Imperatief Programmeren

U kunt de auteur bereiken op het volgende adres: Jeroen Fokker, Vakgroep Informatica, Postbus 80089, 3508 TB Utrecht, e-mail `jeroen@cs.ruu.nl`.

1e druk (CKI-versie) mei 1994
2e druk (CKI-versie) mei 1995
3e druk (informatica-versie) september 1995
4e druk (CKI-versie) maart 1996

Inhoudsopgave

Hoofdstuk 1

Hallo!, *of:* de main-functie

In dit hoofdstuk bekijken we een kort, doch compleet programma. Het enige wat het programma doet is de gebruiker vriendelijk begroeten. We bespreken de elementen die in elk programma, en dus ook in dit programma, terug te vinden zijn. Tenslotte bekijken we hoe je een programma in de computer intikt en kan gebruiken.

1.1 Het programma

Het programma dat in dit hoofdstuk centraal staat heet `hallo`, omdat het de gebruiker begroet. In listing 1 staat de tekst van het programma. Als je dit programma intikt, en de computer vervolgens opdracht geeft het programma uit te voeren (zoals beschreven in hoofdstuk 0) wordt de tekst

```
Hallo!
```

op het scherm geschreven.

In dit korte programma worden al een paar aspecten gedemonstreerd, die in elk C++-programma van belang zijn. In de rest van deze paragraaf worden deze aspecten besproken.

1.2 Spelling

Allereerst: het is belangrijk om nauwkeurig te letten op de symbolen die in een programma gebruikt worden. Achter het woord `main` staat bijvoorbeeld een stel haakjes. Aan het begin van regel 4 en 6 staan daarentegen accolades. Alle andere symbolen in het programma (het hekje `#`, de punthaken `<` en `>`, de aanhalingstekens `"`, het uitroepteken `!`, en niet te vergeten de puntkomma `;`) staan met opzet in het programma en kunnen niet zomaar weggelaten of door iets anders vervangen worden.

In een C++-programma worden hoofdletters en kleine letters als verschillende symbolen beschouwd. Het woord `cout` in regel 5 bijvoorbeeld moet echt op die manier gespeld worden, en dus niet als `Cout` of `COUT`. Ook bij het op papier schrijven van een programma is het verstandig daarop te letten. Veel mensen beginnen bijvoorbeeld automatisch elke zin met een hoofdletter. Een programma is echter geen zin, en het woord `main` wordt dan ook niet met een hoofdletter geschreven (er staat ook geen punt aan het einde...).

```
#include <iostream.h>

int main()
{
5    cout << "Hallo!" << endl ;
}
```

Listing 1: hallo.cc

1.3 Opbouw van een programma

Elk C++-programma bestaat uit een aantal deelprogramma's, *functies* genaamd. De functies hebben ieder een naam, die je als programmeur zelf mag kiezen. Eén van de functies moet echter de naam `main` krijgen. Deze functie bevat het *hoofdprogramma*. De opdrachten in het hoofdprogramma worden door de computer uitgevoerd op het moment dat het programma gestart wordt. In het voorbeeldprogramma is de functie `main` de enige functie; het hele programma bestaat uit slechts één functie.

Achter de naam van de functie kun je, tussen een paar haakjes, aangeven of het gedrag van de functie nog ergens van afhankelijk is. In het voorbeeldprogramma is de functie `main` nergens van afhankelijk, en staat er dus niets tussen de haakjes. De haakjes zelf mag je niet weglaten.

Na de naam van de functie en de haakjes staan de opdrachten waaruit de functie bestaat. Een paar accolades (`{` en `}`) geeft aan welke opdrachten samen de functie vormen. Ook als een functie maar uit één opdracht bestaat, zoals in het voorbeeldprogramma, moeten de accolades er staan. Elke opdracht wordt afgesloten door een puntkomma (`;`).

1.4 Uitvoer

De enige opdracht in de functie `main` van dit programma is

```
cout << "Hallo!" << endl;
```

Met deze opdracht wordt tekst op het beeldscherm getoond. De naam `cout` is een aanduiding van de *uivoer-stream* van het programma. De uitvoer-stream is alle tekst die door het programma op het scherm wordt gezet; het heet een 'stream' omdat er steeds maar meer tekst kan volgen.

Met de symbolen `<<` kan tekst aan een stream worden toegevoegd. De tekst waar het om gaat wordt tussen aanhalingstekens (`"`) gezet. De aanhalingstekens zijn onontbeerlijk, omdat hiermee wordt aangegeven welke tekst letterlijk op het scherm gezet moet worden, en welke tekst tot het programma behoort. In het voorbeeld staat het uitroepteken (`!`) nog tussen de aanhalingstekens, en wordt dus op het scherm gezet. De tweede `<<` staat echter achter het sluit-aanhalingsteken, en behoort weer tot het programma.

Met één opdracht kunnen meerdere dingen naar de uitvoerstream gestuurd worden. Daartoe wordt het pijltjes-symbool `<<` herhaald. Het tweede ding dat in het voorbeeld naar het scherm wordt gestuurd is `endl`. Dit is geen letterlijke tekst, want er staan geen aanhalingstekens omheen. Het is de *naam* van een object dat 'einde van de regel' voorstelt (de naam is een afkorting van 'end line'). Het effect van het sturen van dit object naar de uitvoerstream is dat eventuele latere uitvoer op een volgende regel verschijnt. In dit programma is er weliswaar geen volgende uitvoer, maar toch heeft het sturen van `endl` naar de uitvoerstream zin: de tekst die een volgend programma op het scherm zal zetten, verschijnt nu tenminste op een nieuwe regel, en niet direct tegen de tekst `Hallo!` aan.

1.5 Include-regels

In het programma worden de namen `cout` en `endl` gebruikt, die ieder een bepaald object aanduiden: `cout` duidt de uitvoerstream aan, en `endl` een nieuwe regel op de stream waar hij met `<<` naar toe wordt gestuurd.

Ook `main` is een naam, ditmaal de naam van een functie. De regels 3 t/m 6 van het programma vormen de *definitie* van de functie genaamd `main`. In een programma kun je blijkbaar nieuwe namen definiëren, waardoor die namen een bepaalde betekenis krijgen. In de programma's in de volgende hoofdstukken gebeurt dat dan ook op grote schaal.

De objecten `cout` en `endl` worden echter in het programma niet gedefinieerd. Er is in C++ een groot aantal standaardobjecten en -functies beschikbaar. De definities daarvan zijn ondergebracht in *bibliotheken*. De definitie van de objecten `cout` en `endl` staat in de bibliotheek `<iostream.h>` (een afkorting van 'input/output streams'). Om deze bibliotheek te kunen gebruiken, staat aan het begin van het programma de regel

```
#include <iostream.h>
```

die aangeeft dat in het programma de objecten en functies uit de genoemde bibliotheek nodig zijn. Omdat vrijwel elke programma de uitvoerstream nodig heeft (een programma dat niet laat weten wat er gebeurd is, is weinig zinvol!), staat deze regel in de praktijk aan het begin van elk programma.

1.6 Lay-out

Een C++-programma mag naar believen over de regels van de listing verdeeld worden. Ook mogen op de meeste plaatsen extra spaties staan. Alleen de `#include`-regel moet op een aparte regel staan. Het programma hallo zou dus ook zo geschreven kunnen worden:

```
#include <iostream.h>
int main(){cout<<"Hallo!"<<endl;}
```

of zo:

```
#include <iostream.h>
       int main  (  )
{cout      <<
"Hallo!"<<endl
     ;            }
```

De lay-out van een programma mag dus vrij gekozen worden.

Dat de computer er geen bezwaar tegen heeft als bijna het hele programma op één regel staat, wil nog niet zeggen dat dat ook verstandig is: een programma wordt tenslotte ook nog wel eens door een mens gelezen (een mede-programmeur, of door jezelf). De bizarre lay-out in het laatste voorbeeld maakt het programma er ook niet duidelijker op.

In goede programma's helpt de lay-out je om het programma gemakkelijk te overzien. In listing 1 is bijvoorbeeld de regel met `cout` een stukje ingesprongen, en staat de sluit-accolade recht onder de openings-accolade.

Tussen de aanhalingstekens die rond een tekst staan mogen niet zomaar spaties worden toegevoegd. Of liever gezegd: het mag wel, maar de spaties gaan dan ook deel uitmaken van de tekst. De opdracht

```
cout << "H a l l o !";
```

zet de tekst

```
H a l l o !
```

op het scherm.

De enige plaats waar spaties niet zijn toegestaan is halverwege namen en symbolencombinaties. Als je `co ut` zou schrijven in plaats van `cout`, zal de computer klagen dat er geen objecten `co` en `ut` gedefinieerd zijn. De symbolen `<<` vormen één geheel; hiertussen mag dus geen spatie staan. De symbolen `(` en `)` achter `main` zijn echter twee aparte symbolen (waar toevallig niets tussen staat); hiertussen mogen dus wel spaties en regelovergangen gezet worden.

1.7 Commentaar

Voor de computer mag een lang programma met veel ingewikkelde opdrachten misschien te begrijpen zijn, een menselijke lezer van een programma heeft soms behoefte aan wat toelichting. Daarom bestaat in C++ de mogelijkheid om een programma te voorzien van *commentaar*. Commentaar in een programma wordt door de computer volledig genegeerd, en staat er dus alleen om het programma voor menselijke lezers te verduidelijken.

Commentaar begint met de symbolen // en loopt door tot het einde van de regel. Het commentaar mag uit willkeurige symbolen bestaan. Commentaar mag op elke plaats in het programma staan waar spaties zijn toegestaan. Het is een goede gewoonte om elk programma te beginnen met een stukje commentaar dat het doel van het programma aangeeft, eventueel aangevuld met de naam van de auteur en de datum van de laatste verandering:

```
// Dit programma begroet de gebruiker vriendelijk.
#include <iostream.h>
int main()
{  cout << "Hallo!" << endl;
}
```

Hoofdstuk 2

Rechthoek, *of:* variabelen

In dit hoofdstuk bekijken we een programma dat niet altijd hetzelfde doet, maar de uitvoer laat afhangen van de invoer. Daarvoor zijn *variabelen* nodig. Met de naam van een variabele kan de waarde worden aangeduid, die tijdens het schrijven van het programma nog niet bekend is.

2.1 Het programma

Het programma in dit hoofdstuk heet `rechthoek`, omdat het twee eigenschappen van rechthoeken berekent. In listing 2 staat de tekst van het programma.

Als dit programma uitgevoerd wordt, zal de volgende tekst op het beeldscherm verschijnen:

```
Geef de afmetingen van een rechthoek.
lengte:
```

Daarna wordt de verwerking van het programma tijdelijk stopgezet, zodat de gebruiker de gelegenheid heeft om een waarde in te tikken, bijvoorbeeld `12`. Om aan te geven dat het getal hiermee compleet is, en niet bijvoorbeeld het begin van het getal `125`, moet de gebruiker na het intikken van de waarde op de Return-toets drukken. Het programma wordt dan hervat, todat er opnieuw invoer nodig is:

```
Geef de afmetingen van een rechthoek.
lengte:     12
breedte:
```

De gebruiker van het programma kan nu opnieuw een waarde intikken. Het programma berekent daarna van de rechthoek met de ingevoerde lengte en breedte de omtrek en de oppervlakte, en zet het resultaat van die berekening op de uitvoerstream:

```
Geef de afmetingen van een rechthoek.
lengte:     12
breedte:    5
omtrek:     34
```

```
#include <iostream.h>

int main()
{  int lengte, breedte;

   cout << "Geef de afmetingen van een rechthoek." << endl ;
   cout << "lengte:      " ;
   cin  >> lengte ;
   cout << "breedte:     " ;
   cin  >> breedte ;

   cout << "omtrek:      " << 2*(lengte+breedte) << endl ;
   cout << "oppervlakte: " << lengte*breedte     << endl ;
}
```

Listing 2: `rechthoek.cc`

```
oppervlakte: 60
```

Daarna is het programma beëindigd.

2.2 Invoer

Alle gegevens die bij verwerking van het programma via het toetsenbord worden ingevoerd, kunnen in het programma opgevraagd worden via de *invoerstream* genaamd `cin`. Tijdens het wachten op invoer heeft de gebruiker het initiatief. Pas als die een waarde heeft ingetikt, afgesloten met de Return-toets, wordt het programma hervat. Om de gebruiker er aan te herinneren dat er een waarde verwacht wordt, is het een goede gewoonte om voorafgaand aan een invoer-opdracht een uitvoer-opdracht te geven, waarin wordt gemeld dat er invoer verwacht wordt. De gebruiker zou anders in de veronderstelling kunnen verkeren dat de computer een ingewikkelde berekening aan het uitvoeren is, en zich niet realiseren dat de computer juist op hem/haar aan het wachten is.

In het programma `rechthoek` staan daarom de opdrachten:

```
cout << "lengte:      " ;
cin  >> lengte ;
```

De eerste opdracht zet de letterlijke tekst `lengte:` op het scherm. Er staan immers aanhalingstekens rond deze tekst, dus de tekst wordt letterlijk overgenomen. Ook de zes spaties die nog tussen de aanhalingstekens staan worden letterlijk op het scherm gezet. De uitvoer-opdracht wordt niet gevolgd door het sturen van `endl` naar de uitvoerstream `cout`. Daarom verschijnt de invoer die de gebruiker vervolgens intikt op dezelfde regel.

De waarde die de gebruiker intikt is bij het schrijven van het programma nog niet bekend. Het programma zou ook meerdere keren gebruikt kunnen worden, waarbij telkens een andere waarde wordt ingetikt. Een grootheid met een wel-bepaalde, maar (bij het schrijven van het programma nog) onbekende waarde heet een *variabele*. Zo'n variabele wordt aangeduid met een *naam*. Je kunt je zo'n variabele voorstellen als een (klein) deel van het geheugen van de computer, waarin een waarde kan worden opgeslagen. De waarde van een variabele kan later weer worden opgevraagd, en eventueel ook veranderd worden.

In de meeste wiskundeboeken wordt voor de naam van een variabele steevast x of y gekozen, of anders wel a, b of c. In de meeste computerprogramma's komen zo veel variabelen voor, dat 26 verschillende namen niet genoeg zijn. Variabele-namen mogen in C++-programma's dan ook uit meerdere letters bestaan. Het is goed gebruik om voor variabelen een naam te kiezen die doet blijken waar hij voor bedoeld is. In het programma `rechthoek` heet de variabele waarin de lengte van de rechthoek wordt opgeslagen daarom `lengte`.

Door de opdracht

```
cin  >> lengte ;
```

krijgt de variabele `lengte` bij het uitvoeren van het programma een waarde. Let op de richting van de 'pijltjes': deze staan andersom dan bij gebruik van de uitvoerstream `cout`. De pijltjes wijzen in de richting waarin er gegevens getransporteerd worden; bij uitvoer gaat de tekst naar de stream toe, bij invoer komen de gegevens juist uit de stream vandaan.

In de invoer-opdracht staat de naam `lengte` *niet* tussen aanhalingstekens. Die staan immers alleen rond letterlijke tekst. We willen echter niet hebben dat er een bepaalde letterlijke tekst wordt ingetikt door de gebruiker; er mag een willekeurige waade worden ingetikt, en die moet voortaan bekend staan onder de *naam* `lengte`. Bij invoeropdrachten staan er dus nooit aanhalingstekens: het zou betekenen dat er gevraagd wordt om een vaste tekst in te tikken, en dat is zelden zinvol. In de taal C++ is het zelfs niet toegestaan.

2.3 Expressies

Als de variabele x een bepaalde waarde heeft (bijvoorbeeld 2), en de variabele y heeft ook een waarde (bijvoorbeeld 3), dan is ook de waarde van $x + y$ te bepalen (5 in het voorbeeld). Dit soort wiskundige formules worden *expressies* genoemd. Expressies worden, naast in de wiskunde, ook in programma's veel gebruikt. Ze kunnen onder andere voorkomen in een uitvoeropdracht. In het voorbeeldprogramma is dat het geval in de opdrachten

```
cout << "omtrek:      " << 2*(lengte+breedte) << endl ;
cout << "oppervlakte: " << lengte*breedte     << endl ;
```

In de eerste opdracht wordt de tekst `omtrek:` letterlijk op het scherm gezet (er staan immers aanhalingstekens omheen). De tekst `2*(lengte+breedte)` wordt echter niet letterlijk op het scherm gezet (want er staan geen aanhalingstekens omheen). In plaats daarvan wordt de waarde van de expressie `2*(lengte+breedte)` uitgerekend, gebruik makend van de geldende waarde van de variabelen `lengte` en `breedte`. Die waarde wordt vervolgens op het scherm gezet.

Expressies zijn opgebouwd uit de volgende onderdelen:

- *variabelen*, die (hopelijk) eerder in het programma een waarde hebben gekregen, bijvoorbeeld door een invoeropdracht;
- *constanten*, zoals het getal `2`;
- *operatoren*, zoals de optelling `+` en de vermenigvuldiging `*`;
- *haakjes*, waarmee groepering van deel-expressies kan worden aangegeven.

2.4 Declaraties

Elke variabele in een programma is van een bepaald *type*. Zo is bijvoorbeeld de variabele die aangeduid wordt met de naam `cout` van het type 'stream', en de variabele die aangeduid wordt met de naam `lengte` van het type 'geheel getal'. Andere veelgebruikte typen zijn bijvoorbeeld 'getal met decimale punt', 'letter', en 'tekst'. Maar het is ook mogelijk om zelf nieuwe typen te maken.

Aan de naam van een variabele is (door de computer) niet te zien wat het bedoelde type is. Daarom moet van elke variabele vóór het eerste gebruik worden opgegeven wat het type ervan is. Zo'n opgave heet een *declaratie*.

In het programma `rechthoek` worden de variabelen `lengte` en `breedte` gedeclareerd aan het begin van de functie `main`:

```
int lengte, breedte;
```

Met deze declaratie wordt opgegeven dat er twee variabelen in het programma gebruikt zullen worden, genaamd `lengte` en `breedte`, die allebei van het type `int` zijn. De mogelijke waarden van variabelen van het type `int` zijn de 'integer numbers' (gehele getallen), dus bijvoorbeeld 17 of -5, maar niet 3.2 (niet geheel) of `"aap"` (geen getal).

Het moeten doen van declaraties betekent extra werk voor de programmeur. Toch heeft het feit dat variabelen gedeclareerd moeten worden twee belangrijke voordelen:

- *Controle van spelfouten.*
  Als de naam van de variabele `lengte` op één plaats per ongeluk wordt gespeld als `lente`, dan wordt dit bij het compileren van het programma duidelijk omdat `lente` niet gedeclareerd is. Zou de compiler bijvoorbeeld aannemen dat alle niet-gedeclareerde variabelen het type `int` hebben, dan zou een programma verkeerde berekeningen kunnen uitvoeren (omdat de variabele `lente` wordt gebruikt in plaats van `lengte`). Misschien zou een oplettende gebruiker dat merken, (maar waarschijnlijk is dat niet, want als die de berekening uit het hoofd kan, dan was het hele computerprogramma overbodig). Meestal wordt dit soort fouten pas duidelijk als bijvoorbeeld de brug die op grond van de berekeningen is gebouwd instort.

- *Controle van operatoren.*
  Niet alle operatoren zijn zinvol voor alle typen. Zo kunnen bijvoorbeeld streams niet worden opgeteld, teksten niet worden vermenigvuldigd, en kunnen er met `<<` geen teksten naar getallen worden gestuurd. Omdat de compiler bij het gebruik van operatoren de typen controleert, kunnen dit soort fouten worden opgespoord voordat ze schade kunnen aanrichten.

Declaraties en opdrachten kunnen elkaar in een programma afwisselen. Omdat in het programma `rechthoek` de variabelen `lengte` en `breedte` in de eerste twee opdrachten in de functie `main` niet gebruikt worden, had de declaratie ook pas daarna kunnen plaatsvinden.

Veel programmeurs zetten echter alle declaraties van variabelen aan het begin van een functiedefinitie (in de taal C (zonder ++) was dat verplicht, en dat leren ze niet meer af). Omdat dat ook eigenlijk wel zo overzichtelijk is, zullen we dat in dit diktaat meestal ook doen.

2.5 Variabele-namen

Eerder zagen we al dat namen van variabelen in C++, anders dan in de wiskunde, uit meer dan één letter mogen bestaan. In werkelijkheid is de keuze nog iets vrijer. De regel voor de opbouw van namen is:

> Een naam begint met een letter (hoofdletter of kleine letter), gevolgd door nul of meer letters, cijfers, of onderstrepingstekens (_).

Geldige namen zijn dus bijvoorbeeld `x`, `abC12`, `A1b2`, en `a_2b`.

De keuzevrijheid voor variabele-namen wordt iets ingeperkt doordat 48 namen taboe zijn, omdat ze een vaste betekenis in de taal hebben. Het enige voorbeeld wat we tot nu toe zijn tegengekomen was `int`. Dit woord duidt het type 'gehele getallen' aan, en mag dus niet als naam worden gebruikt. Een aantal van de taboe-woorden zou je nooit zelf verzinnen als naam (zoals `volatile` en `asm`). Andere woorden liggen meer voor de hand (zoals `new` en `this`), dus je moet uitkijken dat je die niet per ongeluk gebruikt. Het volledige lijstje van taboe-namen is:

```
asm     continue  float   new       signed    try
auto    default   for     operator  sizeof    typedef
break   delete    friend  private   static    union
case    do        goto    protected struct    unsigned
catch   double    if      public    switch    virtual
char    else      inline  register  template  void
class   enum      int     return    this      volatile
const   extern    long    short     throw     while
```

Van een 20-tal van deze woorden komt de betekenis in latere hoofdstukken aan de orde; de andere 28 worden zelden gebruikt.

Als je het programma ook voor mensen (en dan vooral voor anderen dan jezelf) leesbaar wilt houden, is het verstandig om in de namen van de variabelen hun bedoeling tot uiting te laten komen. Daarom noemden we de lengte en breedte van de rechthoek in het programma `lengte` en `breedte` en niet `x` en `y`. Nu zou dat in zo'n kort programma ook nog wel te begrijpen zijn geweest, maar in langere programma's worden dat soort namen een onplezierige puzzel. We zullen ons éénletterige namen in het vervolg dan ook alleen nog permitteren als de variabele op slechts een paar plaatsen vlak bij elkaar wordt gebruikt.

In de praktijk zijn een paar conventies ontstaan voor het gebruik van variabele-namen. Zo zijn bijvoorbeeld èrg lange namen zoals `ditiseenhelelangenaam` lastig te lezen. Spaties zijn niet toegestaan in een variabelenaam. Als surrogaat kan het onderstrepings-teken worden gebruikt: `dit_is_een_hele_lange_naam`. Andere programmeurs schrijven liever `ditIsEenHeleLangeNaam`. Ga in ieder geval niet afkorten, want een week later weet je niet meer wat `diehln` betekent. Ook is het verstandig om niet te overdrijven: `langeNaam` is al mooi genoeg. De duidelijkste variabele-namen hebben meestal tussen de 4 en 10 letters. Hoofdletters worden zelden gebruikt in variabele-namen (anders dan voor het leesbaar maken van meer-woordige namen). Variabelen heten dus veeleer `max` dan `MAX`. Hoofdletterwoorden worden bewaard voor variabelen die altijd een vaste waarde houden.

Hoofdstuk 3

abc-formule, *of:* keuze

In dit hoofdstuk bekijken we een programma waarin niet altijd dezelfde opdrachten worden uitgevoerd. Afhankelijk van door de gebruiker ingevoerde waarden wordt een keuze gemaakt uit verschillende opdrachten.

In het programma wordt het onnodig tweemaal uitrekenen van expressies voorkomen door expliciete toekenning aan een variabele. Verder wordt gebruik gemaakt van 'floating-point'-getallen, en van een tweede bibliotheek naast `<iostream.h>`.

3.1 Het programma

Het programma in dit hoofdstuk heet `abc`, omdat het is gebaseerd op de welbekende *abc*-formule. In listing 3 staat de tekst van het programma.

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

int main()
{  double a, b, c, discr, wortel;

   cout << "Geef de coefficienten van een tweedegraadsvergelijking." << endl ;
   cout << "a: ";
   cin  >> a ;
   cout << "b: ";
   cin  >> b ;
   cout << "c: ";
   cin  >> c;

   discr = b*b - 4*a*c;

   if (discr<0.0)
       cout << "Geen oplossingen." << endl ;
   else if (discr==0.0)
       cout << "Een oplossing: " << -b/(2*a) << endl ;
   else
   {
      wortel = sqrt(discr);
      cout << "Twee oplossingen: " << (-b+wortel)/(2*a)
           << " en "               << (-b-wortel)/(2*a) << endl ;
   }
}
```

Listing 3: `abc.cc`

Ter opfrissing van middelbare-schoolwiskunde: de *abc*-formule luidt

$$\frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

en kan gebruikt worden om de oplossingen van de tweedegraadsvergelijking $ax^2 + bx + c = 0$ te vinden. De deelexpressie $b^2 - 4ac$ heet de *discriminant* van de vergelijking. Als de discriminant groter is dan 0, zijn er twee oplossingen. Is de discriminant gelijk aan 0, dan is er één oplossing (of volgens sommigen: twee 'samenvallende' oplossingen). Is de discriminant kleiner dan 0, dan kan de wortel niet worden uitgerekend (hoewel sommigen beweren dat er in dit geval toch twee zogeheten 'complexe' oplossingen zijn).

Het programma begint met het vragen om waardes van de coëfficiënten a, b en c van de vergelijking (we gaan er maar even van uit dat de gebruiker van het programma weet wat dat zijn, en waarom het leuk is om de oplossing van zo'n tweedegraadsvergelijking te bepalen). Vervolgens berekent het programma de discriminant. Aan de hand daarvan wordt het aantal oplossingen gemeld, plus de eventuele waarde(n) daarvan.

Een sessie met dit programma zou er als volgt uit kunnen zien:

```
Geef de coefficienten van een tweedegraadsvergelijking.
a: 1
b: -0.5
c: -3
Twee oplossingen: 2 en -1.5
```

3.2 Floating-point getallen

Bij het oplossen van tweedegraadsvergelijkingen is het gebruikelijk om op het domein der reële getallen te werken. We claimen dus zonder omhaal dat -1.5 een oplossing is van $x^2 - 0.5x - 3$, zonder ons druk te maken om het feit dat min anderhalf schapen niet bestaan (als de vergelijking opgesteld was om een kudde eerlijk onder twee herders te verdelen, dan moet er maar een beest gemeenschappelijk geroosterd worden...).

Ook voor de coëfficiënten a, b en c zijn gebroken getallen toegestaan. We kunnen in dit programma dus niet volstaan met variabelen van het type `int`. Die kunnen weliswaar negatief zijn, maar zijn altijd geheel, en hebben dus geen deel 'achter de punt'. Gelukkig is er in C++ speciaal voor dit doel een type `float` beschikbaar. De waarden van dit type zijn het soort getallen dat ook op een rekenmachientje gebruikt wordt: soms staan er een heleboel cijfers achter de punt, zoals in 3.1415926, en soms een heleboel ervoor, zoals in 273.15; de punt kan als het ware 'drijven' door het getal. Dit soort getallen heet daarom 'floating point' getallen, afgekort tot `float`. Toch gebruiken we in het programma `abc` niet dit type, maar het type `double`. Dit is het type van 'double precision floating point numbers', oftewel 'extra nauwkeurige' getallen.

Net als op de rekenmachine kunnen in een floating-point variabele zeer grote getallen worden opgeslagen. Daarvoor wordt de 'wetenschappelijke' notatie gebruikt. Het getal van Avogadro bijvoorbeeld heeft de waarde $6.0220 \cdot 10^{23}$. Meer dan vijf cijfers van dit getal zijn niet zo interessant, maar wel het feit dat het er 23 zijn! Dit soort getallen kunnen in een programma worden opgeschreven als `6.0220e23`.

Het verschil tussen de typen `float` en `double` is het aantal significante cijfers dat aangegeven kan worden: bij `float` is dit 7, bij `double` is het 15. Ook het grootste representeerbare getal verschilt: de grootste `float` is `3.4e38`, de grootste `double` is `1.7e308`.

Nu we het toch over het bereik van getallen hebben: ook de waarde van een `int` is aan een maximum gebonden. Om het ingewikkeld te maken kan dit maximum afhankelijk zijn van de gebruikte compiler. Op PC's is de maximaal representeerbare integer vaak 32767, terwijl op grotere machines de grootst mogelijke integer meestal ruim 2 miljard is (om precies te zijn: 2147483648).

Een belangrijk verschil tussen de type `int` en `float`/`double` is dat berekeningen met een `int` altijd exact zijn, terwijl bij `float` afrondfouten een rol kunnen spelen. Dit is de reden dat je niet klakkeloos een `float` moet gebruiken waar een `int` ook voldoet. Het type `int` moet gebruikt worden bij variabelen die een 'aantal' of een 'volgnummer' voorstelen; de typen `float` en `double` kunnen gebruikt worden bij continue grootheden zoals afstanden, gewichten en temperaturen. (Er zijn natuurlijk uitzonderingen op deze vuistregel: het getal van Avogadro is eigenlijk een aantal, maar te groot om in een `int` op te slaan, en vaak is voor afstanden het geheel aantal kilometers uitstekend in een `int` op te slaan).

3.3 Toekenningen aan variabelen

In het programma `abc` zijn de variabelen `a`, `b` en `c` als `double` gedeclareerd. (We zondigen meteen al tegen het principe dat variabelen een duidelijke naam moeten krijgen – hoewel, de *abc*-formule is zó bekend, dat `a`, `b` en `c` toevallig de duidelijkst denkbare namen zijn).

In de *abc*-formule wordt een belangrijke rol gespeeld door de discriminant. Het zou handig zijn als we de waarde van de discriminant konden aanduiden met een variabele-naam. Gelukkig is er een opdracht waarmee dat mogelijk wordt:

```
discr = b*b - 4*a*c;
```

Met deze opdracht wordt de waarde van de variabele `discr` gelijk gemaakt aan de waarde van de expressie `b*b-4*a*c`. Dit soort opdrachten heet een *toekenningsopdracht*, omdat er een nieuwe waarde mee wordt toegekend aan een variabele. Net als de invoer- en uitvoer-opdrachten wordt een toekenningsopdracht afgesloten met een puntkomma.

Het `=`-symbool kun je het beste uitspreken als 'wordt'. De waarde van `discr` *wordt* immers gelijk aan die van de expressie. Het is verleidelijk om het symbool uit te spreken als 'is'. Dat is echter minder juist: de waarde van `discr` was misschien nog helemaal niet gelijk aan die van de expressie; hij *wordt* het pas door deze opdracht. Bovendien is 'blablabla is blablabla' taalkundig gezien niet zozeer een opdracht die uitgevoerd kan worden, maar een constatering die geverifieerd kan worden. Het gaat hier echter wel degelijk om een opdracht die de computer kan uitvoeren – '*wordt*' zeggen dus, en niet '*is*'.

3.4 Voorwaardelijke opdrachten

In het programma wordt afhankelijk van de waarde van de discriminant (die inmiddels beschikbaar is in de variabele `discr`) een verschillende tekst getoond ('geen oplossing' – 'een oplossing' – 'twee oplossingen'). Deze gevalen kunnen worden onderscheiden met een *voorwaardelijke opdracht*. Een voorwaardelijke opdracht wordt opgebouwd met behulp van de speciaal daarvoor gereserveerde woorden `if` en `else`. Het programma luidt ruwweg:

```
if (discr < 0.0)
    cout << "nul" ;
else if (discr == 0.0)
    cout << "een" ;
else
    cout << "twee" ;
```

Achter het woord `if` staat tussen haakjes een *voorwaarde*. Een voorwaarde is een expressie waarin een vergelijkings-operator gebruikt wordt (zoals `<` met de betekenis 'is kleiner dan'). Zo'n expressie heet ook wel een *logische expressie*. Filosofen noemen het een *predicaat*. Hoe je het ook noemt (voorwaarde, logische expressie of predicaat), er zijn twee mogelijke uitkomsten: ja/nee, waar/onwaar, true/false of hoe je het ook maar wilt noemen. Als de uitkomst van de voorwaarde 'ja' is, wordt de opdracht er direct achter uitgevoerd. Is de uitkomst echter 'nee', dan wordt de opdracht achter het woord `else` uitgevoerd. In het voorbeeld is de opdracht achter `else` toevallig opnieuw een voorwaardelijke opdracht, waarin afhankelijk van een tweede voorwaarde de tekst `een` of de tekst `twee` wordt getoond.

In logische expressies kunnen de volgende operatoren gebruikt worden:

- `<` kleiner dan
- `>` groter dan
- `<=` kleiner dan of gelijk aan ('hoogstens')
- `>=` groter dan of gelijk aan ('minstens')
- `==` gelijk aan
- `!=` ongelijk aan

Speciale aandacht behoeft de gelijkheids-test. Hiervoor moet een *dubbel* is-teken gebruikt worden. Het enkele is-teken wordt immers al voor toekenningsopdrachten gebruikt. Het dubbele is-teken mag wèl worden uitgesproken als 'is'.

Erg gevaarlijk is het om je te vergissen in het symbool. Gebruik je bij de gelijkheidstest per ongeluk een enkel is-teken, dan wordt de gelijkheid niet *getest* maar *tot stand gebracht*! Dat is in het geheel niet de bedoeling in een situatie waarin het nou juist interessant is om de test uit te voeren. Deze fout wordt, ook door ervaren programmeurs, veel gemaakt, en is erg lastig te vinden. Gelukkig wordt er door veel compilers voor gewaarschuwd met de boodschap: 'suspicious assignment in conditional'. 'Verdachte toekenningsopdracht' – zeg dat wel!

3.5 Wiskundige functies

Terug naar het programma `abc`. Behalve het aantal oplossingen moeten ook de oplossingen zèlf worden getoond. Daartoe moet de wortel van de discriminant worden berekend (als die tenminste positief is). Nu doet een computer natuurlijk niet onder voor een rekenmachientje, en zijn naast de vier bekende operatoren ook allerlei wiskundige functies te gebruiken. De meeste functies hebben bekende namen, zoals `sin`, `arctan` en `log`. De wortelfunctie wordt in de wiskunde met $\surd$ aangeduid, maar omdat dat symbool wat lastig is in te tikken heeft in C++ ook de wortelfunctie een naam gekregen: `sqrt`, een afkoring van 'square root'. De waarde van de wortel van de discriminant kan worden berekend met de expressie `sqrt(discr)`.

De waarden van de twee oplossingen kunnen worden berekend door deze expressie te gebruiken als deel van een grotere expressie: `(-b+sqrt(discr))/(2*a)`, respectievelijk `(-b-sqrt(discr))/(2*a)`. Het is echter zonde van het werk om de wortel tweemaal uit te rekenen, terwijl hij in beide expressies hetzelfde is. Handiger is het om de waarde van de expressie `sqrt(discr)` apart op te slaan in een nieuwe variabele, zeg `wortel`, en daarna die variabele te gebruiken bij de berekening van de twee oplossingen. We krijgen dan:

```
wortel = sqrt(discr);
cout << "Twee oplossingen: " << (-b+wortel)/(2*a)
     << " en "               << (-b-wortel)/(2*a) ;
```

Hoewel dit programmafragment drie regels beslaat, gaat het toch maar om twee opdrachten (te herkennen aan de twee puntkomma's). Om de regel niet te vol te maken is de tweede opdracht over twee regels gesplitst; dat mag tenslotte altijd. In een nog royalere bui hadden we er ook wel vijf regels van kunnen maken, en op een zuinig moment hadden de twee opdrachten ook op één regel gekund...

De namen van de wiskundige functies, waaronder `sqrt`, zijn niet zonder meer beschikbaar. Om er gebruik van te kunnen maken, moet de bibliotheek waarin ze worden gedefinieerd in het programma worden opgenomen. De functies staan in de bibliotheek `<math.h>`. Daarom wordt naast de bekende `<iostream.h>` nu ook deze bibliotheek gebruikt.

Door een wat rottig ontwerp van de organisatie van bibliotheken is de `#include`-regel niet voldoende om de compiler ervan te overtuigen dat we de wiskunde-bibliotheek willen gebruiken. Behalve met deze regel in het programma moet dit ook worden meegedeeld op de commandoregel van de compiler, en wel als volgt:

```
CC abc.cc -lm
```

Met de letters `-lm` wordt aangegeven dat de **l**ibrary voor **m**athematics nodig is. Hartstikke dubbelop natuurlijk in combinatie met de `#include`-regel, maar zo moet het nou eenmaal. Bij het gebruik van computers moet je niet altijd vragen waarom de dingen zijn zoals ze zijn...

3.6 Samengestelde opdrachten

In de vorige paragraaf werd besloten dat het handig was om (bij positieve discriminant) eerst een waarde aan de variabele `wortel` toe te kennen, om die waarde in de daaropvolgende uitvoeropdracht twee keer te gebruiken. We moeten dus aangeven dat deze twee opdrachten *samen* tot het `else` gedeelte van de tweede voorwaarde behoren. Ook deze situatie is bij het ontwerp van de taal voorzien. In dit soort gevallen kun je meerdere opdrachten met accolades groeperen tot één opdracht. Zo'n opdracht heet een *samengestelde* opdracht.

Door dus te schrijven

```
if (discr==0.0)
    cout << "Een oplossing: " << -b/(2*a) ;
```

```
    else
    {
        wortel = sqrt(discr);
        cout << "Twee oplossingen: " << (-b+wortel)/(2*a)
             << " en "               << (-b-wortel)/(2*a) ;
    }
```

worden de twee opdrachten (toekenning aan `wortel` en de uitvoer-opdracht) samen alleen uitgevoerd als niet aan de voorwaarde voldaan is.

3.7 Samenvatting

Tot nu toe zijn er drie taalconstructies behandeld: opdracht, expressie, en declaratie. De vorm van die constructies kan als volgt worden samengevat.

Opdracht

- *stream* `<<` *expressie* `<<` *expressie* ... `;`
- *stream* `>>` *variabele* `;`
- *variabele* `=` *expressie* `;`
- `if (` *expressies* `)` *opdracht* `else` *opdracht*
- `{` *meerdere opdrachten en eventueel declaraties* `}`

Expressie

- *getal-constante*
- `"` *tekst* `"`
- *variabele-naam*
- *expressie operator expressie*
- *functienaam* `(` *expressie* `)`
- `(` *expressie* `)`

Declaratie

- *type variabele* `,` *variabele* ... `;`

Opgaven

1. Niet alle wiskundige operaties zijn voor alle getallen gedefinieerd. Je kunt bijvoorbeeld niet de wortel van een negatief getal trekken, en deling is niet mogelijk als de noemer nul is. In het programma `abc` wordt de wortelfunctie adequaat beveiligd met voorwaarden. De delingen zijn echter nog niet *fool proof*. Breid het programma uit zodat het programma voor *alle* getaalen `a`, `b` en `c` zinvolle uitvoer genereert.

2. Stel dat de `sqrt`-functie niet beschikbaar is. Het is dan toch mogelijk om wortels te bepalen. De methode daarvoor berust op het volgende principe: 'als y een benadering is voor $\sqrt{x}$, dan is het gemiddelde van y en x/y een betere benadering'. Uitgaande van die betere benadering kun je een nog betere benadering maken, enzovoort. Schrijf een programma dat een getal inleest, en de wortel daarvan afdrukt. De oplossing moet zó nauwkeurig zijn, dat het kwadraat ervan niet meer dan 10^{-6} afwijkt van het oorspronkelijke getal.

3. Als je het volgende programma compileert geeft de compiler de foutmelding 'unexpected `else`'. Waarom?

