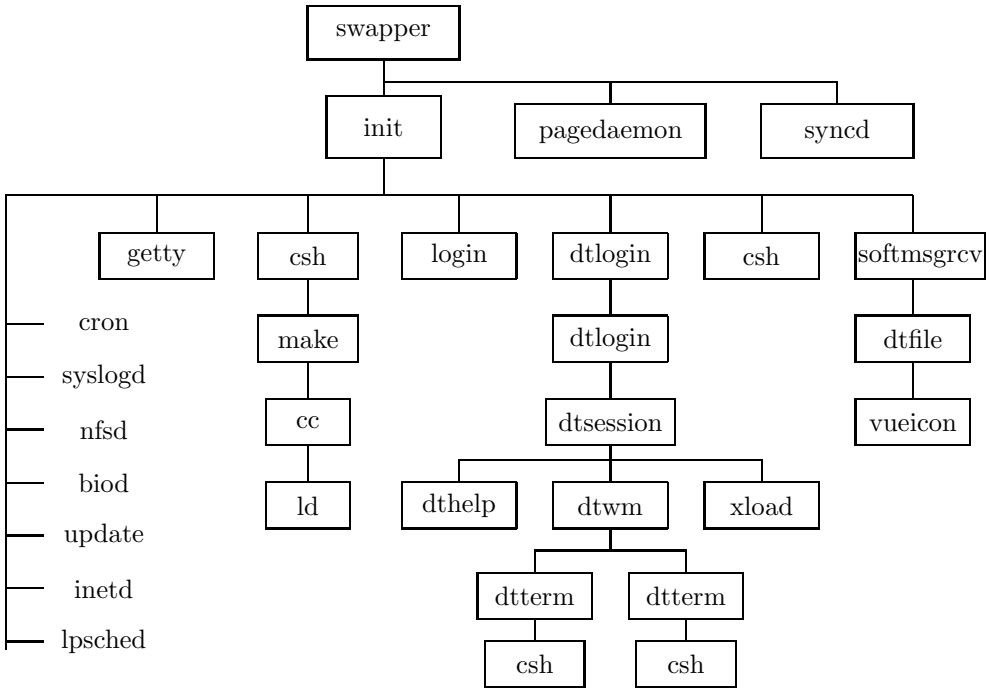
kill: stuurt een signaal naar een proces, meestal met de bedoeling het proces te laten stoppen (lijst met mogelijke signalen: `kill -l`);

CTRL C: breekt een proces af (op sommige toetsenborden BREAK);

CTRL Z: onderbreekt een proces;

fg/bg: een proces wordt voortgezet in de voorgrond/de achtergrond.

Het proces systeem is georganiseerd als een boom. Op proces 0 na, wordt elk proces gecreëerd door een ouderproces:



Enkelvoudige commando's en primitieven van de shell vormen bouwstenen om complexere commando's te bouwen:

&	het proces wordt in de achtergrond gestart
|	pipe: de output van het eerste proces is de input van het tweede proces
<	haalt de gegevens (standaard input) uit het aangegeven bestand
>	stuurt de resultaten (standaard output) naar het aangegeven bestand (eventueel wordt dit eerst gecreëerd)
>>	voegt de resultaten achteraan aan een bestaand bestand bij
tee	stuurt de input zowel naar standaard output als naar het aangegeven bestand

A.6 Het netwerk systeem

Naar de gebruiker toe wordt het netwerk vooral gebruikt voor het transport van bestanden tussen DOS en UNIX via ftp. Dit gebeurt met de volgende procedure.

Op PC:

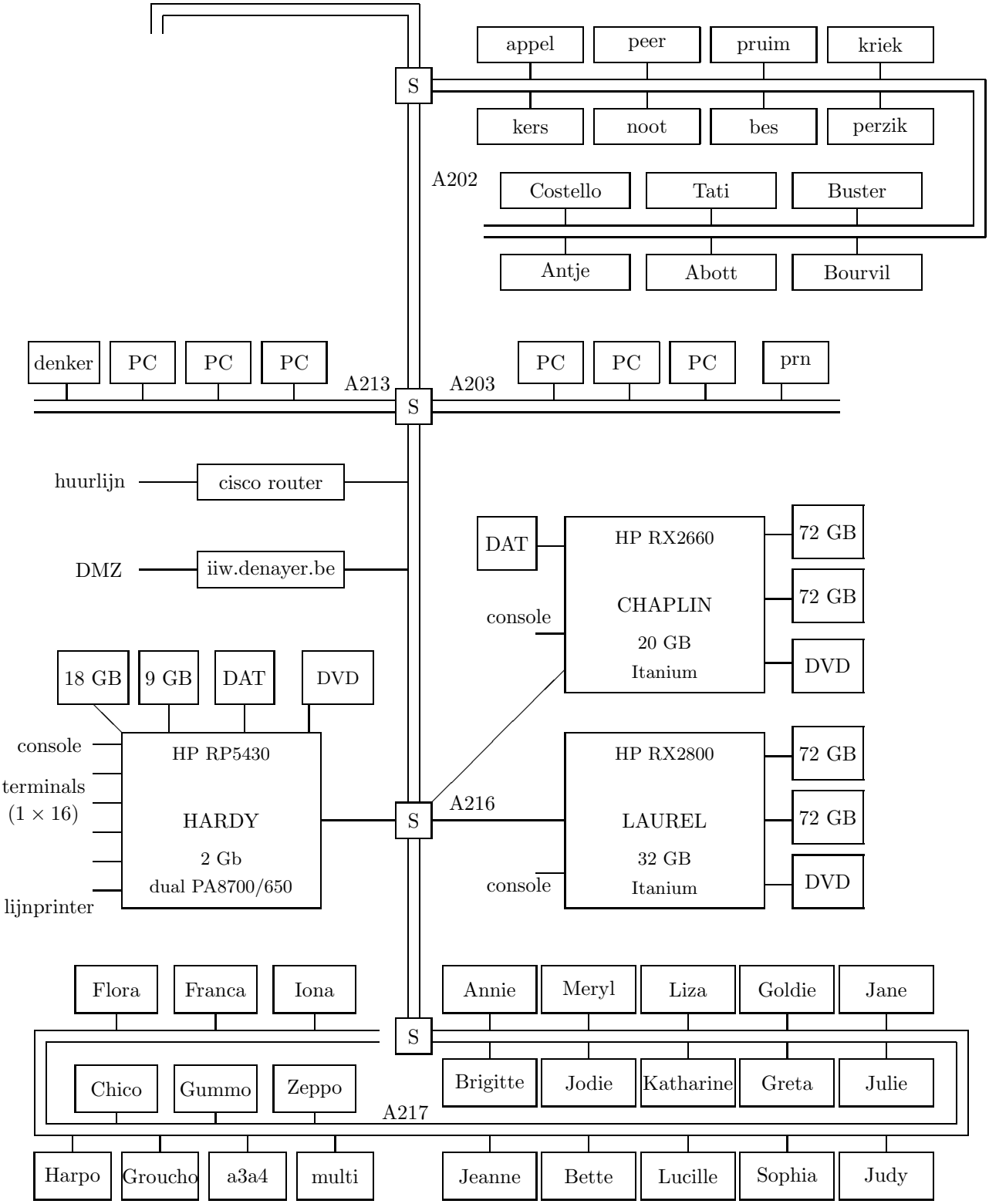
1. inbrengen diskette/USB-stick in a: of x: drive
2. opstarten: *ftp chaplin* (naam van een UNIX machine, dus abott is ook mogelijk)
3. hier moet ingelogd worden op chaplin: loginnaam: ..... paswoord: .....
4. de prompt is ftp>
5. *drive a:* (3.5 inch) OF *drive x:* (USB-stick)
6. lcd \dos-directory (naar keuze)
7. cd unix-directory
8. van DOS->UNIX: *mput *.c* (bijvoorbeeld)
9. van UNIX->DOS: *mget *.c* (bijvoorbeeld)
10. andere nuttige bevelen:

    help : lijst van commands
    lmkdir : dos-directory (op de diskette dus)
    mkdir : unix-directory (op chaplin)
    lpwd : print directory van dos
    pwd : print directory van unix
    ldir : inhoud lokale directory van dos
    ls : inhoud remote directory van unix
    dir : lange inhoud remote directory van unix

11. transfer gebeurt in een bepaalde mode, is op te vragen met status

    bin : binair (geen enkele conversie van tekens)
    ascii : vertaling:
        UNIX->DOS (get) : \n wordt \n\r
        DOS->UNIX (put) : \n\r wordt \n

12. drive c:
13. stoppen met *quit* of *bye*.



Figuur A.1: Configuratieschema

B Oefening 1: Schema van Horner.

Deling van een veelterm door een lineaire factor:

$$\frac{a_0x^n + a_1x^{n-1} + a_2x^{n-2} + \ldots + a_{n-1}x + a_n}{x - x_0} = b_0x^{n-1} + b_1x^{n-2} + \ldots + b_{n-2}x + b_{n-1} + \frac{b_n}{x - x_0}$$

De b_i coëfficiënten kunnen als volgt berekend worden:

$$\begin{cases} b_0 &= a_0 \\ b_i &= a_i + x_0 \times b_{i-1} \quad \text{voor } i = 1, 2, \ldots, n \end{cases}$$

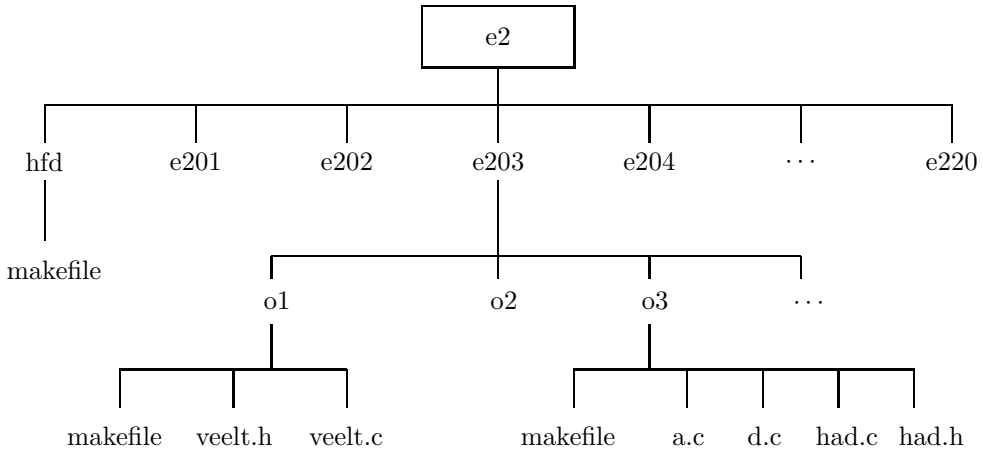
Indien b_n gelijk is aan 0, dan is x_0 een nulpunt van de veelterm.

$$\text{Voorbeeld: } x^4 + x^3 - 16x + 8 = (x - 2)(x^3 + 3x^2 + 6x - 4)$$

Opgave.

- Schrijf een C-functie, met de arrays a en b, de graad n en een waarde x_0 als parameters, die volgens het schema van Horner het quotiënt en de rest in de array b berekent. De returnwaarde is de rest.
- Schrijf een `main`-functie die de graad van de veelterm (max 9) in de variabele `n` inleest en de coëfficiënten in de elementen `a[0], a[1], ... a[n]` van een array `a`. Dan wordt de functie een aantal maal opgeroepen met voor het vierde argument de waarden -2, -1, 0, 1 en 2. Indien de teruggeefwaarde nul is, wordt het quotiënt uitgeschreven; in het andere geval de rest.