```
#include <iostream.h>
int main()
{   int getal;
    cin >> getal;
    if (getal<0);
        cout << "negatief";
    else
        cout << "positief";
}
```

Hoofdstuk 4

Gemiddelde, *of:* iteratie

In dit hoofdstuk bekijken we een programma waarin dezelfde opdracht meer dan een keer uitgevoerd wordt. We bespreken twee manieren om een lopend programma te stoppen. Verder zullen we zien dat opdrachten en expressies meer met elkaar te maken hebben dan het aanvankelijk scheen.

4.1 Het programma

Het programma in dit hoofdstuk heet gemiddeld, omdat het gemiddelde van een rij getallen ermee berekend kan worden. In listing 4 staat de tekst van het programma.

Zoals gewoonlijk meldt het programma netjes dat er invoer van de gebruiker wordt verwacht. Vervolgens worden niet een vast aantal waarden ingelezen, maar willekeurig veel. Deze waarden worden niet allemaal permanent opgeslagen in variabelen; wel worden er twee variabelen gebruikt om het totaal en het aantal van de ingelezen waarden bij te houden.

Een sessie zou er als volgt uit kunnen zien:

```
Geef getallen, afgesloten door control-D.
3
6.5
12
 control-D
```

```
#include <iostream.h>

int main()
{
    double totaal, getal;
    int aantal;

    cout << "Geef getallen, afgesloten door control-D." << endl ;

    totaal = 0.0;
    aantal = 0;

    while ( cin >> getal )
    {
        totaal += getal;
        aantal ++;
    }

    cout << "totaal:    " << totaal        << endl ;
    if (aantal>0)
         cout << "gemiddeld: " << totaal/aantal << endl ;
}
```

Listing 4: gemiddeld.cc

```
totaal:    19.5
gemiddeld: 6.5
```

4.2 Neveneffecten

In de programma's tot nu toe is er een belangrijk onderscheid geweest tussen de taalconstructies 'opdracht' en 'expressie':

- een *opdracht* kan worden *uitgevoerd*, met als gevolg dat de *toestand* van de variabelen of het beeldscherm wordt gewijzigd
- een *expressie* kan worden *uitgerekend*, waardoor een *waarde* wordt bepaald.

Het kan echter het geval zijn dat tijdens het uitrekenen van de waarde van een expressie óók de toestand van variabelen wordt gewijzigd. Het berekenen van zo'n expressie heeft dan een zogeheten *neveneffect* tot gevolg.

Soms ben je meer geïnteresseerd in het neveneffect dat bij het berekenen van een expressie optreedt, dan in de waarde zelf. Je behandelt de expressie dan eigenlijk als een opdracht; het gaat om de toestandswijziging die door het neveneffect tot stand is gebracht. En de waarde van de expressie – ach, die wordt wel uitgerekend, maar die kun je meteen weer vergeten. Voor dit geval is er een aparte notatie beschikbaar in C++:

> elke expressie met een puntkomma er achter kan als opdracht worden gebruikt.

Het is dus bijvoorbeeld toegestaan om de volgende opdracht in een programma te schrijven:

```
15 + 8 ;
```

Dit is een opdracht, want het is een expressie met een puntkomma er achter. Uitvoeren van deze opdracht is overigens niet zo zinvol: de waarde van de expressie wordt berekend (23) en meteen weer vergeten. Enig blijvend resultaat zou het gevolg moeten zijn van de neveneffecten van het uitrekenen van de expressie; de operator `+` heeft echter geen nevenefecten.

Een voorbeeld van een expressie mèt neveneffecten, die als opdracht wordt gebruikt, is het volgende:

```
cin >> getal ;
```

Het effect van deze opdracht is bekend: er wordt een waarde ingelezen, en opgeslagen in de variabele `getal`. Omdat dit neveneffect van de expressie `cin>>getal` vaak het enige is wat van belang is, hebben we dit tot nu toe gemakshalve een 'opdracht' genoemd. Maar de expressie heeft ook een waarde, die af en toe van pas komt. Om in te zien wat deze waarde voorstelt, moeten we eerst iets meer over streams spreken.

Tot nu toe hebben we gedaan alsof streams aldoor maar voortkabbelen zonder dat er een eind aan komt, maar dat is niet helemaal waar. Behalve het toetsenbord kan namelijk ook de inhoud van een file als stream gebruikt worden, en vroeg of laat is zo'n file aan zijn eind. Zelfs als het toetsenbord als invoerstream wordt gebruikt, kan de gebruiker het einde van de stream aankondigen. Dat gebeurt door de toetsencombinatie control-D in te tikken (control- of CTRL-toets ingedrukt houden en gelijktijdig op de D-toets drukken). Voor het programma lijkt het dan alsof de stream beëindigd is.

De *waarde* van de expressie `cin>>getal` kan gebruikt worden als waarheidswaarde, die aangeeft of er nog informatie beschikbaar is in de stream. Het *neveneffect* van de expressie is dat (indien er inderdaad informatie is) er informatie wordt ingelezen en toegekend aan de variabele `getal`. De waarheidwaarde zou je bijvoorbeeld kunnen gebruiken als voorwaarde in een `if`-opdracht.

In het programma gemiddeld wordt de waarde van `cin>>getal` echter op een andere manier gebruikt:

```
while ( cin >> getal )
{
    totaal += getal;
    aantal ++;
}
```

Net als `if` is `while` een gereserveerd woord. De vorm van deze opdracht lijkt op die van een voorwaardelijke opdracht: een gereserveerd woord, een voorwaarde tussen haakjes, gevolgd door een opdracht (in dit geval een samengestelde opdracht).

Wordt de opdracht achter de voorwaarde van `if` éénmaal uitgevoerd indien de voorwaarde 'waar' is, de opdracht achter `while` wordt *steeds opnieuw* uitgevoerd *zolang* de voorwaarde waar is. De waarde van de voorwaarde wordt elke keer opnieuw uitgerekend, en eventuele neveneffecten treden steeds opnieuw op. In het voorbeeld worden dus de twee opdrachten tussen de accolades steeds opnieuw uitgevoerd, althans zolang de expressie `cin>>getal` 'waar' is (dus zolang het einde van de stream nog niet bereikt is). Als neveneffect van de berekening van de voorwaarde is er steeds een nieuw getal beschikbaar in de variabele `getal`.

4.3 Ophogen

In het programma gemiddeld moeten het totaal en het aantal van de ingelezen waarden bijgehouden worden. Daaruit kan immers het gemiddelde berekend worden. In het programma worden daarom twee variabelen gedeclareerd:

```
double totaal;
int aantal;
```

De variabele `totaal` is van het type `double`, zodat het ook mogelijk is om het gemiddelde van niet-gehele getallen te berekenen. De variabele `aantal` is echter van type `int`, omdat een aantal altijd een geheel getal is.

Deze twee variabelen moeten te alle tijde het totaal (respectievelijk het aantal) van alle tot nu toe ingelezen waarden bevatten. Dit moet ook het geval zijn vóórdat er iets is ingelezen. Daarom staan er in het programma twee opdrachten die deze variabelen van een zinvolle beginwaarde voorzien:

```
totaal = 0.0;
aantal = 0;
```

Steeds als er een getal is ingelezen, moet de waarde van het totaal worden bijgewerkt. Dat zou kunnen met de opdracht

```
totaal = totaal + getal ;
```

Omdat dit soort opdrachten (waarin de waarde een variabele wordt verhoogd met een andere waarde) veel gebruikt worden, is er een kortere notatie voor beschikbaar:

```
totaal += getal ;
```

Dit zou je kunnen uitspreken als 'totaal wordt opgehoogd met getal'. Met gebruikmaking van deze notatie zou ook de waarde `aantal` aangepast kunnen worden:

```
aantal += 1 ;
```

Omdat het ophogen met de waarde 1 relatief vaak voorkomt, is hier een nòg kortere notatie voor:

```
aantal++ ;
```

Het bijwerken van de waardes van `totaal` en `aantal`, steeds als er een getal is ingelezen, gebeurt dus met de volgende `while`-opdracht:

```
while ( cin >> getal )
{   totaal += getal;
    aantal ++;
}
```

Nadat alle getallen zijn ingelezen (doordat de gebruiker met control-D het einde van de stream aankondigt), zijn in de variabelen `totaal` en `aantal` de gegevens opgebouwd die nodig zijn om het gemiddelde te berekenen. Met de laatste opdrachten in het programma wordt het totaal en het gemiddelde aan de gebruiker meegedeeld:

```
cout << "totaal:    " << totaal << endl ;
if (aantal>0)
     cout << "gemiddeld: " << totaal/aantal << endl ;
```

Omdat het niet mogelijk is om het gemiddelde van nul getallen uit te rekenen, wordt het gemiddelde alleen uitgerekend als er inderdaad getallen zijn ingevoerd. Het is verstandig om overal waar in een programma de delings-operator / gebruikt wordt na te gaan, of er situaties kunnen voorkomen waarin de noemer van de breuk nul is. Kun je als programmeur niet garanderen dat dit nooit het geval zal zijn, dan is het verstandig om met `if` te verzekeren dat de deling alleen wordt uitgevoerd wanneer dit zinvol is.

4.4 Nooit meer stoppen

Pas met gebruik van `while`-opdrachten wordt het mogelijk om de computer voor langere tijd aan het werk te houden. De `while`-constructie wordt dan ook veel gebruikt in programma's.

Je moet wel een beetje op je woorden gaan passen bij het redeneren over programma's. Als je zegt 'opdracht A komt vóór opdracht B', bedoel je dan dat de opdracht eerder in het programma staat, of dat hij eerder zal worden uitgevoerd? Het kan zijn dat deze begrippen toevallig samenvallen, maar er zijn ook gevallen denkbaar waarbij dat niet het geval is.

Een ander gevaar van `while`-opdrachten is dat er soms nooit een einde aan komt (qua uitvoeren dus, niet qua programmatekst!). Zo'n opdracht is gemakkelijk te schrijven. Met

```
while (1==1)
    cout << "hallo!";
```

heb je in een mum van tijd je hele scherm vol behang. De expressie `1==1` is namelijk altijd 'waar', zodat de opdracht steeds opnieuw uitgevoerd wordt.

In dit programma was die 'oneindige iteratie' wellicht de bedoeling, maar vaak slaat een `while`-opdracht ook op hol als gevolg van een programmeerfout. Bijvoorbeeld in

```
aantal = 0;
while (aantal<10)
    cout << "hoi" ;
    aantal++;
```

Het is de bedoeling dat dit programma 10 keer de tekst `hoi` afdrukt. Helaas heeft de programmeur vergeten om de twee opdrachten samen te smeden tot één samengestelde opdracht. Daardoor wordt alleen de opdracht `cout<<"hoi";` herhaald, en op die manier wordt de waarde van `aantal` natuurlijk nooit groter dan 10. Na afloop van de iteratie zou de opdracht `aantal++;` éénmaal worden uitgevoerd, maar daaraan komt de computer niet eens meer toe.

Een programmeerfout die nog lastiger te vinden is, is de volgende:

```
while (aantal<10) ;
{    cout << "hoi" ;
     aantal++;
}
```

Er staan nu wel netjes accolades rond de te herhalen opdrachten. Toch lijkt de computer helemaal niets te doen. De oorzaak is de extra puntkomma die achter de eerste regel staat. Die is er de oorzaak van dat de 'lege opdracht' die tussen de `)` en de `;` staat steeds opnieuw wordt uitgevoerd. Ook nu zal de variabele `aantal` nooit de nagestreefde waarde bereiken...

Het zou jammer zijn als je, na een computer met een ongelukkig geplaatste puntkomma in coma gebracht te hebben, het dure apparaat weg zou moeten gooien omdat hij steeds maar bezig blijft met dat ene programma. Gelukkig is er een manier om met geweld de uitvoering van het programma te beëindigen, ook al is het nog niet voltooid. De gebruiker kan hiertoe de toetsencombinatie control-C indrukken. Het programma wordt dan direct gestopt, en de gebruiker kan bijvoorbeeld een ander programma starten.

Er is een belangrijk verschil tussen het indrukken van control-C en control-D. Bij control-C wordt de uitvoering van het programma afgebroken. Bij control-D wordt slechts aangegeven dat de invoer-stream beëindigd is. De verwerking van het programma kan echter gewoon worden voortgezet, mits daarbij de invoer-stream niet gebruikt wordt.

Opgaven

1. Geef een voorbeeld van een programmafragment waarin waarin opdracht A in de programmatekst vóór opdracht B staat, maar bij het uitvoeren van het programma erná wordt uitgevoerd.

2. Schrijf een programma dat een geheel getal inleest en bepaalt uit hoeveel cijfers dat getal bestaat.

Hoofdstuk 5

Rente, *of:* meer iteratie

In dit hoofdstuk bekijken we een programma waarin opdrachten eveneens herhaald worden. Het aantal keer dat de iteratie plaatsvindt wordt nu echter niet door de gebruiker bepaald, maar is in het programma vastgelegd.

5.1 Het programma

Het programma in dit hoofdstuk heet `rente`, omdat het berekent hoe een tegen aantrekkelijke rente vastgezet kapitaal aangroeit (of, zo je wilt, hoe een tegen een minder aantrekkelijke rente aangegane schuld verergert). In listing 5 staat de tekst van het programma.

Het programma begint te vragen naar het startkapitaal (of -schuld) en het rentepercentage. Nadat die zijn ingevoerd, wordt de prognose voor de komende tien jaar berekend. Een sessie zou er als volgt uit kunnen zien:

```
kapitaal: 200
rentepercentage: 5
na 1 jaar: 210
na 2 jaar: 220.5
na 3 jaar: 231.525
na 4 jaar: 243.101
na 5 jaar: 255.256
na 6 jaar: 268.019
na 7 jaar: 281.42
na 8 jaar: 295.491
na 9 jaar: 310.266
na 10 jaar: 325.779
```

In het eerste jaar verdienen we een tientje: 5% van 200 gulden. In het tweede jaar is de vermeerdering groter, want ook over de uitbetaalde rente wordt in dit jaar rente betaald. Deze 'rente op rente' zorgt ervoor dat we na tien jaar niet tien tientjes hebben verdiend (gedokt), maar mooi een geeltje extra.

5.2 Probleemanalyse

In elk programma wordt een of ander probleem opgelost. Steeds moet daarbij worden nagedacht welke variabelen een rol spelen, en dus door de gebruiker ingevoerd moeten kunnen worden. Bovendien moet het proces waar het om gaat gecodeerd worden in een programma. Niet altijd is dat even gemakkelijk. Vaak helpt het dan om aan de hand van een voorbeeld te bekijken wat er moet gebeuren.

Bij de kapitaalsontwikkeling bekijken we bijvoorbeeld een startbedrag van 200 gulden en een rentepercentage van 5%. 'Procent' betekent letterlijk 'honderdste', dus een percentage van 5% betekent een rente van $\frac{5}{100}$ van het ingelegde bedrag. De rente op een kapitaal k is dus $\frac{5}{100} * k$. Het kapitaal plus de rente na 1 jaar bedraagt dus $k + \frac{5}{100} * k$, oftewel $(1 + \frac{5}{100}) * k$. Dit geldt natuurlijk ook voor andere rentepercentages dan 5. In het algemeen kunnen we zeggen dat bij een rentepercentage van r% een kapitaal k na 1 jaar is aangegroeid tot $(1 + \frac{r}{100}) * k$, anders gezegd is vermenigvuldigd met $1 + \frac{r}{100}$. Dat getal is blijkbaar belangrijk, want als het rentepercentage niet wijzigt geldt deze zelfde groeifactor ook in het tweede jaar.

```
#include <iostream.h>

int main()
{
    float kapitaal, rente, groeifactor;
    int   jaar;

    cout << "kapitaal: ";
    cin  >> kapitaal;
    cout << "renteprecentage: ";
    cin  >> rente;

    groeifactor = 1 + rente/100;

    for (jaar=1; jaar<=10; jaar++)
    {
        kapitaal *= groeifactor;
        cout << "na " << jaar << " jaar: " << kapitaal << endl;
    }
}
```

Listing 5: rente.cc

In het programma slaan we deze belangrijke waarde op in een aparte variabele. En omdat programmeren geen wiskunde is, gebruiken we meteen maar duidelijkere namen voor de variabelen:

```
groeifactor = 1 + rente/100;
```

Gebruikmakend van deze variabele kunnen we de variabele `kapitaal` steeds met de groeifactor vermenigvuldigen:

```
kapitaal *= groeifactor;
cout << "na 1 jaar: " << kapitaal << endl;
kapitaal *= groeifactor;
cout << "na 2 jaar: " << kapitaal << endl;
kapitaal *= groeifactor;
cout << "na 3 jaar: " << kapitaal << endl;
```

Dat is echter wel erg veel schrijf-/programmeerwerk. Vooral als het om langlopende leningen gaat...

Het valt op dat er eigenlijk driemaal vrijwel hetzelfde gebeurt. Het enige wat verschilt is het getal in de uitvoer-opdracht (1, 2 en 3). We kunnen ons het driemaal opschrijven van deze twee opdrachten besparen door ze maar éénmaal op te schrijven, en ze door middel van een `while`-opdracht driemaal te laten uitvoeren. Voor datgene wat steeds verandert (het getal 1, 2 en 3) voeren we een extra variabele in, die in de opeenvolgende iteraties steeds een andere waarde heeft. Deze variabele kunnen we ook goed gebruiken om te controleren of we de opdrachten al vaak genoeg hebben uitgevoerd.

In plaats van bovenstaand programmafragment kunnen we dus schrijven:

```
jaar = 1 ;
while (jaar<=3)
{
    kapitaal *= groeifactor;
    cout << "na " << jaar << " jaar: " << kapitaal << endl;
    jaar++;
}
```

Het programma is er alleen maar langer op geworden (7 regels inplaats van 6), en tot overmaat van ramp hebben we ook nog moeten nadenken. Toch is het wel de moeite waard geweest, want nu kunnen we de 3 moeiteloos veranderen in 10 of 30. Of nog mooier: we kunnen hier een variabele neerzetten en de gebruiker de gewenste looptijd vragen in te voeren.

5.3 Herhaling met een teller

De constructie uit de vorige paragraaf is in de praktijk erg vaak nodig. Het gaat dus om programmafragmenten met de volgende vorm:

```
x = startwaarde;
while (x <= eindwaarde )
{
        doe iets leuks;
        x++;
}
```

Deze opzet komt zo vaak voor, dat er een aparte notatie voor is ontworpen, namelijk:

```
for (x = startwaarde; x <= eindwaarde; x++)
{
        doe iets leuks;
}
```

Als 'iets leuks' dan ook nog uit één opdracht bestaat, kunnen we bovendien besparen op accolades.

Een `for`-opdracht (`for` is een gereserveerd woord) heeft altijd dezelfde vorm; tussen de haakjes staan altijd precies twee puntkomma's. De betekenis is nogal omslachtig om te beschrijven: in `for` $(A;B;C)D;$ wordt eerst A eenmaal uitgevoerd, en vervolgens D en C indien en zolang de logische expressie B waar is. Hierin zijn A, B en C alledrie expressies (als je de puntkomma's niet meetelt). Bij A en C is het je in de praktijk te doen om het neveneffect, terwijl het bij B juist om de waarde gaat.

Als de `for`-opdracht met bovenstaande opbouw D uiteindelijk driemaal uitvoert, zijn achtereenvolgens uitgerekend/uitgevoerd: $ABDCBDCBDCB$. Het lijkt allemaal ingewikkelder dan het is. Zie bijvoorbeeld de uiteindelijke formulering van de herhaald uitgevoerde opdrachten in listing 5.

Opgaven

1. Bepaal de uitvoer van het volgende programmafragment:

```
a = 0; b = 1; c = 6;
for (k=0; k<100; k++)
{   a += b;
    b += c;
    c += 6;
}
cout << a;
```

Hoofdstuk 6

Driehoek van Pascal, *of:* functies

In het vorige hoofdstuk bleken slim opgestelde iteratie-opdrachten je een boel schrijfwerk te kunnen besparen. In dit hoofdstuk bekijken we een nog veel flexibelere methode om schrijfwerk te besparen. Zodra je jezelf erop betrapt vrijwel identieke tekst tweemaal op te schrijven, valt het te overwegen om daar een *functie* van te maken. Voor de verschillen tussen de twee bijna-identieke teksten vul je een extra variabele in. De nieuw gemaakte functie kan vervolgens meer malen worden *aangeroepen*.

6.1 Het programma

Het programma in dit hoofdstuk heet pascal, omdat het de 'driehoek van Pascal' berekent. Dit zijn een aantal in een driehoek gerangschikte getallen. Het geheel is vernoemd naar de wiskundige/filosoof Blaise Pascal (1623–1662), die de eigenschappen ervan beschreef. (Er is overigens ook een programmeertaal naar Pascal vernoemd, omdat de man ook de eerste was die een rekenmachine ontwierp. Die taal heeft verder niets met dit programma te maken, en is bovendien de concurrent van C++, dus daar hebben we het maar niet meer over...) In listing 6 staat de tekst van het programma.

Het programma krijgt geen invoer, en genereert dus altijd dezelfde uitvoer, namelijk

```
1
1    1
1    2    1
1    3    3    1
1    4    6    4    1
1    5   10   10    5    1
1    6   15   20   15    6    1
1    7   21   35   35   21    7    1
1    8   28   56   70   56   28    8    1
1    9   36   84  126  126   84   36    9    1
```

Het zou eenvoudig zijn om het programma aan te passen zodat het een door de gebruiker in te voeren aantal regels van de driehoek laat zien, dus desgewenst ook minder of meer dan 10 regels. Dat zou echter maar de aandacht afleiden van waar het in dit hoofdstuk in werkelijkheid om gaat.

De driehoek is op verschillende manieren te beschrijven. Een manier is: in de linkerkolom en op de diagonaal staat steeds het getal 1, en verder is elk getal de som van het getal er recht boven en er schuin links boven. Een tweede manier is: op de k-de plaats van de n-de rij (beginnend te tellen bij 0) staat de waarde $\binom{n}{k}$, wat gedefinieerd is door

$$\binom{n}{k} = \frac{n!}{k!\,(n-k)!}$$

In het programma wordt deze tweede methode gebruikt om de driehoek te berekenen.

De driehoek van Pascal komt op allerlei plaatsen in de wiskunde voor. Bijvoorbeeld in de kansrekening: $\binom{n}{k}$ is het aantal manieren om een steekproef van k uit n te trekken. Of in de algebra: de getallen op de n-de rij zijn de coëfficiënten van het polynoom $(1+x)^n$.

```
#include <iostream.h>

int fac( int n )
{
    int res, t;
    res = 1;
    for (t=1; t<=n; t++)
        res *= t;
    return res;
}

int boven( int n, int k )
{
    return  fac(n) / ( fac(k) * fac(n-k) );
}

int main()
{
    int x, y;
    for (x=0; x<10; x++)
    {
        for (y=0; y<=x; y++)
        {
            cout.width(5);
            cout << boven(x,y);
        }
        cout << endl;
    }
}
```

Listing 6: pascal.cc

6.2 De faculteit-functie

In de wiskunde wordt de notatie $n!$ (uit te spreken als 'n faculteit') gebruikt voor het het product van de getallen 1 tot en met n (alle getallen van 1 t/m n met elkaar vermenigvuldigd). Apart gedefinieerd is $0! = 1$. Je zou hopen dat er in de bibliotheek `<math.h>` een functie is gedefinieerd waarmee de faculteit van een getal kan worden uitgerekend, maar helaas.

Met behulp van een `for`-opdracht is het echter niet zo moeilijk om een programmatje te schrijven dat de faculteit van een getal berekent. Zo'n programma is bijvoorbeeld (zelf nog even afmaken met `<iostream.h>` en `main`):

```
int n, t, res;
cout << "getal: " ;
cin  >> n;
res = 1;
for (t=1; t<=n; t++)
    res *= t ;
cout << getal << "! = " << res;
```

Precies volgens de definitie wordt een variabele `res` (van 'resultaat') gelijk gemaakt aan 1, en daarna vermenigvuldigd met alle getallen van 1 t/m n. De extra vermenigvuldiging met 1 aan het begin kan geen kwaad. Mocht n toevallig 0 zijn, dan wordt de opdracht `res*=t;` helemaal niet uitgevoerd, en blijft `res` de waarde 1 houden – dat komt dus goed uit.

6.3 Binomiaalcoëfficiënten

De waarde $\binom{n}{k}$ heet een 'binomiaalcoëfficënt', uit te spreken als 'n boven k'. De waarde ervan kan worden berekend met de formule $n!/k!\,(n-k)!$. Wiskundig is dat makkelijk opgeschreven, maar in C++ wordt het wat onverkwikkelijker:

```
a = 1;
for (t=1; t<=n; t++)
    a *= t ;
b = 1;
for (t=1; t<=k; t++)
    b *= t ;
c = 1;
for (t=1; t<=n-k; t++)
    c *= t ;
cout << n << " boven " << k << " is " << a/(b*c);
```

En dat is dan nog alleen het programma om één binomiaalcoëfficiënt te berekenen; als je er meer wilt hebben wordt het alleen nog maar ingewikkelder.

In het programmafragment staat driemaal vrijwel dezelfde tekst, om de drie faculteiten uit te rekenen. Dat is zonde van het werk. Voor dit soort saaie klusjes hebben we nu juist computers; als programmeur zou je dat soort klerkenwerk niet hoeven te doen. Met een behoorlijke tekstverwerker is het weliswaar niet zo moeilijk om een aantal regels te kopiëren en er daarna nog kleine wijzigingen in aan te brengen, maar denk ook eens aan je arme collega's die jouw programma moeten lezen en aanpassen als de klant over twee jaar een verandering in het programma wil, en jij hebt inmiddels een betere baan elders... Die moeten al die troep weer lezen, en komen pas na lange studie tot de conclusie dat er eigenlijk driemaal hetzelfde staat, en dat het enige wat verschilt de bovengrens in de `for`-opdracht is: achtereenvolgens `n`, `k` en `n-k`.

6.4 Functies definiëren

De manier om het schrijfwerk te vermijden is om zelf een nieuwe functie te definiëren. In de `<math.h>` bibliotheek is al een functie `sqrt` beschikbaar, maar helaas geen functie `fac`; welnu, die kunnen we zelf maken. De definitie van de functie `fac` lijkt qua vorm op die van de functie `main`:

```
int fac( int n )
{   int res, t;
    res = 1;
    for (t=1; t<=n; t++)
        res *= t ;
    return res;
}
```

Eerst het woord `int`, dan de naam van de functie, dan iets tussen haakjes, en tenslotte tussen accolades declaraties en opdrachten.

Het enige verschil met het eerder behandelde complete faculteit-programma is de manier van invoer en uitvoer. De functie `fac` haalt de invoer niet van het toetsenbord (uit de stream `cin`). Functies kunnen hun invoer namelijk ontvangen als *parameter*. In bovenstaande definitie heet de parameter `n` en is van het type `int`. De variabele `n` is daarmee meteen gedeclareerd, zodat na de accolade alleen nog maar `t` en `res` gedeclareerd hoeven te worden.

De uitvoer van functies verloopt evenmin via streams. De functie gaat niet op eigen houtje op het beeldscherm kliederen. In plaats daarvan wordt het resultaat `res` teruggegeven aan wie de functie gebruikt. Daarvoor is een speciale opdracht beschikbaar, genoteerd met het gereserveerde woord **`return`**. Het woord `int` waar de hele functiedefinitie mee begint, is een aanwijzing dat het type van de resultaat van de functie `int` is.

De zelfgedefinieerde functie kan net zoals de standaardfunctie `sqrt` in expressies gebruikt worden. Zo kan elders in het programma de expressie `fac(3) + fac(4)` staan, waarmee netjes de waarde 30 berekend wordt. Wat er met die waarde gebeurt (op het beeldscherm gezet, opgeslagen in een variabele, gebruikt in een andere berekening, of misschien helemaal genegeerd) hangt ervan af in welke context de expressie voorkomt. De vier genoemde voorbeelden krijg je met respectievelijk:

```
cout << fac(3) + fac(4) ;
x = fac(3) + fac(4) ;
```

```
y = 2 * (fac(3) + fac(4)) ;
fac(3) + fac(4) ;
```

Het laatste voorbeeld is natuurlijk weinig zinvol, omdat de functie `fac` en de operator `+` geen neveneffecten hebben.

De functie `fac` kan in het programma `pascal` goed gebruikt worden om de binomiaalcoëfficiënten uit te rekenen. De wiskundige definitie $n!/k!\,(n-k)!$ is direct te vertalen naar de C++-expressie

```
fac(n) / (fac(k) * fac(n-k))
```

Deze expressie zou in het hoofdprogramma (de functie `main`) gebruikt kunnen worden in een uitvoer-opdracht, waarbij de variabelen `n` en `k` met een `for`-opdracht een aantal verschillende waarden doorlopen.

In listing 6 is het echter op een andere manier aangepakt: ook voor het berekenen van de binomiaalcoëfficënten is een aparte functie gedefinieerd, genaamd `boven`:

```
int boven( int n, int k )
{
    return  fac(n) / ( fac(k) * fac(n-k) );
}
```

Deze functie heeft twee parameters, en de hele definitie bestaat uit slechts één opdracht: de `return`-opdracht. De accolades zijn bij functiedefinities desondanks altijd verplicht. In het hoofdprogramma wordt de functie `boven` éénmaal aangeroepen, met als parameters `x` en `y`, die op hun beurt een aantal waarden doorlopen.

Het definiëren van de functie `boven` maakt het programma niet korter. Integendeel. Toch is het een goede gewoonte om duidelijk afgebakende functies ook inderdaad als functie te definiëren, ook al wordt hij maar een keer gebruikt. Het maakt het programma overzichtelijker, en bovendien bereid je je voor op de toekomst: misschien schrijf je later nog eens een programma waarin je ook de functie `boven` nodig hebt, en wel meermalen. Je kunt hem dan kant-en-klaar uit het eerder geschreven programma plukken.

6.5 Lokale variabelen

Op het eerste gezicht lijkt het misschien zorgelijk dat de parameter van `fac` de naam `n` heeft, en de eerste parameter van `boven` ook. Sterker nog, in de derde aanroep van `fac` in de definitie van `boven` bijvoorbeeld, correspondeert de waarde `n-k` van `boven` met de variabele `n` van `fac`. Gelukkig hoef je je over dit soort zaken nooit zorgen te maken. Declaraties gelden namelijk altijd alleen *lokaal* binnen de functie waarin ze gedeclareerd zijn. De variabele `k` kan dus alleen in de functie `boven` gebruikt worden. Daarbuiten, dus in de functies `fac` en `main` is hij domweg niet bekend. Probeer je het toch, dan krijg je van de compiler een foutmelding dat de variabele niet gedeclareerd is.

Het voordeel van de lokale naamgevingsregels is dat je de functies rustig onafhankelijk van elkaar kunt schrijven. Vooral bij grote programma's, die door meerdere personen samen worden geschreven, is dat belangrijk. Ook kun je zonder probleem functies die je in andere programma's gebruikt hebt overnemen in een nieuw programma. Tenslotte zijn bibliotheekfuncties moeiteloos te gebruiken: het zou een mooie boel worden als je in je eigen programma de variabelenamen `x` en `yw` niet mag gebruiken omdat die eventjes nodig waren bij de definitie van `sqrt`.

Het enige wat je hoeft te weten van functies die je wilt aanroepen is het type van de parameters en het type van het resultaat. Hierover moeten teams van programmeurs het dan ook eens worden als ze samen een programma schrijven, maar de naamgeving van de variabelen is de zorg voor de schrijver van de functie.

6.6 Object-methoden

Er is nog één opdracht in listing 6 die toelichting behoeft, namelijk

```
cout.width(5);
```

Streams, zoals `cout`, zijn voorbeelden van wat men in C++ noemt 'objecten'. Objecten kunnen privé-functies hebben die een neveneffect hebben op het gedrag van het object. Dat soort functies heten *methoden*. De functie `width` is zo'n methode.

Alle uitvoer-streams hebben een methode `width`. De methode heeft een parameter, waarmee je de breedte kunt aangeven de volgende waarde die met `<<` naar de stream wordt gestuurd. Door de aanroep `cout.width(5)` vlak vóór de uitvoer van getallen te zetten, wordt voor elk getal evenveel ruimte gebruikt, namelijk 5 posities. Daardoor wordt de driehoek van Pascal in mooie regelmatige kolommetjes opgebouwd.

Methoden van een object kun je aanroepen door de naam van het object, een punt, en de naam van de methode te noemen, gevolg door eventuele parameters. In latere hoofdstukken zullen we zien hoe je ook zelf objecten en bijbehorende methodes kunt definiëren.

Opgaven

1. Schrijf een functie die de absolute waarde van een `double` berekent.
2. Maak een functie rondom het programmafragment van vorige week waarin je zelf de wortel uitrekende van een getal. Maak gebruik van de functie uit opgave 1.
3. Schrijf een functie die 1 oplevert als zijn parameter een priemgetal is, en 0 als dat niet het geval is. Je mag ervan uitgaan dat de parameter een natuurlijk getal ≥ 2 is. (Een getal is een priemgetal als het alleen maar deelbaar is door 1 en zichzelf. Deelbaarheid kun je testen door te controleren of de rest-bij-deling nul is.)

Hoofdstuk 7

Wisselgeld, *of:* void-functies

In dit hoofdstuk bekijken we een programma waarin een functie gedefinieerd wordt zonder resultaatwaarde. De functie laat zijn blijvende sporen in de plaats daarvan achter in de vorm van een neveneffect. Dit neveneffect treedt op door middel van uitvoer naar het scherm, èn toekenning aan een veranderbare parameter. Verder wordt in het programma een tekst als parameter meegegeven aan een functie. Tenslotte wordt gedemonstreerd hoe getallen afgerond kunnen worden.

7.1 Het programma

Het programma in dit hoofdstuk heet `wisselgeld`, omdat het berekent hoe een bedrag kan worden uitbetaald in verschillende coupures. In listing 7 staat de tekst van het programma.

Het programma accepteert als invoer via het toetsenbord geldbedragen in guldens (althans getallen die zo'n bedrag voorstellen. De biljetten zelf mogen via de floppy-drive gedoneerd worden.) Voor elk bedrag wordt aangegeven hoe het met biljetten en munten kan worden samengesteld. Daarna kan het volgende bedrag worden ingevoerd, totdat de invoer-stream wordt afgesloten met control-D.

Een sessie zou er als volgt uit kunnen zien:

```
Geef bedragen:
38.1
1 geeltje
1 tientje
1 rijksdaalder
2 kwartjes
1 duppie
0.8
3 kwartjes
1 stuiver
```

Het programma betaalt steeds met zo groot mogelijke coupures. Van het resterende bedrag wordt zoveel mogelijk met een lagere coupure betaald, enzovoort. Apparaat wisselt niet. Door het Nederlandse systeem van muntopbouw wordt hiermee tevens het totaal aantal briefjes/munten geminimaliseerd. Drie gulden wordt gepast met een rijksdaalder en twee kwartjes, maar dat zijn net zoveel munten als drie losse guldens.

7.2 Afronden

In het hoofdprogramma wordt het bedrag ingevoerd als `float` getal. De 7 significante cijfers zijn genoeg om een ton in centen nauwkeurig te coderen. Het rekenen gaat, zoals spoedig zal blijken, echter gemakkelijker met integers. Direct na het inlezen van een bedrag wordt dit daarom geconverteerd naar centen door het met 100 te vermenigvuldigen:

```
double geld;
int centen;
while (cin>>geld)
{
    centen = 100*geld;
```

Als bijvoorbeeld het bedrag `38.1` wordt ingevoerd, krijgt de variabele `centen` de waarde 3810. Hierbij treedt een speciale afrondingsregel in werking. De variabele `geld` is namelijk een `float`,

```
#include <iostream.h>

void coupure( int & bedrag, int waarde, char * naam )
{
    int aantal;

    aantal = bedrag / waarde;
    bedrag = bedrag % waarde;

    if (aantal>0)
    {
        cout << aantal << " " << naam ;
        if (aantal>1)
            cout << "s";
        cout << endl;

    }
}

int main()
{
    double geld;
    int centen;

    cout << "Geef bedragen:" << endl;
    while (cin >> geld)
    {
       centen = geld * 100;
       coupure( centen, 10000, "meier"   );
       coupure( centen,  2500, "geeltje" );
       coupure( centen,  1000, "tientje" );
       coupure( centen,   500, "vijfje"  );
       coupure( centen,   250, "rijksdaalder" );
       coupure( centen,   100, "gulden"  );
       coupure( centen,    25, "kwartje" );
       coupure( centen,    10, "duppie"  );
       coupure( centen,     5, "stuiver" );
    }
}
```

Listing 7: wisselgeld.cc

en de expressie `100*geld` dus ook. Omdat `centen` een `int` is, wordt het eventuele breuk-gedeelte van `100*geld` verwaarloosd. Mocht de gebruiker dus het bedrag `38.123` invoeren, dan vervallen er drie-tiende cent aan de bank.[1] In het algemeen is het niet veilig om een float-waarde aan een int-variabele toe te kennen. De compiler geeft daarom een waarschuwing bij deze opdracht. In dit programma kunnen we die waarschuwing echter voor kennisgeving aannemen en negeren, omdat de verdwenen breukdelen van centen toch niet in munten gepast kunnen worden.

Een tientje is een duizend-cent biljet. Om te bepalen hoeveel tientjes er in een bedrag (in centen) gaan, kan dit bedrag door 1000 gedeeld worden. Als bij de delings-operator beide operanden van het type `int` zijn, is het resultaat dat ook. Een eventueel breukdeel wordt weer verwaarloosd. Pas als één van beide (of allebei de) operanden een `float` of `double` zijn, wordt de deling exact uitgevoerd. (In het programma gemiddeld hebben we daar stilzwijgend gebruik van gemaakt.) Omdat er geen halve tientjes bestaan, komt de verwaarlozing van het breukdeel in dit geval goed

[1] In een van de eerste gevallen van computerfraude (zestiger jaren) liet een programmeur bij een bank deze door afronding ontstane bedragen op zijn eigen rekening bijschrijven, en dan natuurlijk zonder afronden. Geen enkele klant miste wat, en programmeur werd na vele miljoenen transacties toch behoorlijk rijker!

van pas: 3810 centen zijn 3 tientjes, en dan blijven er nog 810 centen over.

In het programma zou daarom een opdracht kunnen staan als

```
aantal = centen / 1000;
cout << aantal << " tientjes";
```

Het resterende bedrag kan berekend worden met

```
centen = centen - (aantal*1000);
```

Mathematisch gesproken zou hierdoor aan de waarde `centen` niet veranderen, want $c - (c/1000) * 1000$ is gelijk aan c. bij int-deling is vermenigvuldigen echter niet precies het omgekeerde van delen, doordat de afronding bij het delen niet ongedaan wordt gemaakt. Het berekende aantal (3 in het voorbeeld) vermenigvuldigd met 1000 is de waarde van de drie tientjes, en afgetrokken van het gehele bedrag levert dit het resterende bedrag (810 cent).

Een andere manier om deze 810 cent te berekenen is

```
centen = centen % 1000 ;
```

De operator `%` berekent de 'rest bij deling' – het getal dat je helemaal onderin overhoudt bij een staartdeling (zou je dat nog leren op de basisschool, tegenwoordig?). Omdat de formulering met de `%`-operator overzichtelijker is dan dat gepriegel met eerst delen met afronden en dan weer vermenigvuldigen, is het deze formulering die we gebruiken in listing 4. Tussen haakjes: deze operator komt ook goed van pas bij het rekenen met cyclische getallen, zoals uren of weekdagen. Als het nu bijvoorbeeld 10 uur is, dan is het over 100 uur `(10+100)%24` is 14 uur.

Terug naar het programma. Dit zou als volgt de verschillende coupures kunnen berekenen:

```
aantal = centen / 1000;
centen = centen % 1000;
cout << aantal << " tientjes";
aantal = centen / 500;
centen = centen % 500;
cout << aantal << " vijfjes";
aantal = centen / 100;
centen = centen % 100;
cout << aantal << " guldens";
```

en zo verder voor hogere en lagere coupures.

7.3 Void-functies

Aan het regelmatige patroon hierin kun je al zien dat dit vast handiger moet kunnen met behulp van functies. Er staat tenslotte driemaal (of eigenlijk tienmaal, als we van honderdje tot stuiver alles meenemen) vrijwel hetzelfde. Het enige wat verschilt is het getal dat de waarde van de coupure beschrijft (1000, 500, 100 enz.) en de tekst met de naam van de coupure (`"tientje"`, `"vijfje"` en `"gulden"`).

In de functie komen de drie opdrachten te staan waarin het aantal bepaald wordt, het restantbedrag uitgerekend wordt, en het aantal op de uitvoer-stream wordt geschreven. Parameters van de functie worden gebruikt om dat wat steeds verschilt (de waarde van de coupure en de tekst met de naam ervan) te kunnen aanduiden. De vraag is echter wat de functie als resultaatwaarde moet opleveren. Daarvoor komt eigenlijk niets in aanmerking. De functie berekent namelijk niet een waarde, zoals `fac` en `boven` in het vorige hoofdstuk. Het blijvende effect van de functie wordt tot stand gebracht door het neveneffect van de functie: het tonen van tekst op de uitvoer-stream.

Toch moet de functie een resultaat-type hebben. In de definitie moet vóór de naam van de functie nu eenmaal het resultaattype worden vermeld. Voor dit soort gevallen is er daarom een speciaal type beschikbaar, `void` genaamd. Het woord 'void' betekent letterlijk 'leeg'. Een functie met een resultaat van type `void` wil zeggen dat de functie geen nuttige resultaatwaarde oplevert, en dat alle actie dus van het neveneffect verwacht moet worden. In functies met een `void` resultaat is een `return`-opdracht overbodig.

De parameters van de functie zijn minder een probleem. De waarde van de coupure (1000, 500, 100 enz.) heeft het type `int`. Ook een tekst (zoals `"gulden"` of `"kwartje"`) kan als parameter aan een functie worden meegegeven. Als type van zo'n parameter moet `char *` gebruikt worden. In variabelen van het type `char` kan één letterteken worden opgeslagen. Een sterretje achter het type wil zeggen dat niet één letter als parameter wordt meegegeven, maar een verwijzing naar een hele serie lettertekens – een tekst dus.

Er is nog een derde parameter nodig. De functie moet namelijk de waarde van de variabele `centen` kennen. Die variabele is in de functie `main` gedeclareerd. In het vorige hoofdstuk bleek dat declaraties *lokaal* gelden in een functie; de variabelen mogen dus niet in een andere functie gebruikt worden. De functie krijgt daarom een derde parameter (van type `int`), waarmee de waarde van de variable `centen` aan de functie wordt doorgespeeld.

Als naam voor de functie zullen we `coupure` gebruiken, omdat de functie een bedrag opsplitst in coupures. De eerste regel van de definitie zou er als volgt uit kunnen zien:

```
void coupure( int bedrag, int waarde, char* naam )
```

De functie kan in het hoofdprogramma dan meermalen aangeroepen worden:

```
coupure( centen, 1000, "tientje" );
coupure( centen,  500, "vijfje"  );
coupure( centen,  100, "gulden"  );
```

De parameter `bedrag` is om een andere reden aanwezig dan `waarde` en `naam`. Voor die laatste twee wordt bij elke aanroep een verschillende waarde meegegeven; de parameter `bedrag` is uitsluitend aanwezig om de waarde van de lokale variabele `centen` aan de functie door te spelen.

Er is nog één extra complicatie. De functie heeft niet alleen de waarde van de variabele `centen` nodig, hij moet er ook een nieuwe waarde aan toekennen. Via onze parameter heet dit inmiddels `bedrag`, dus de opdracht in de functie moet luiden:

```
bedrag = bedrag % waarde;
```

Het probleem met deze opdracht is dat niet de waarde van de variabele `centen` wordt veranderd, maar de waarde van de *lokale kopie* daarvan (die `bedrag` heet). Maar als de functie `coupure` verderop aan `bedrag` refereert is de waarde inderdaad veranderd. Als de functie `main` de controle weer overneemt, is de waarde van `centen` onveranderd gebleven. Deze waarde wordt aan de volgende aanroep van `coupure` opnieuw meegegeven, zodat na uitbetaling van de tientjes hetzelfde bedrag ook nog eens in guldens wordt afgepast, in plaats van alleen het restantbedrag.

Ook voor deze situatie is een oplossing voorhanden:

> als een functie de waarde van een meegegeven parameter moet kunnen *veranderen*, dan moet bij de declaratie van de parameter een `&`-symbool achter het type gezet worden.

Bij aanroep van de functie moet op de overeenkomstige parameterpositie dan wel een variabele worden meegegeven. Je kunt moeilijk de waarde 37 meegeven, en verwachten dat deze waarde 'veranderd' wordt door de functie. Wat zou dat immers betekenen: dat 37 voortaan 91 is? De compiler controleert of voor 'veranderbare parameters' inderdaad variabelen (van het juiste type) worden meegegeven.

Al met al luidt de definitie van de functie `coupure` als volgt:

```
void coupure( int & bedrag, int waarde, char * naam )
{
    int aantal;
    aantal = bedrag / waarde;
    bedrag = bedrag % waarde;
    if (aantal>0)
        cout << aantal << " " << naam << endl ;
}
```

De declaratie van de variabele `aantal` kan rustig lokaal gebeuren. Deze variabele wordt immers alleen binnen de functie `coupure` gebruikt. In listing 7 is de procedure nog iets verfijnd, zodat na de naam van de coupure een `"s"` wordt getoond als het meer dan één exemplaar betreft.

7.4 `return` of `cout`?

Bij het definiëren van een functie is het goed om even stil te staan bij de vraag of deze functie zijn resultaat naar de aanroepende expressie moet retourneren met een `return`-opdracht, of dat het handiger is dat de berekende waarde via `cout` op het scherm gezet wordt. Omgekeerd kan de invoer voor een functie als parameter worden doorgespeeld, maar het is ook mogelijk dat de functie zelf de invoer-stream `cin` inspecteert.

In het algemeen zijn `void`-functies die hun resultaat direct in de uitvoer-stream zetten het geschikst als er sprake is van ingewikkelde layout-zaken. In de functie `coupure` is dat bijvoorbeeld het geval bij het gepruts om de meervoud-s al of niet te tonen. Meer klassieke 'wiskunde-achtige' functies moeten echter vooral niet op eigen houtje de uitvoer-stream bekladden. Stel je voor dat de functie `fac` in het vorige hoofdstuk na het berekenen van de waarde deze direct op het beeldscherm zou tonen. Tijdens het berekenen van een binomiaalcoëficiënt zouden dan op het beeldscherm allerlei tussenresultaten getoond worden, die helemaal niet voor de gebruiker bestemd zijn.

In grote programma's is er meestal sprake van een groot aantal functies. De meeste van die functies retourneren hun waarde via `return`-opdrachten, of veranderen hun veranderbare parameters. Slechts een klein aantal functies verzorgt de layout van de uitvoer, en 'mag' daartoe de uitvoer-stream `cout` gebruiken.

Opgaven

1. Schrijf een functie die de haakjes uitwerkt in het polynoom $(1+x)^n$, waarbij n een parameter is. Het resultaat moet naar de uitvoerstream gestuurd worden. Voor parameter 4 moet bijvoorbeeld afgedrukt worden:

   ```
   x^4 + 4x^3 + 6x^2 + 4x + 1
   ```

2. Schrijf een functie `verwissel` die de waarden van twee integer variabelen kan omkeren. Het programmafragment

   ```
   x = 3;
   y = 5;
   verwissel(x,y);
   cout << x << " " << y ;
   ```

   moet dus `5 3` op de uitvoer-stream zetten.

Hoofdstuk 8

Cijferfrequentie, *of:* arrays

In dit hoofdstuk bekijken we een programma waarin met één declaratie een hele rij variabelen wordt gedeclareerd. In dit geval betreft het een rij `int`-variabelen, waarin aantallen kunnen worden opgeslagen. Zo'n rij aantallen kunnen we beschouwen als een soort 'turf-tabel'.

8.1 Het programma

Het programma in dit hoofdstuk heet `frequ`, omdat het de frequentie van cijfers in een cijferlijst bepaalt. In listing 8 staat de tekst van het programma.

Elke docent wordt van tijd tot tijd geconfronteerd met een lijst tentamenuitslagen. Het programma `frequ` biedt hulp bij de analyse daarvan. Het programma vraagt om een lijst cijfers, die zoals gewoonlijk wordt afgesloten met de einde-stream-indicatie control-D. Vervolgens beeldt het programma een tabelletje af met de behaalde uitslagen, bijvoorbeeld:

```
4: 3x
5: 4x
6: 6x
7: 2x
8: 4x
10: 1x
```

Ook de gemiddelde score wordt berekend:

```
gemiddeld: 6.2
```

Tenslotte wordt de uitslag ook nog eens grafisch weergegeven met een histogram:

```
 0:
 1:
 2:
 3:
 4: ***
 5: ****
 6: ******
 7: **
 8: ****
 9:
10: *
```

In het tabelletje zijn de cijfers met nul uitslagen weggelaten, maar in het histogram terwille van het totaalbeeld niet.

8.2 Rijen variabelen

Het programma `frequ` is in feite een uitbreiding van het programma `gemiddeld`. Ook nu moeten, ten behoeve van de berekening van het gemiddelde, het totaal en aantal van ingelezen cijfers tijdens het inlezen worden bijgehouden. Maar daarnaast moet nu ook *per cijfer* het aantal worden bijgehouden. Al die aantallen zijn onafhankelijk van elkaar; voor elk is dus een aparte variabele nodig.

Een eerste idee is om hiervoor elf (!) variabelen te declareren:

```
#include <iostream.h>

void verslag(int cijfer, int aantal )
{
   if (aantal>0)
       cout << cijfer << ": " << aantal << "x" << endl;
}

void histo(int cijfer, int aantal )
{  int i;
   cout.width(2);
   cout << cijfer << ": " ;
   for (i=0; i<aantal; i++)
       cout << "*";
   cout << endl;
}

int main()
{
    int totaal, getal;
    int aantal;
    int turf[11];

    totaal = 0;
    aantal = 0;
    for (getal=0; getal<11; getal++)
        turf[getal] = 0;

    cout << "Geef cijfers, afgesloten door control-D." << endl ;
    while ( cin >> getal )
    {
        if (getal<0 || getal>10)
            cout << "cijfer moet tussen 0 en 10 liggen" << endl;
        else
        {
            totaal += getal;
            aantal ++;
            turf[getal] ++;
        }
    }

    for (getal=0; getal<11; getal++)
        verslag( getal, turf[getal] );

    if (aantal>0)
        cout << "gemiddeld: " << totaal/aantal << endl ;

    for (getal=0; getal<11; getal++)
        histo( getal, turf[getal] );
}
```

Listing 8: frequ.cc

```
int n0, n1, n2, n3, n4, n5, n6, n7, n8, n9, n10;
```

Al deze variabelen moeten van een zinvolle beginwaarde worden voorzien:

```
n0=0; n1=0; n2=0; n3=0; n4=0;
n5=0; n6=0; n7=0; n8=0; n9=0; n10=0;
```

Bij het inlezen van de cijfers kan dan de betreffende variabele worden opgehoogd:

```
while (cin >> cijfer)
{
    if      (cijfer==0) n0++;
    else if (cijfer==1) n1++;
    else if (cijfer==2) n2++;
    else if (cijfer==3) n3++;
```

enzovoort, enzovoort. Tenslotte moeten de waarden van de tien variabelen worden afgedrukt.

Met het nodige tikwerk is het programma hiermee wel te schrijven. Maar zo'n lange rij vrijwel dezelfde opdrachten (en de behoefte aan het woord 'enzovoort'!) is verdacht. Trouwens, voor schoolcijfers is dit programma misschien nog wel te ontwikkelen, maar voor het turven van bijvoorbeeld postcodes wordt het tikwerk toch wel erg vervelend.

Veel handiger zou het zijn als je in één opdracht zou kunnen specificeren: 'bij inlezen van cijfer x moet variabele `n`x worden opgehoogd'. Zoiets is inderdaad mogelijk. Een mislukte poging is de volgende opdracht:

```
while (cin >> cijfer)
    ncijfer ++;
```

dit zou immers beschouwd worden als de opdracht om de variabele met de 7-letterige naam `ncijfer` te verhogen, en zo'n variabele is nergens gedeclareerd. Daarom moet de expressie die het volgnummer van de variabele aangeeft tussen vierkante haken geschreven worden:

```
while (cin >> cijfer)
    n[cijfer] ++;
```

De declaratie van de variabelen moet als volgt geschieden:

```
int n[11];
```

De naam die hiermee wordt gedeclareerd is eigenlijk alleen `n`. Deze naam duidt echter een rij van elf variabelen aan, die kunnen worden aangesproken door de naam `n` gevolgd door hun volgnummer tussen vierkante haken.

8.3 Arrays

Een rij variabelen met één naam heet (heel origineel) een *array*. Bij de declaratie van een array wordt het type en het aantal variabelen in de rij gespecificeerd. Dit aantal moet reeds aan de compiler bekend zijn. Het is dus niet mogelijk om een getal in te lezen, en vervolgens een array te declareren met zoveel elementen. Wel mag je, als je daar zin in zou hebben, het aantal elementen met een expressie specificeren, als die expressie maar door de compiler uit te rekenen is:

```
int n[5+6];
```

Als je een array `a` met drie elementen hebt gedeclareerd met de declaratie

```
int a[3]
```

dan zijn die drie variabelen te benoemen met `a[0]`, `a[1]` en `a[2]`. Een volgnummer van variabelen in een array heet een *index* (meervoud *indices*). De indices van de variabelen in een array beginnen dus bij 0, dat wil zeggen: het 'eerste' element is `a[0]`, het 'tweede' element is `a[1]` en het 'derde' is `a[2]`. Overigens is het in sommige kringen van programmeurs en wiskundigen gebruik om te beginnen te tellen bij 0; dit soort lieden zegt dan ook gewoonweg 'het nulde element is `a[0]`, het eerste element is `a[1]`, het tweede element is `a[2]` en een derde element is er niet'.

Bij de declaratie wordt het *aantal* elementen van de array opgegeven. Doordat de telling bij nul begint is het uitkijken geblazen: een array die gedeclareerd is met `int a[3]` heeft géén element `a[3]`; het laatste element is `a[2]`. Daarom wordt in het programma frequ de array met aantallen voor elk schoolcijfer gedeclareerd met

```
int aantal[11]
```

In deze 11 variabelen kunnen de aantallen voor de cijfers 0 t/m 10 opgeslagen worden; `a[10]` is het laatste element.

Het handige van arrays is dat bij gebruik van de variabelen (anders dan bij de declaratie ervan) de index een *expressie* mag zijn, met een waarde die aan de compiler nog niet bekend hoeft te zijn. Daardoor is het mogelijk om te schrijven

```
while (cin >> getal)
    aantal[getal] ++;
```

Elementen uit een array, aangeduid door een naam en een index tussen vierkante haken, mogen op alle plaatsen gebruikt worden waar een variabele mag staan. Dus bijvoorbeeld in een expressie, maar ook aan de linkerkant van een toekenningsopdracht (toekenning aan een variabele met `=`, of afgekorte vormen met `+=` of `++`), als resultaat-variabele achter `cin>>`, en bij aanroep van een functie met veranderbare parameter.

8.4 Arraygrenzen

Arrays zijn gevaarlijk! (Maar alle dingen die het leven leuk maken zijn gevaarlijk.) Je moet er als programmeur namelijk altijd voor zorgen dat de index van een array wel binnen het gedeclareerde bereik valt. De waarde van de index moet *kleiner* zijn dan het aantal elementen (en dus niet gelijk, of groter, en natuurlijk ook niet negatief).

Wat gebeurt er als de index tòch groter is dan toegestaan? Vooraleerst merk je daar niets van. De compiler slikt het programma zonder zeuren. Allicht, want de compiler kan de gebruikte index vaak niet eens berekenen, dus ook niet controleren. Ook bij het uitvoeren van het programma gaat het programma gewoon door na schendingen van de array-limiet. Dit is anders dan bijvoorbeeld bij delen door nul, waarbij het programma wordt gestopt. Maar welke waarde heeft een arrayelement met een te grote index dan?

Welnu, net als andere variabelen worden arrays in het geheugen van de computer opgeslagen. De elementen van de array liggen netjes naast elkaar, zodat bij een gegeven index het juiste element er netjes uitgepikt kan worden. Gebruik je een index die 1 te groot is, dan krijg je de waarde te zien die toevallig naast de array staat opgeslagen. Bij andere verboden index-waarden krijg je misschien variabelen van andere functies, stukjes programma (want ook dat staat in het geheugen opgeslagen) of delen van de interne administratie van functie-aanroepen.

Een typisch symptoom van array-indicering buiten de limiet is bijvoorbeeld uitvoer als:

```
cijfer 11: -3481x
cijfer 12: 17x
```

Als je geluk hebt duidelijke nonsens-waarden die direct opvallen; als je minder geluk hebt nonsens-waarden die zich goed vermomd hebben als 'echte' waarden.

Nog gevaarlijker is *toekenning* aan niet-bestaande array-elementen. Je zit dan in onbeschermd geheugen te peuteren, met als gevolg dat bijvoorbeeld andere variabelen van het programma worden veranderd. Dat kan zeer rare effecten opleveren, vooral omdat dit soort fouten soms pas veel later indirect aan het licht komt(misschien omdat de onverhoeds veranderde variabele ook weer wordt gebruikt als index een een array, die daarmee buiten zijn grenzen loopt, waardoor...)

Is er dan helemaal niets aan te doen? Nee. Hoewel, je kunt de grootste rampen voorkomen door op onderstaande dingen te letten:

- Als je niet zeker weet dat een index in het toegestane bereik ligt, controleer dat dan met een `if`-opdracht. In het programma `frequ` wordt bijvoorbeeld na het inlezen van een getal gecontroleerd of dit een geldig rapportcijfer is (en daarmee een geldige index). Zo nee, dan wordt de indicering niet uitgevoerd. Ook parameters van een functie kun je aan extra controle onderwerpen, als je niet zeker weet vanaf welke plaatsen de functie wordt aangeroepen.

- De elementen van een array worden vaak doorlopen met een `for`-opdracht. Gebruik daarin de volgende formulering:

  ```
  int a[5];
  for (t=0; t<5; t++) a[t] = 0 ;
  ```

  De test of de bovengrens bereikt is, maakt gebruik van de operator `<` en niet van `<=`. Daardoor kun je in deze test van hetzelfde getal gebruik maken als in de declaratie van de array. De kans op vergissingen is op deze manier kleiner dan wanneer je zou schrijven `t<=4`.

- Bedenk: de geldige indices van een array lopen van 0 *tot en zonder* het aantal elementen.

8.5 Opzet van een programma

In het programma frequ worden twee functies gedefinieerd: `verslag` en `histo`. Beide functies zijn `void`-functies die hun neveneffect bewerkstelligen door informatie in de uitvoer-stream te zetten.

Strikt genomen was het niet nodig om deze functies te definiëren. Ze worden allebei slechts één keer aangeroepen. De tekst van de definitie had daarom ook wel op de plaats van de aanroep geschreven kunnen worden. Toch is er in het programma voor gekozen om de functies wèl te definiëren. Het hoofdprogramma is er namelijk een stuk overzichtelijker door geworden. Je kunt nu direct zien dat de turftabel tweemaal doorlopen wordt, waarbij de waarden aan de functies `verslag` en `histo` worden meegegeven. Ben je geïnteresseerd hoe een regel van het histogram precies wordt opgebouwd, dan kun je dat opzoeken in de functie `histo`. Met de details wordt je echter niet lastiggevallen bij bestudering van het hoofdprogramma.

Psychologisch onderzoek heeft aangetoond dat een mens slechts verrassend weinig 'dingen' tegelijk kan overzien. En een programmeur is ook maar een mens...

8.6 Niet afronden

In het programma gemiddeld waren de ingelezen getallen en hun totaal van type `double`. Het aantal getallen was natuurlijk een `int`. Het gemiddelde werd berekend met de expressie `totaal/aantal`. Als bij een deling één van beide operanden (of allebei) een reëel getal is, wordt de deling exact uitgevoerd.

In het programma frequ is het totaal van de ingelezen getallen echter een `int`. Bij berekening van het gemiddelde met de expressie `totaal/aantal` zijn dus beide operanden van het type `int`. In dat geval is het resultaat van de deling ook van type `int`, en wordt het breukdeel verwaarloosd. Er wordt nog niet eens afgerond, maar echt verwaarloosd; dus 6.9 wordt 6 en niet 7.

Om ervoor te zorgen dat toch de exacte deling gebruikt wordt, moet tenminste een van beide operanden tot `float` of `double` gemaakt worden. Dat kan met behulp van een extra variabele:

```
int totaal, aantal; float extra;
extra = totaal;
cout << "gemiddeld: " << extra/aantal;
```

Dat is echter onnodig omslachtig. Indien nodig kan een int namelijk worden omgezet in een float door er `(float)` voor te zetten. Dat heet een *cast* (letterlijk: 'gietvorm'; de int wordt als het ware in de gietvorm van een float geperst). In het programma wordt gebruik gemaakt van zo'n cast om de variabele `totaal` als `float` te beschouwen, waardoor de exacte deling uitgevoerd wordt:

```
cout << "gemiddeld: " << (float)totaal / aantal;
```

Opgaven

1. Hoeveel palen heb je nodig voor een dranghek van tien segmenten? Hoe oud ben je? En met je hoeveelste levensjaar ben je bezig? Welk jaar leven we in? En de hoeveelste eeuw is het? In welk jaar is Jezus eigenlijk geboren? Wanneer vieren we dus het begin van het volgende millennium? En wat heeft dit allemaal met array-declaraties te maken?

2. De 'spreiding' van een lijst uitslagen is de gemiddelde afwijking van het gemiddelde. Bij een kleine spreiding liggen de meeste cijfers in de buurt van het gemiddelde, bij een grote spreiding zijn er veel uitschieters naar onder en naar boven. Breid het programma frequ uit, zodat ook de spreiding berekend wordt.

3. In het programma pascal worden de binomiaalcoëfficiënten uitgerekend met de formule met de faculteiten erin. In het diktaat wordt ook een andere methode genoemd: in de linkerkolom en op de diagonaal staat steeds het getal 1, en verder is elk getal de som van het getal er recht boven en er schuin links boven. Schrijf een programma dat (een deel van) de driehoek afdrukt met gebruikmaking van deze methode. Gebruik een array waarin je een complete rij berekent, die je vervolgens afdrukt. Je kunt vervolgens de waarden in de array gebruiken om de volgende rij uit te rekenen. Let wel op dat je variabelen in de array niet verandert zolang de oude waarde nog nodig is!

Hoofdstuk 9

Kassa!, *of:* parameters

In dit hoofdstuk bekijken we een programma waarin gegevens uit een file kunnen worden gelezen, zodat de gebruiker niet steeds alle informatie via het toetsenbord hoeft in te voeren. Verder bespreken we hoe een array als parameter van een functie kan worden meegegeven. Ook de functie `main` blijkt parameters te kunnen accepteren.

9.1 Het programma

Het programma in dit hoofdstuk heet `kassa`, omdat het bij een kassa gebruikt kan worden om bedragen op te zoeken bij streepjescodegetallen. In listing 9 staat de tekst van het programma.

Tot nu toe is alle invoer voor programma's via het toetsenbord verlopen. Programma's werden gestart door simpelweg de naam ervan te noemen (meestal `a.out`). Het is echter ook mogelijk om aan programma's via de commandoregel nog extra informatie toe te spelen. Een voorbeeld daarvan is de compiler (zelf ook een programma): via de commandoregel wordt de naam van het programma meegedeeld dat gecompileerd moet worden.

Het programma `kassa` maakt gebruik van deze mogelijkheid. Via de commandoregel kan de naam van een file worden doorgespeeld, waarin van een aantal produkten de streepjescode en de prijs staan gegeven (voor elk produkt een regel). Streepjescodes bestaan meestal uit 13 cijfers. Het getal dat daardoor gevormd wordt is te groot om in een `int` op te slaan (de maximale waarde is immers 2.1 miljard). Om het niet te ingewikkeld te maken, beperken we ons hier tot produkten met een 'korte' streepjescode van 8 cijfers (dat past wel in een `int`[1]). Een voorbeeld van twee regels uit de file met produkten (een pot Mellona-honing en een doosje Pottertjes[2]):

```
87101621  5.45
87233025  1.69
```

Het programma leest al deze gegevens uit de file, en slaat ze op in twee arrays. Vervolgens accepteert het programma als invoer van de gebruiker streepjescode-getallen, waarbij het bijbehorende bedrag wordt opgezocht:

```
87101621
5.45
12345678
bedrag onbekend
```

9.2 Array-parameters

We kijken eerst naar de tweede helft van het programma. In de functie main worden arrays gedeclareerd waarin de codes en bijbehorende bedragen worden opgeslagen:

```
int   codes[1000],    code;
float bedragen[1000], bedrag;
```

Behalve de arrays declareren we ook twee losse variabelen, waarin steeds één waarde kan worden ingelezen. Zoals je ziet hoeft bij het declareren van een array en een losse variabele van hetzelfde type de naam van het type niet herhaald te worden. Omdat het van te voren niet bekend is hoeveel

[1] behalve op een PC waarop `int`s beperkt zijn tot 32767 – *kids, don't try this at home!*
[2] sterk, maar goed!