Organisatie van de bestanden.
De login-directories van de verschillende groepen studenten bevinden zich in de klas directory **e2**. In deze klas directory is er ook een *hfd* directory aanwezig. In deze directory staan een aantal bestanden die tijdens de oefeningen moeten gebruikt worden.



In de login directory worden een aantal subdirectories gemaakt, één per oefening. In zo'n subdirectory komen de verschillende bronbestanden (`.c` en `.h`), de makefile, de objectbestanden en het uitvoerbaar bestand.
In de **hfd** directory vindt men meestal per oefening een include-bestand en een bronbestand met het hoofdprogramma. Deze dingen moeten dus niet meer ingetikt worden.
In directory `o1` worden een include bestand `veelt.h` (voor de defines en de prototypes) en een bronbestand gecreëerd: *veelt.c* voor de main en deling functies.

Demo+verslag: tijdens de 1-e labo-zitting.

C Oefening 2: bewerkingen op bestanden

Gegeven een tekstbestand met per lijn volgende gegevens: hoteltype, hotelnaam, plaats, aantal kamers, omzet, land.

Deze velden zijn van elkaar gescheiden met enkele spaties.
Een voorbeeld van een gegeven tekstbestand:

```
rust      Luca_Faliraki     Rhodos       200    1000.0     Griekenland
sport     Sheraton          Parijs       400    4000.5     Frankrijk
rust      Hilton            Brussel      205    2300.4     Belgie
zon       Aqua_Sol          Paphos       120     234.5     Cyprus
```

Schrijf een programma dat een aantal bewerkingen kan uitvoeren op het gegeven bestand en op twee bij te maken binaire bestanden. Deze bewerkingen worden via een menu geselecteerd.

```
char *menutekst[] =
{   "",
    " tekst->binair",
    " lijst",
    " indiceren",
    " zoeken",
    " stoppen",
      0
};
```

De bewerkingen zijn:

1. Het **omvormen** van een tekstbestand naar een binair bestand.
   Met behulp van `fscanf` kan een lijn van het tekstbestand gelezen worden. Op basis van de spaties tussen de verschillende woorden en getallen splitst deze functie zo'n lijn in drie strings, een integer, een float en een string. Deze kunnen in een structure van type `Hotelinfo` ingevuld worden. Het veld `nr` is een volgnummer te beginnen vanaf nul. Wanneer de structure volledig ingevuld is, kan die weggeschreven worden mbv `fwrite`.

   ```
   nr     naam           plaats        land          type  aantal  omzet
   012301234567890123456701234567890123012345678901230123450123012 3
      0 Luca_Faliraki    Rhodos        Griekenland   rust   200   1000.0
      1 Sheraton         Parijs        Frankrijk     sport  400   4000.5
      2 Hilton           Brussel       Belgie        rust   205   2300.4
      3 Aqua_Sol         Paphos        Cyprus        zon    120    234.5
   ```

2. Het maken van een **lijst** met de gegevens van het binair bestand.
   De structures van het binair bestand worden één voor één gelezen en afgedrukt. Eventueel kan de naam van het *type* ingegeven worden, zodat alleen hotels van dat *type* getoond worden.

3. Het maken van een indexbestand op dit binair bestand (**indiceren**).
   De structures van het binair bestand worden één voor één gelezen. Met de binaire zoekmethode wordt bepaald waar het element in de tabel moet toegevoegd worden. De `naam` samen met de structure nummer wordt toegevoegd in een tabel van maximaal 100 `Tabel` elementen zodat de tabel alfabetisch geordend blijft.

   ```
   eerste stap        tweede stap       derde stap         vierde stap
   index  naam        index  naam       index  naam        index  naam
      0   Luka_Faliraki  0  Luka_Faliraki 2   Hilton          3   Aqua_Sol
                         1  Sheraton      0   Luka_Faliraki   2   Hilton
                                          1   Sheraton        0   Luka_Faliraki
                                                              1   Sheraton
   ```

4. Het binair **zoeken** van een record met behulp van het indexbestand.
   Er wordt een naam van het toetsenbord ingelezen. Het indexbestand wordt mbv `fread`

gelezen in een tabel met 100 elementen van type `Tabel`. Met het binair-zoek-algoritme kan snel nagegaan worden of de te zoeken naam in de tabel voorkomt. De bijhorende index kan gebruikt worden om de structure uit het binair bestand te lezen.

Voor het algoritme van *binair zoeken* kan volgende functie als inspiratie gebruikt worden:

```
//Een gegeven array a[] opgevuld met n integers en zoeken naar x

int binzoek( int a[],int n,int x)
{
    int midden,links=0,rechts = n-1;

    if ( x<= a[links])
        return 0;
    if ( x > a[rechts])
        return n;
    while ( rechts != links )
    {
        midden = ( links + rechts) /2;
        if ( x == a[midden] )
            return midden;
        if ( a[midden ] < x )
            links = midden + 1;
        else
            rechts = midden;
    }
    return links;
}
```

De gemeenschappelijke definities en declaraties worden verzameld in een *header*bestand:

```
/*
 * hotel.h  : definities en declaraties voor bestanden
 */
#define MAX 100
#define NLEN    6
#define LEN 14
#define HLEN    18
#define TNAAM "org.txt"
#define BNAAM "org.dat"
#define INAAM "org.ndx"

typedef struct
        {
            int     nr;
            char    hotel[HLEN];
            char    plaats[LEN];
            char    land[LEN];
            char    type[NLEN];
            int     aantkamers;
            float   omzet;
        } Hotelinfo;
```

```
typedef struct
        {
            int     index;
            char    naam[HLEN];
        } Tabel;

/*
 * hieronder: externe declaraties van functies en variabelen
 */
```