```
#include <fstream.h>

int zoek( int code, int tabel[], int aantal )
{   int i;
    for (i=0; i<aantal; i++)
        if (code==tabel[i])
            return i;
    return -1;
}

int main(int aantalNamen, char *namen[] )
{
    // namen[0] is de naam van het programma
    // namen[1] is de naam van de eerste parameter
    // als er 1 parameter is, geldt dus aantalNamen==2

    int    code,   codes[1000];
    double bedrag, bedragen[1000];
    int i, aantal;

    if (aantalNamen!=2)
    {
        cout << "Gebruik dit programma met 1 parameter" << endl;
        return 1;
    }

    ifstream invoer( namen[1] );

    cout << "Bezig met inlezen tabel..." << endl;
    aantal = 0;
    while( invoer >> code >> bedrag )
    {
        codes[aantal]    = code;
        bedragen[aantal] = bedrag;
        aantal++;
    }

    cout << "Gereed om codes op te zoeken." << endl;
    while ( cin >> code )
    {
         i = zoek(code, codes, aantal);
         if (i>=0)
             cout << "bedrag: " << bedragen[i] << endl;
         else
             cout << "bedrag onbekend." << endl;
    }
}
```

Listing 9: kassa.cc

produkten er in de file zullen staan, maken we de arrays maar lekker groot. Tijdens het inlezen van de file zullen we het werkelijke aantal bepalen, en opslaan in de variabele `aantal`:

```
int aantal;
```

Laten we even aannemen dat de arrays al ingelezen zijn. De interactie met de gebruiker kan dan beginnen:

```
cout << "Gereed om codes op te zoeken." << endl;
```

```
while ( cin >> code )
{
```

In de body van de `while`-opdracht moet nu worden opgezocht op welke plaats `code` in de array `codes` staat. Vervolgens kan het bijbehorende bedrag worden getoond. Om het hoofdprogramma niet te ingewikkeld te maken, maken we een aparte functie die het zoeken van de goede index voor zijn rekening neemt.

Deze functie noemen we `zoek`, en om te beginnen moeten we nadenken over de parameters en het resultaat ervan. Als parameters krijgt de functie de code die opgezocht moet worden, maar ook de tabel waarin alle codes staan opgeslagen, en het getal dat aangeeft hoeveel plaatsen in die tabel daadwerkelijk zijn ingevuld. De tabel met bedragen hoeft niet aan de functie te worden doorgespeeld; de functie hoeft immers alleen de index van de code op te zoeken. Het hoofdprogramma zal deze index gebruiken om het bijbehorende bedrag op te zoeken. Het resultaat van de functie `zoek` is een `int`, namelijk de index van de code in de tabel.

Nu moeten we nog verzinnen wat de functie zal retourneren als de code niet aanwezig is in de tabel. Een speciale waarde, die nooit als echte index kan voorkomen. Nul is een speciale waarde die vaak als foutcode in functieresultaten gebruikt wordt. Hier kan dat echter niet, omdat 0 een toegestane index in de tabel is: arrays beginnen immers te tellen bij 0. Nou, -1 dan maar; dat kan zeker geen index zijn. We spreken dus met onszelf af: de functie levert -1 op als de code niet in de tabel voorkomt, en anders de index in de tabel waar hij voorkomt.

De eerste regel (ook wel *header* genaamd) van de definitie van `zoek` is:

```
int zoek( int code, int tabel[], int aantal )
```

De tweede parameter is de array waarin gezocht moet worden. Array-parameters worden net zo gedeclareerd als array-variabelen. Alleen het aantal elementen hoeft (of liever: mag) niet worden ingevuld. Het aantal elementen wordt immers bepaald op de plaats waar de array aan de functie wordt meegegeven. De functie moet dus op een andere manier te weten komen hoeveel elementen van de array gebruikt mogen worden. In dit geval hebben we dat opgelost met een derde parameter.

De body van de functie `zoek` is als volgt:

```
{   int i;
    for (i=0; i<aantal; i++)
        if (code==tabel[i])
            return i;
    return -1;
}
```

Met een `for`-opdracht wordt één voor één van elk tabelelement geprobeerd of het misschien de gezochte code is. Zo ja, dan wordt direct de bijbehorende index geretourneerd. Het uitvoeren van de `return`-opdract heeft naast het teruggeven van een waarde tot gevolg dat de verwerking van de functie wordt stopgezet. Dat komt goed uit, want als de gewenste index gevonden is, hoeven de overige elementen niet meer doorzocht te worden.

Overleeft de functie echter het hele zoekproces, dan komt de code blijkbaar nergens voor in de tabel. In dat geval zouden we de waarde -1 retourneren.

In het hoofdprogramma kan nu lekker overzichtelijk de functie `zoek` worden aangeroepen:

```
i = zoek(code, codes, aantal);
if (i>=0) cout << bedragen[i] << endl;
else      cout << "bedrag onbekend." << endl;
```

9.3 Programma-parameters

In programma's die geïnteresseerd zijn in de commandoregel waarmee ze worden gestart, moet de functie `main` op een andere manier worden gedefinieerd dan we tot nu toe gedaan hebben. De functie krijgt namelijk twee parameters:

```
int main(int aantalNamen, char* namen[] )
```

De tweede parameter is een *array met teksten*, waar we zelf een naam voor mogen kiezen (hier: `namen`). Het type `char*` geeft aan dat het om teksten gaat, en de haken `[]` dat een een array ervan betreft. Het aantal elementen van die array dat gebruikt mag worden wordt doorgegeven via de eerste parameter. Het is dezelfde methode die we bij de functie `zoek` hebben gebruikt: een array-parameter, en een aparte int die de lengte ervan aangeeft.

In de tekst `namen[0]` is de naam van het programma beschikbaar. Eventueel op de commandoregel gespecificeerde namen zijn via de volgende elementen van de array te inspecteren: `namen[1]` is de eerste extra naam op de commandoregel, `namen[2]` de tweede, enzovoort. De `int`-parameter geeft het aantal ingevulde array-elementen. Bij een programma zonder extra informatie op de commandoregel is dat 1, en als er één filenaam gespecificeerd is de waarde 2.

Het programma kassa veronderstelt dat er precies één naam wordt doorgegeven via de commandoregel. Voor de zekerheid wordt dit gecontroleerd. Indien het niet het geval is, wordt een speciale foutmelding gegeven, en het programma afgebroken:

```
if (aantalNamen!=2)
{
    cout << "Gebruik dit programma met 1 parameter" << endl;
    return 1;
}
```

Afbreken van het programma kan gedaan worden door de functie `main` te beëindigen. Net als bij andere functies kan dat met een `return`-opdracht; de functie `main` is tenslotte gewoon een functie.

Het resultaattype van de functie `main` is `int`. De functie kan dus een getalswaarde retourneren aan wie hem heeft aangeroepen, dat is aan het operating systeem. Gebruikelijk is het om bij abnormale beëindiging een waarde ongelijk aan nul te retourneren. Voor de volledigheid kan aan het normale eind van het programma de waarde 0 teruggegeven worden.

Als het programma de controle op de waarde van `aantalNamen` overleeft, kan het er verder op vertrouwen dat de tekst `namen[1]` de op de commandoregel gespecificeerde filenaam bevat.

9.4 Files als streams

De inhoud van een file kan, net als de 'inhoud' van het toetsenbord, aangesproken worden als stream. Een stream kan, net als variabelen, aangeduid worden met een naam. Willen we een nieuwe stream gebruiken, dan zal die naam gedeclareerd moeten worden. Net als andere declaraties bestaat deze declaratie uit de naam van een type en een zelfbedachte naam. Het type van streams die een invoerfile toegankelijk maken is `ifstream`. (Dat 'if' in de naam heeft niets te maken met `if`-opdrachten; het is een wat onduidelijke afkorting van 'input file'.)

De declaratie van nieuwe streams wijkt in zoverre af van declaraties van integers en dergelijke, dat er bij de declaratie ook nog een parameter meegegeven moet worden. De declaratie heeft dus ook iets van een functie-aanroep. Het blijft echter wel degelijk een declaratie; functie-aanroepen kan je tenslotte ook in expressies gebruiken, en dat kan niet met declaraties zoals deze. De declaratie ziet er al met al zo uit:

```
ifstream invoer( namen[1] );
```

De parameter bij de declaratie is een tekst, waarin de naam van de file staat die aan de stream `invoer` gekoppeld moet worden.

Na deze declaratie kan `invoer` als een gewone stream behandeld worden. De regels van de file worden dus als volgt gelezen en in de arrays opgeslagen:

```
aantal = 0;
while( invoer >> code >> bedrag )
{
    codes[aantal]    = code;
    bedragen[aantal] = bedrag;
    aantal++;
}
```

Anders dan de type-namen `int`, `double` en `char` is de type-naam `ifstream` geen gereserveerd woord in de taal. Het is namelijk mogelijk om zelf nieuwe typen te definiëren. Later zullen we dat ook zelf gaan doen, maar voorlopig maken we dankbaar gebruik van de type-definitie voor `ifstream` die in een programmabibliotheek wordt gegeven. Deze bibliotheek heet `<fstream.h>`; aan het begin van het programma staat dan ook de regel

```
#include <fstream.h>
```

De bibliotheek `<fstream.h>` omvat ook de inhoud van de bibliotheek `<iostream.h>`, die dus niet meer apart vermeld hoeft te worden.

Opgaven

1. Wat kan er fout gaan met de arrays `codes` en `bedragen`? Pas het programma aan zodat de ongewenste situatie in ieder geval wordt gemeld aan de gebruiker.
2. De variabelen `code` en `bedrag` zijn in feite overbodig bij het inlezen van de file. Hoe kan het eenvoudiger?

Hoofdstuk 10

Woordenboek, *of:* strings

In dit hoofdstuk bekijken we een programma waarin tekst gemanipuleerd wordt. We gaan nader in op het type `char*` waarmee teksten gedeclareerd kunnen worden. We zullen teksten uit files inlezen en in arrays opslaan, en we bespreken functies waarmee teksten gekopieerd en vergeleken kunnen worden.

10.1 Het programma

Het programma in dit hoofdstuk heet `woordenboek`, omdat het gebruikt kan worden als woordenboek. Dat wil zeggen: de gebruiker kan woorden intikken, en in een woordenlijst zoekt het programma de vertaling ervan (indien aanwezig) op. In listing 10 staat de tekst van het programma.

Het gebruik van het programma lijkt erg op dat van het programma `kassa`. Het verschil is dat de file nu geen streepjescode-getallen met bijbehorende bedragen bevat, maar tweetallen woorden. De gebruiker tikt nu ook geen getallen meer in, maar woorden.

Dit programma zou zelfs helemaal in de plaats van `kassa` gebruikt kunnen worden. Een getal is tenslotte te beschouwen als een bijzonder soort woord. Het programma is dus universeler bruikbaar dan `kassa`. Mits voorzien van een geschikte lijst kan het ook gebruikt worden als telefoonboek. In feite kan het gebruikt worden voor elke situatie waarin woorden (of namen, getallen e.d.) geassocieerd worden met andere woorden.

10.2 Het type van teksten

Omdat in dit programma teksten een belangrijke rol spelen, is het nodig dat we wat dieper ingaan op het type van teksten.

Tot nu toe hebben we teksten op twee manieren gebruikt. Ten eerste de vaste teksten (genoteerd tussen aanhalingstekens) zoals `"hallo"` in opdrachten als

```
cout << "hallo" ;
coupure( centen, 1000, "tientje" );
```

Ten tweede werden teksten als parameter meegegeven, zoals in de functie `coupure`. Dan moest deze gedeclareerd worden met het type `char*`, zoals in

```
void coupure( int& bedrag, int waarde, char* naam )
```

De aldus gedeclareerde `naam` kon overal gebruikt worden in plaats van vaste teksten, bijvoorbeeld achter `cout<<`.

We hebben echter nog nooit teksten *ingelezen*. Om een tekst in te lezen, is het niet voldoende om er een te declareren met het type `char*`, op dezelfde manier als je een integer declareert om te kunnen inlezen. De reden hiervoor is dat het type `char*` een *verwijzing* is naar een reeds *bestaande* tekst. Een nieuw te creëren tekst moet echter anders gedeclareerd worden.

Een tekst is opgebouwd uit meerdere lettertekens. Voor het opslaan van lettertekens is er het type `char`. Om ruimte te creëren voor een nieuwe tekst declareer je een *array* met elementen van type `char`. Een tekst van maximaal 100 lettertekens declareer je bijvoorbeeld met

```
char tekst[100];
```

Zo'n tekst zou je kunnen inlezen en vervolgens op het beeldscherm afbeelden met

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

char *kopie(char *orig)
{   int lengte;
    char *res;

    lengte = strlen(orig);
    res = new char[lengte+1];
    strcpy(res, orig);
    return res;
}

int zoek( char *woord, char *tabel[], int aantal )
{   int i;
    for (i=0; i<aantal; i++)
        if (strcmp(woord, tabel[i])==0)
            return i;
    return -1;
}

int main(int aantalNamen, char *namen[] )
{   char *woorden[1000], *vertalingen[1000];
    char woord[100], vertaling[100];
    int i, aantal;

    if (aantalNamen!=2)
    {   cout << "Gebruik dit programma met 1 parameter" << endl;
        return 1;
    }

    ifstream invoer( namen[1] );
    cout << "Bezig met inlezen woordenboek..." << endl;
    aantal = 0;
    while( invoer >> woord >> vertaling )
    {   woorden[aantal]     = kopie( woord );
        vertalingen[aantal] = kopie( vertaling );
        aantal++;
    }

    cout << "Gereed om woorden te vertalen." << endl;
    while ( cin >> woord )
    {    i = zoek(woord, woorden, aantal);
         if (i>=0)
             cout << vertalingen[i] << endl;
         else
             cout << "vertaling onbekend." << endl;
    }
}
```

Listing 10: woordenboek.cc

```
cin  >> tekst;
cout << tekst;
```

De tekst mag ook megegeven worden als parameter aan functies. Die parameter moet in de functie dan gedeclareerd zijn als `char*` of als `char[]` (mag allebei).

In het vorige hoofdstuk zagen we echter een probleem met arrays als parameters: de functie kan niet raden hoe lang de array is, en daarom is er een extra parameter nodig om de lengte door te geven. Voor teksten is dat wel erg omslachtig. Je zou bij elke tekst dan apart zijn lengte moeten bewaren, en iedere keer ook meegeven als je een tekst als parameter meegeeft. Daarom is er voor teksten een andere methode bedacht om de lengte door te geven: het einde van de tekst wordt met een speciaal symbool gemarkeerd. Functies met een tekst als parameter mogen alle opeenvolgende symbolen daarvan bekijken, tot ze dit speciale symbool tegenkomen. Daarachter is het bestaan van de array niet meer gegarandeerd. Doordat het speciale eind-symbool ook ruimte inneemt, kan in een array met 100 elementen dus in feite slechts een tekst met 99 symbolen worden opgeslagen.

Over de eindmarkering volgt in een paragraaf 12.3 nog wat achtergrondinformatie, maar voorlopig weten we genoeg om het programma `woordenboek` te begrijpen.

10.3 Functies op teksten

Het komt nogal eens voor dat je de lengte van een string wilt weten. Het is niet zo moeilijk om een functie te schrijven die de lengte van een string bepaalt: deze functie hoeft alleen maar achtereenvolgende elementen van de tekst te inspecteren, tot hij de eindmarkering tegenkomt, en ondertussen steeds een tellertje ophogen. Nog gemakkelijker is om anderen dit werk te laten doen: de functie die de lengte van een tekst bepaalt is al aanwezig in een functiebibliotheek. De bibliotheek heet `<string.h>` en de functie heet `strlen` (*string* is het technische woord voor 'rijtje letters' oftewel 'tekst').

Omdat een tekst eigenlijk een array is, kunnen teksten niet zomaar met `=` gekopieerd worden. Wel is hiervoor een functie aanwezig, die het hele array elementsgewijs kopieert totdat de eindmarkering bereikt is. Deze functie staat ook in de bibliotheek `<string.h>` en heet `strcpy`. De eerste parameter is een array waarin de kopie moet komen te staan; de tweede parameter is het origineel. Zo kan met de opdracht

```
strcpy( tekst, "hallo" );
```

de tekst `"hallo"` in de array `tekst` gekopieerd worden, compleet met eindmarkering. De programmeur moet er wel op letten dat de array `tekst` genoeg ruimte heeft om het resultaat te bevatten, in dit geval minstens 6 (!) elementen.

Ook de vergelijkingsoperator `==` kan niet op teksten gebruikt worden. Maar natuurlijk is ook hiervoor weer een funcie voorhanden. De functie `strcmp` ('string compare') vergelijkt twee teksten en levert een integer op: 0 als de teksten precies gelijk zijn, een negatief getal als de eerste tekst alfabetisch kleiner is, en een positief getal als hij alfabetisch groter is.

10.4 Woordenlijsten

De geheugenruimte voor één woord met maximaal 99 lettertekens kun je declararen met

```
char tekst[100];
```

Wil je een lijst met 1000 van dit soort woorden, dan kun je declareren

```
char lijst[100][1000];
```

Met deze ene declaratie wordt geheugenruimte voor 100000 lettertekens gereserveerd, gegroepeerd in 1000 teksten van 100 tekens. Zo raak je wel snel door je dure geheugen heen, zeker als je bedenkt dat een beetje woordenboek wel meer dan 1000 woorden bevat. De declaratie is dan ook wel erg royaal bemeten. Er zullen misschien een paar woorden zijn met tegen de 100 lettertekens, maar de meeste woorden zijn veel korter. We gaan het dus anders aanpakken.

In plaats van een array met 1000 elementen die ieder een array met 100 elementen zijn, maken we een array met 1000 *verwijzingen* naar bestaande teksten, die niet allemaal even lang hoeven te zijn. Zo'n verwijzing kost niet zoveel geheugen, en de teksten waar deze verwijzingen naar verwijzen gaan we precies op maat maken voor de woorden die opgeslagen moeten worden. In het programma `woordenboek` staat de volgende declaratie:

```
char *woorden[1000], *vertalingen[1000];
```

Het type `char*` is een verwijzing naar een tekst, en daar willen we er tweemaal 1000 van hebben. Merk op dat bij de tweede declaratie het woord `char` niet herhaald hoeft te worden, maar het sterretje wel!

Verder declareren we nog twee arrays, waarin één woord met zijn vertaling in ingelezen kan worden:

```
char woord[100], vertaling[100];
```

Na het creëren van een invoerstream **invoer** (net als in het programma kassa) kunnen de woorden ingelezen worden. Een (verkeerde) poging hiertoe is

```
while( invoer >> woord >> vertaling )
{
    woorden[aantal]     = woord ;
    vertalingen[aantal] = vertaling ;
    aantal++;
}
```

Dit gaat niet goed, want een verwijzing moet verwijzen naar een *bestaande* tekst. Alle verwijzingen worden nu gezet naar dezelfde array. We eindigen op deze manier dus met 1000 verwijzingen naar één array met daarin het laatste woord. De juiste manier om de array met verwijzingen te vullen is

```
while( invoer >> woord >> vertaling )
{
    woorden[aantal]     = kopie( woord );
    vertalingen[aantal] = kopie( vertaling );
    aantal++;
}
```

Rest ons nog om een functie **kopie** te schrijven die een kopie maakt van een tekst.

10.5 Dynamisch geheugen creëren

De *header* (eerste regel) van de functie kopie is als volgt:

```
char *kopie(char *orig)
```

De functie levert, gegeven een verwijzing naar een bestaande string, een verwijzing op naar een speciaal daartoe gecreëerde nieuwe string, met dezefde inhoud als het origineel. We noemen het een 'kopie' omdat als het origineel later veranderd wordt, de kopie niet meeverandert.

Het probleem bij het schrijven van de functie **kopie** is: waar halen we zo'n nieuwe string vandaan? Deze string kan om twee redenen niet gewoon gedeclareerd worden:

- Bij declaraties van arrays moet de lengte aan de compiler bekend zijn. Dat kan hier niet, want we willen de lengte juist laten afhangen van een ingelezen string.
- Declaraties die in een functie gedaan worden, zijn *lokaal* voor die functie. Als de functie is afgelopen, ben je ook je variabele kwijt.

Gelukkig is er nog een manier om geheugen te creëren: *dynamische creatie*. Het dynamische hiervan is, dat er pas tijdens het uitvoeren van het programma (en niet al tijdens het compileren) toe besloten wordt.

Voor dit doel is er een apart soort expressie voorzien, bestaande uit het (voor dit doel) gereserveerde woord **new** en een type. Als dit type een array is, mag de lengte daarvan een dynamisch bepaalde expressie zijn. De waarde van de expressie **new** t is een verwijzing naar een nieuwe variabele van het type t. Deze variabele is naamloos (er is nergens een naam voor gedeclareerd), maar dat is niet erg, want we hebben er een verwijzing naar. Deze verwijzing laten we als resultaat opleveren door de functie **kopie**, zodat hij uiteindelijk in de grote array met verwijzingen **woorden** terecht komt.

De definitie van de functie **kopie** verloopt nu als volgt. We hebben twee lokale variabelen nodig, waarin we tijdelijk de lengte van de string en een verwijzing naar de nieuwe geheugenruimte in kunnen opslaan:

```
int lengte;
char *res;
```

Allereerst bepalen we de lengte van het origineel:

```
lengte = strlen(orig);
```

We hebben een array nodig die één element langer is dan deze lengte (voor de eindmarkering). Dat gebeurt met de speciale **new**-constructie:

```
res = new char[lengte+1];
```

In de nieuwe ruimte zetten we een kopie van de originele tekst (past precies):

```
strcpy( res, orig );
```

De verwijzing naar de aldus gecreëerde en van waarde voorziene tekst is het resultaat van de functie `kopie`:

```
return res;
```

Lees nu listing 10 nog eens in zijn geheel door en controleer of je elk sterretje begrijpt.

Opgaven

1. Geef een definitie van de functies `strlen` en `strcpy` zonder de bibliotheek `<string.h>` te gebruiken.

Hoofdstuk 11

Snel woordenboek, *of:* binary search

In dit hoofdstuk passen we het programma uit het vorige hoofdstuk aan zodat het zoeken veel sneller verloopt. Het programma zal aannemen dat de woordenlijst die het inleest alfabetisch geordend is. Het zoeken kan dan gebeuren volgens de zogeheten *binary search* methode.

11.1 Het programma

Het programma in dit hoofdstuk heet snelwdb, omdat het gebruikt kan worden als woordenboek waarin *snel* in opgezocht kan worden. Het grootste deel van het programma is hetzelfde als het programma woordenboek in het vorige hoofdstuk. Alleen de functie `zoek` wordt vervangen. In listing 11 staat de nieuwe tekst van die functie.

In de oude functie `zoek` werd de woordenlijst vanaf het begin doorzocht. Als het woord niet in

```
int zoek( char *woord, char *tabel[], int aantal )
{   int vanaf, tot, mid, cmp;

    vanaf = 0;
    tot   = aantal;

    while (tot!=vanaf)
    {
        mid = (vanaf+tot) / 2;
        cmp = strcmp(woord, tabel[mid]);

        if (cmp<0)
        {   // woord is kleiner dan middelste tabelelement
            // dus zoek in eerste helft van de tabel

            tot = mid;
        }
        else if (cmp>0)
        {
            // woord is groter dan middelste tabelelement
            // dus zoek in tweede helft tabel

            vanaf = mid+1;
        }
        else return mid;
    }

    return -1;
}
```

Listing 11: snelwdb.cc

de woordenlijst voorkomt bekijkt de functie alle elementen voordat het tot die conclusie komt. Als het woord ergens in het begin van de lijst staat heb je hem snel gevonden, maar je hebt ook wel eens de pech dat een woord helemaal aan het eind staat. Gemiddeld zal je ongeveer de halve woordenlijst moeten doorzoeken. Als de woordenlijst lang is, gaat dat toch de nodige tijd kosten. Dit is dan ook niet de manier waarop je zelf een woord in een woordenboek zou opzoeken. Omdat je weet dat een woordenboek alfabetisch geordend is, weet je ongeveer waar je moet zoeken. Je bekijkt eens een woord in het boek, en als het gezochte woord verderop in het alfabet staat blader je een eind vooruit, en vooral niet achteruit.

11.2 De halveringsmethode

De functie `zoek` krijgt als parameter het aantal woorden in de woordenlijst mee. De gezochte index moet liggen in het interval vanaf 0 tot `aantal`. Het woord 'tot' moet je hier lezen als 'tot en zonder', omdat arrays bij 0 beginnen te tellen. Als wiskundig interval genoteerd: het zoekinterval is $[0, n\rangle$. In het programma wordt een variabele i gedeclareerd, die aan het begin gelijk aan 0 gemaakt wordt. Het zoekinterval aan het begin is dus $[i, n\rangle$. Bij elke stap van de `while`-opdracht wordt dit zoekinterval één element kleiner gemaakt. Het te doorzoeken interval kun je steeds blijven aangeven met $[i, n\rangle$, maar de waarde van i wordt steeds groter. Op het moment dat i gelijk aan n is geworden, is het zoekinterval dus $[n, n\rangle$. Dit is een leeg interval, en we kunnen veilig aannemen dat de gezochte waarde niet voorkomt.

Met de 'halveringsmethode' gaan we proberen het zoekinterval met wat grotere stappen te verkleinen. Stel dat het zoekinterval op een gegeven moment $[k, n\rangle$ is. Als het zoekinterval leeg is, dan kunnen we stoppen met zoeken: het gezochte element komt dan niet voor in de array, en we kunnen -1 opleveren om dit te melden. Als het zoekinterval uit meer dan één element bestaat, kijken we ongeveer halverwege het interval, zeg bij index m. Nu zijn er drie mogelijkheden:

- Op positie m staat het gezochte woord. Mooi, dan kunnen we stoppen met zoeken en de index m opleveren.
- Op positie m staat een *eerder* woord dan het gezochte.
  Het zoekinterval wordt nu gereduceerd tot $[m+1, n\rangle$.
- Op positie m staat een *later* woord dan het gezochte.
  Het zoekinterval wordt nu gereduceerd tot $[k, m\rangle$.

Als we er voor zorgen dat m halverwege het zoekinterval ligt, dan wordt bij elke stap het zoekinterval gehalveerd. Daarom heet deze zoekmethode de *halveringsmethode* of *binary search*.

In het programma wordt deze methode gevolgd. We gebruiken twee variabelen, die de index van het begin (inclusief) en het eind (exclusief) van het zoekinterval aangeven. Een derde variabele gebruiken we om het midden te bepalen:

```
int vanaf, tot, mid;
```

Om te beginnen zetten we de variabelen zo, dat ze het hele zoekinterval aanduiden:

```
vanaf = 0;
tot   = aantal;
```

Als `tot` gelijk is aan `vanaf`, is het zoekinterval leeg (het is immers 'tot en zonder'). Dit blijkt uit de voorwaarde van de `while`-opdracht:

```
while (tot != vanaf)
```

Het midden van het interval kunnen we berekenen als het gemiddelde van `tot` en `vanaf`:

```
{
    mid = (tot+vanaf) / 2;
```

Als het interval een even aantal elementen heeft, is er natuurlijk geen index die precies in het midden ligt. Doordat de deling naar beneden afrondt krijgen we echter wel een index die vlakbij het midden ligt.

De waarde die in de array de `mid`-index heeft, wordt vergeleken met de gezochte waarde. Omdat het in dit geval om een tabel met teksten gaat, moet dat met de functie `strcmp`. In het geval van een tabel met getallen (zoals in het programma `kassa`) kan de operator `<` gebruikt worden. Afhankelijk van de uitkomst van de vergelijking wordt de midden-index opgeleverd, dan wel het zoekinterval aan de voorkant of de achterkant aangepast.

Na afloop van het zoekproces geldt niet meer `tot!=vanaf`, anders was het zoekproces nog doorgegaan. Er geldt dus `tot==vanaf`. Het zoekinterval is dus leeg, en de waarde -1 wordt opgeleverd om aan te geven dat het woord niet in de tabel aanwezig is.

11.3 Analyse van de zoektijd

Eerder in dit hoofdstuk zagen we al dat je bij het lineair doorzoeken van een tabel gemiddeld de halve tabel moet bekijken. Als de tabel dus n elementen heeft, kost het zoeken $\frac{1}{2}n$ stappen.

Hoeveel stappen kost het doorzoeken met de halveringsmethode? Stel een tabel heeft 1000 elementen. Na één vergelijking is het zoekinterval al gehalveerd tot 500 elementen (dat kunnen de eerste 500 zijn of de tweede 500, maar in ieder geval zijn het er 500). Daarna wordt het interval in elke stap weer gehalveerd: 500 - 250 - 125 - 63 - 32 - 16 - 8 - 4 - 2 - 1. Dat zijn dus 10 stappen. Met de extra vergelijking op het eind is het totaal 11 stappen. Als je de gezochte waarde onderweg toevallig tegenkomt is het aantal stappen nog minder, maar dat zal in de praktijk niet zo vaak voorkomen. Bij de analyse houden we er dus maar geen rekening mee, dan kan het altijd nog meevallen.

Het aantal keren dat je een getal n kunt halveren wordt wiskundig beschreven met behulp van de logaritme-functie: ${}^2\log n$. Het aantal stappen dat nodig is bij het zoeken met de halveringsmethode is dus ${}^2\log n+1$. De volgende tabel geeft een indruk van het aantal stappen dat de 2 zoekmethodes nodig hebben bij verschillende waardes van n:

omvang n	lineair $\frac{1}{2}n$	binair ${}^2\log n + 1$
2	1	2
10	5	5
100	50	8
1000	500	11
10.000	5000	15
100.000	50.000	18
1.000.000	500.000	21

Vooral bij grote tabellen is de winst die je boekt door binair te zoeken in plaats van lineair dramatisch. Helemaal eerlijk is de vergelijking niet, omdat de 'stappen' bij de binaire zoekmethode wat groter zijn dan bij de lineaire: behalve een `strcmp` moet ook het een middenpositie worden uitgerekend en een extra test worden gedaan. Maar al zou je de stappen van de binaire zoekmethode voor 3 tellen, dan nog is de binaire methode al voor $n > 15$ sneller dan de lineaire.

Opgaven

1. Stel dat een aanroep van `strcmp` 1 microseconde duurt (een miljoenste seconde). In een woordenboek staan zo'n 50000 woorden. Maak een ruwe schatting hoe lang het op spelling controleren van een boek van 100 pagina's kost zonder en met de halveringsmethode.

2. Controleer dat de berekening

```
mid = (tot+vanaf) / 2;
```

voor intervallen met een oneven lengte inderdaad het midden bepaalt. Welk element krijg je bij intervallen met even lengte?

3. Wat gebeurt er als in het programma de opdracht `vanaf = mid+1` vervangen wordt door `vanaf = mid` ?

4. Wat gebeurt er als je met *binary search* zoekt in een tabel die niet alfabetisch geordend is?

Hoofdstuk 12

Letterfrequentie, *of:* classes

In dit hoofdstuk bekijken we een programma waarin de frequentieverdeling van letters op de invoerstream wordt bepaald. In het programma zullen we zien hoe je in plaats van getallen of woorden losse lettertekens kunt inlezen, en hoe je een array met lettertekens kunt indiceren. Het belangrijkste verschil met het cijferfrequentie-programma in een eerder hoofdstuk is echter dat we de frequentietabel ditmaal als een 'object' declareren, compleet met daarop werkende 'methodes'. Echt nodig is dat niet, maar door deze manier van declareren kunnen we het programma moeiteloos uitbreiden tot een veel ingewikkelder programma.

12.1 Het programma

Het programma in dit hoofdstuk heet `freqtab`, omdat het een frequentietabel van letters berekent, maar vooral ook omdat het object 'frequentietabel' er een centrale rol in speelt. In listing 12 staat de tekst van het programma.

Dit programma meldt, in tegenstelling tot eerdere programma's, niet dat het invoer verwacht. Het gaat gewoon stilletjes op invoer zitten wachten. De gebruiker mag vervolgens naar believen letters en andere symbolen intikken, afgesloten met control-D. Het programma zal dan een frequentietabel afdrukken van de ingetikte tekens. Een voorbeeld van een sessie:

```
aabaac
ar+ b
 : 1x (7%)
+: 1x (7%)
a: 5x (38%)
b: 2x (15%)
c: 1x (7%)
r: 1x (7%)
```

Alle symbolen die de gebruikter intikt worden meegenomen in de tabel, dus ook de spatie en het plus-teken. Als extra'tje (ten opzicht van het programma `frequ`) geeft `freqtab` niet alleen de absolute frequentie, maar ook de relatieve frequentie van de symbolen.

12.2 Letters inlezen

In eerdere hoofdstukken hebben we getallen en woorden ingelezen uit invoerstreams (het toetsenbord of een file). Voor getallen ging dat met

```
int getal;
cin >> getal;
```

en voor teksten met

```
char tekst[100];
cin >> tekst;
```

Bij het inlezen op deze manier gelden spaties en regelovergangen als scheiding tussen de in te lezen waarden. Bij getallen is dat nogal logisch: de invoer `31 17` telt als twee getallen: eenendertig en zeventien. Bij teksten werkt het op dezelfde manier: als de invoer de symbolen `aap monkey` bevat, dan zal het inlezen van één tekst de string `"aap"` opleveren, en het inlezen van de volgende tekst de string `"monkey"`.

```
#include <iostream.h>
const char MIN=32, MAX=127;

class Freqtab
{
    int aantal;
    int turf[MAX];
public:
         Freqtab();
    void verhoog( char);
    void verslag( char);
};

Freqtab::Freqtab()
{   char ch;
    aantal = 0;
    for (ch=0; ch<MAX; ch++)
        turf[ch] = 0;
}

void Freqtab::verhoog(char ch)
{   aantal++;
    turf[ch]++;
}

void Freqtab::verslag(char ch)
{   if (turf[ch]>0)
       cout << ch << ": " << turf[ch] << "x"
            << " (" << 100*turf[ch]/aantal << "%)"
            << endl;
}

int main()
{   char letter;
    Freqtab freq;

    while ( cin.get(letter) )
        freq.verhoog(letter);

    for (letter=MIN; letter<MAX; letter++)
        freq.verslag( letter );
}
```

Listing 12: freqtab.cc

Het is ook mogelijk om losse letters in te lezen. Die kun je declareren met als type `char`, en op dezelfde manier inlezen als andere types:

```
char ch;
cin >> ch;
```

Ook hierbij tellen spaties en regelovergangen niet mee. Met de opdracht `cin>>ch` lees je dus de eerstvolgende niet-spatie in. Dat is niet altijd de bedoeling; in het programma freqtab willen we bijvoorbeeld ook de spaties meenemen in de frequentieanalyse.

Gelukkig is er een tweede manier om letters in te lezen, waarbij de spaties wèl als apart symbool gelden. In het programma wordt die manier gebruikt:

```
while ( cin.get(letter) )
```

De functie `get` is een *methode* van behorend bij streams. Ter herinnering: een methode is een functie die hoort bij een bepaald object. De functie kan worden aangeroepen met de constructie '*objectnaam.functienaam*'. De methode `get` heeft als parameter een veranderbare `char`. Het neveneffect van de methode is dat de meegegeven variabele gelijk wordt aan het volgende symbool op de stream, terwijl het resultaat van de functie gebruikt kan worden als waarheidswaarde die aangeeft of er eigenlijk wel zo'n symbool was.

12.3 Codering van letters

Constante lettertekens kun je in een programma aanduiden door het symbool tussen `'`-tekens te zetten. Dat zijn eigenlijk apostrofs, maar in dit verband worden ze meestal 'enkele aanhalingstekens' genoemd. Let op het verschil met de echte (dubbele) aanhalingstekens, die gebruikt worden om een constante tekst in het programma op te schrijven. Je kunt op deze manier individuele letters van een string veranderen. Bijvoorbeeld met:

```
char tekst[10];
strcpy( tekst, "hallo" );
tekst[1] = 'e';
cout << tekst;
```

wordt de tekst `hello` getoond.

Er is een tweede manier om symbolen (letters, cijfers, leestekens enz.) aan te duiden. Elk symbool heeft een codenummer. Het symbool `'A'` heeft bijvoorbeeld code 65, en het symbool `'B'` code 66. Kleine letters hebben aparte codes: symbool `'a'` heeft code 97. Alle intikbare symbolen hebben een code: de code van `'*'` is 42, en van `'!'` 33. De cijfersymbolen hebben codes tussen 48 (voor de `'0'`) en 57 (voor de `'9'`). De spatie heeft code 32. Alle codes liggen tussen 32 en 126.

De codes 0 t/m 31 hebben een speciale betekenis. Bijvoorbeeld: code 10 staat voor een regelovergang, code 7 voor 'piep', en code 0 is de einde-string markering. Let op het verschil tussen het symbool met code 0, en het symbool `'0'` (dat code 48 heeft).

In plaats van een symbool tussen enkele aanhalingstekens, mag je ook het codenummer gebruiken (maar dan natuurlijk zonder aanhalingstekens). Dat wordt eigenlijk vooral voor de codes 0 t/m 31 gebruikt, omdat het in andere gevallen duidelijker is om het symbool zelf in het programma te zetten. Dan heb je tenminste geen codetabel nodig bij het programmeren.

Door de code 0 in een tekst te zetten, kun je zelf einde-string markeringen plaatsen. Het volgende programmafragment kort de string `"hallo"` in tot de string `"ha"`:

```
char tekst[10];
strcpy( tekst, "hallo" );
tekst[2] = 0;
cout << tekst;
```

In het programma `freqtab` gebruiken we een speciale declaratie om twee belangrijke constanten een naam te geven: de kleinste en (1 meer dan de) grootste code voor intikbare symbolen geven we de namen `MIN` en `MAX`:

```
const char MIN=32, MAX=127;
```

In zo'n constante-declaratie staat vóór het type het gereserveerde woord `const`, en na de gedeclareerde naam een =-teken en de waarde. Een naam die als `const` gedeclareerd wordt mag in het programma niet veranderd worden.

Waarom zou je zo'n constante declareren, en niet gewoon overal in het programma het getal 32 resp. 127 zetten? Daar zijn twee redenen voor:

- *duidelijkheid*: als je in een programma een getal als 127 ziet staan, moet je je steeds afvragen wat dat getal ook al weer te betekenen heeft. Zie je de naam `MAX` staan dan is dat sneller duidelijk.
- *veranderbaarheid*: als je later bedenkt dat je alleen maar geïnteresseerd bent in hoofdletters, dan kun je de constanten alsnog veranderen in 65 en 90 (en het programma opnieuw compileren). Dat hoeft dan maar op één plaats in het programma te gebeuren, en niet op drie verschillende plaatsen. Met de constante-declaratie kun je er niet per ongeluk eentje vergeten, of per ongeluk een 32 die niets met lettercodes te maken heeft ook veranderen.

Overal waar in het programma alle symbolen in volgorde doorlopen moeten worden (bijvoorbeeld bij het afdrukken van de frequentietabel) staat een opdracht die begint met

```
for (ch=MIN; ch<MAX; ch++)
```

Codegetallen mogen nooit groter zijn dan 127. Doe je dat toch, dan is het gevolg onvoorspelbaar.

12.4 Objecten

Omdat we in dit programma behalve de absolute frequenties ook de relatieve frequenties willen rapporteren, is er een extra variabele nodig, waarin het totale aantal symbolen wordt geteld. De relatieve frequentie wordt immers berekend door de absolute frequentie door het totale aantal te delen.

In het programma `frequ` hadden we functies `verslag` en `histo` gemaakt, die als parameter een index van de turftabel en het bijbehorende aantal meekregen. Nu moeten deze functies als extra parameter ook nog het totale aantal meekrijgen, zodat ze de relatieve frequentie kunnen berekenen.

Dit is allemaal wel een heel gedoe: wil je je programma een beetje aanpassen, en dan moeten gelijk allerlei functies extra parameters krijgen. Misschien vind je het niet heel schokkend, maar bedenk dat bij grotere programma's het aantal functies dat met een bepaalde tabel te maken heeft in de tientallen kan lopen, die bovendien nog door verschillende programmeurs geschreven kunnen zijn. Eerder zagen we dat in teams van programmeurs ieder relatief ongestoord zijn gang kan gaan, zolang het aantal parameters van functies maar niet veranderd wordt. En om nou voor elk wissewasje de onderhandelingen daarover te heropenen wordt al gauw vervelend.

Daarom gaan we het in dit programma anders aanpakken. Alle informatie die bij een frequentietabel hoort (dat zijn de eigenlijke turftabel en de variabele die het totale aantal bevat) groeperen we in een *object*. Van de functies die met een frequentietabel te maken hebben maken we *methodes* van dat object. Van dit soort methodes maken we er drie: de functie `verslag`, een nieuwe functie om een turfje in de tabel te zetten, en een functie waarmee de tabel van zinvolle beginwaardes wordt voorzien (allemaal nullen).

Het handige van objecten is dat je ze net zo kunt declareren als gewone variabelen. In het hoofdprogramma komen nog maar twee variabelen te staan:

```
char letter;
Freqtab freq;
```

De frequentietabel `freq` is van het zelfbedachte type `Freqtab`. We gaan ervoor zorgen dat objecten van dat type de methodes `verhoog` en `verslag` tot hun beschikking hebben. Het hoofdprogramma wordt daarmee lekker overzichtelijk:

```
while ( cin.get(letter) )
    freq.verhoog(letter);
for (letter=MIN; letter<MAX; letter++)
    freq.verslag( letter );
```

Geen gepruts met arrays meer in het hoofdprogramma: die details zijn weggewerkt in de methodes. Het hoofdprogramma geeft echt de essentie van het turfproces weer: eerst letters lezen en turfjes zetten, en daarna voor elke letter graag een verslagje.