De verschillende routines moeten in verschillende bronbestanden geplaatst worden:

`hotel.c`	main:	hoofdprogramma met menu-keuze
`txtdat.c`	omvormen:	omvormen van tekst naar binair bestand
`lijst.c`	lijst:	afdrukken van de lijst uit het binair bestand
`index.c`	indiceren:	maken van een index bestand
`zoek.c`	zoeken:	zoeken van een structure via binair zoeken

De functie `binzoek()` wordt zowel in `indiceren()` als in `zoeken()` opgeroepen. Toch moet de definitie van deze functie maar eenmaal geschreven worden, bijvoorbeeld in het bronbestand `index.c`.
Om te compileren en te linken kan men een `makefile` gebruiken.

Uitbreiding: twee extra acties: verwijderen en toevoegen.
Bij *verwijderen* wordt de naam ingelezen; de bijhorende structure wordt via "binair zoeken" opgezocht en het `hotel` veld wordt volledig op `\0` gezet.
Bij *toevoegen* worden de zes velden via een eenvoudige invoerfunctie ingelezen van het toetsenbord en de resulterende structure wordt achteraan aan het binair bestand toegevoegd (of op een vrije plaats).
In beide gevallen moet ook de inhoud van het indexbestand aangepast worden.

Demo+verslag: tijdens de 3-e labo-zitting.
Het verslag bevat een listing van de geschreven routines en een korte tekst (+ figuren) waarin de verschillende routines besproken worden.

D Oefening 3: Geheugenbeheer.

Gegevensstructuur:
Het geheugen wordt voorgesteld door een 1-dimensionale array van TOTAAL elementen (bijv. 50). Het nul-de element van deze array heeft een speciale functie; de volgende elementen vormen telkens één geheugenplaats. Stukken van deze array kunnen gealloceerd en daarna terug gedealloceerd worden. Welke delen vrij en welke delen bezet zijn, wordt aangegeven door een ketting van vrije zones: het eerste element van elke vrije zone (4 bytes) geeft het aantal vrije plaatsen van deze zone (2 bytes) en de index in de array waar de volgende vrije zone start (2 bytes); de volgende elementen hebben geen bijzondere betekenis. Het eerste vrije blok van de ketting wordt aangegeven door het nul-de element van de array.
Bij initialisatie is het volledige geheugen vrij. Dus

```
mem[0] = (1<<16) | 0     /* eerste vrije zone op index 1 */
mem[1] = (-1<<16) | 49   /* er is geen volgende vrije zone */
```

Allocatie routine: 2 argumenten:
mem : de array met vrije plaatsen.
gevraagd : het aantal te alloceren elementen.
Deze routine zoekt in de ketting van vrije zones de eerste zone die voldoende groot is en hiervan worden *gevraagd* elementen vooraan afgenomen, de vrije zone wordt dus kleiner.

```
Voorbeeld: VOOR       mem[ 1] = (-1<<16) | 49 ;
           NA "a 20"  mem[ 1] tot mem[20]: gealloceerd
                      mem[0] = (21<<16) | 0
                      mem[21] = (-1<<16) | 29
```

Deallocatie routine: 3 argumenten:
mem : de array met vrije plaatsen.
waar : vanaf welke index elementen vrijgegeven moeten worden;
gevraagd : het aantal vrij te geven elementen.
Op de plaats *waar* wordt een nieuwe vrije zone in de ketting van vrije zones ingelast.

```
Voorbeeld: VOOR     mem[0] = (21<<16) | 0
                    mem[21] = (-1<<16) | 29
           NA "d 11 8"     mem[0] = (11<<16) | 0
                           mem[11] = (21<<16) |  8
                           mem[21] = (-1<<16) | 29
```

Merk op. Indien de vrije zone vooraan of achteraan aansluit bij een andere vrije zone, dan wordt deze bestaande vrije zone uitgebreid.

```
Voorbeeld: VOOR    mem[0] = (11<<16) | 0
                   mem[11] = (21<<16) |  8
                   mem[21] = (-1<<16) | 29
           NA "d 19 2"   mem[0] = (11<<16) | 0
                         mem[11] = (-1<<16) | 39
```

Testprogramma.
Om de twee routines te testen is een hoofdprogramma geschreven, dat strings van de vorm **a 20** en **d 20 10** leest, ontleedt en de corresponderende routines oproept. Er is ook een routine voorzien die gans het geheugen afdrukt zodat de ketting van vrije zones kan gecontroleerd worden. Hierbij is de onderstelling gemaakt dat een element met waarde TOTAAL vrij is en een element met waarde -TOTAAL bezet.
In de **hfd** directory vindt men per oefening een include-bestand en een bronbestand met het hoofdprogramma. Deze dingen moeten dus niet meer ingetikt worden. De bestanden *had.c* en *had.h* horen bij oefening 3.

Uitbreiding.
Allocatie : in plaats van de eerste vrije zone die groot genoeg is, te alloceren (*first fit*) kan gezocht worden naar de vrije zone die het dichtst het gevraagd aantal vrije plaatsen benadert (*best fit*).
Deallocatie : toevoegen van testen om te zien of geen vrij geheugen opnieuw vrij gegeven wordt.

Demo+verslag: begin van de 6-e labo-zitting.
Het verslag bevat een listing van de geschreven routines en een korte tekst (+ figuren) waarin de twee routines besproken worden.

Listing van hoofdprogramma:

```
/*
 * had.h : allocatie en deallocatie : deklaraties en definities
 */
#define TOTAAL  50
#define LEN  40
#define NILP    -1

int alloc(int mem[], int gevraagd) ;
int dealloc(int mem[], int waar, int gevraagd) ;
```

```
/*
 * had.c : allocatie en deallocatie algoritme
 */
#include   <stdio.h>
#include   <stdlib.h>
#include   "had.h"

int main( int argc, char *argv[] )
{
   register int   n, p, waar, gevraagd ;
   int   mem[TOTAAL];
   char   str[LEN] ;
   char   *ptr ;

   while ( gets(str) != NULL )
   {
      switch(str[0])
      {
      case 'a' :
         gevraagd = atoi( &str[2] ) ;
         n = alloc(mem, gevraagd) ;
         break ;
      case 'd' :
         waar = strtol(&str[2], &ptr, 10);
         gevraagd = strtol(ptr+1, (char **)NULL,10);
         n = dealloc(mem, waar, gevraagd) ;
         break ;
      case 'v' :
         for ( p=0, n=0; p != (int)NILP ; p = mem[p]>>16 )
         {
            printf("%4d (%2x) : %4d %4d\n", p,p,mem[p]>>16,mem[p]&0xffff);
            n += mem[p]&0xffff ;
```

```
33              }
                printf("Totaal vrij: %4d\n", n) ;
35              break ;
            case 'z' :
37              for (p=0; p<TOTAAL; p++)
                {
39                  printf("%3x:", p);
                    if ( mem[p] == TOTAAL )
41                      printf("%8c", ' ');
                    else if ( mem[p] == -TOTAAL )
43                      printf("  ++++++");
                    else
45                      printf("%8x", mem[p]);
                    printf("%c", ((p+1)%6 ? ' ':'\n') );
47              }
                printf("\n") ;
49              break ;
            case 'i' :
51              mem[0] = (1<<16) | 0 ;
                mem[1] = (NILP<<16) | (TOTAAL-1) ;
53              for(p=2; p<TOTAAL; p++ )
                    mem[p] = TOTAAL ;
55              break ;
            case 't' :
57              for(p=1; p<TOTAAL; p++ )
                     mem[p] = TOTAAL ;
59              mem[0] = (11<<16)|0;
                for(p=1; p<=10; p++ )
61                   mem[p] = -TOTAAL ;
                mem[11] = (21<<16) | (8) ;
63              for(p=19; p<=20; p++ )
                     mem[p] = -TOTAAL ;
65              mem[21] = (NILP<<16) | (29) ;
                break ;
67          default:
                printf("q(uit) i(nit) t(est) z(ien) v(rij) a<aant> d<waar aant>\n");
69              break ;
            case 'q' :
71              exit(0);
            }
73      }
    }
```

E Oefening 4: Recursie

1. Een recursieve functie voor de vermenigvuldiging van twee positieve gehele getallen m en n, door middel van een opeenvolging van optellingen.

2. Een recursieve functie voor de grootste gemene deler van twee positieve gehele getallen a en b gebaseerd op het algoritme van Euclides: indien $a > b$ dan is $\text{ggd}(a, b) = \text{ggd}(a - b, b)$ en $\text{ggd}(a, 0) = a$.

3. Een recursieve functie voor de machtsverheffing x^n, met n een positief geheel getal: $x^n = x^{n/2} \times x^{n/2}$ (met n even) en $x^n = x \times x^{n/2} \times x^{n/2}$ (met n oneven).

4. Een recursieve functie voor het zoeken van het grootste element in een array a met elementen van 1 tot n: $\text{rmax}(a, n) = \max \{ \text{rmax}(a, n - 1) \ , \ a[n] \}$.

5. Een recursieve functie om na te gaan of a een macht is van b: een getal a is een macht van getal b indien het getal a deelbaar is door het getal b en indien a/b een macht is van b.

6. Een recursieve functie voor het oplossen van het probleem van Josephus (zie hoofdstuk 9).

Demo+verslag: tijdens de 6-e labo-zitting.

F Oefening 5: Gelinkte lijsten

Gegeven: een binair bestand met records van het type Hotel (zie oefening 2):

```
typedef struct
        {
            int     nr;
            char    hotel[HLEN];
            char    plaats[LEN];
            char    land[LEN];
            char    type[NLEN];
            int     aantkamers;
            float   omzet;
        } Hotelinfo;
```

Gevraagd:

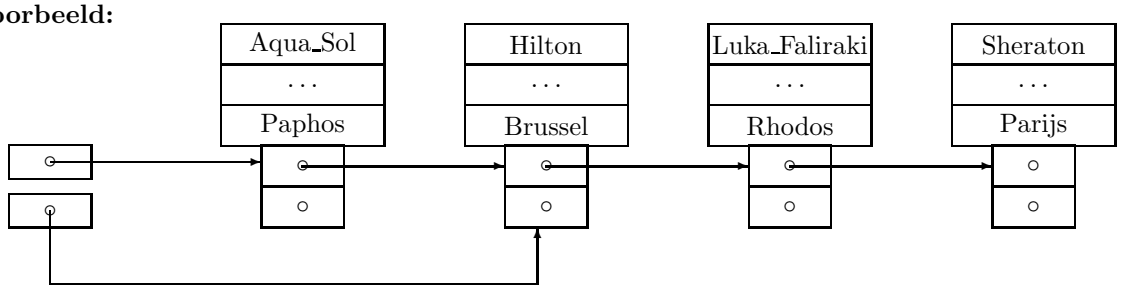
1. Schrijf een functie die een lineaire lijst maakt van deze records. Lees het bestand sequentieel. Voeg telkens de ingelezen record in de lijst in zodat een op `naam` alfabetisch geordende lijst ontstaat.
2. Schrijf een functie die de elementen van deze lineaire lijst in een circulaire lijst plaatst zodat een op `plaats` alfabetisch geordende lijst ontstaat.
3. Schrijf een functie die deze lineaire lijst doorloopt en nagaat of er in een element een bepaalde gegeven substring voorkomt.
4. Schrijf een programma dat deze functies oproept en voorzie afdrukfuncties zodat de goede werking kan gedemonstreerd worden.