12.5 Classes

Hoe krijgen we het voor elkaar dat we frequentietabel-objecten kunnen declareren alsof het gewone variabelen zijn? Daartoe moeten we eerst het type `Freqtab` definiëren. Het type van een object wordt een *class* genoemd. We geven daarom de volgende *class-definitie*:

```
class Freqtab
{
    int aantal;
    int turf[MAX];
public:
           Freqtab();
    void verhoog( char );
    void verslag( char );
};
```

In een class-definitie worden de variabelen opgesomd die samen het object vormen. Van dit setje variabelen krijg je dus voor elk object dat je declareert een exemplaar tot je beschikking. Achter het woord `public:` worden de *methoden* opgesomd die op objecten van deze klasse werken. Van elke methode wordt de header van de functiedefinitie gegeven, waarbij de namen van de parameters achterwege kunnen blijven. Eén methode heeft dezelfde naam als de class; deze methode heeft geen resultaat-type (zelfs niet `void`). Puntkomma's sluiten de headers af, en ook na de gehele class-definitie moet een puntkomma staan (dat is eigenlijk een beetje inconsequent, want achter functiedefinities staat geen puntkomma).

De drie methodes die in de class-definitie werden aangekondigd, moeten apart gedefinieerd worden. Een methode-definitie lijkt sterk op een gewone functie-definitie, maar er zijn twee verschillen:

- in de header moet worden aangegeven bij welke class de methode hoort;
- behalve van de lokale variabelen en parameters, mag ook gebruik worden gemaakt van de variabelen die in de class-definitie werden opgesomd.

Een eerste voorbeeld is de definitie van de methode `verhoog`:

```
void Freqtab::verhoog(char ch)
{
    aantal++;
    turf[ch]++;
}
```

Tussen het resultaattype en de naam staat de klasse waar deze methode bij hoort (daaraan kan de compiler zien dat het hier om een methode-definitie gaat, en niet om een gewone functie-definitie). In de definitie wordt gebruik gemaakt van `ch`, wat een parameter van de methode is, maar ook van `aantal` en `turf`, die bij het object horen waarbij de methode wordt aangeroepen. Met deze methode wordt simpelweg een turfje gezet op de gewenste plaats, en wordt het totale aantal opgehoogd. Het object waarvan de turftabel gebruikt moet worden, wordt bij de aanroep gespecificeerd:

```
freq.verhoog(letter);
```

De definitie van de methode `verslag` verloopt op een vergelijkbare manier; zie listing 12.

De derde methode die in de class-definitie wordt genoemd is een speciale: de zogenaamde *constructor*-methode. Deze methode heeft dezelfde naam als de klasse, en hij heeft geen resultaattype. De definitie lijkt desalniettemin sterk op een gewone methode-definitie:

```
Freqtab::Freqtab()
{   char ch;
    aantal = 0;
    for (ch=0; ch<MAX; ch++)
        turf[ch] = 0;
}
```

In deze methode worden de variabelen van het object van een zinvolle beginwaarde voorzien. In dit geval worden alle tellertjes in de turftabel, en ook de totaalteller `aantal` op nul gezet.

Het bijzondere van deze methode is dat hij automatisch wordt aangeroepen bij declaratie van een object. Op deze manier krijgt bij de declaratie

```
Freqtab freq;
```

het object `freq` automatisch een goede beginwaarde. Je kunt je waarschijnlijk wel voorstellen dat het hoogst ongebruikelijk is om in zo'n initialisatie-methode de `cout`-stream te gaan beschrijven. Niettemin kan het instructief zijn om dat voor jezelf toch eens te doen, om te zien wanneer de initialisatie nu eigenlijk precies plaatsvindt.

Opgaven

1. Schrijf een programma dat het aantal regels van een te specificeren file telt. (Hint: zoek naar de symbolen in de file met het codegetal dat 'regelovergang' betekent).

2. Definieer een class `String` met als variabele een `char*`, en methodes voor het bepalen van de lengte en het omzetten van kleine letters naar hoofdletters. De initialisatie-methode krijgt als parameter een `char*` die aangeeft van welke tekst het nieuwe object een kopie moet gaan bevatten. (Hint: voor de omzet-methode kun je de symbolen van de string beschouwen als getallen, waardoor je waardes bij hun codenummers kunt optellen).

Hoofdstuk 13

Taalherkenning, *of:* de grote oogst

In dit hoofdstuk bekijken we een programma waarin allerlei dingen uit voorgaande hoofdstukken samenkomen. We maken een array van classes (!), waarvan ook nog eens de lengte dynamisch bepaald wordt. Aan de class wordt een methode toegevoegd, waarmee objecten zichzelf kunnen inlezen vanaf een stream of vanaf een file. Verder schrijven we een functie met objecten als parameter. Kortom, dit programma is de oogst van het voorafgaande!

13.1 Het programma

Het programma in dit hoofdstuk heet `zoektaal`, omdat het gebruikt kan worden om de taal te bepalen waarin een bepaalde tekst geschreven is. In listing 13 staat de definitie van de class en de methoden daarin die in het programma gebruikt worden; in listing 14 staan de andere functiedefinities.

De frequentieverdeling van de letters in een tekst is karakteristiek voor de taal waarin de tekst is opgesteld. Zo komen er bijvoorbeeld in het Engels veel meer letters 'y' voor dan in het Nederlands, terwijl het Nederlands weer veel vaker de letter 'j' gebruikt. Omdat we inmiddels vrij goed zijn geworden in het opstellen van frequentietabellen, gaan we deze kennis toepassen in een programma dat geheel automatisch de taal van een tekst bepaald.

Natuurlijk kan zo'n programma nooit voor 100% werken. Speciaal geconstrueerde teksten, waarin bijvoorbeeld veel zeeëendeëieren en zomerpostzeeëgels worden opgevoerd worden door het programma wellicht fout geklassificeerd. Maar voor 'gewone' teksten zal het programma vrij aardig kunnen werken.

Eerst moet het programma 'leren' over de frequentieverdelingen in verschillende talen. Daartoe geven we het programma op de commandoregel de namen van files die een 'typische' tekst bevatten in een aantal verschillende talen. Vervolgens zal het programma invoer van het toetsenbord accepteren. Na beëindiging van de stream vergelijkt het programma de frequentiekarakteristiek van de tekst met die van alle voorbeeldteksten. De best passende frequentietabel zal dan wel bij de taal horen waarin de nieuwe tekst is geschreven.

13.2 De klasse

De class `Freqtab` uit het vorige hoofdstuk breiden we in dit programma uit met een paar nieuwe methodes. De class-definitie wordt:

```
class Freqtab
{
    int aantal;
    int turf[MAX];
    void verhoog(char);
    void verslag(char);
public:
         Freqtab();
    int  relatief(char);
    void lees(char*);
    void lees(istream &);
    void resultaat();
};
```

```
#include <fstream.h>
const char MIN=32, MAX=127;

class Freqtab
{   int aantal;
    int turf[MAX];
    void verhoog(char);
    void verslag(char);
public:
         Freqtab();
    int  relatief(char);
    void lees(char*);
    void lees(istream &);
    void resultaat();
};

Freqtab::Freqtab()
{   char ch;
    aantal = 0;
    for (ch=0; ch<MAX; ch++)
        turf[ch] = 0;
}

void Freqtab::verhoog(char ch)
{   aantal++;
    turf[ch]++;
}

int Freqtab::relatief(char ch)
{   return 100*turf[ch]/aantal;
}

void Freqtab::verslag(char ch)
{   if (turf[ch]>0)
       cout << ch << ": " << turf[ch] << "x"
            << " (" << relatief(ch) << "%)" << endl;
}

void Freqtab::lees(char *naam)
{   ifstream f( naam );
    lees(f);
}

void Freqtab::lees( istream &s )
{   char ch;
    while ( s.get(ch) )
        verhoog(ch);
}

void Freqtab::resultaat()
{   char ch;
    for (ch=MIN; ch<MAX; ch++)
        verslag( ch );
}
```

Listing 13: zoektaal1.cc

```
int verschil( Freqtab &t1, Freqtab &t2 )
{   char ch;
    int x, totaal;

    totaal = 0;
    for (ch=MIN; ch<MAX; ch++)
    {
        x = t1.relatief(ch) - t2.relatief(ch);
        if (x<0) x = -x;
        totaal += x;
    }
    return totaal;
}

int main(int aantal, char *namen[])
{
    int i, x, beste, besteverschil;
    Freqtab *f = new Freqtab[aantal];

    for (i=1; i<aantal; i++)
       f[i].lees( namen[i] );
    f[0].lees(cin);

    beste = 0;
    besteverschil = 999999;

    for (i=1; i<aantal; i++)
    {   x = verschil( f[0], f[i] );
        if (x<besteverschil)
        {
            besteverschil = x;
            beste = i;
        }
        cout << namen[i] << " heeft verschil " << x << endl;
    }
    if (beste>0)
        cout << "invoer lijkt het meeste op " << namen[beste] << endl;
}
```

Listing 14: zoektaal2.cc

De methode `relatief` bepaalt de relatieve frequentie van een bepaalde letter. Dat gebeurde in het vorige programma als deel van de methode `verslag`, maar het is eigenlijk wel zo overzichtelijk om er een aparte methode van te maken.

De methode `resultaat` is een nieuwe methode die de methode `verslag` aanroept voor alle symbolen tussen `MIN` en `MAX`, iets wat vroeger in het hoofdprogramma gebeurde. De methode `verslag` hoeft nu dus niet meer door het hoofdprogramma aangeroepen te worden; daarom is hij verhuisd van de `public` sectie naar de privé-afdeling. De methode mag nu alleen nog maar gebruikt worden door de andere methoden (door `resultaat` dus), maar niet meer door buitenstaanders zoals `main`. Een handige manier waarop objecten zich kunnen beschermen tegen ongewenste intimiteiten.

De nieuwe methode `lees` bouwt de frequentietabel op aan de hand van de invoer die hij aantreft op de `istream` die hij als parameter meekrijgt. De functie `verhoog` waarmee het eigenlijke turven gebeurt, kan daarom verhuizen naar de privé-sectie. Er is ook een versie van `lees` die niet een stream meekrijgt maar de naam van een file die gelezen moet worden.

De initialisatie-methode `Freqtab` en de methode `verhoog` zijn niet veranderd. De methode `verslag` ook niet, behalve dat de berekening van de relatieve frequentie is verplaatst naar de methode `relatief`. In de methode `lees` met een stream als parameter staat de tekst die vroeger in het hoofdprogramma stond:

```
void Freqtab::lees( istream &s )
{   char ch;
    while ( s.get(ch) )
        verhoog(ch);
}
```

Natuurlijk wordt nu in plaats van de stream `cin` de stream die de methode als parameter meekrijgt gebruikt. De methoden van een class mogen elkaar aanroepen alsof het functies zijn. De methode `verhoog`, die vanuit `main` aangeroepen moest worden als `freq.verhoog(ch)` kan nu aangeroepen worden zonder te vermelden over welk object het gaat. Als object wordt hetzelfde object gebruikt waarmee `lees` wordt aangeroepen.

De andere methode `lees` maakt een nieuw stream-object, gekoppeld aan de file waarvan hij de naam meekrijgt, en gebruikt vervolgens de eerder gedefinieerde methode `lees` voor het lezen van die stream:

```
void Freqtab::lees(char *naam)
{   ifstream f( naam );
    lees(f);
}
```

Merk op dat de compiler aan het type van de parameter kan zien welke versie van de methode `lees` hij moet gebruiken.

13.3 Het hoofdprogramma

In het programma moeten we coderen wanneer twee frequentietabellen 'op elkaar lijken'. We nemen daarvoor een heel eenvoudig criterium: voor elk symbool bepalen we in beide tabellen de relatieve frequentie. Hoe groter het verschil, hoe minder de tabellen op elkaar lijken. De verschillen voor alle symbolen worden bij elkaar opgeteld om 'het' verschil van twee tabellen te bepalen. Daarbij wordt het verschil wel steeds positief genomen (de absolute waarde), zodat uitschieters naar twee kanten elkaar niet kunnen opheffen. Dit alles is gecodeerd in de functie `verschil` in listing 14.

In de functie `main` wordt een array van frequentietabellen gedeclareerd. Dankzij onze class is dat net zo gemakkelijk als het declareren van een array met integers (voor de programmeur tenminste. De computer initialiseert zich rot.) De lengte van de array hangt af van het aantal namen op de commandoregel. De array wordt daarom dynamisch gecreëerd, op dezelfde manier als in het programma woordenboek dynamisch een tekst werd gecreëerd:

```
Freqtab *f = new Freqtab[aantal];
```

We hebben nog één extra tabel nodig om de frequentiekarakteristiek van de toetsenbord-invoer in te bepalen, maar daar kunnen we mooi het nulde element van de array voor gebruiken. De diverse elementen van de array `f` worden gevuld met informatie door aanroepen van de methode `lees`; voor de voorbeeld-teksten met een `char*` parameter, en voor de nieuwe tekst met `cin` als stream-parameter:

```
for (i=1; i<aantal; i++)
   f[i].lees( namen[i] );
f[0].lees(cin);
```

Tenslotte kan de best-passende taal gevonden worden door tabel `f[0]` te vergelijken met alle voorbeeld-tabellen, en daarbij de best-passende en zijn index te onthouden. Met die index kan de filenaam van de bijbehorende file worden teruggevonden en aan de gebruiker worden meegedeeld.

Opgaven

1. Pas het programma aan zodat het let op de frequenties van *tweetallen* letters. De typisch Nederlandse combinatie 'ij' wordt dan bijvoorbeeld beter herkend, en ook de typsich Engelse combinatie 'th'. Verzin zelf een passende aktie voor leestekens en andere niet-letters.

2. Schrijf een programma dat in teksten het gemiddelde en de spreiding bepaalt van het aantal woorden per zin, het aantal letters per woord, en het aantal komma's per zin. Ontwerp eerst

een class waarmee statistische gegevens gemakkelijk kunnen worden bijgehouden.

3. Zou je met het programma uit opgave 2 teksten van verschillende schrijvers uit elkaar kunnen houden? Zo niet, hoe wel?

4. Mag iemand veroordeeld worden wegens het vermeend schrijven van een tekst op grond van zo'n soort analyse? (Deze vraag vormt het onderwerp van een ruzie tussen Teun de Vries (vindt van wel) en Gerrit Komrij (vindt van niet)).

Hoofdstuk 14

Onbegrensde data, *of:* lijsten

In eerdere hoofdstukken hebben we arrays gebruikt om gegevens op te slaan. Nadeel daarbij was, dat van tevoren bekend moest zijn hoeveel elementen zo'n array heeft. In dit hoofdstuk bekijken we een methode waarmee gegevens opgeslagen kunnen worden, *zonder* dat vooraf bekend is hoeveel het er zullen zijn. Daarvoor gebruiken we *lijsten*. In tegenstelling tot de array is de lijst geen nieuwe taalconstructie; het is een programmeertechniek die gebruik maakt van taalconstructies die al in eerdere hoofdstukken werden ingevoerd: dynamisch aangemaakt geheugen met `new`, en classes.

14.1 Het programma

In dit hoofdstuk zullen we geen interessante toepassing behandelen. Dat zou de aandacht maar afleiden van de hoofdzaak: het aanmaken van een lijst met een onbegrensde (dat wil zeggen: niet vooraf bekende) lengte. We bekijken daarom een erg simpel probleem: druk de volgnummers af van woorden op de invoerstream die met een hoofdletter beginnen. Als de invoer van dit programma is:

```
Aap
noot
Mies
```

dan is de uitvoer van het programma:

```
1
3
```

omdat het eerste en het derde woord met een hoofdletter beginnen. (Komt de invoerstream van het toetsenbord, dan zal de uitvoer vermengd worden met de invoer.)

In het programma dat dit probleem oplost zijn geen arrays nodig en al helemaal geen lijsten. De programmeermethode waar het in dit hoofdstuk over gaat (gebruik van lijsten) is pas nodig als we de probleemstelling aanpassen: de volgnummers van de woorden met een hoofdletter moeten in *omgekeerde* volgorde worden afgedrukt. De uitvoer zou bij bovenstaande invoer dus worden:

```
3
1
```

14.2 Hoofdletters

Om te beginnen bekijken we het basisprobleem: het afdrukken van de volgnummers van woorden die met een hoofdletter beginnen. Het probleem wordt opgelost in programma hoofdlet in listing 15. Het programma spreekt geheel voor zich. Er is alleen één regel die iets anders is geformuleerd dan tot nu toe. In plaats van de tellende variabele eerst te declareren met

```
int r;
```

en vervolgens te gebruiken in een `for`-opdracht:

```
for (r=1; cin>>woord; r++)
```

wordt de variabele tegelijkertijd gedeclareerd en van beginwaarde voorzien:

```
for (int r=1; cin>>woord; r++)
```

Deze constructie hebben we tot nu toe alleen nog maar gebruikt voor constanten:

```
#include <iostream.h>

int hoofdletter(char w[])
{   return (w[0]>='A' && w[0]<='Z');
}

int main()
{   char woord[100];

    for (int r=1; cin>>woord; r++)
       if (hoofdletter(woord))
           cout << r;
    return 0;
}
```

Listing 15: hoofdlet.cc

```
const int AANTAL = 100;
```

Bij constanten is declaratie en gelijktijdige toekenning verplicht (constanten kunnen immers maar één keer een waarde krijgen). Bij variabelen is de constructie van declaratie en gelijktijdige toekenning echter ook toegestaan; voor de aardigheid hebben we dat hier eens gebruikt. Veel programmeurs vinden het duidelijk om variabelen zo dicht mogelijk bij het gebruik ervan te declareren; welnu: dichterbij dan op deze manier is niet mogelijk. (Andere programmeurs vinden het daarentegen overzichtelijker om alle variabelen aan het begin van de functie bij elkaar te declareren).

Deze `for`-opdracht is tevens een mooi voorbeeld van een geval waarin de tellende variabele (`r`) niet voorkomt in de voorwaarde-om-door-te-gaan (`cin>>woord`). Vaak is dat verdacht, omdat het stoppen van de opdracht daarmee niet direct duidelijk is. In dit geval is het echter duidelijk dat de opdracht zal stoppen zodra de invoerstream beëindigd wordt.

14.3 Onbegrensde data

Het programma hoofdlet is niet gemakkelijk aan te passen aan de gewijzigde probleemstelling dat de woordnummers in omgekeerde volgorde afgedrukt moeten worden. De af te drukken getallen moeten dan immers bewaard worden totdat alle regels zijn ingelezen, om dan in omgekeerde volgorde afgedrukt te worden. Een array ligt voor de hand om de woordnummers in op te slaan. Maar hoe lang moet deze array zijn? Het hangt van de omstandigheden af welke oplossing je hiervoor kiest:

- De programmeur kan voorspellen wat het maximale aantal woordnummers is dat opgeslagen moet worden. In dit geval kan natuurlijk wel gewoon een array gedeclareerd worden. Het is verstandig om een constante te definiëren waarin dit aantal wordt bepaald, en die constante te gebruiken bij de declaratie van de array; op die manier is de keuze later eventueel nog te wijzigen.

  ```
  const int AANTAL = 1000;
  int nummers[AANTAL];
  ```

  Bij wijziging van `AANTAL` moet het programma echter wel opnieuw gecompileerd worden.

- De programmeur weet het maximale aantal woordnummers niet, maar de gebruiker van het programma wel. Het aantal kan bij elk gebruik van het programma verschillend zijn, maar moet wel bekend zijn voordat de gegevens ingelezen worden. In dit geval kan de geheugenruimte voor de array aangemaakt worden met `new`. De naam wordt gedeclareerd als verwijzing naar de array, en op het moment dat de lengte bekend is wordt de verwijzing gezet naar een nieuw gecreëerde array:

  ```
  int *nummers;
  int aantal;
  ```

```
cout << "hoeveel regelnummers maximaal? ";
cin  >> aantal;
nummers = new int[aantal];
```

In het type dat achter `new` geschreven wordt mag het aantal elementen (anders dan bij declaraties) variabel zijn; daar wordt hier dankbaar gebruik van gemaakt.

- De gebruiker kan of wil aan het begin van de sessie niet het maximaal aantal elementen in de array opgeven (of de programmeur is te bescheiden om er om te vragen). In dit geval kan er geen array gebruikt worden; we zullen een *lijst* moeten gebruiken, waaraan steeds nieuwe elementen toegevoegd kunnen worden, zonder dat aan het begin het maximale aantal bekend hoeft te zijn.

Voor de opbouw van een lijst zullen we een speciale klasse `ListElem` (voor: lijst-element) definiëren. In objecten die deze klasse als type hebben kan één integer opgeslagen worden. Maar bovendien zit er in de klasse een verwijzing naar het *volgende* lijst-element. De klasse heeft dus de volgende struktuur:

```
class ListElem
{
    int      data;
    ListElem *next;
};
```

De verwijzing `next` in een object verwijst naar een volgend object, waarin een tweede integer te vinden is, alsmede een verwijzing naar het daaropvolgende element. Dit gaar zo door, todat het laatste element bereikt is, waar een speciale verwijzing naar 'niets meer' staat.

In listing 16 staan alle `class`-definities die voor het opbouwen van een lijst noodzakelijk zijn. In listing 17 staat het programma waar het allemaal om begonnen was (het in omgekeerde volgorde afdrukken van woordnummers); dit programma maakt gebruik van de klassen in listing 16. In de volgende paragrafen zullen we beide listings bespreken.

14.4 Lijstelementen en lijsten

Een lijst*element* is nog geen complete lijst. Daarom maken we naast de klasse `ListElem` ook nog een klasse `List`. De struktuur van de klasse `List` is zeer eenvoudig: het is simpelweg een verwijzing naar het eerste lijstelement. Via de `next`-component van dat lijstelement zijn de volgende elementen desgewenst wel te bereiken.

De reden dat we ook van `List` een aparte klasse maken (hoewel deze maar één component heeft), is dat we in deze klasse een aantal methoden kunnen definiëren die op een lijst werken. (Ter herinnering: een 'methode' is een functie die de componenten van een object mag gebruiken). In dit geval zijn dat methoden om een element aan de lijst toe te voegen, een methode om te bepalen of de lijst leeg is, en een methode om een nieuwe lijst te creëren. De declaratie van de klasse zou als volgt kunnen zijn:

```
class List
{
    ListElem    *first;
public:
                List  ();
    int         empty ();
    void        add   (int x);
};
```

Zoals bij alle klassen is er ook hier een methode met dezelfde naam als de klasse, die geen resultaattype heeft. Dit is de methode die automatisch wordt aangeroepen als er een object uit deze klasse wordt gecreëerd. Dat is bijvoorbeeld het geval bij een lokale declaratie:

```
List mijnlijst;
```

of bij uitrekenen van een expressie met `new` erin:

```
List *v;
...
v = new List;
```

De variabele `first` verwijst naar het eerste element van de lijst. Maar zolang er nog geen enkel element aan een zojuist gecreëerde lijst is toegevoegd, is er nog geen eerste element. Voor dit doel is er een speciale verwijzings-waarde beschikbaar, `NULL` geheten. Deze waarde betekent 'wijst nergens naar', en kan gebruikt worden als waarde van `first` zolang er nog geen elementen zijn. In de constructor-methode `List` wordt deze waarde toegekend:

```
List::List()
{  first = NULL;
}
```

De methode `empty`, waarmee gecontroleerd kan worden of een lijst leeg is, kijkt of de waarde van `first` nog steeds `NULL` is:

```
int List::empty()
{  return  first==NULL ;
}
```

De buitenwereld mag de variabele `first` niet direct bekijken; de declaratie ervan staat immers niet in het `public:` gedeelte. Alleen via de methode `empty` kan vastgesteld worden of een lijst al dan niet leeg is.

De twee methode-definities zijn zeer kort: de body bestaat uit maar één opdracht. Om het geheel overzichtelijker te houden is het toegestaan om de methode-definitie al direct te geven in de class-definitie. In de class-definitie kan dus behalve de header van de methoden ook al direct de definitie ervan staan. De class-definitie `List` zou daarmee worden:

```
class List
{
    ListElem    *first;
public:
                 List  ()      {first=NULL; } ;
    int          empty ()      {return first==NULL; } ;
    void         add   (int x);
};
```

Aparte methode-definities voor `List::List` en `List::empty` zijn dan niet meer nodig.

Wel moeten we nog een definitie voor `List::add` geven. Deze methode is bedoeld om een lijstelement aan de lijst toe te voegen. Daarvoor moeten we eerst de class `ListElem` verder uitwerken.

14.5 Constructor-methoden met parameters

Eerder zagen we al welke gegevens er in een object van het type `ListElem` opgeslagen staan: een integer genaamd `data`, en een verwijzing naar een volgend lijstelement genaamd `next`. Natuurlijk is er ook in de klasse `ListElem` een constructor-methode nodig, die een zinvolle beginwaarde geeft aan deze twee variabelen.

De constructor-methode van `List` kon aan de variabele `first` een neutrale waarde geven (`NULL`). Bij een nieuw gecreëerd lijstelement is het echter nodig dat de variabele `data` de waarde van het nieuwe lijstelement gaat bevatten; deze waarde kan per element verschillen. Ook de waarde van de variabele `next` verschilt in elk lijstelement: elk lijstelement heeft immers een andere opvolger. We geven de constructor-methode daarom twee parameters, waarin de gewenste waarden van `data` en `next` kunnen worden opgegeven.

De body van de methode `ListElem` bestaat simpelweg uit het toekennen van de waarden van de twee parameters aan de twee in de klasse gedeclareerde variabelen.

Moet de constructor-methode `ListElem` nu een publieke methode worden? Enerzijds wel, want de methode `add` (die bij een andere klasse hoort: `List`) moet een nieuw lijstelement kunnen creëren. Anderzijds zouden we de constructor liever privé houden, want het is niet de bedoeling dat het hoofdprogramma op eigen houtje, dus zonder `List::add` te gebruiken, lijstelementen kan creëren.

Gelukkig is er uitgerekend voor deze situatie een tussenvorm mogelijk: we houden de methode privé, maar verklaren dat de methoden van de klasse `List` tot de 'vriendjes' behoren die hem wèl mogen gebruiken. Voor dit doel is er een apart gereserveerd woord beschikbaar: `friend`. In een klasse-definitie kunnen achter dit woord de bevriende klassen worden genoemd. De definitie van de klasse `ListElem` wordt daarmee:

```
class List;
class All;

class ListElem
{
    friend List;
    friend All;

    int       data;
    ListElem *next;
              ListElem (int a, ListElem *b) {data=a; next=b; };
};

class List
{
    friend All;

    ListElem    *first;
public:
                 List  ()      {first=NULL; };
    int          empty ()      {return first==NULL; };
    void         add   (int x) {first = new ListElem(x, first); };
};

class All
{
    ListElem    *curr;
public:
                 All   (List list)  {curr = list.first; };
    int          valid ()           {return curr!=NULL; };
    void         next  ()           {curr = curr->next; };
    int          value ()           {return curr->data; };
};
```

Listing 16: lijst0.cc

```
class ListElem
{   friend List;
    int       data;
    ListElem *next;
              ListElem (int a, ListElem *b) {data=a; next=b; };
};
```

Nu we een definitie van **ListElem** hebben, en de klasse **List** tot vriendje verklaard is, kunnen we verdergaan met de definitie van **List::add**.

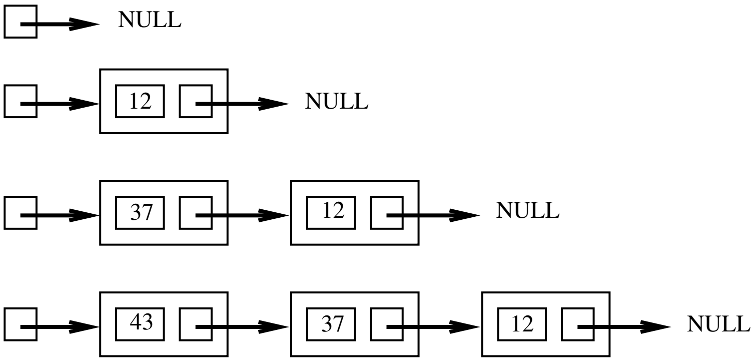
De methode **add** moet een nieuw lijstelement maken. Omdat hierbij echt nieuwe geheugenruimte aangemaakt moet worden, gebruiken we daarvoor een expressie met het gereserveerde woord **new**. Het resultaat daarvan is een verwijzing naar het nieuwe lijstelement. Om het nieuwe element de eerste in de keten te laten worden, slaan we deze verwijzing op in de variabele **first**. In het nieuwe lijstelement wordt de integer-waarde opgeslagen die aan de lijst toegevoegd moet worden. Als **next**-component van het nieuwe lijstelement gebruiken we de eerdere waarde van **first**, zodat de oude lijstelementen achter het nieuwe element geschakeld worden. Al met al wordt de definitie van **List::add** als volgt:

```
void List::add(int x)
{
    first = new ListElem(x, first);
}
```

Figuur 14.1 maakt duidelijk hoe een aanvankelijk lege lijst er uitziet na toevoegen van de waarden

```
#include <iostream.h>
#include "lijst0.cc"

int hoofdletter(char w[])
{   return (w[0]>='A' && w[0]<='Z');
}

int main()
{
    List lijst;
    char woord[100];

    for (int r=1; cin>>woord; r++)
       if (hoofdletter(woord))
          lijst.add( r );

    for (All x(lijst); x.valid(); x.next() )
       cout << x.value() << endl;

    return 0;
}
```

Listing 17: lijst0test.cc



Figuur 14.1: Lijst is een verwijzing naar eerste van drie lijstelementen

12, 37 en 43.

Omdat `add` in feite maar uit één opdracht bestaat kan de definitie ook wel direct in de klasse-definitie worden geschreven. Dit is gebeurd in listing 16.

14.6 Iteratie over lijsten

Het opbouwen van een lijst met `add` is één ding, maar de elementen van de lijst moeten natuurlijk ook nog wel eens gebruikt kunnen worden. Anders zou het opslaan van de elementen weinig zin hebben. In principe kunnen alle elementen van de lijst bereikt worden via de `next`-verwijzingen in de lijstelementen.

Echter, alle componenten van de klasse `ListElem` zijn privé; ze mogen alleen door de bevriende klasse `List` gebruikt worden. Dat willen we eigenlijk zo houden: gebruikers van de klasse moeten niet lastiggevallen worden met de precieze manier waarop lijstelementen aan elkaar gekoppeld zijn; ze willen gewoon kunnen zeggen: 'doe iets met alle elementen van de lijst'.

Voor het maken van een nieuwe lijst en het toevoegen van elementen daaraan hebben we de klasse `List` ontworpen. Voor het langslopen van alle elementen gaan we ook een aparte klasse maken, genaamd `All`. Elke keer als alle elementen van een lijst langsgelopen moeten worden, declareert de programmeur van het hoofdprogramma een object met type `All`. Verborgen in dat object zit een verwijzing naar het 'huidige' lijstelement. Dat hoeft de programmeur die een `All` declareert echter niet te weten; alles wat die nodig heeft zijn de vier publieke methoden van `All`:

- de constructormethode `All`, met als parameter een lijst
- een methode `valid`, die aangeeft of er een valide huidig element is
- een methode `next`, waarmee het volgende element tot 'huidig' element gemaakt wordt
- een methode `value`, waarmee de waarde van het huidige element gevonden kan worden.

Al deze vier methoden bestaan uit één opdracht, die we gemakshalve direct in de klasse-definitie erbij schrijven:

```
class All
{
    ListElem    *curr;
public:
                All   (List list)  {curr = list.first; };
    int         valid ()           {return curr!=NULL; };
    void        next  ()           {curr = curr->next; };
    int         value ()           {return curr->data; };
};
```

In de constructor-methode wordt het huidige element `curr` gelijk gemaakt aan de variabele `first` van de lijst in kwestie. Een `All`-object heeft een '`valid`' element beschikbaar als het huidige element nog niet de verwijzing-naar-niks `NULL` is.

In de methode `next` komt een nieuwe notatie voor: het symbool `->` waarin met enige goede wil een pijltje te herkennen is. Links van dit symbool moet de naam van een *verwijzing naar een object* staan; rechts ervan de naam van een component. De betekenis van `v->c` is: neem van het object waar `v` naar verwijst de component `c`.

Er zijn dus twee notaties om een component (variabele of methode) van een object aan te spreken:

- `obj.c` waarbij `obj` een object is
- `verw->c` waarbij `verw` een *verwijzing naar* een object is

Beide notaties zijn nodig in de methoden van de klasse `All`. Omdat in deze klasse de privé-component `first` van de klasse `List`, en `data` en `next` van de klasse `ListElem` gebruikt worden, moet de klasse `All` tot `friend` van zowel `List` als `ListElem` verklaard worden.

Namen van functies en methoden die je in een C++-programma gebruikt, moeten eerder in het programma gedefinieerd zijn (of met `#include` uit een bibliotheek gehaald zijn). Omdat de klasse `List` de constructor-methode van `ListElem` gebruikt, moet `ListElem` dus vóór `List` gedefinieerd worden. De klasse `All` gebruikt dingen uit zowel `List` als `ListElem`, en moet dus als laatste gedefinieerd worden.

Een probleempje is dat in de klasse `ListElem`, die als eerste wordt gedeclareerd, al aan de klassen `List` en `All` gerefereerd wordt, namelijk waar ze tot `friend` verklaard worden. Omdat de compiler op dat moment nog niet kan weten dat `List` en `All` klassen zijn, moet dat vóór de definitie van `ListElem` alvast worden aangekondigd met:

```
class List;
class All;
```

Deze twee regels lichten de compiler alvast in dat `List` en `All` inderdaad klassen zijn, waarvan de definitie later verwacht kan worden.

Hiermee is het complete pakket klasse-definities is listing 16 besproken.

14.7 Gebruik van de lijst-klassen

We hebben de klasse-definities voor `List` geschreven, omdat we een programma moesten schrijven dat een van tevoren onbekende hoeveelheid gegevens moet opslaan: een programma dat woorden inleest, en de volgnummers van woorden die met een hoofdletter beginnen *in omgekeerde volgorde* afdrukt. Gebruikmakend van de nieuw gedefinieerde klassen is dat niet moeilijk meer. De tekst van het programma staat in listing 17.

De tekst van de klasse-definities hoeft niet eens meer in het programma opgenomen te worden; met de regel

```
#include "lijst0.cc"
```

kunnen we aangeven dat de klassen uit listing 16 in het programma nodig zijn. Op deze manier kun je dus je eigen bibliotheken schrijven: lijsten van getallen zijn in andere programma's vast ook nog wel eens van nut. In de `#include`-regel worden systeembibliotheken van zelfbedachte bibliotheken onderscheiden door de symbolen die er omheen staan: aanhalingstekens bij zelfgemaakt bibliotheken zoals `"lijst0.h"`, punthaken bij standaardbibliotheken zoals `<iostream.h>`.

In het programma in listing 17 wordt een object van type `List` gedeclareerd, en ruimte om één woord in te kunnen lezen:

```
List lijst;
char woord[100];
```

De variabele `lijst` wordt door de constructor-methode van een zinvolle beginwaarde voorzien (de lege lijst). Vervolgens kunnen we woorden inlezen, en indien gewenst het bijbehorende regelnummer `r` in de lijst opslaan:

```
for (int r=1; cin>>woord; r++)
   if (hoofdletter(woord))
      lijst.add( r );
```

Tenslotte kunnen we met behulp van een object van type `All` alle elementen van de lijst langslopen:

```
for (All x(lijst); x.valid(); x.next() )
   cout << x.value() << endl;
```

De vier methoden van objecten van type `All` zijn eigenlijk op maat gemaakt om in een `for`-opdracht te gebruiken: de constructor voor de initialisatie, `valid` als test om nog door te gaan, `next` in plaats van het ophogen van een teller, en `value` in de body.

Let op de overeenkomst tussen de `for`-opdracht die een integer gebruikt om regels langs te lopen, en de `for`-opdracht waar een `All`-object gebruikt wordt om lijstelementen langs te lopen. In beide gevallen wordt er een variabele gedeclareerd (van type `int`, resp. `All`) met een naam (`r`, resp. `x`) en een beginwaarde (`1`, resp. `lijst`).

Dat de lijstelementen inderdaad van achter naar voren worden afgedrukt komt doordat we de methode `add` zo geschreven hebben dat de laatst toegevoegde elementen aan het begin van de lijst staan. Als je dus een lijst langsloopt, kom je de laatst toegevoegde elementen het eerst tegen.

Opgaven

1. Voeg een methode `append` toe aan `List` waarmee een element kan worden toegevoegd aan het *eind* van de lijst.

2. De methode `append` kan efficiënter worden gemaakt door een extra component op te nemen in de klasse `List`. Welke is dat, en hoe op welke plaatsen moet deze component worden aangepast?