Gebruik volgende type-definitie voor de elementen van de lijst:

```
#define  LIN   0
#define  CIRC  1
typedef  struct snode
         {
             Hotelinfo  a;
             struct snode *next[2];
         } Node;
#define  NULN  ((Node *)NULL)
```

In de eerste functie wordt met behulp van `next[LIN]` de lineaire lijst gebouwd. Het veld `next[CIRC]` wordt gebruikt voor de circulaire lijst.
De verschillende routines moeten in verschillende bronbestanden geplaatst worden: `hfd.c`, `linlijst.c`, `circlijst.c` en `druklijst.c`. Maak hiervoor een passende `makefile`.

Voorbeeld:



Demo+verslag: tijdens de 9-e labo-zitting.

G Oefening 6: Huffman boom

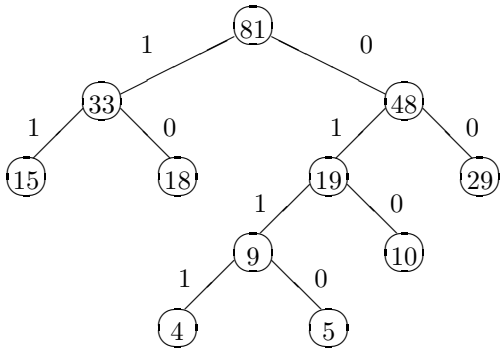
Opgave: schrijf een C-programma met drie hoofdfuncties.

- Een functie die een tekstbestand leest met per lijn een letter en bijhorende frequentie waarde. Op basis van de resulterende tabel kan een Huffman boom gebouwd worden.
- Een codeer-functie met als argument een woord: op basis van de geconstrueerde boom, wordt de bijhorende bitstring berekend.
- Een decodeer-functie met als argument een bitstring: op basis van de geconstrueerde boom, wordt het bijhorende woord berekend.

De main-functie heeft als argument de naam van het bestand. In deze functie kunnen van toetsenbord woorden en bitstring gelezen worden om de tweede en derde functie te testen.

Een tweede voorbeeld:

letter	frequen	bit code
r	4	
t	5	
u	10	
a	15	
d	18	
e	29	



Uitbreidingen:

- een gecodeerde bitrij hexadecimaal per 4 bits uitschrijven; eventueel moet de laatste nible achteraan aangevuld worden met nullen;
- een tekst inlezen; hieruit een tabel met frequenties opstellen en de ingelezen tekst coderen. Is er een compressie?

Demo+verslag: tijdens de laatste labo-zitting.

H ANSI C standard library.

H.1 <stdio.h>

`FILE` Type which records information necessary to control a stream.

`stdin` Standard input stream. Automatically opened when a program begins execution.

`stdout` Standard output stream. Automatically opened when a program begins execution.

`stderr` Standard error stream. Automatically opened when a program begins execution.

`FILENAME_MAX` Maximum permissible length of a file name

`FOPEN_MAX` Maximum number of files which may be open simultaneously.

`TMP_MAX` Maximum number of temporary files during program execution.

`FILE* fopen(const char* filename, const char* mode);`
Opens file `filename` and returns a stream, or NULL on failure.

`FILE* freopen(const char* filename, const char* mode, FILE* stream);`
Opens file `filename` with the specified mode and associaties with it the specified stream. Returns stream or NULL on error. Usually used to change files associated with stdin, stdout, stderr.

`int fflush(FILE* stream);`
Flushes stream `stream`. Effect undefined for input stream. Returns EOF for write error, zero otherwise. `fflush(NULL)` flushes all output streams.

`int fclose(FILE* stream);`
Closes stream `stream` (after flushing, if output stream). Returns EOF on error, zero otherwise.

`int remove(const char* filename);`
Removes file `filename`. Returns non-zero on failure.

`int rename(const char* oldname, const char* newname);`
Changes name of file `oldname` to `newname`. Returns non-zero on failure.

`FILE* tmpfile();`
Creates temporary file (mode `"wb+"`) which will be removed when closed or on normal program termination. Returns stream or NULL on failure.

`int setvbuf(FILE* stream, char* buf, int mode, size_t size);`
Controls buffering for stream `stream`.

`void setbuf(FILE* stream, char* buf);`
Controls buffering for stream `stream`.

`int fprintf(FILE* stream, const char* format, ...`
Converts (with format `format`) and writes output to stream `stream`. Number of characters written [negative on error] is returned.

`int printf(const char* format, ... );` Equivalent to `fprintf (stdout, f, ...)`

`int sprintf(char* s, const char* format, ... );`
Like `fprintf()`, but output written into string `s`, which must be large enough to hold the output, rather than to a stream. Output is NULL-terminated. Return length does not include the NULL.

`int vfprintf(FILE* stream, const char* format, va_list arg);`
Equivalent to `fprintf()` except that the variable argument list is replaced by `arg`, which must have been initialised by the `va_start` macro and may have been used in calls to `va_arg`. See `<stdarg.h>`

`int vprintf (const char* f ormat, va-list arg)`
Equivalent to `printf()` except that the variable argument list is replaced by `arg`, which must have been initialised by the `va_start` macro and may have been used in calls to `va_arg`. See `<stdarg.h>`

`int vsprintf (char* s, const char* f ormat, va-list arg)`
Equivalent to `sprintf()` except that the variable argument list is replaced by `arg`, which must have been initialised by the `va_start` macro and may have been used in calls to `va_arg`. See `<stdarg.h>`

`int fscanf(FILE* stream, const char* format, ...`
Performs formatted input conversion, reading from stream `stream` according to format `format`. The function returns when `format` is fully processed. Returns EOF if end-of-file or error oceurs before any conversion; otherwise, the number of items converted and assigned. Each of the arguments following `format` must be a pointer.

`int scanf(const char* format, ...` Equivalent to `fscanf(stdin, f , ... )`

`int sscanf(char* s, const char* format, ...`
Like `fscanf()`, but input read from string `s`.

`int fgetc(FILE* stream);`
Returns next character from stream `stream` as an unsigned char, or EOF on end-of-file or error.

`char* fgets(char* s, int n, FILE* stream);`
Reads at most the next n-1 characters from stream `stream` into `s`, stopping if a newline is encountered (after copying the newline to `s`). `s` is NULL-terminated. Returns `s`, or NULL on end-of-file or error.

`int fputc(int c, FILE* stream);`
Writes `c`, converted to unsigned char, to stream `stream`. Returns the character written, or EOF on error.

`int fputs(const char* s, FILE* stream);`
Writes `s`, which need not contain `'\n'` on stream `stream`. Returns non-negative on success, EOF on error.

`int getc(FILE* stream);` Equivalent to `fgetc()` except that it may be a macro.

`int getchar();` Equivalent to `getc(stdin)`.

`char* gets(char* s);`
Reads next line from stdin into `s`. Replaces terminating newline with `'\0'`. Returns `s`, or NULL on end-of-file or error.

`int putc(int c, FILE* stream);` Equivalent to `fputc()` except that it may be a macro.

`int putchar(int c);` Equivalent to `putc(c, stdout)`.

`int puts(const char* s);`
Writes `s` and a newline to stdout. Returns non-negative on success, EOF on error.

`int unget(int c, FILE* stream);`
Pushes c (which must not be EOF), converted to unsigned char, onto stream `stream` such that it will be returned by the next read. Only one character of pushback is guaranteed for a stream. Returns `c`, or EOF on error.

`size_t fread(void* ptr, size_t size, size_t nobj, FILE* stream);`
Reads at most `nobj` objects of size `size` from stream `stream` into `ptr`. Returns the number of objects read. `feof` and `ferror` must be used to determine status.

`size_t fwrite(const void* ptr, size_t size, size_t nobj, FILE* stream);`
Writes to stream `stream`, `nobj` objects of size `size` from array `ptr`. Returns the number of objects written (which will be less than `nobj` on error).

`int fseek(FILE* stream, long offset, int origin);`
Sets file position for stream `stream`. For a binary file, position is set to `offset` characters from `origin`, which may be `SEEK_SET` (beginning), `SEEK_CUR` (current position) or `SEEK_END` (end-of-file); for a text stream, `offset` must be zero or a value returned by `ftell` (in which case origin must be `SEEK_SET`). Returns non-zero on error.

`long ftell(FILE* stream);`
Returns current file position for stream `stream`, or -1L on error.

`void rewind(FILE* stream);`
`rewind (stream)` is equivalent to `fseek (stream, OL, SEEK_SET) ; clearerr (stream)`.

`int fgetpos(FILE* stream, fpos_t* ptr);`
Assigns current position in stream `stream` to `*ptr`. Type `fpos_t` is suitable for recording such values. Returns non-zero on error.

`int fsetpos(FILE* stream, const fpos_t* ptr);`
Sets current position of stream `stream` to `*ptr`. Returns non-zero on error.

`void clearerr(FILE* stream);`
Clears the end-of-file and error indicators for stream `stream`.

`int feof(FILE* stream);`
Returns non-zero if end-of-file indicator for stream `stream` is set.

`int ferror(FILE* stream);`
Returns non-zero if error indicator for stream `stream` is set.

`void perror(const char* s);`
Prints `s` and implementation-defined error message corresponding to `errno`:
`fprintf(stderr, "%s: %s\n", s, "error message")` See `strerror`.

H.2 <stdlib.h>

`double atof(const char* s);`
Returns numerical value of `s`. Equivalent to `strtod(s, (char**)NULL)`.

`int atoi(const char* s);`
Returns numerical value of `s`. Equivalent to `(int) strtol (s, (char**) NULL, 10)`.

`long atol(const char* s);`
Returns numerical value of `s`. Equivalent to `strtol (s, (char**) NULL, 10)`.

`double strtod(const char* s, char** endp);`
Converts prefix of `s` to double, ignoring leading quite space. Stores a pointer to any unconverted suffix in `*endp` if `endp` non-NULL. If answer would overflow, `HUGE_VAL` is returned with the appropriate sign; if underflow, zero returned. In either case, `errno` is set to ERANGE.

`long strtol(const char* s, char** endp, int base);`
Converts prefix of `s` to long, ignoring leading quite space. Stores a pointer to any unconverted suffix in `*endp` (if `endp` non-NULL). If `base` between 2 and 36, that `base` used; if zero, leading OX or Ox implies hexadecimal, leading O implies octal, otherwise decimal. Leading OX or Ox permitted for base 16. If answer would overflow, `LONG_MAX` or `LONG_MIN` returned and `errno` is set to ERANGE.

`unsigned long strtoul(const char* s, char** endp, int base);`
As for `strtol` except result is unsigned long and error value is `ULONG_MAX`.

`int rand();` Returns pseudo-random number in range 0 to `RAND_MAX`.

`void srand(unsigned int seed);`
Uses `seed` as seed for new sequence of pseudo-random numbers. Initial seed is 1.

`void* calloc(size_t nobj, size_t size);`
Returns pointer to zero-initialised newly-allocated space for an array of nobj objects each of size size, or NULL if request cannot be satisfied.

`void* malloc(size_t size);`
Returns pointer to uninitialised newly-allocated space for an object of size size, or NULL if request cannot be satisfied.