3. Schrijf een methode `List::elem(int)` die een element met een gegeven volgnummer oplevert.

4. Vergelijk de tijd die benodigd is om `elem` aan te roepen met de tijd die nodig is om een array-element te selecteren.

5. Definieer een klasse `DLijst` ('dubbel-gelinkte lijst'), waarin bij elk element niet alleen een verwijzing naar het volgende, maar ook een naar het vorige element staat. Voeg aan de bijbehorende klasse `DLijstElem` een methode toe, waarmee het element uit een lijst verwijderd kan worden.

Hoofdstuk 15

Hergebruik, *of:* templates

In dit hoofdstuk bekijken we hoe klasse-bibliotheken, zoals `"lijst0.cc"`, zo algemeen mogelijk bruikbaar gemaakt kunnen worden. Vooral nuttig is het daartoe om klassen te kunnen *parametriseren* met een type, dat wil zeggen een variabel type te kunnen gebruiken in de klasse.

15.1 Containers

In objecten van het type `List`, zoals gedefinieerd in listing 16 in het vorige hoofdstuk, kunnen lijsten van integers worden opgeslagen. Door de klasse-definities in een aparte file te zetten, kunnen andere programma's eenvoudig gebruik maken van deze klassen, door deze bibliotheek met een `#include`-regel mee te nemen. Omdat programmeren toch wel erg veel werk is, streven velen ernaar om een persoonlijke bibliotheek op te bouwen van klassen die in veel programma's van nut zijn. De klasse `list` is een goed voorbeeld van zo'n algemeen bruikbare klasse.

In sommige programma's zijn er echter lijsten nodig van andere dingen dan integers. De opdrachtgever van het programma in het vorige hoofdstuk zou zich bij het zien van een demonstratie van het programma bijvoorbeeld kunnen bedenken dat in plaats van de regelnummers van woorden met een hoofdletter, ook de woorden zelf wel interessant zijn. We kunnen het hoofdprogramma gemakkelijk aanpassen, maar in plaats van een lijst van integers hebben we nu een lijst van teksten nodig. Er moet dus een nieuwe klasse gedefinieerd worden. Deze lijkt weliswaar sterk op de klasse lijst-van-integers (vervang overal `int` door `char*`), maar het is wel veel werk om de klasse daadwerkelijk aan te passen. Vooral bij grote bibliotheken is dat erg vervelend. Bovendien zijn er behalve integers en teksten nog wel andere dingen denkbaar waarvan je lijsten zou willen bijhouden.

Bij de definitie van `List` en `ListElem` valt het op, dat het feit dat de opgeslagen informatie integers zijn nergens echt gebruikt wordt. Daarom is het ook mogelijk om `int` te vervangen door `char*` of door een ander type, waarbij het geheel nog steeds werkt. Klassen zoals `List` heten *container* klassen: er worden dingen bewaard, en alle methoden zijn er alleen op gericht om de bewaarde dingen terug te vinden; het type van de bewaarde dingen is niet van belang.

Het zou prettig zijn als je zo'n container-klasse maar éénmaal hoefde te definiëren. Je zou bijvoorbeeld eenmalig een lijst-van-'dingen' kunnen definiëren, zodat het niet meer nodig is om apart een lijst-van-integers en een lijst-van-teksten en een lijst-van-floats en een lijst-van-frequentietabellen te definiëren. Zo'n definitie is dan meteen ook voorbereid op alle objecten die je in toekomstige programma's misschien nog gaat definiëren.

Dat kan. Het is mogelijk om in een klasse-definitie een type-*variabele* te gebruiken. Zo kun je dus een lijst-van-T definiëren, waarbij het type T nog open gelaten wordt. Het te gebruiken type T is als het ware een parameter van de klasse-definitie.

15.2 Templates

We zullen de klasse-definities van `List` en vrienden nog een keer opnieuw schrijven, maar nu niet voor elementen van type `int` maar voor elementen van willekeurig type. In de objecten van type `ListElem` zullen we *verwijzingen* opnemen naar de eigenlijke elementen. Zo wordt het mogelijk om lijsten te maken van reeds bestaande objecten, zonder van die objecten weer een nieuwe kopie te maken.

Voor het schrijven van een klasse waarin variabele types gebruikt worden, zijn twee nieuwe notaties nodig:

- 'dit is de definitie van een klasse `List` waarin een typevariabele `T` gebruikt wordt' wordt genoteerd als:

  ```
  template <class T>  class List
  {
  ```

  Het woord `template` is een gereserveerd woord.
- 'ik heb een object `x` van klasse `List` nodig, met als elementen ditmaal waarden van het type `char`' wordt genoteerd als

  ```
  List<char> x;
  ```

Gebruikmakend van deze twee nieuwe notaties gaan we de klasse-definities uit listing 16 aanpassen. Om te beginnen de klasse `ListElem`.

Als uitgangspunt kunnen we de definitie van `ListElem` uit listing 16 nemen:

```
class ListElem
{   int       data;
    ListElem *next;
              ListElem (int a, ListElem *b) {data=a; next=b; };
};
```

Omdat we de in de klasse een typevariabele `T` gaan gebruiken, komt er de tekst `template <class T>` te boven te staan. In plaats van integers worden in de nieuwe definitie verwijzingen-naar-`T` opgeslagen. We vervangen dus `int data` door `T *data` en `int a` door `T *a`. Op de drie plaatsen waar het type `ListElem` wordt gebruikt (bij de declaratie van `next` en parameter `b` van de constructor-methode, en de naam van de constructor-methode zelf) moeten we specificeren dat het ook hier lijsten-van-`T` betreft. We vervangen dus op drie plaatsen `ListElem` door `ListElem<T>`. Het resultaat is:

```
template <class T>
class ListElem
{   T           *data;
    ListElem<T> *next;
                 ListElem<T> (T *a, ListElem<T> *b) {data=a; next=b; };
};
```

Om het iets overzichtelijker te maken is er nog een vereenvoudiging toegestaan: binnen de definitie van de met `T` geparameteriseerde klasse `ListElem` mogen we in plaats van `ListElem<T>` ook `ListElem` schrijven.

Op dezelfde manier nemen we de klassen `List` en `All` onder handen. Het eindresultaat staat in listing 18.

15.3 Gebruik van de template-klasse

Om het gebruik van de nieuwe lijst-bibliotheek te demonstreren passen we het voorbeeldprogramma uit het vorige hoofdstuk aan. Nog steeds worden er woorden ingelezen. Van de woorden die met een hoofdletter beginnen moet het regelnummer afgedrukt worden èn het woord zelf, nog steeds in omgekeerde volgorde.

Het voorbeeldprogramma staat in listing 19. Zoals gewoonlijk begint het programma met het noemen van de gewenste bibliotheken:

```
#include <iostream.h>
#include <string.h>
#include "lijst.cc"
```

De bibliotheek `<iostream>` is nodig voor de in- en uitvoer, `<string.h>` is nodig voor de functie `strlen`, en `"lijst.cc"` is de vernieuwde lijst-bibliotheek uit listing 18.

De dingen die in een lijst bewaard moeten worden zijn ditmaal combinaties regelnummer–woord. Combinaties van gegevens die bij elkaar horen zijn objecten; we moeten dus een klasse definiëren waarin een regelnummer en een woord gedeclareerd worden. Als we deze klasse `Regel` noemen, moeten we de lijst declareren als `List<Regel> lijst;`.

De definitie van de klasse `Regel` is gemakkelijk te geven; de benodigde gegevens zijn een `int` (het regelnummer) en een `char*` (het bijbehorende woord). Er is natuurlijk een constructormethode, met als parameters de beginwaarden voor deze twee variabelen. Verder is er een methode waarmee

```
template <class T> class List;
template <class T> class All;

template <class T>
class ListElem
{
    friend List<T>;
    friend All<T>;

    T        *data;
    ListElem *next;
                 ListElem (T *a, ListElem *b) {data=a; next=b; };
};

template <class T>
class List
{
    friend All<T>;

    ListElem<T> *first;
public:
                 List  ()     {first=NULL; };
    int          empty ()     {return first==NULL; };
    void         add   (T *x) {first = new ListElem<T> (x, first); };
};

template <class T>
class All
{
    ListElem<T> *curr;
public:
                 All    (List<T> list) {curr = list.first; };
    int          valid  ()             {return curr!=NULL; };
    void         next   ()             {curr = curr->next; };
    T           *value  ()             {return curr->data; };
};
```

Listing 18: lijst.cc

zo'n `Regel` geprint kan worden; vergelijk dit met de methode `verslag` in de klasse `Freqtab` in hoofdstuk 12. Om het helemaal mooi te maken geven we aan de functie `print` als parameter de stream mee waarop geprint moet worden. De complete definitie is:

```
class Regel
{   int   nummer;
    char *woord;
public:
          Regel(int n, char*w) { nummer=n; woord=w; };
    void  print(ostream &str)  { str<<nummer<<": "<<woord<<endl;  };
};
```

Dit voorbeeld geeft wel aan dat een klassedefinitie helemaal niet groot hoeft te zijn. Zodra er in een programma gegevens op een logische wijze bij elkaar horen (zoals hier een regelnummer en een woord) is het al snel de moeite waard om er een klassedefinitie voor te schrijven.

Het hoofdprogramma lijkt sterk op het programma in het vorige hoofdstuk. In plaats van een `List` van integers declareren we nu echter een `List<Regel>` van verwijzingen naar `Regel`-objecten. Aan de methode `add` wordt nu dan ook geen integer meegeven, maar een met `new` nieuw gecreëerde `Regel`:

```
#include <iostream.h>
#include <string.h>
#include "lijst.cc"

class Regel
{
    int   nummer;
    char *woord;
public:
          Regel(int n, char*w) { nummer=n; woord=w; };
    void  print(ostream &str)  { str<<nummer<<": "<<woord<<endl;  };
};

char *kopie(char *s)
{   return strcpy( new char[strlen(s)+1], s );
}

int hoofdletter(char w[])
{   return (w[0]>='A' && w[0]<='Z');
}

int main()
{
    List<Regel> lijst;
    char woord[100];

    for (int r=1; cin>>woord; r++)
       if (hoofdletter(woord))
           lijst.add( new Regel(r, kopie(woord)) );

    for (All<Regel> x(lijst); x.valid(); x.next() )
       x.value()->print(cout);

    return 0;
}
```

Listing 19: lijsttest.cc

```
for (int r=1; cin>>woord; r++)
   if (hoofdletter(woord))
       lijst.add( new Regel(r, kopie(woord)) );
```

De nieuw gecreëerde regel wordt gelijktijdig van waarden voorzien: het regelnummer `r`, en een kopie van het gelezen woord. We moeten een kopie van het woord maken omdat in `Regel`-objecten *verwijzingen* naar woorden staan. Zouden we geen kopie maken, dan wijzen al die verwijzingen uiteindelijk naar dezelfde array `woord`, die op dat moment als waarde het laatst ingelezen woord heeft (dit verschijnsel zagen we ook al in het programma woordenboek in hoofdstuk 10). In figuur 15.1 is afgebeeld op welke manier objecten en referenties aan elkaar geknoopt zijn na inlezen van een aantal woorden.

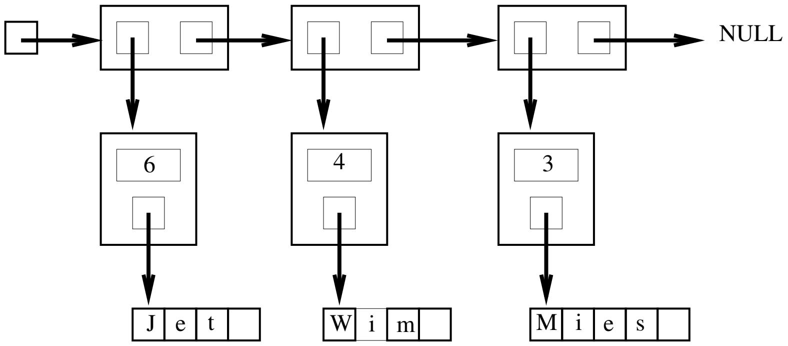
In het programma in het vorige hoofdstuk werden de lijstelementen geprint met de opdracht

```
for (All x(lijst); x.valid(); x.next() )
   cout << x.value() << endl;
```

Deze opdracht veranderen we in

```
for (All<Regel> x(lijst); x.valid(); x.next() )
   x.value()->print(cout);
```

Het verschil is dat we het type `All<Regel>` gebruiken in plaats van `All`. Daardoor is het type van `x.value()` nu `Regel*` in plaats van `int`. Gelukkig is er in de klasse `Regel` een methode `print`



Figuur 15.1: Lijst met verwijzingen naar `Regel`s, waarin verwijzingen naar teksten

waarmee een regel afgedrukt kan worden. Omdat `x.value()` een *verwijzing* naar een `Regel`-object is, gebruiken we de pijltje-notatie om de methode van het object te selecteren, en niet de punt-notatie.

De functie `kopie` is dezelfde functie die we in hoofdstuk 10 geschreven hadden om een nieuwe kopie van een tekst te maken, hoewel hij hier wat compacter is opgeschreven.

Opgaven

1. Vergelijk de definitie van de functie `kopie` in listing 19 met die in listing 10. Wat kun je concluderen over het resultaat van de standaardfunctie `strcpy`?

Hoofdstuk 16

Zoeknetwerken, *of:* ontwerp van klassen

In dit hoofdstuk bekijken we een programma dat gegeven een landkaart met steden en wegen, een weg zoekt tussen twee steden. Dit probleem wordt behandeld in hoofdstuk 4 van het boek 'Artificial Intelligence' van Patrick Henry Winston. In dit hoofdstuk wordt besproken hoe de diverse elementen in het probleem kunnen worden beschreven met behulp van klassen.

In het genoemde boek wordt een zoekalgoritme beschreven. Daarbij wordt gebruik gemaakt van een 'queue'. We beginnen dit hoofdstuk daarom met het schrijven van een klasse `Queue`. We maken die klasse net zo algemeen bruikbaar als de klasse `List`, die overigens ook nu weer goede diensten zal bewijzen.

Vervolgens zullen we klassen schrijven waarmee steden, wegen en landkaarten worden beschreven. Het zoekalgoritme waarmee een weg tussen twee steden gevonden kan worden zullen we in een methode van de klasse 'landkaart' coderen. Tenslotte schrijven we een programma, `netmain` geheten, die gebruikmakend van al die klassen op verzoek van een gebruiker een landkaart opbouwt en daarin wegen kan zoeken.

16.1 Een queue als klasse

Het voordeel van een datastructuur die van verwijzingen aan elkaar hangt, zoals een lijst, is de flexibiliteit: elementen kunnen naar believen worden toegevoegd, en de datastructuur is nooit 'vol'. Het nadeel is echter dat de elementen nogal lastig toegankelijk zijn. Ze allemaal op een rij langslopen is nog wel te doen, maar bijvoorbeeld zoeken volgens de halveringsmethode is in een lijst nauwlijks te doen.

In gevallen waar de maximale omvang van de datastructuur vooraf te bepalen is, is het vaak eenvoudiger om een array te gebruiken. Met arrays is het echter uitkijken: een vergissing met array-indices is gauw gemaakt. Daarom is het verstandig om als dat mogelijk is een array 'in te pakken' in een klasse, en hem alleen aan te spreken via een paar welomschreven methoden.

In deze paragraaf zullen we een *queue* maken van een array. Een queue is een rij waarden, waarbij aan het begin en aan het eind waarden kunnen worden toegevoegd en weggehaald. We maken dus geen methoden die waarden in het midden van de rij manipuleren; die moeten wachten tot ze aan één van de uiteinden vrij komen te liggen.

Wat het type is van de elementen die in de queue staan opgeslagen maakt voor de klasse-definitie niet uit. We maken er daarom weer een template-klasse van, dus een klasse waarin een typevariabele `T` wordt gebruikt. Net als bij de lijst zetten we *verwijzingen* naar reeds bestaande `T`'s in de queue. De klassedefinitie zou als volgt kunnen beginnen:

```
template <class T>
class Queue
{    T *data[100];
```

Maar om de klasse toch nog iets flexibeler te laten zijn, maken we de array dynamisch: de lengte wordt niet vastgelegd in het programma, maar bij creatie door middel van `new`. Net als bij de dynamische arrays in hoofdstuk 10, declareren we daarom niet een array `data`, maar een verwijzing naar een later aan te maken exemplaar. Omdat de elementen van de array ook al verwijzingen zijn, is `data` dus een *verwijzing naar* (later aan te maken) *verwijzingen*, in de declaratie aangeduid door twee sterretjes. De geheugenruimte wordt met `new` aangemaakt in de constructor-methode. De te gebruiken lengte is parameter van de constructor-methode, zodat bij elke declaratie van een queue de lengte apart bepaald kan worden. De klassedefinitie wordt dus iets als:

```
template <class T>
class Queue
{   T **data;
public:
    Queue(int n)  { data = new T*[n]; };
```

Behalve de (verwijzingen naar) gegevens zelf, bestaat een queue-object uit een paar integers die aangeven op welke plaats in de array de rij begint en eindigt, wat de huidige lengte is, en wat de maximale lengte is. Deze vier variabelen declareren we door:

```
int begin, end, lengte, maxlengte;
```

In de klasse zullen we vier methoden definiëren voor het toevoegen en verwijderen van elementen aan het begin en aan het eind van de queue. Bij deze vier methoden zullen we er voor zorgen dat steeds de volgende eigenschappen gelden:

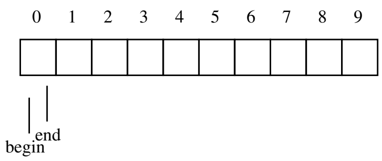
- de waarde `begin` is de index van het eerste element
- de waarde `end` is de index net voorbij het laatste element
- als `lengte` nul is, dan is `begin` niet de index van het eerste element (want er is geen eerste element); `begin` is in dit geval gelijk aan `end`.

De vier methoden mogen ervan uitgaan dat deze eigenschappen aan het begin gelden, maar ze moeten ervoor zorgen dat ze na afloop ook weer gelden. Als er bijvoorbeeld een element uit de rij verwijderd wordt, dan moeten de waarden van `begin`, `end` en `lengte` dienovereenkomstig veranderd worden. Verder maken we een vijfde methode `empty` waarmee getest kan worden of de queue leeg is. De complete klasse-definitie (waarin de methode `empty` ook alvast wordt gedefinieerd) is als volgt:

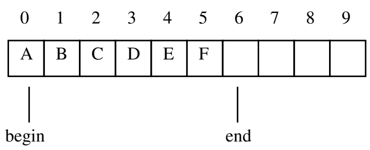
```
template <class T>
class Queue
{
    T    **data;
    int  begin, end, lengte, maxlengte;
public:
         Queue        (int);
    int  empty        ()     {return lengte==0; };
    void addAtBegin   (T*);
    void addAtEnd     (T*);
    T*   getFromBegin ();
    T*   getFromEnd   ();
};
```

In de constructor-methode `Queue` wordt niet alleen de geheugenruimte voor `data` aangemaakt, maar krijgen bovendien de vier integer-variabelen hun beginwaarde; zie listing 20.

Doordat gegevens zowel aan het begin als aan het eind kunnen worden toegevoegd en weggehaald, zullen tijdens het gebruik van de queue de eindpunten `begin` en `end` door de array heen en weer bewegen. Bekijk als voorbeeld een array met 10 elementen (genummerd van 0 tot en met 9). Bij aanvang hebben zowel `begin` als `end` de waarde 0:



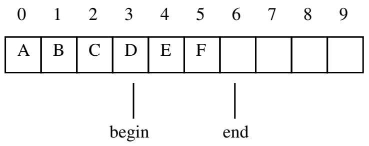
Na toevoegen van 6 elementen, in het plaatje aangeduid met A t/m F, zal de waarde van `end` 6 zijn geworden:



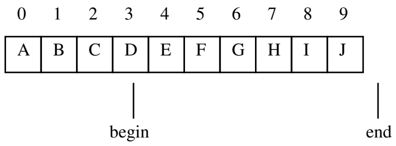
dit getal geeft immers de index net voorbij het eind van de queue. Als er vervolgens drie elementen van het begin van de queue worden verwijderd, ontstaat de volgende situatie:

```
template <class T>
class Queue
{
    T    **data;
    int  begin, end, lengte, maxlengte;
public:
         Queue        (int);
    int  empty        ()     {return lengte==0; };
    void addAtBegin   (T*);
    void addAtEnd     (T*);
    T*   getFromBegin ();
    T*   getFromEnd   ();
};

template <class T>
Queue<T>::Queue(int len)
{   maxlengte = len;
    data   = new T*[len];
    begin  = 0;
    end    = 0;
    lengte = 0;
}

template <class T>
void Queue<T>::addAtBegin(T *t)
{   if (begin==0) begin=maxlengte;
    begin--;
    data[begin] = t;
    lengte++;
}

template <class T>
void Queue<T>::addAtEnd( T *t)
{   data[end] = t;
    end++;
    if (end==maxlengte) end=0;
    lengte++;
}

template <class T>
T* Queue<T>::getFromBegin()
{   T *res  = data[begin];
    begin++;
    if (begin==maxlengte) begin=0;
    lengte--;
    return res;
}

template <class T>
T* Queue<T>::getFromEnd()
{   if (end==0) end=maxlengte;
    end--;
    lengte--;
    return data[end];
}
```

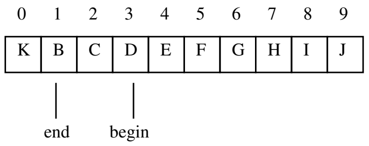
Listing 20: queue.cc



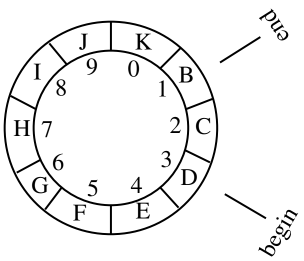
De elementen A, B en C laten we gewoon in de array staan, maar voor de queue tellen ze niet meer mee: de queue begint immers bij de indexwaarde `begin`. Na toevoegen van nog vier elementen bereikt `end` de rand van de array:



Toch is de queue nog niet vol, want de plaatsen 0, 1 en 2 van de array zijn nog vrij. Een volgend element kan als volgt worden toegevoegd:



Pas als het `end` een 'hele ronde' voorligt op `begin` is de queue vol. Het is misschien handiger om je de array als cirkel voor te stellen:



Het cyclisch gebruiken van de ruimte in de array doe je door bij ophogen van grenzen te controleren of de bovengrens is overschreden, en zonodig bij 0 te beginnen. Zo komt na 9 niet 10 maar 0. Op dezelfde manier moet je opletten bij het verlagen van grenzen: voor 0 komt niet -1 maar 9.

Als eerste bekijken we het toevoegen van een element aan het einde van de queue. De methode-header is:

```
template <class T>
void Queue<T>::addAtEnd( T *t)
{
```

Omdat de waarde van `end` de index is net voorbij het eind van de queue, is dit dus de index van de eerste vrije plaats. De nieuwe waarde kan op die plaats worden neergezet:

```
    data[end] = t;
```

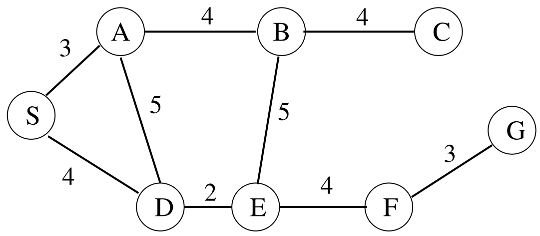
De `end`-grens moet een plaats opschuiven, en de cirkel dichtmaken (d.w.z. opnieuw bij 0 beginnen) als de rand van de array bereikt is:

```
    end++;
    if (end==maxlengte) end=0;
```

Tenslotte moet de lengte van de queue worden aangepast:

```
    lengte++;
}
```

Het toevoegen van een element aan het begin van de queue lijkt hier sterk op. Het verschil is dat `begin` niet naar een lege plaats wijst, maar naar een gebruikte plaats. De waarde moet dus *eerst* worden verlaagd (met controle op bereiken van de rand van de array):



Figuur 16.1: Voorbeeld van een `Net`

```
{   if (begin==0) begin=maxlengte;
    begin--;
```

Dan kan het element worden neergezet, waardoor vanzelf weer de gewenste eigenschap geldt dat `begin` het eerste element van de queue aanwijst:

```
    data[begin] = t;
    lengte++;
}
```

Het weghalen van elementen uit de queue verloopt op een vergelijkbare manier; het belangrijkste waar je op moet letten is dat het aanpassen van de grens en het pakken van de waarde in de goede volgorde gebeurt. Deze methoden zijn uitgewerkt in listing 20.

16.2 Klassen voor een zoek-netwerk

Niet alle dingen op een landkaart zijn voor het in de inleiding geschetste probleem van belang. In feite is het alleen belangrijk welke steden er zijn, welke steden verbonden zijn door wegen, en hoe lang die wegen zijn. Zo'n abstracte versie van een landkaart noemen we een netwerk, of kortweg een *net*. In het boek 'Artificial Intelligence' wordt een voorbeeld gegeven van zo'n net. Dit voorbeeld is overgenomen in figuur 16.1. De rondjes stellen steden voor en worden *nodes* genoemd. De nodes zijn verbonden door wegen, *links* genaamd. Bij de wegen staat een getal die de lengte van de weg aangeeft. In plaats van het woord 'lengte' wordt ook vaak het woord 'kosten' gebruikt; het getal hoeft namelijk niet de fysieke afstand tussen de steden aan te duiden, maar kan ook rekening houden met andere aspecten die het gebruik van zo'n weg al of niet aantrekkelijk maken (wegdek, kans op files, enzovoorts).

We gaan een bibliotheekje schrijven waarin voor de drie soorten objecten die een rol spelen een klasse wordt gedefinieerd: `Net`, `NetNode` en `NetLink`. In elke klasse worden variabelen gedeclareerd die voor het betreffende object van belang zijn. Het gaat om de volgende declaraties (deze klassedefinities zijn nog niet compleet, want er staan nog geen methoden bij):

```
class NetNode
{   char          *naam;
    List<NetLink> wegen;
};
class NetLink
{   NetNode       *doel;
    int           kosten;
};
class Net
{   List<NetNode> nodes;
};
```

De klasse `List` uit hoofdstuk 15 komt goed van pas. Een `NetNode` bestaat uit een naam en een lijst van (verwijzingen naar) wegen die ervandaan vertrekken. Een `NetLink` bevat een verwijzing naar de node waar de weg naartoe leidt, en een getal dat de kosten van het gebruik van de weg aangeeft. Een netwerk bestaat uit een lijst van (verwijzingen naar) nodes waaruit het is opgebouwd. De wegen worden niet apart opgeslagen in objecten van het type `Net`; ze maken er natuurlijk wel deel van uit, omdat ze via de component `wegen` van de afzonderlijke nodes bereikbaar zijn.

In elk van de drie klassen zal natuurlijk een constructor-methode worden gedefinieerd. Die van `Net` maakt een leeg netwerk, die van `NetNode` maakt een node met een gegeven naam zonder uitgaande

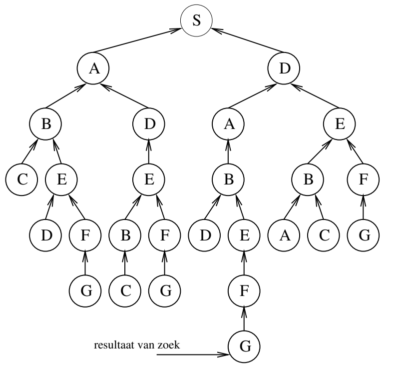
```
class NetNode;
class NetLink;
class Net;
class TreeNode;

class NetNode
{   friend          Net;
    friend          NetLink;
    char           *naam;
    List<NetLink>   wegen;
                    NetNode   (char*a)                 {naam=a; };
    void            addlink   (NetNode*, int);
public:
    void            print     (ostream &);
};

class NetLink
{   friend          NetNode;
    friend          Net;
    NetNode        *doel;
    int             kosten;
                    NetLink   (NetNode*a, int b)       {doel=a; kosten=b; };
public:
    void            print     (ostream &);
};

class Net
{
    List<NetNode>   nodes;
    NetNode        *findnode  (char*);
public:
                    Net       ()                       {};
    void            addnode   (char*);
    void            addlink   (char*, char*, int);
    void            print     (ostream&);
    TreeNode       *zoek      (char*, char*);
};

class TreeNode
{   friend          Net;
    NetNode        *netnode;
    TreeNode       *parent;
public:
                    TreeNode (NetNode *n,TreeNode *p) {netnode=n; parent=p; };
    void            print    (ostream &);
    int             contains (NetNode*);
};
```

Listing 21: net.h

wegen, en die van **NetLink** maakt een weg naar een gegeven node met gegeven kosten.

In elke klasse definiëren we een methode `print`. Daarmee zijn objecten op elk moment af te drukken. Vooral tijdens het testen van het programma komt zo'n methode vaak goed van pas; het is dan ook een goede gewoonte om in elke klasse zo'n methode te definiëren. Parameter van de methode `print` is steeds de stream waarop de uitvoer geschreven moet worden.

De constructor-methoden zijn niet voldoende om elk gewenst netwerk op te bouwen: ze maken een leeg netwerk, respectievelijk een node zonder wegen. We maken daarom nog een paar methoden



Figuur 16.2: Voorbeeld van een tree met `TreeNode`s

waarmee het netwerk uitgebreid kan worden. In de klasse `Net` zijn dat:

```
void addnode (char*);
void addlink (char*, char*, int);
```

Hiermee kan een node (met gegeven naam), respectievelijk een link tussen twee nodes (waarvan de namen gegeven zijn) aan het netwerk worden toegevoegd. Deze twee methoden zijn publiek; het is de bedoeling dat ze vanuit het hoofdprogramma worden aangeroepen om een netwerk op te bouwen.

Bij de methode `addlink` worden de namen van de twee te verbinden nodes opgegeven. Deze namen zijn hopelijk eerder toegevoegd als nieuwe node met `addnode`. Om de verwijzing naar het node-object te vinden bij een gegeven naam maken we nog een extra methode

```
NetNode *findnode (char*);
```

Deze methode is alleen voor intern gebruik door de andere methoden van `Net`; hij staat daarom in de privé-sectie.

De toegevoegde links moeten in de lijst van wegen van hun begin- en eindpunt worden opgeslagen. We maken daarom ook een methode `addlink` in de klasse `NetNode`:

```
void addlink (NetNode*, int);
```

waarmee de lijst van wegen vanuit een bepaalde node kan worden uitgebreid. Het is niet de bedoeling dat het hoofdprogramma deze methode op eigen houtje gaat gebruiken; deze methode wordt door `Net` gebruikt bij het uitbreiden van het netwerk. De methode `NetLink::addlink` is daarom privé, en alleen toegankelijk voor methoden van `Net`, die voor dit doel tot `friend` verklaard is.

In het met al deze methoden opgebouwde netwerk kan nu een pad tussen twee nodes gezocht worden. Daarvoor maken we nog een extra methode in `Net`:

```
TreeNode *zoek (char*, char*);
```

Deze methode krijgt de namen van twee nodes als parameter, en levert als resultaat een pad tussen die nodes (als die tenminste bestaat). Zoals in het boek 'Artificial Intelligence' is betoogd, kunnen mogelijke paden vanaf een bepaald startpunt worden beschreven door een boom-vormige datastructuur. Zo'n boom bestaat ook uit splitspunten, die echter niet verward moeten worden met de nodes van het netwerk. In figuur 16.2 is een boom getekend waarmee paden van node S naar een aantal andere nodes worden beschreven.

Om dit soort bomen op te bouwen maken we een nieuwe klasse `TreeNode`. Elk object van type `TreeNode` bevat een verwijzing naar een node van het netwerk, en een verwijzing naar een `TreeNode` eerder op het pad (hoger in de boom). Daar is er altijd maar één van. De verbindingen in de boom van boven naar beneden slaan we, anders dan door het plaatje in 'Artificial Intelligence' wordt gesuggereerd, niet op. Natuurlijk heeft ook de klasse `TreeNode` de nodige methoden: een constructor-methode, een methode `print`, en een methode `contains` waarmee bepaald kan worden of het pad dat in de betreffende `TreeNode` eindigt een bepaalde `NetNode` bevat.

In listing 21 zijn de klasse-definities van de vier benodigde klassen uitgewerkt.

```
char *kopie(char *s)
{   return strcpy( new char[strlen(s)+1], s );
}

void NetNode::addlink(NetNode *nn, int w)
{   wegen.add( new NetLink(nn,w) );
}

void Net::addnode(char *s)
{   nodes.add( new NetNode(kopie(s)) );
}

void Net::addlink(char *a, char *b, int k)
{   NetNode *na, *nb;
    na = findnode(a);
    nb = findnode(b);
    na -> addlink(nb, k);
    nb -> addlink(na, k);
}

NetNode* Net::findnode(char *s)
{   for (All<NetNode> x(nodes); x.valid(); x.next() )
        if ( strcmp( x.value()->naam, s)==0 )
            return x.value();
    return NULL;
}
```

Listing 22: netbuild.cc

16.3 Opbouwen van een zoek-netwerk

De schrijver van het hoofdprogramma wordt geacht een netwerk op te bouwen door aanroep van de methoden `addnode` en `addlink` van een `Net`-object. In listing 22 worden deze methoden gedefinieerd.

In een `Net`-object is een lijst van alle nodes opgeslagen. De methode `addnode` maakt om te beginnen een nieuwe node, door middel van de expressie `new NetNode` (met parameters). De waarde van de `new`-expressie is een verwijzing naar de nieuwe node; deze wordt aan de lijst nodes toegevoegd door de methode `add` aan te roepen van het object `nodes`. Bedenk dat `nodes` een `List`-object is (een `List<NetNode>` om precies te zijn), en dus een methode `add` kent om er een element aan toe te voegen:

```
void Net::addnode(char *s)
{   nodes.add( new NetNode(kopie(s)) );
}
```

Als naam van de nieuwe node wordt een kopie gebruikt van de naam die `addnode` als parameter krijgt. De kopie wordt gemaakt met de inmiddels welbekende functie `kopie`.

Het toevoegen van een nieuwe link is wat ingewikkelder. De methode `addlink` krijgt als parameter twee namen van te verbinden nodes:

```
void Net::addlink(char *a, char *b, int k)
```

Als eerste worden (verwijzingen naar) de bijbehorende nodes opgezocht met behulp van de nog te schrijven methode `findnode`:

```
NetNode *na, *nb;
na = findnode(a);
nb = findnode(b);
```

Van de aldus gevonden nodes kan de methode worden aangeroepen die een weg toevoegt naar de andere node:

```
void NetNode::print(ostream &str)
{   str << naam;
}

void NetLink::print(ostream &str)
{   str << kosten << ": " ;
    doel->print(str);
}

void Net::print(ostream &str)
{   for (All<NetNode> node(nodes); node.valid(); node.next() )
    {   node.value()->print(str);
        str << endl;
        for (All<NetLink> link(node.value()->wegen); link.valid(); link.next() )
        {    link.value()->print(str);
             str << endl;
        }
    }
}

void TreeNode::print(ostream &str)
{   if (parent!=NULL)
    {   parent->print(str);
        str << " - " ;
    }
    netnode->print(str);
}
```

Listing 23: netprint.cc

```
na -> addlink(nb, k);
nb -> addlink(na, k);
```

hiermee wordt in beide nodes de weg naar de andere node toegevoegd aan de lijst van uitgaande wegen. De methode `NetNode::addlink` werkt op een zelfde manier als `Net::addnode`. Er wordt een nieuwe link gecreëerd, en die wordt toegevoegd aan de lijst van wegen die in elk `NetNode`-object wordt bijgehouden.

In listing 22 staat tenslotte de definitie van `Net::findnode`. Met deze methode kan de node gevonden worden met een bepaalde naam. De methode gebruikt een `All`-object om alle elementen van de lijst nodes te bekijken. Van elk element wordt met de string-vergelijkfunctie `strcpy` de naam vergeleken met de gezochte naam. Is de naam gelijk, dan wordt direct een verwijzing naar de bijbehorende node geretourneerd. Mocht de naam onverhoopt nergens worden aangetroffen, dan is de resultaatwaarde `NULL`, de verwijzing 'naar nergens'.

16.4 Printen van een zoek-netwerk

In listing 23 worden de `print`-methoden gedefinieerd voor de vier klassen. De methoden hebben steeds een stream als parameter die gebruikt wordt om het resultaat naar toe te sturen. De definities zijn verder niet moeilijk. Een node wordt afgedrukt door de naam naar de stream te sturen.

Een link wordt afgedrukt door de kosten ervan als getal te tonen, gevolgd door de node waar de link naar toe voert. Voor dat laatste wordt meteen al de zojuist gedefinieerde `NetNode::print` gebruikt.