`void* realloc (void* p, size_t size)`
Changes to size the size of the object to which p points. Contents unchanged to minimum of old and new sizes. If new size larger, new space is uninitialised. Returns pointer to the new space or, if request cannot be satisfied NULL leaving `p` unchanged.

`void free(void* p);`
Deallocats space to which `p` points. `p` must be NULL, in which case there is no effect, or a pointer returned by `calloc()`, `malloc()` or `realloc()`.

`void abort();` Causes program to terminale abnormally, as if by raise(SIGABRT).

`void exit(int status);`
Causes normal program termination. Functions installed using `atexit` are called in reverse order of registration, open files are flushed, open streams are closed and control is returned to environment. `status` is returned to environment in implementation-dependent manner. Zero indicaties successful termination and the values `EXIT_SUCCESS` and `EXIT_FAILURE` may be used.

`int atexit(void (*fcm)(void));`
Registers `fcm` to be called when program terminates normally. Non-zero returned if registration cannot be made.

`int system(const char* s);`
Passes `s` to environment for execution. If `s` is NULL, non-zero returned if command processor exists, return value is implementation-dependent if `s` is non-NULL.

`char* getenv(const char* name);`
Returns (implementation-dependent) environment string associated with `name`, or NULL if no such string exists.

```
void* bsearch(const void* key, const void* base, size_t n, size_t size,
              int (*cmp)(const void*, const void*)
```
Searches `base[0] ... base[n-1]` for item matching `*key`. Comparison function `cmp` must return negative if first argument is less than second, zero if equal and positive if greater. The `n` items of `base` must be in ascending order. Returns a pointer to the matching entry or NULL if not found.

```
void qsort (void* base, size_t n, size_t size,
                int (*cmp)(const void*, const void*)
```
Arranges into ascending order the array `base[0] ... base[n-1]` of objects of size `size`. Comparison function `cmp` must return negative if first argument is less than second, zero if equal and positive if greater.

`int abs(int n);` Returns absolute value of n.

`long labs(long n);` Returns absolute value of n.

`div_t div(int num, int denom);`
Returns in fields `quot` and `rem` of structure of type `div_t` the quotient and remainder of `num/denom`.

`ldiv_t ldiv(long num, long denom);`
Returns in fields `quot` and `rem` of structure of type `ldiv_t` the quotient and remainder of `num/denom`.

H.3 <string.h>

`char* strcpy(char* s, const char* ct);`
Copy `ct` to `s` including terminating NULL. Return `s`.

`char* strncpy(char* s, const char* ct, int n);`
Copy at most `n` characters of `ct` to `s`. Pad with NULLs if `ct` is of length less than `n`. Return `s`.

`char* strcat(char* s, const char* ct);`
Concatenate `ct` to `s`. Return `s`.

`char* strncat(char* s, const char* ct, int n);`
Concatenate at most `n` characters of `ct` to `s`. Terminate `s` with NULL and return it.

`int strcmp(const char* cs, const char* ct);`
Compare `cs` and `ct`. Return negative if `cs < ct`, zero if `cs == ct`, positive if `cs > ct`.

`int strncmp(const char* cs, const char* ct, int n);`
Compare at most `n` characters of `cs` and `ct`. Return negative if `cs < ct`, zero if `cs == ct`, positive if `cs > ct`.

`char* strchr(const char* cs, int c);`
Return pointer to first occurrence of `c` in `cs`, or NULL if not found.

`char* strrchr(const char* cs, int c);`
Return pointer to last occurrence of `c` in `cs`, or NULL if not found.

`size_t strspn(const char* cs, const char* ct);`
Return length of prefix of `cs` consisting entirely of characters in `ct`.

`size_t strcspn(const char* cs, const char* ct);`
Return length of prefix of `cs` consisting entirely of characters not in `ct`.

`char* strpbrk(const char* cs, const char* ct);`
Return pointer to first oceurrence within `cs` of any character of `ct`, or NULL if not found.

`char* strstr(const char* cs, const char* ct);`
Return pointer to first occurrence of `ct` in `cs`, or NULL if not found.

`size_t strlen(const char* cs);` Return length of `cs`.

`char* strerror(int n);`
Return pointer to implementation-defined string corresponding with error `n`.

`char* strtok(char* s, const char* t);`
A sequence of calls to `strtok` returns tokens from `s` delimted by a character in `ct`. Non-NULL `s` indicaties the first call in a sequence. `ct` may differ on each call. Returns NULL when no such token found.

`void* memcpy(void* s, const void* ct, int n);`
Copy `n` characters from `ct` to `s`. Return `s`. Does not work correctly if objects overlap.

`void* memmove(void* s, const void* ct, int n);`
Copy `n` characters from `ct` to `s`. Return `s`. Works correctly even if objects overlap.

`int memcmp(const void* cs, const void* ct, int n);`
Compare first `n` characters of `cs` with `ct`. Return negative if `cs < ct`, zero if `cs == ct`, positive if `cs > ct`.

`void* memchr(const char* cs, int c, int n);`
Return pointer to first occurrence of `c` in first `n` characters of `cs`, or NULL if not found.

`void* memset(char* s, int c, int n);`
Replace each of the first `n` characters of `s` by `c`. Return `s`.

H.4 <stdarg.h>

Facilities for stepping through a list of function arguments of unknown number and type.

`void va-start (va list ap, lastarg) ;`
Initialisation macro to be called once before any unnamed argument is accessed. `ap` must be declared as a local variable, and `lastarg` is the last named parameter of the function.

`type va-arg(va list ap, type);`
Produce a value of the type (`type`) and value of the next unnamed argument. Modifies `ap`.

`void va-end(va list ap);`
Must be called once after arguments processed and before function exit.