Een compleet `Net` kan worden afgedrukt door alle nodes af te drukken, elk met een overzicht van de uitgaande wegen. Met een `All`-object worden daartoe alle nodes langsgelopen, en bij elke node worden, ook weer met een `All`-object, alle wegen langsgelopen.

```
int TreeNode::contains(NetNode *nn)
{   if (netnode==nn) return 1;
    if (parent!=NULL) return(parent->contains(nn));
    return 0;
}

TreeNode *Net::zoek(char *s, char *g)
{   NetNode *ns, *ng;
    TreeNode *tn, *ntn;
    Queue<TreeNode> queue(200);

    ns = findnode(s);
    ng = findnode(g);

    tn = new TreeNode(ns,NULL);
    queue.addAtBegin( tn );

    while (!queue.empty())
    {
        tn = queue.getFromBegin();
        if (tn->netnode==ng)
            return tn;

        for (All<NetLink> link(tn->netnode->wegen);
             link.valid(); link.next()             )
        {
            if (!tn->contains(link.value()->doel))
            {   ntn = new TreeNode( link.value()->doel, tn );
                queue.addAtBegin(ntn);
            }
        }
    }
    return NULL;
}
```

Listing 24: netzoek.cc

Het afdrukken van een `TreeNode` is het leukste. Zoals uit figuur 16.2 blijkt, stelt elke treenode het eindpunt van een pad voor. Als zo'n treenode wordt afgedrukt, willen we het hele pad te zien krijgen. Daarom wordt in de methode `TreeNode::print` niet alleen de bijbehorende `NetNode` afgedrukt, maar ook het deel van het pad dat daar naartoe leidt. Dat deel is beschikbaar in de `parent`-component van de `TreeNode`. En omdat het type daarvan (een verwijzing naar) óók een `TreeNode` is, kunnen we daarvoor meteen al weer de methode `print` gebruiken, die we nu juist aan het schrijven zijn. Dat alles natuurlijk alleen als we nog niet bij het startpunt van het pad zijn aangeland, dus als `parent!=NULL`.

16.5 Zoeken in een zoek-netwerk

Het eigenlijke zoekwerk naar een pad tussen twee nodes in het netwerk gebeurt in de methode `Net::zoek`. Deze tekst van deze methode staat in listing 24.

Als parameter krijgt de methode de namen van twee nodes. Om te beginnen worden de bijbehorende nodes opgezocht met de methode `findnode`. De gebruikte zoekmethode is een regelrechte uitwerking van het algoritme op blz. 68 van 'Artificial Intelligence'. Als eerste wordt de wortel van de paden-boom geconstrueerd:

```
tn = new TreeNode(ns,NULL);
```

De bijbehorende node is `ns`, het startpunt in het netwerk. Omdat het de wortel van de paden-boom is, wordt bij de constructie `NULL` gezet op de plaats van de `parent`. De gecreëerde `TreeNode` wordt daarna in een daartoe aangemaakte `Queue` geplaatst:

```
Queue<TreeNode> queue(200);
queue.addAtBegin( tn );
```

Vervolgens wordt steeds één element uit de queue gehaald. Dit element is een `TreeNode`, en stelt dus een pad voor. Eindigt dit pad in de gewenste bestemming (`tn->netnode==ng`), dan kan het pad worden opgeleverd als resultaat. Anders is de treenode `tn` het eindpunt van een pad dat nog niet in de bestemming eindigt. Waar eindigt dit pad dan wel? Deze treenode staat vermeld in `tn->netnode`. Van dit netnode wordt de lijst van uitgaande wegen geïnspecteerd (`tn->netnode->wegen`). Dit is een lijst, waarvan de afzonderlijke elementen langsgelopen worden: `for (All<NetLink> link(tn->netnode->wegen);`. Van elke aldus gevonden link wordt de doelnode bekeken: `link.value()->doel`. Als die nog niet op het pad voorkomt, dan wordt er een nieuwe treenode gemaakt en aan het begin van de queue toegevoegd.

De tweede methode die in listing 24 wordt gedefinieerd is `TreeNode::contains`. Deze methode controleert of een bepaalde netnode al voorkomt in een pad dat gerepresenteerd wordt door de treenode. Mocht de netnode toevallig het eindpunt zijn van het pad, dan is het antwoord 'ja' (`return 1;`). Anders wordt de rest van het pad bekeken, door de methode `contains` opnieuw aan te roepen met als parameter `parent`. Het resultaat daarvan is dan tevens het eindresultaat. Is er geen parent meer, dan is het antwoord 'nee, de netnode werd niet op het pad aangetroffen' (`return 0;`).

16.6 Gebruikersinterface voor een zoek-netwerk

Met de definities van een paar klassen en de methoden daarin heb je nog geen werkend programma: er moet nog een hoofdprogramma `main` geschreven worden. In het hoofdprogramma vindt de communicatie met de gebruiker plaats. Het is de enige plaats waar gelezen en geschreven wordt (behalve dan in de `print`-methoden van de diverse klassen, maar die methoden worden alleen maar vanuit het hoofdprogramma aangeroepen). In listing 25 staat het hoofdprogramma van `netmain`.

In het programma wordt één netwerk gedeclareerd. Het programma accepteert regels van de gebruiker zoals:

```
stad Utrecht
stad Amsterdam
stad Hilversum
weg  Utrecht   Amsterdam  40
weg  Utrecht   Hilversum  20
weg  Hilversum Amsterdam  30
print
zoek Utrecht   Amsterdam
```

Als de regel met `stad` begint, dan wordt de betreffende stad aan het netwerk toegevoegd met behulp van `Net::addnode`. Als de regel met `weg` begint, dan wordt een link toegevoegd tussen de opgegeven steden met het opgegeven getal als kosten. Met het commando `print` kan de gebruiker het hele netwerk uitprinten. Het commando `zoek` tenslotte kan gebruikt worden om een weg te zoeken tussen twee steden. Daarna kunnen overigens nog nieuwe steden en wegen worden toegevoegd, en/of nieuwe zoek-opdrachten verstrekt worden.

Het programma vraagt niet netjes om z'n invoer, maar dat kan er desgewenst makkelijk aan worden toegevoegd. Het programma zal trouwens waarschijnlijk meestal gebruikt worden met de invoer-stream gekoppeld aan een file, waarbij het vragen om invoer niet noodzakelijk is.

In het programma wordt een nieuwe opdracht-vorm gebruikt: de `switch`-opdracht. Bij deze opdracht wordt de expressie tussen de haakjes achter `switch` uitgerekend. Vervolgens wordt de uitvoering van de opdrachten tussen de accolades erachter begonnen op het punt waar het woord `case` staat met erachter de waarde die de expressie had. De speciale opdracht `break` breekt het uitvoeren van de gehele `switch`-opdracht af, zodat niet ook de opdrachten achter de andere `case`-s uitgevoerd worden.

```
#include <iostream.h>
#include <string.h>
#include "lijst.cc"
#include "queue.cc"
#include "net.h"
#include "netprint.cc"
#include "netbuild.cc"
#include "netzoek.cc"

int main()
{
    char command[20], van[20], naar[20]; int kosten;
    Net net; TreeNode *tn;

    while (cin >> command)
    {
        switch(command[0])
        {
         case 's': cin >> van;
                   net.addnode(van);
                   break;

         case 'w': cin >> van >> naar >> kosten;
                   net.addlink(van,naar,kosten);
                   break;

         case 'p': net.print(cout);
                   break;

         case 'z': cin >> van >> naar;
                   tn = net.zoek(van,naar);
                   if (tn!=NULL)
                       tn->print(cout);
                   else
                       cout << "geen pad gevonden";
                   cout << endl;
        }
    }
    return 0;
}
```

Listing 25: netmain.cc

Opgaven

1. Is het zoekalgoritme *breadth-first* of *depth-first*? Verander de methode `Net::zoek` zodat het andere algoritme wordt gebruikt.
2. Breid het programma uit zodat elk pad dat tijdens het zoeken geprobeerd wordt geprint wordt. Test dit uit met zowel het breadth-first als het depth-first algoritme.

Hoofdstuk 17

Personen, *of:* subklassen

In dit hoofdstuk bekijken we hoe je door middel van *subklassen* eerder gedefinieerde klassen kunt uitbreiden met nieuwe componenten en methoden. Dit mechanisme wordt gebruikt voor de window-interfaces in de volgende hoofdstukken.

17.1 Subklassen

In de meeste voorbeelden die we tot nu toe zijn tegengekomen hadden klassen (of liever gezegd: de objecten die die klassen als type hebben) slechts enkele componenten en methoden. Daartoe hoeft het echter niet beperkt te blijven: in de praktijk kunnen objecten tientallen componenten hebben. Het is natuurlijk veel werk om zo'n klasse te definiëren, maar als dat eenmaal gebeurd is, is het gemak des te groter. Vaak zullen de klassen in een bibliotheek gedefinieerd zijn, klaar om te gebruiken in verschillende programma's.

Juist bij klassen met veel componenten dreigt echter het gevaar dat de klasse in de bibliotheek nèt niet biedt wat je nodig hebt. Soms is er de behoefte aan objecten met dezelfde componenten zoals beschreven in een bibliotheek-klasse, maar dan met één of twee componenten of methoden meer. Je zou natuurlijk een nieuwe klasse kunnen maken, daarin hetzelfde definiëren wat je in de bibliotheek aantreft, plus de extra componenten. Maar dit is niet zo'n slimme benadering, om twee redenen:

- Veel leveranciers van bibliotheken stellen de `.cc`-files van de bibliotheek niet beschikbaar, alleen de `.h`-files met de typen, en de gecompileerde versie van de bibliotheek. Je kunt dan dus niet 'spieken' hoe een bepaalde klasse in elkaar zit.
- Al een paar keer eerder in dit diktaat hebben we gezien: als je teksten moet overschrijven of kopiëren ben je verkeerd bezig; daar hebben we de computer nou juist voor.

Daarom is er een mechanisme in de taal waarmee *bestaande klassen uitgebreid* kunnen worden met nieuwe componenten en methoden. Zo'n uitgebreide klasse heet een *subklasse* van de originele klasse; omgekeerd heet de originele klasse de *superklasse* van de subklasse.

Behalve dat subklassen een handige notatie bieden om veel schrijfwerk te besparen, bieden subklassen nog iets extra's ten opzichte van helemaal overnieuw gedefinieerde klassen. Je mag objecten van subklassen namelijk op de plaats gebruiken (bijvoorbeeld als parameter) waar objecten van de superklasse verwacht worden. Daarover meer in paragraaf 17.3.

17.2 Voorbeeld van subklasse

Stel dat in een bibliotheek een klasse is gedefinieerd, in de objecten waarvan persoonsgegevens opgeslagen kunnen worden:

```
class Persoon
{
   char *naam;
   int inkomen;
public:
   Persoon(char*,int);
   void Print();
};
```

(heel erg veel componenten heeft deze klasse nog niet, maar dat is om deze tekst niet te lang te laten worden. Laat ons aannemen dat de klasse `Persoon` zo groot is, dat de lust om hem helemaal opnieuw te schrijven ons vergaan is. De methode `Print` zou bijvoorbeeld ingewikkelde lay-out problemen voor zijn rekening kunnen nemen.) Merk op dat de constructor-methode twee parameters heeft, waarmee bij constructie de twee componenten geïnitialiseerd kunnen worden.

In een bepaald programma is het noodzakelijk om gegevens van studenten op te slaan. Studenten hebben behalve een naam en een inkomen ook een studierichting die van belang kan zijn. We kunnen hiervoor een subklasse `Student` definiëren:

```
class Student : public Persoon
{
   char *studie;
public:
   Student(char*,char*);
   void Print();
};
```

De naam van de superklasse (`Persoon`) wordt genoemd achter de naam van de nieuwe klasse, daarvan gescheiden door een dubbele punt en het woord `public`. Objecten van de klasse `Student` hebben alle componenten van de superklasse (`naam` en `inkomen`), en daarnaast ook de nieuwe component `studie`. De klasse heeft een eigen constructor, en de methode `Print` zal opnieuw gedefinieerd worden. Van het recht om nieuwe methoden toe te voegen hebben we hier geen gebruik gemaakt.

Er is hier gekozen om de constructormethode van `Student` slechts twee parameters te geven. Hiermee worden bij creatie twee componenten geïnitialiseerd. De component `inkomen` is immers voor alle studenten dezelfde basisbeurs, en hoeft dus niet bij creatie van een student-object gespecificeerd te worden. De constructormethode voor `Student` kan als volgt gedefinieerd worden:

```
Student::Student(char* n, char *s) : Persoon( n, 470 )
{   studie = s;
}
```

Er is hier een nieuwe notatie gebruikt. In de header van de methode wordt aangegeven welke constructor voor wat betreft de superklasse gebruikt moet worden: de `n`(aam)-parameter wordt daarbij doorgespeeld aan de constructor van `Persoon`, terwijl voor het inkomen de vaste waarde `470` wordt gebruikt. De `s`(tudie)-parameter wordt gebruikt om de `studie`-component te initialiseren.

In de opnieuw gedefinieerde methode `Print` is nog een nieuwe notatie nodig:

```
void Student::Print()
{
    Persoon::Print();
    cout << studie;
}
```

Door in de body van de methode de aanroep `Print()` expliciet te kwalificeren met een klassenaam, wordt hiervoor niet de methode zelf recursief aangeroepen, maar wordt de methode `Print` zoals gedefinieerd in `Persoon` gebruikt. Dit is een handige manier waarop methodes kunnen worden 'uitgebreid' om de nieuwe componenten in aanmerking te nemen.

In het hoofdprogramma kan de nieuwe klasse worden gebruikt:

```
int main()
{
   Persoon jo("ritzen", 14010 );
   Student st("jansen", "cki");
   jo.Print();
   st.Print();
}
```

Dit programma declareert twee objecten: een persoon-object `jo` met de naam `"ritzen"` en een maandinkomen van 14010 (schaal 18 bezoldigingsbesluit rijksambtenaren), en een student-object `st` met de naam `"jansen"` en de studie `"cki"`, die bij creatie automatisch een riante basisbeurs toebedeeld krijgt. Van beide objecten wordt de methode `Print` aangeroepen. Door het type weet de compiler welke methode in de twee gevallen gebruikt moet worden.

17.3 Gebruik van subklasse-objecten

Functies kunnen een object als parameter hebben. Als type fungeert de naam van de klasse. Deze functie `PrintRegel` heeft bijvoorbeeld een `Persoon`-object als parameter:

```
void PrintRegel( Persoon &p )
{
    p.Print();
    cout << endl;
}
```

Deze functie roept van de als parameter meegegeven persoon de `Print`-methode aan, en schrijft daarna nog een nieuwe regel – een erg nuttige functie dus.

De functie kan worden aangeroepen met `jo` als parameter; vanzelfsprekend, want `jo` is een `Persoon`.

```
main()
{
    ...
    PrintRegel( jo );
    PrintRegel( st );
}
```

De functie kan echter óók worden aangeroepen met `st` als parameter. De student `st` is immers ook een persoon. Voor deze gelegenheid doet ze even afstand van haar student-kenmerken, en is daarmee acceptabel voor `PrintRegel`. Objecten uit een subklasse hebben immers alle componenten uit de superklasse; de extra componenten zitten niet in de weg en worden door de functie gewoon niet gebruikt.

Andersom is het niet mogelijk. Een functie met een `Student`-parameter, die mogelijkerwijs gebruik maakt van de `studie`-component daarvan, zal `jo` niet als parameter accepteren.

17.4 Virtuele methoden

In dit voorbeeld is er nog een extra complicatie. De functie `PrintRegel` roept van zijn `Persoon`-parameter de `Print`-functie aan. Welke methode is dat, wanneer `st`, in haar hoedanigheid van `Persoon`, als parameter aan `PrintRegel` wordt meegegeven?

Bij het beantwoorden van deze vraag kijkt de compiler normaliter naar het type van het object waarbij de methode wordt aangeroepen. In de body van `PrintRegel` staat de aanroep

```
p.Print();
```

De declaratie van `p` is ondubbelzinnig een `Persoon`. De originele `Print`-functie uit de bibliotheek zal dus worden aangeroepen, ook wanneer voor `p` toevallig een `Student` wordt meegegeven.

In sommige gevallen is dit niet de gewenste aktie; je zou misschien willen dat in het geval dat `PrintRegel` met `jo` wordt aangeroepen, de `Print`-methode van `Persoon` wordt gebruikt, maar als `PrintRegel` met `st` wordt aangeroepen, de `Print`-methode van `Student`. Dat kan. Het moet dan echter worden aangegeven bij de declaratie van de methode `Print`, zowel in de superklasse als in de subklasse. Dat gaat met het gereserveerde woord `virtual` voor de header van de methode in de klasse-definitie:

```
class Student : public Persoon
{
   char *studie;
public:
   Student(char*,char*);
   virtual void Print();
};
```

Door het gebruik van `virtual` methoden ontstaat een leuke constructie. Een schrijver van een bibliotheek kan in een bibliotheek-functie (zoals `Print`) methoden aanroepen die op dat moment nog helemaal niet bestaan, maar die later in een door een andere programmeur te schrijven subklasse gedefinieerd zullen worden. Dit verklaart ook het woord *virtual*, dat hier in de betekenis 'latent' gebruikt wordt.

Virtuele methoden worden veel gebruikt in window-systemen. In een bibliotheek kan een klasse gedefinieerd waarvan de objecten lege windows voorstellen (dat is moeilijk genoeg, want windows moeten verplaatst kunnen worden, kunnen elkaar half bedekken, enzovoorts). De programmeur van een programma waarin ook niet-lege windows nodig zijn kan een subklasse definiëren, waarin de nodige virtuele functies opnieuw gedefinieerd worden.

In het volgende hoofdstuk bespreken we een window-bibliotheek die op deze manier gebruikt kan worden. Het feit dat virtuele methoden daarin zo mooi gebruikt kunnen worden, is de voornaamste reden dat we dit ingewikkelde onderwerp in dit hoofdstuk besproken hebben.

Opgaven

1. Eerder in dit diktaat zijn we al een voorbeeld tegengekomen van een subklasse (zonder dat met zoveel woorden te noemen). Je kon het herkennen aan het feit dat een object van een subklasse werd geaccepteerd als parameter, op de plaats waar een object van een superklasse werd verwacht. Welk voorbeeld was dat?

Hoofdstuk 18

Lijnen, *of:* window input/output

In alle programma's tot nu toe verliep de interactie met de gebruiker via de streams `cin` en `cout`, en eventueel via streams die aan files waren gekoppeld. Vrijwel alle computersystemen kunnen echter ook met een muis bediend worden, waarbij de uitvoer in een grafisch scherm verschijnt. Dit vereist een andere manier van programmeren. In dit hoofdstuk wordt een bibliotheek besproken waarmee grafische invoer en uitvoer mogelijk is. Met gebruik van deze bibliotheek schrijven we een programma waarmee de gebruiker met de muis een lijntekening kan maken in een window.

18.1 Windowsystemen

Programma's in C++ zijn onafhankelijk van de machine waarop ze uitgevoerd worden. Als je een programma op geschreven hebt op een Unix-systeem, kun je het door het opnieuw te compileren ook gebruiken op een PC of op een Mac. De taal is gestandaardiseerd en op alle machines hetzelfde; het enige wat kan verschillen is de precisie van integers en dergelijke. Ook veel bibliotheken zijn standaard; zo zijn `<iostream.h>`, `<math.h>` en `<string.h>` overal te gebruiken. Andere bibliotheken zijn specifiek voor een bepaalde machine. Op Unix-machines zijn bijvoorbeeld bibliotheken aanwezig om te communiceren tussen parallel verlopende processen, en op PC's zijn er specifieke bibliotheken voor het werken met geluid.

Op veel computers kan een window-systeem gebruikt worden, waarbij met de muis dingen op het scherm aangewezen en gemanipuleerd kunnen worden. Op veel Unix-computers wordt een window-systeem genaamd 'X' gebruikt, PC's gebruiken Microsoft Windows of ook 'X' onder Linux, en de Mac, Atari en Amiga hebben ieder hun eigen windowsysteem. Hoewel de verschillende windowsystemen veel kenmerken gemeen hebben, worden ze aangesproken via machine-specifieke bibliotheken. Het gebruik van deze bibliotheken is vaak niet eenvoudig; zelfs voor zeer simpele programma's zijn vaak al ingewikkelde constructies nodig.

We kunnen hier niet al deze window-systemen gaan bespreken. Zelfs als we ons zouden beperken tot één machine kan daar een cursus op zich aan gewijd worden. Het leuke van C++ is echter dat je een bibliotheek kunt schrijven waarin de meeste details weggewerkt kunnen worden. In de bibliotheek `"simwin.h"` ('simpele windows') is dat gebeurd. Deze bibliotheek is verre van standaard: hij is speciaal voor deze cursus ontwikkeld. Het voordeel ervan is echter dat hij zeer klein en overzichtelijk is, en dat toch vele aspecten van programmeren in een window-omgeving ermee gedemonstreerd worden. De bibliotheek maakt achter de schermen gebruik van een machine-specifieke bibliotheek.

Wie de simwin bibliotheek goed begrijpt, zal minder moeite hebben om in voorkomende gevallen een machine-specifieke bibliotheek te doorgronden, vooral als in die bibiotheek net als in simwin een object-georiënteerde benadering wordt aangehouden, zoals de Object Windows bibliotheek behorend bij de C++-compiler van Borland voor een PC onder Microsoft Windows.

18.2 Interactie met een windowsysteem

In programma's die streams en files gebruiken ligt het initiatief grotendeels bij de computer. De computer voert opdrachten in het programma uit. Af en toe wordt dit onderbroken om te wachten op invoer. De gebruiker mag dan even wat tekst invoeren, en daarna gaat het programma weer zijn eigen gang.

In een muis/window programma zijn de rollen omgedraaid. De gebruiker heeft het initiatief, en kan op elk moment beslissen om met de muis een punt op het scherm aan te klikken, een toets op

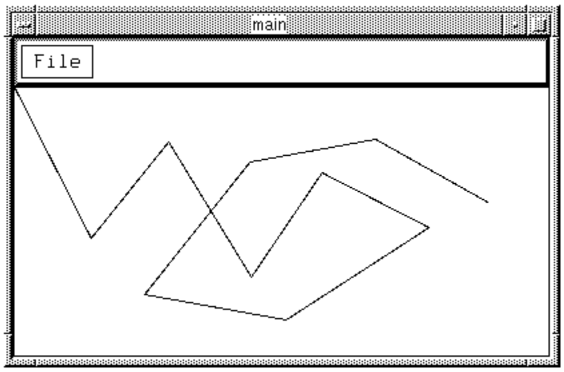
```
class Grafisch
{
public:
     int breedte, hoogte;

     Grafisch(int br   // breedte
             ,int h    // hoogte
             ,char ** =NULL
                       // menu-items (strings beginnend met # zijn titel)
             ,int r=1  // r==1: dingen worden automatisch getekend
                       // r==0: er wordt pas getekend als je Teken() aanroept
             );

     // Deze functies kun je aanroepen vanuit je programma

     void Run();
     void DefinieerKleur( int i, int r, int g, int b );  // i<100, rgb<256
     void ZetKleur(int i);
     void LijnDikte(int);
     void Lijn(int,int,int,int);
     void Punt(int,int);
     void Rechthoek(int,int,int,int);
     void Tekst(int,int,char*);
     void Tekst(int,int,int);
     void Schoon();

     // Deze functies kun je herdefinieren in een subklasse van Grafisch

     virtual void Klik(int,int);
     virtual void Toets(char);
     virtual void Menu(char*);
     virtual void Teken();
     virtual void Idle();
};

class StringDialog
{
public:
            Grafisch *grafisch;
            char     *waarde;
                      StringDialog (Grafisch&,char*, char*);
                     ~StringDialog ();
    virtual void      apply        ();
    virtual void      close        ();
};
```

Listing 26: Simwin.h

het toetsenbord in te drukken, of misschien iets uit een dropdown-menu te kiezen. Het programma wordt geacht daar direct op te reageren.

We kunnen het ons dus niet meer permitteren om het programma gedurende langere tijd te laten rekenen; elk moment moet het programma klaarstaan om nieuwe invoer te verwerken. Het programma moet echter ook weer niet lijdzaam op invoer gaan wachten, zoals dat bij `cin>>x` het geval is. Hoe moet het dan wel?

In listing 26 staat de definitie van een klasse `Grafisch`, althans van het publieke gedeelte daarvan. In het hoofdprogramma moet een object uit deze klasse worden gedeclareerd. Als parameter van de constructor-methode moeten de breedte en de hoogte van het hoofdwindow van het programma



Figuur 18.1: Het programma lijnen in aktie

opgegeven worden. In de klasse zijn methoden aanwezig om dingen op het window te tekenen: `Punt`, `Lijn`, `Rechthoek` en `Tekst`. Voorlopig mogen deze echter nog niet aangeroepen worden; de eerste methode die gebruikt moet worden is `Run`. Door aanroep van deze methode wordt het initiatief overgedragen aan het windowsysteem. De methode eindigt nooit; de aanroep van `Run` is dus het laatste wat in het hoofdprogramma gebeurt. Gelukkig kan het programma het initiatief ook wel weer eens terugkrijgen.

Als reactie op acties van de gebruiker zal de methode `Run` op zijn beurt namelijk andere methoden aanroepen. Welke methoden dat zijn, is afhankelijk van de acties van de gebruiker: als reactie op het klikken van de muis zal de methode `Klik` worden aangeroepen, op het indrukken van een toets volgt aanroepen van de methode `Toets`, enzovoort. Als er echt helemaal niets anders te doen is wordt de methode `Idle` aangeroepen.

De methoden `Klik` en `Toets` en dergelijke zijn `virtual` methoden. Dat betekent dat ze *in een subklasse van* `Grafisch` *hergedefinieerd* kunnen worden. En dat is precies wat de programmeur moet doen. In een programma waarin bijvoorbeeld muiskliks van belang zijn, definiëer je een subklasse van `Grafisch`, waarin de methode `Klik` op een zinvolle manier wordt ingevuld. Als reactie op zo'n klik kun je dan bijvoorbeeld iets in het window tekenen. Na voltooiing van de methode `Klik` krijgt de methode `Run` het initiatief weer terug, totdat deze bij een volgende actie van de gebruiker weer een van de vijf virtual methoden aanroept.

18.3 Lijnen trekken

Een voorbeeld van het gebruik van simwin is het programma lijnen. In listing 27 staat de tekst van dit programma. In figuur 18.1 is te zien hoe gebruik van het programma er uitziet. Het programma creëert een window; in dit window worden als reactie op muiskliks van de gebruiker rechte lijnen getrokken, te beginnen in de linkerbovenhoek.

Het programma begint met een `#include`-regel waarmee de simwin-bibliotheek gebruikt wordt. De bibliotheek `<iostream.h>` is niet eens meer nodig, omdat in het programma geen 'klassieke' uitvoer gepleegd wordt. In het programma wordt een subklasse `Lijnen` van de klasse `Grafisch` gedefinieerd. In het hoofdprogramma `main` wordt er één object uit die klasse gedeclareerd. Van dit object wordt direct de methode `Run` aangeroepen.

Als extra variabelen worden in de klasse `Lijnen` twee integers gedeclareerd, `x0` en `y0`. Samen duiden deze het 'huidige punt' aan. Verder is er natuurlijk een constructor-methode `Lijnen`, en wordt de virtuele methode `Klik` hergedefinieerd. De constructormethode van `Lijnen` speelt de binnenkomende breedte en hoogte van het window direct door aan de constructormethode van `Grafisch`. Daarnaast worden `x0` en `y0` van een beginwaarde voorzien.

De methode `Klik` zal aangeroepen worden als reactie op het klikken van de muis. In dat geval wordt met de methode `Lijn` een lijn in het window getekend, waarna het 'huidige punt' wordt aangepast.

```
#include "simwin.h"

class Lijnen : public Grafisch
{
   int x0, y0;
public:
   Lijnen(int,int);
   virtual void Klik(int,int);
};

Lijnen::Lijnen (int br, int h) : Grafisch(br,h)
{
    x0 = 0;
    y0 = 0;
}

void Lijnen::Klik(int x1, int y1)
{
    Lijn(x0, y0, x1, y1);
    x0 = x1;
    y0 = y1;
}

int main(char **, int )
{
    Lijnen t(400,200);
    t.Run();
}
```

Listing 27: lijnen.cc

18.4 Separate compilatie

In de hoofdstukken 14–16 werd de tekst van zelfgeschreven bibliotheken met `#include`-regels in het programma opgenomen, bijvoorbeeld

```
#include "netprint.cc"
#include "netbuild.cc"
#include "netzoek.cc"
```

Dat zou met de bibliotheek simwin ook kunnen. De bibliotheek is echter nogal groot, en het is zonde van de tijd om hem elke keer opnieuw te compileren als het hoofdprogramma veranderd wordt.

Bibliotheken kunnen daarom apart worden gecompileerd. De gecompileerde versie van `simwin.cc` staat in de file `simwin.o`. In het hoofdprogramma hoeft nu niet meer de bibliotheek `"simwin.cc"` opgenomen te worden, maar alleen de klasse-definities daaruit: `"simwin.h"`. Het hoofdprogramma wordt ook apart gecompileerd, en daarna worden de twee resulterende files samengevoegd tot één programma.

De Unix-commando's die gegeven moeten worden om files apart te compileren, en ze daarna samen te voegen waarbij dan ook nog de bibliotheken meegenomen worden die simwin op zijn beurt gebruikt, zijn nogal ingewikkeld. Omdat het vervelend is die elke keer in te tikken, is dit geautomatiseerd. Daartoe heb je een file `Makefile` nodig, waarin staat gespecificeerd hoe programma's aan elkaar geknoopt moeten worden.

De `Makefile` die bij simwin beschikbaar is, maakt dit wel erg makkelijk. Je moet een hoofdprogramma maken, laten we zeggen `hoofd.cc`, waarin de bibliotheek `"simwin.h"` wordt gebruikt. Zorg dan dat de `Makefile` aanwezig is in dezelfde directory als `hoofd.cc`. Geef vervolgens het Unix-commando:

```
make hoofd
```

Er wordt dan, door een aantal malen `CC` te gebruiken, volautomatisch een programma `hoofd` gemaakt, dat uitgevoerd kan worden. Het programma `hoofd` komt dus in plaats van de file `a.out` die meestal het resultaat is van compilatie.

Opgaven

1. Breid het programma lijnen in hoofdstuk 18 uit, zodat met een toetsaanslag de dikte en kleur van de lijnen ingesteld kan worden: de toetsen 1 t/m 9 moeten lijnen veroorzaken met die dikte. De toetsen 'Z', 'R', 'G', 'B', 'Y', 'M', en 'C' moeten zwarte, rode, groene, blauwe, gele, magenta (paars/roze) en cyane (groenachtig blauw) lijnen tot gevolg hebben.

   Aanwijzing: standaard zijn de kleuren in simwin als volgt opgebouwd. Laat x de volgende kleur-intensiteiten aanduiden: 0=geen kleur, 1=donker, 2=midden, 3=intens; dan is $1 * x$ de kleur blauw in de betreffende intensiteit, $4 * x$ groen, en $16 * x$ rood. Kleuren kunnen worden gemengd door de getallen op te tellen. Zo is bijvoorbeeld rood+groen=geel. Alle kleuren samen geeft wit.

2. Breid het programma lijnen nog eens uit, zodat alle getekende lijnen ook worden opgeslagen in een lijst, zodat bij verlaten van het programma met het File–Quit menu de coordinaten daarvan op `cout` geschreven kunnen worden. Herdefinieerd daartoe de virtuele methode `Menu`.

Hoofdstuk 19

Mandelbrot, *of:* mooie plaatjes

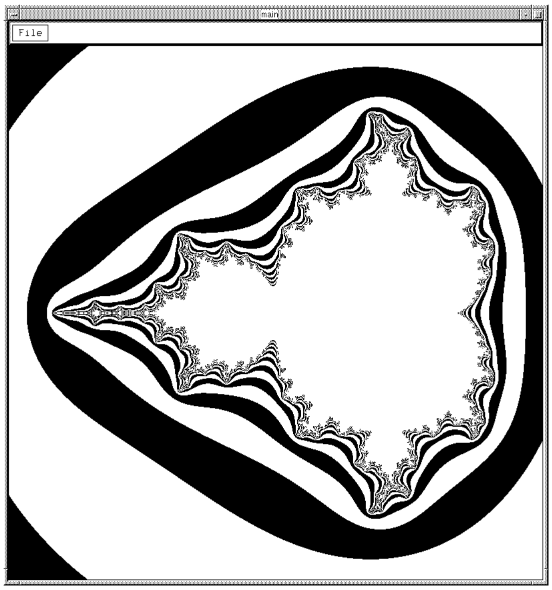
In dit hoofdstuk bekijken we een uitgebreider voorbeeld van een grafisch programma. Het programma mandel tekent de Mandelbrot-verzameling, en geeft de gebruiker gelegenheid om op bepaalde delen van het plaatje in te zoomen.

19.1 Het programma

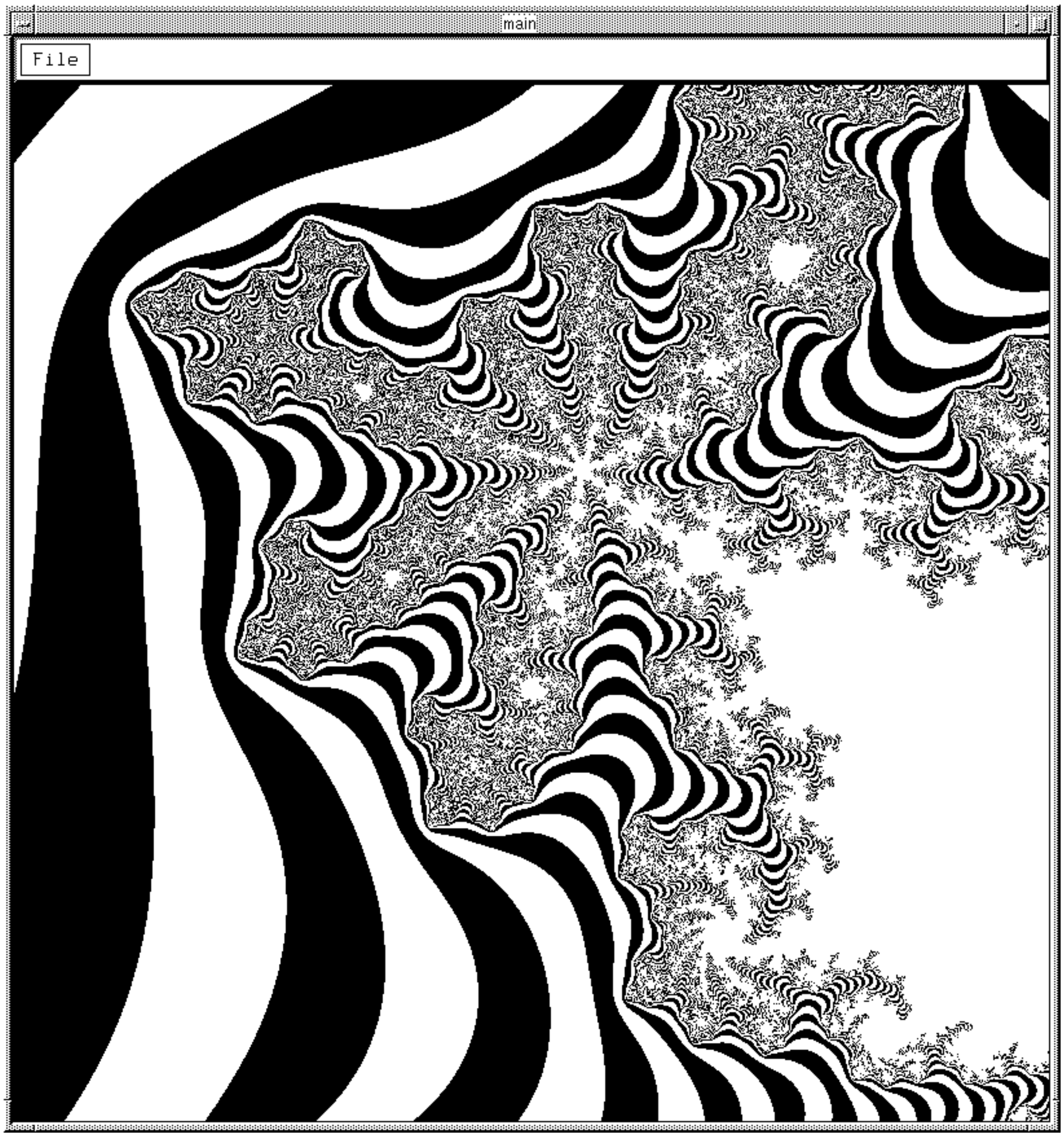
In het programma lijnen uit het vorige hoofdstuk bepaalde de gebruiker steeds wat er gaat gebeuren: elke keer als hij/zij op de muis klikt, tekent het programma als reactie een lijn. Vervolgens gebeurt er niets, tot de volgende actie van de gebruiker.

In dit hoofdstuk bekijken we een programma waarin nog van alles gebeurt, ook als de gebruiker passief is. Het programma heet mandel, omdat het de Mandelbrot-verzameling tekent. De Mandelbrot-verzameling is een voorbeeld van een *fractal*: een figuur waarin een bepaald patroon zich steeds op kleinere schaal herhaalt. De Mandelbrot-verzameling is genoemd naar de Frans-Amerikaanse wiskundige Benoit Mandelbrot, die deze figuur in 1979 voor het eerst construeerde. In figuur 19.1 is het programma in aktie te zien; de figuur toont tevens hoe de Mandelbrot-verzameling er ongeveer uitziet.

Het tekenen van de figuur kost het nodige rekenwerk. Bij het uitvoeren van het programma zal de figuur langzaam opgebouwd worden: de gebruiker ziet hem van links naar rechts getekend worden. Toch heeft het programma tijdens het tekenen niet alleen het initiatief; de gebruiker kan op elk moment tijdens het tekenproces ingrijpen, door met de muis op een bepaald punt te klikken. Dat is een signaal dat hij op dat punt wil *inzoomen*, om een deel van de figuur nauwkeuriger te bekijken. Op deze manier ontstaan de figuren 19.2 en 19.3.



Figuur 19.1: Mandelbrot-figuur, $x_0 = -0.75$, $y_0 = 0$, $b = 3$, $maxk = 20$



Figuur 19.2: Mandelbrot-figuur, $x_0 = -0.675$, $y_0 = -0.445$, $b = 0.1$, $maxk = 60$

Het is typerend voor muisgestuurde programma's dat de gebruiker op elk moment interactie kan plegen, en daarmee dus niet hoeft te wachten tot de hele figuur getekend is. Dat heeft consequenties voor de manier waarop het programma is opgebouwd; we kunnen niet rustig de hele figuur tekenen, maar moeten steeds rekening houden met de mogelijkheid dat er de gebruiker nieuwe wensen heeft.

19.2 De Mandelbrot-figuur

In de Mandelbrot-figuur is een deel van de twee-dimensionale ruimte $\mathbf{R}^2$ afgebeeld. In figuur 19.1 is het middelpunt van de figuur het punt $(-0.75, 0)$ en de breedte van het plaatje 3. De x-waarden lopen dus van -2.25 tot 0.75 en de y-waarden van -1.5 tot 1.5. Elk punt van het vlak krijgt een bepaalde *kleur*. In een zwartwit-plaatje is dat zwart of wit, maar je kunt ook een kleuren-afbeelding maken waarin meerdere kleuren gebruikt worden.

Welke kleur een punt (x, y) krijgt wordt bepaald door de functie `Kleur` in listing 28. De functie gebruikt twee variabelen `a` en `b` om een ander punt aan te duiden. Aan het begin is $(a, b) = (0, 0)$:

```
a = 0.0;
b = 0.0;
```

Vervolgens wordt het punt (a, b) op een nieuw punt afgebeeld volgens een bepaalde formule:

```
#include <iostream.h>
#include "simwin.h"
#include "mandelbrot.h"
#include "mandelbrot.cc"

int Kleur(double x, double y, int maxk)
{
    double a, b, newa, newb;
    int k;

    a = 0.0;
    b = 0.0;
    k = 0;

    while (a*a+b*b<16 && k<=maxk)
    {
        newa = a*a - b*b + x;
        newb = 2*a*b + y;
        a = newa;
        b = newb;
        k = k+1;
    }
    return k%256;
}

int main(char **, int )
{
    Mandelbrot t(800,800);
    t.Run();
}
```

Listing 28: mandel.cc

```
int Kleur(double,double,int);

class Mandelbrot : public Grafisch
{
    double x, x0, y0, interval, stap;
    int xp, maxk;
public:
            Mandelbrot   (int,int);
    virtual void Klik    (int,int);
    virtual void Idle    ();
            void Opnieuw ();
};
```

Listing 29: mandelbrot.h

```
newa = a*a - b*b + x;
newb = 2*a*b + y;
```

Deze formule is afhankelijk van x en y, het punt waarvan we de kleur willen bepalen. De transformatie van het punt (a, b) wordt steeds opnieuw toegepast. Daarbij telt de variabele `k` hoe vaak dat moet gebeuren voordat het punt (a, b) afstand > 4 tot de oorsprong heeft. Dat is het geval als $\sqrt{a^2 + b^2} > 4$ of anders gezegd $a^2 + b^2 > 16$. Het is niet gezegd dat dit ooit het geval zal zijn. Om niet in een oneindig durende herhaling te vervallen, geeft de functie het op als het al te lang gaat duren. Het aantal stappen dat maximaal geprobeerd wordt is de parameter `maxk` van de functie.

```
Mandelbrot::Mandelbrot(int w, int h) : Grafisch(w,h,NULL,0)
{   x0 = 0.0;
    y0 = 0.0;
    Opnieuw();
}

void Mandelbrot::Opnieuw()
{   cout << "lengte interval: ";  cin >> interval;
    cout << "maxk: "   ;  cin >> maxk;
    stap = interval / breedte;
    x = x0 - interval/2;
    xp = 0;
    Schoon();
    cout << "x0: " << x0 << "  y0: " << y0 << "  lengte: " << interval << endl;
}

void Mandelbrot::Klik(int x1, int y1)
{   x0 += (x1-breedte/2) * stap;
    y0 += (y1-breedte/2) * stap;
    Opnieuw();
}

void Mandelbrot::Idle()
{   double y;
    int yp, k;

    if (xp<breedte)
    {    y = y0 - interval/2;
         for (yp=0; yp<hoogte; yp++)
         {   k = Kleur(x,y,maxk);
             ZetKleur(k);
             Punt(xp,yp);
             y += stap;
         }
         xp++;
         x += stap;
         if (xp%16==0) Teken();
    }
    else
    {    maxk *= 2;
         xp = 0;
         x = x0 - interval/2;
    }
}
```

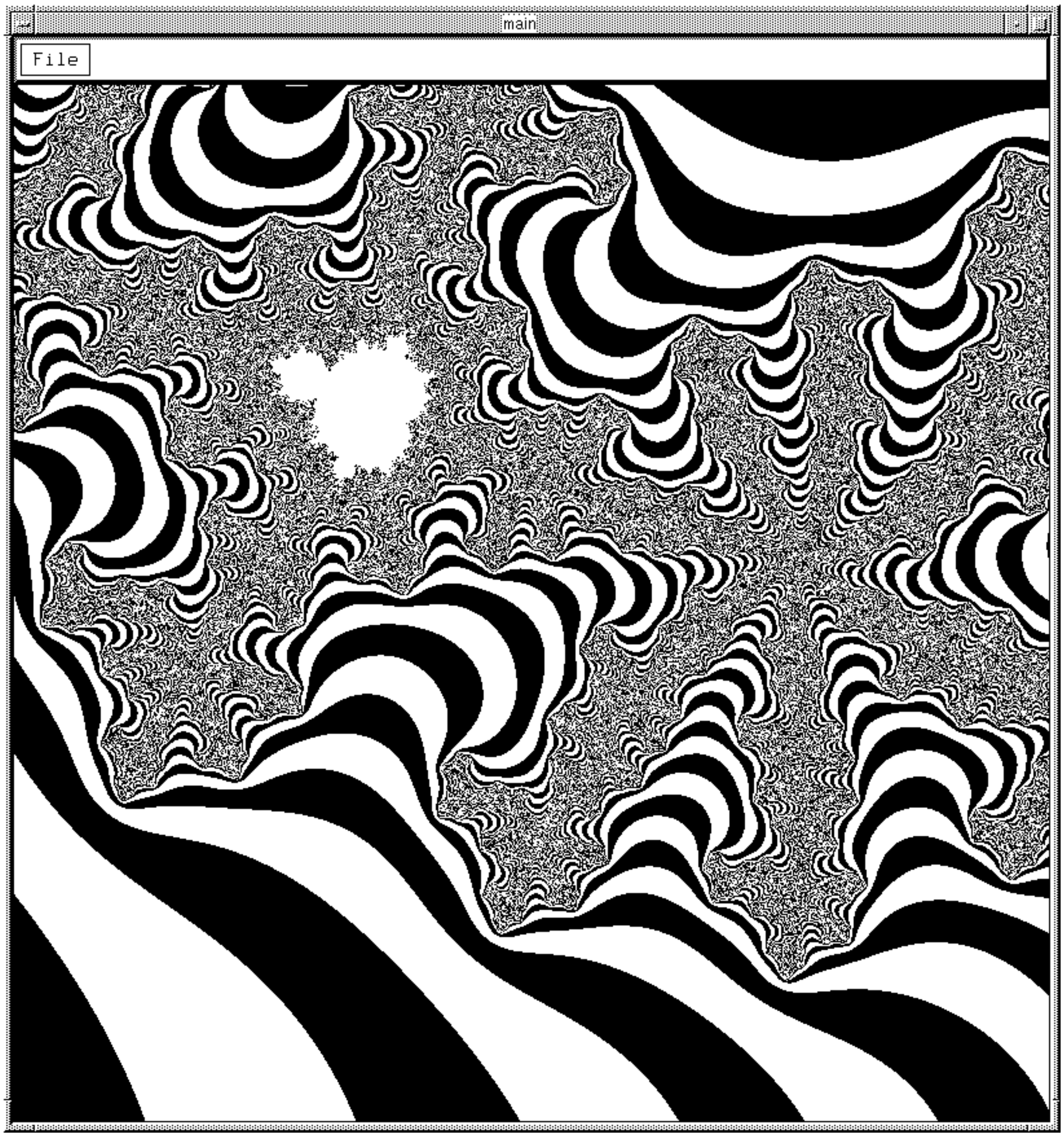
Listing 30: mandelbrot.cc

19.3 De klasse Mandelbrot

De Mandelbrot-figuur mag dan een deel van het reële vlak voorstellen, maar niet elk reëel punt kunnen we op het beeldscherm afbeelden. De punten in een window worden aangeduid met integers. Als we de Mandelbrot-figuur tussen $x = -2.25$ en $x = 0.75$ willen bekijken, moeten we dit interval afbeelden op integers tussen 0 en de breedte van het window.

In listing 29 wordt een klasse `Mandelbrot` gedefinieerd. Objecten uit deze klasse beschrijven een plaatje van een deel van de Mandelbrot-figuur. De klasse is een subklasse van `Grafisch`, zodat we ook methoden kunnen schrijven waarmee het plaatje daadwerkelijk getekend wordt.

In de klasse worden reële getallen (`double`) gedeclareerd voor het centrum van het plaatje (`x0` en



Figuur 19.3: Mandelbrot-figuur, $x_0 = -0.6895$, $y_0 = -0.4635$, $b = 0.01$, $maxk = 150$

y0) en de lengte van het af te beelden interval (interval). De variabele maxk bepaalt hoeveel pogingen de functie Kleur maximaal moet doen. Verder is er een variabele stap die aangeeft hoe groot de stap is tussen de punten die corresponderen met twee aangrenzende beeldpunten. Die variabele kan berekend worden uit de lengte van het interval interval en de variabele breedte uit de klasse Grafisch, die het aantal beeldpunten aangeeft:

```
stap = interval / breedte;
```

De figuur zou nu als volgt getekend kunnen worden:

```
int xp, yp, k;
double x, y;
for (xp=0; xp<breedte; xp++)
    for (yp=0; yp<hoogte; yp++)
    {   x = x0 - interval/2 + xp*stap;
        y = y0 - interval/2 + yp*stap;
        k = Kleur(x,y,maxk);
        ZetKleur(k);
        Punt(xp,yp);
    }
```

De methode Punt is een methode uit de klasse Grafisch waarmee een punt getekend wordt in de kleur die tevoren met `ZetKleur` ingesteld is.

De berekening van `x` hoeft eigenlijk niet voor elke waarde van `yp` uitgevoerd te worden. Bovendien kan de berekening van `y` nog wat efficiënter gebeuren, door de expressie `interval/2` slechts één keer uit te rekenen:

```
x = x0 - interval/2;
for (xp=0; xp<breedte; xp++)
{   y = y0 - interval/2;
    for (yp=0; yp<hoogte; yp++)
    {   k = Kleur(x,y,maxk);
        ZetKleur(k);
        Punt(x,y);
        y += stap;
    }
    x += stap;
}
```

Deze opdrachten moeten uitgevoerd worden om het complete plaatje te tekenen. Als we hier echter één functie van zouden maken, zou het tekenen door de gebruiker niet te onderbreken zijn. We maken daarom een functie die alle waarden van y doorloopt bij *één vaste* x. Deze functie tekent dus één beeldlijn van het plaatje. Het tekenen van een één beeldlijn gaat snel genoeg om als een ononderbroken geheel uit te voeren.

Beeldlijnen kunnen getekend worden als het programma niets beters te doen heeft (namelijk het reageren op invoer van de gebruiker). Voor deze situatie is een speciale virtual methode beschikbaar in `Grafisch`: de methode `Idle`. Deze methode wordt (door `Run`) elke keer aangeroepen als er geen andere gebeurtenissen (muiskliks, toetsenbord-invoer, menu-keuzes) opgetreden zijn. De methode `Idle` kan in subklassen van `Grafisch`, zoals `Mandelbrot`, hergedefinieerd worden om iets nuttigs te doen. Het is daarbij goed gebruik om niet teveel werk tegelijk te doen; het volgende beetje werk kan bij de volgende aanroep van `Idle` uitgevoerd worden.

In listing 30 wordt de methode `Idle` hergedefinieerd om één beeldlijn te tekenen. De variabelen `x` en `xp` kunnen nu niet meer lokaal gedeclareerd worden in de tekenfunctie. In plaats daarvan worden ze gedeclareerd in de klasse `Mandelbrot`. De variabelen in een `Mandelbrot`-object specificeren samen een plaatje-in-wording: `x0`, `y0`, `lengte` bepalen de uitsnede, `maxk` de detaillering, en `x` en `xp` tot hoever het werk gevorderd is. Door inspectie van deze zes variabelen weet de methode `Idle` precies wat hem te doen staat.

Als het hele plaatje getekend is zonder dat de gebruiker het heeft onderbroken, zou het programma op zijn lauweren kunnen gaan rusten en wachten tot de gebruiker aangeeft dat er ingezoomd moet worden. Om de tussentijd toch nuttig te gebruiken verdubbelt het programma echter de nauwkeurigheid `maxk`, en zet de variabelen `xp` en `x0` weer aan het begin:

```
else
{    maxk *= 2;
     xp = 0;
     x = x0 - interval/2;
}
```

Hiermee begint het tekenen van vooraf aan; de uitsnede is hetzelfde, maar door de grotere `maxk` worden meer 'contourlijnen' onderscheiden, en zal dus meer detaillering zichtbaar worden. Op deze manier wordt eerst snel een ruwe schets van de tekening gemaakt, waarna de tekening steeds verder gedetailleerd wordt.

19.4 Inzoomen

De methode `Opnieuw` kan worden aangeroepen om het tekenen van het plaatje te starten. De methode vraagt de gebruiker om de lengte van het af te beelden interval en de te gebruiken waarde van `maxk`. Dan wordt het scherm schoongemaakt door aanroep van de methode `Schoon`, die voor dit doel in de klasse `Grafisch` aanwezig is. Tenslotte worden de variabelen `x0` en `xp` op hun beginwaarde gezet. De methode `Opnieuw` tekent niet zelf het plaatje; dat gebeurt door de methode `Idle` die –als de gebruiker zich een beetje rustig houdt– zeer binnenkort zal worden aangeroepen.

Als de gebruiker op een punt van het plaatje klikt, zal de functie `Klik` worden aangeroepen. Die is in `Mandelbrot` hergedefinieerd. Uit de aangeklikte coördinaten wordt het nieuwe middelpunt berekend, waarna door aanroep van `Opnieuw` het tekenproces opnieuw wordt gestart.

Als je de uitvoer van het programma in figuur 19.1 nader bekijkt, dan zie je dat er in het midden een wit figuur uitgespaard blijft. Dit is het gebied waar de `while`-opdracht in `Kleur` wordt afgebroken, omdat `k>maxk`. De rand van dit gebied (dat ook wel 'appelmannetje' wordt genoemd) is erg grillig. Het precieze verloop van de rand kun je pas bij grote waarden van `maxk` goed zien. De punten die voor elke `maxk` binnen het gebied liggen, vormen samen de Mandelbrot-verzameling. Strikt genomen vormt dus alleen het appelmannetje de Mandelbrot-verzameling; de contourlijnen eromheen vormen slechts franje, die aangeven vanaf welke waarde van `maxk` het punt toch buiten de Mandelbrot-verzameling bleek te liggen.

Inzoomen op de Mandelbrot-figuur is het interessantst in de buurt van de rand van de verzameling. In figuur 19.2 staat een uitvergroting van een van de bolletjes op de 'schouder' van het appelmannetje. Het blijkt dat dit bolletje een soort appelmannetje-in-het-klein is. Nog bizarder wordt het als we het witte vlekje op ongeveer een kwart van links en van boven sterk uitvergroten: als je `maxk` maar groot genoeg maakt blijkt het appelmannetje overal weer op te duiken! (zie figuur 19.3). Wat zou er te zien zijn als je op de schouder van dìt appelmannetje zou inzoomen?

Het is bijna onvoorstelbaar dat een zo fraaie figuur als de Mandelbrot-verzameling tot uiting komt als resultaat van een zo eenvoudige functie als `Kleur`. Maar pas op: dat een eenvoudige functie een ingewikkelde figuur kan beschrijven, betekent nog niet dat elke ingewikkelde figuur ook door een eenvoudige functie te beschrijven is! (Dit is een denkfout die nogal eens gemaakt wordt in verband met fractals).

Een opmerking voor kenners van complexe getallen: in feite is de kleur-functie nog simpeler dan hij lijkt. Als c en z complexe getallen zijn die zijn opgebouwd volgens $c = a + bi$ en $z = x + yi$, dan is de functie die in de kleur-functie wordt geïtereerd: $c \mapsto c^2 + z$. Het 'appelmannetje' kan dus omschreven worden als de getallen z waarvoor het itereren van $c^2 + z$ begrensd blijft (het 'attractiegebied' van deze functie).

19.5 Gebufferd tekenen

Bij aanroep van de methode `Punt` wordt het punt direct op het scherm getekend. Ook bij `Lijn`, `Rechthoek` en `Tekst` is het resultaat direct zichtbaar. Het blijkt bij sommige window-systemen sneller te werken om een aantal punten tegelijk te tekenen, in plaats van één voor één. Hiervoor is in simwin een voorziening gemaakt.

Bij creatie van het `Grafisch`-object worden twee parameters opgegeven: de breedte en de hoogte van het window. Het is mogelijk om ook een derde en een vierde parameter mee te geven. Met de derde parameter kan een menu gespecificeerd worden. Voorlopig volstaat daar de waarde `NULL`. Met de vierde parameter kun je aangeven of elke punt of lijn direct getekend moet worden. Waarde 1 is 'ja', maar met waarde 0 kan het direct tekenen worden afgezet. In dat geval zie je geen direct resultaat van aanroepen van `Punt` en `Lijn`. De veranderingen worden pas in één klap zichtbaar als je de methode `Teken` aanroept.

In het programma mandel wordt van deze faciliteit gebruik gemaakt. Daar waar de constructor-methode van de klasse `Mandelbrot` zijn superklasse `Grafisch` initialiseert, wordt als vierde parameter de waarde 0 meegegeven:

```
Mandelbrot::Mandelbrot(int w, int h) : Grafisch(w,h,NULL,0)
```

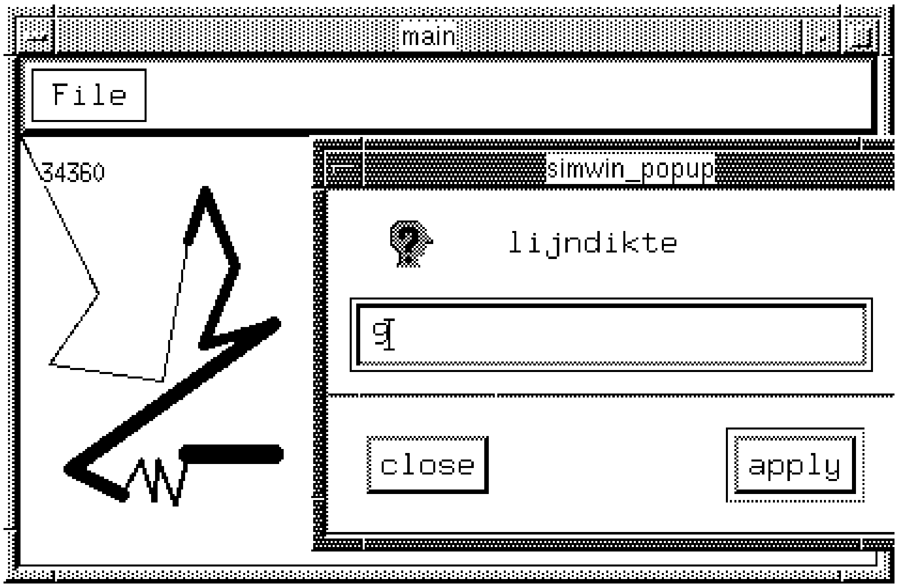
In de methode `Idle` wordt na elke zestiende beeldlijn de methode `Teken` aangeroepen:

```
if (xp%16==0) Teken();
```

Hierdoor verloopt het tekenen een stuk sneller. Als gebruiker zie je inderdaad het plaatje opgebouwd worden in blokken van 16 beeldlijnen tegelijk.

Opgaven

1. De invoer van de lengte van het interval en de waarde van `maxk` via het toetsenbord is minder fraai. In feite moet je er naar streven om `cin` en `cout` niet te gebruiken in window-programma's: bij invoer komt de hele interactie stil te liggen. Pas het programma daarom aan dat je kunt inzoomen door *twee* punten aan te klikken, die samen het gewenste gebied afbakenen. Expliciete invoer van de lengte van het interval is dan overbodig geworden. Het



Figuur 19.4: Het programma lijndikte in aktie

veranderen van de waarde van **maxk** kun je mogelijk maken door in het programma te reageren op het toetsenbord. Daartoe moet je de methode **Toets** herdefiniëren. Je zou bijvoorbeeld de waarde van **maxk** kunnen verhogen bij indrukken van de +-toets en verlagen bij de --toets.

2. Voor kenners van complexe getallen: ga na dat de formules in de functie **Kleur**

```
newa = a*a - b*b + x;
newb = 2*a*b + y;
```

inderdaad de complexe functie $c \mapsto c^2+z$ met $c = a+bi$ en $z = x+yi$ voorstellen. Hoe luiden de formules voor de complexe functie $c \mapsto c^3 + z$? Gebruik deze formules in het programma en kijk hoe de Mandelbrot-figuur er dan uit ziet.

Hoofdstuk 20

Lijndikte, *of:* dialogen

20.1 Dialogen

In window-programma's is het eigenlijk ongewenst om nog op de klassieke manier (via `cin` en `cout`) communicatie met de gebruiker te voeren. Deze uitvoer verschijnt namelijk in het window van waaruit het programma werd opgestart. Zo'n window is echter niet altijd aanwezig, bijvoorbeeld als het programma op een of andere manier met de muis wordt opgestart, iets wat in window-omgevingen niet ongebruikelijk is. In die gevallen zou alle uitvoer naar `cout` verdwijnen, en invoer van `cin` is natuurlijk ook niet mogelijk.

Invoer van waarden naar een window-programma dient te verlopen via *dialogen*. Een voorbeeld van het gebruik van een dialoog is te zien in figuur 19.4. Een dialoog is een apart window waarin de gebruiker tekst kan invullen. In het voorbeeld stelt die tekst de dikte van de lijnen voor. Aan de tekening te zien heeft de gebruiker deze dialoog al meermalen gebruikt...

Voor het voeren van dialogen is in simwin een speciale klasse `StringDialog` beschikbaar. De publieke onderdelen hiervan staan in listing 26. Er is een constructormethode, en er zijn twee virtual methoden `apply` en `close`. Die twee methoden worden aangeroepen als de gebruiker de gelijknamige buttons in de dialoog indrukt. Door een subklasse van `StringDialog` te maken waarin de methoden worden hergedefinieerd, kan deze gebeurtenis in het programma onderschept worden.

20.2 Een dialoog in 'lijnen'

In het programma lijndikte in listing 31 wordt inderdaad een subklasse `Dikte` van `StringDialog` gedefinieerd. Daarvan wordt de virtual methode `apply` geherdefinieerd.

De constructormethode van `StringDialog` krijgt als parameter een tekst mee die in de dialoog ter informatie wordt vermeld, en een beginwaarde voor de in te voeren string. Verder moet het object van type `Grafisch` dat het hoofdwindow voorstelt worden meegegeven. In de methoden van `StringDialog` is deze dan weer beschikbaar onder de naam `grafisch`. Dat komt goed uit, want nu kan in de methode `Dikte::apply` de lijndikte van het hoofdwindow worden aangepast. De waarde van de in de dialoog ingevoerde string is beschikbaar in de variabele `waarde` in `StringDialog`-objecten.

```
#include <iostream.h>
#include "simwin.h"

class Lijnen : public Grafisch
{   int x0, y0;
public:
    Lijnen(int,int);
    virtual void Klik(int,int);
    virtual void Idle();
};

class Dikte : public StringDialog
{
public:
    Dikte(Grafisch &g, char*a, char *b) : StringDialog(g,a,b) {};
    virtual void apply();
};

void Dikte::apply()
{   int n = waarde[0] - '0';
    if (n>0 && n<10)
        grafisch->LijnDikte(n);
}

Lijnen::Lijnen(int br, int h) : Grafisch(br, h)
{   x0 = 0;
    y0 = 0;
}

void Lijnen::Klik(int x1, int y1)
{   Lijn(x0, y0, x1, y1);
    x0 = x1;
    y0 = y1;
}

void Lijnen::Idle()
{   static int n=0;
    Tekst(10, 20, n++);
}

int main(char **, int )
{   Lijnen t(400,200);
    Dikte d ( t, "lijndikte", "1" );
    t.Run();
}
```

Listing 31: lijndikte.cc

Hoofdstuk 21

Life, *of:* een bitmap-editor

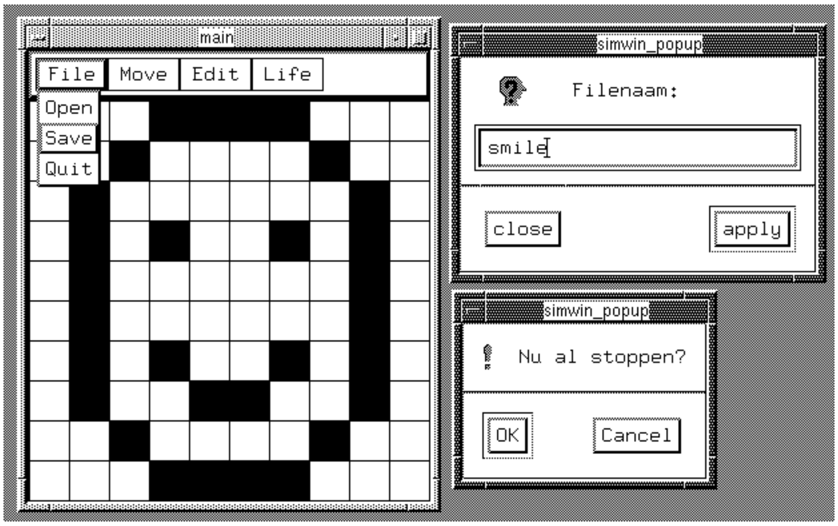
In dit hoofdstuk bekijken we een wat groter grafisch programma. Met het programma kunnen *bitmaps* worden getekend, bewerkt en opgeslagen als file. Verder kun je met het programma 'life' spelen, een simulatie van een evoluerend systeem.

21.1 Het programma

Het programma in dit hoofdstuk heet bitedit, omdat het een editor voor *bitmaps* is. Een bitmap is een tekening die is opgebouwd uit losse zwarte puntjes, die in een rechthoekig patroon zijn gerangschikt. Bitmaps zijn daardoor uiterst geschikt om op een beeldscherm weer te geven. Het programma kan bijvoorbeeld worden gebruikt om ikonen, cursors en lettertypen te ontwerpen, die in andere programma's gebruikt kunnen worden.

In figuur 21.1 is het programma in werking te zien. Met de muis kunnen de zwarte vierkantjes in het raster aan en uit worden gezet: met de linker muisknop wordt het vierkantje zwart, en met de rechter muisknop wit. Met menu's zijn er operaties te kizen die het plaatje in z'n geheel veranderen. In het Move-menu zijn er opties om het plaatje Up, Down, Left en Right te schuiven. In het Edit-menu maakt Clear het hele plaatje schoon. De optie Bold zet de lijnen van de tekening extra dik aan, door links van elk punt een extra punt te zetten. De optie Outline laat alleen de punten aan de rand van de tekening staan, en maakt grote opgevulde vlakken wit. In het File-menu kan met Open een bitmap uit een file gelezen worden, en met Save juist weggezet worden voor later gebruik. De te gebruiken file kan in een aparte dialoog worden gespecificeerd.

Een aardig extra'tje biedt het Life-menu. Hiermee kan een eenvoudige simulatie van een evoluerend systeem worden uitgevoerd, oorspronkelijk beschreven door John Conway. Elk punt van de bitmap kan hierbij beschouwd worden als een beestje (een bacterie ofzo). Een volgende generatie beestjes kan worden bepaald volgens een eenvoudige regel: alleen beestjes in cellen met 2 of 3 buur-beestjes



Figuur 21.1: Het programma bitedit in aktie

```
class BitEdit : public Grafisch
{
    int    diameter;
    int    playlife;
    char   filename[100];
public:
    Bitmap bitmap;
           BitEdit  (int,char**);
    void   Set      (int,int);     virtual void   Klik  (int,int);
    void   Clr      (int,int);     virtual void   Klik3 (int,int);
    void   Sync     ();            virtual void   Menu  (char*);
    void   Grid     ();            virtual void   Idle  ();
    void   SetName  (char* s);
};
```

Listing 32: bme.h

```
const char VOL   = '*';
const char LEEG  = '.';

class Bitmap
{
    char data[SIZE][SIZE];
    int  Neighb     (int,int);
    void Xor        (Bitmap &);
    void Or         (Bitmap &);
    void Copy       (Bitmap *);
public:
         Bitmap     ();
    void Set        (int,int);      void Write    (ostream&);
    void Clr        (int,int);      void Read     (istream&);
    char Get        (int,int);      void Write    (char*);
    int  Tst        (int,int);      void Read     (char*);
    void ShiftLeft  ();             void Clear    ();
    void ShiftRight ();             void Outline  ();
    void ShiftUp    ();             void Bold     ();
    void ShiftDown  ();             void Life     ();
};
```

Listing 33: bm.h

overleven (de anderen gaan dood door verstikking of eenzaamheid), en in lege cellen met precies 3 buren die als verzorgers kunnen dienen worden nieuwe beestjes geboren. De optie Step laat een volgende generatie zien. Na het kiezen van de optie On worden voortdurend nieuwe generaties berekend en getoond, totdat dit wordt stopgezet met de optie Off.

21.2 De klassen

Omdat bitedit een interaktief grafisch programma is, wordt de simwin-bibliotheek gebruikt. Zoals gewoonlijk wordt in het programma een subklasse van de klasse `Grafisch` gedefinieerd, namelijk `BitEdit`. Deze klasse staat gedefinieerd in listing 32. Het complete plaatje zoals dat onder handen wordt genomen is één object `bitmap` van het type `Bitmap`. Verder is er een variabele `diameter` die de diameter van de zwarte vierkantjes in de tekening aangeeft, en een waarheidswaarde `playlife` die aangeeft of er op dit moment nieuwe generaties in de life-simulatie berekend en getekend moeten worden. De tekst-variabele `filename` bewaart de naam die gebruikt wordt als er een file geschreven of gelezen wordt.

In de klasse `BitEdit` zijn er teken-functies `Grid` (om de lijnen te tekenen die de vakjes afbakenen) en `Set` en `Clr` om een vakje zwart, respectievelijk wit te maken. Met de methode `SetName` kan de niet-publieke variabele `filename` worden veranderd. Verder is er een functie `Sync` die de hele afbeelding in overeenstemming brengt ('synchroniseert') met de bitmap die is opgeslagen in de variabele `bitmap`.

Daarnaaast worden er vier `virtual` methoden hergedefinieerd, waardoor interactie met de gebruiker mogelijk wordt. De methode `Klik` reageert op het indrukken van de linker muisknop, de methode `Klik3` op de rechter muisknop. De methode `Menu` handelt gekozen menu-opties af, en de methode `Idle` zal als er niets anders te doen is een nieuwe life-generatie berekenen.

Het eigenlijke plaatje wordt opgeslagen in een object van het type `Bitmap`. De opbouw van zo'n object en de methoden die erop werken staat aangegeven in listing 33. De zwarte en witte punten staan opgeslagen in een array van arrays genaamd `data`. De elementen hiervan zijn characters: sterretjes voor zwarte stippen, en punten voor witte. Er zijn vier methoden die op de individuele stippen werken, zodat de array `data` privé kan blijven. De methode `Set` maakt een stip zwart, `Clr` wit, `Get` levert de letter op, en `Tst` bepaalt of het een zwarte stip betreft. Er zijn methoden `Read` en `Write` om de bitmap uit een file te lezen, respectievelijk in een file te schrijven. Van deze methoden zijn er twee versies: met de naam van de te gebruiken file als parameter, of met een stream als parameter. Dan zijn er drie methoden die een tweede bitmap als parameter hebben: `Copy` zorgt dat de 'eigen' bitmap een kopie daarvan wordt, `Or` voegt de zwarte punten van de tweede bitmap toe aan de eigen bitmap, en `Xor` laat alleen de punten zwart die in één van beide bitmaps zwart zijn.

De overige methoden van `Bitmap` komen overeen met de operaties die de gebruiker via menukeuzes op de bitmap kan uitvoeren: `Clear`, `Shift`... `Outline` en `Bold`. De methode `Life` berekekent een nieuwe life-generatie.

21.3 De klasse BitMap

De klasse `BitMap` kent veel methoden, die echter ieder op zich niet moeilijk zijn. Dit is iets om na te streven bij het ontwerpen van een klasse: zodra iets te ingewikkeld wordt, is het beter om het werk te spliten in aparte methoden. Het geheel blijft daardoor overzichtelijk.

In listing 34 staan de basisoperaties op een bitmap. De constructor-methode `Bitmap` gebruikt `Clear` om de bitmap leeg te maken. De methoden `Set`, `Clr`, `Get` en `Tst` werken direct op het betreffende element van de `data`-arrays. De methoden `Clear`, `Copy`, `Read`, `Write`, `Xor` en `Or` gebruiken twee in elkaar geschoven `while`-opdrachten om alle combinaties van `x`- en `y`-coördinaten te behandelen. De methode `Or` gebruikt daarbij de `||`-operator om een stip zwart te maken als het 'eigen' òf het andere punt (of allebei) zwart is/zijn; in `Xor` wordt de `^`-operator gebruikt om de stip zwart te maken als één van beide (maar niet allebei de) punten zwart zijn.

In listing 35 worden de vier schuif-operaties gedefinieerd. Je moet hierbij vooral opletten om aan de goede kant te beginnen met schuiven, zodat je nog geen punten verandert waarvan je de waarde nog niet op een andere plaats hebt weggeschreven. Het schuiven naar links gebeurt daarom van links naar rechts, en het schuiven naar rechts juist van rechts naar links. Als je ook nog uitkijkt niet een stap te ver door te gaan, waardoor de arraygrenzen geschonden zouden worden, kan er verder weinig mis gaan. In alle vier de methoden wordt de vrijkomende rand aangevuld met lege cellen.

De meer interessante methoden staan in listing 36. De methode `Bold` zet de lijnen van een figuur dikker aan door de bitmap te combineren met een verschoven kopie van zichzelf. In de bitmap `twee` wordt zo'n verschoven kopie aangemaakt, naar links verschoven, en met `Or` gecombineerd met de originele bitmap. Om ook in verticale richting dikkere lijnen te krijgen, wordt dit nog eens herhaald, maar nu met verschuiving naar beneden. Speciale aandacht behoeft de parameter bij de aanroep van `Copy`. Die methode verwacht een verwijzing naar een bitmap. We willen hier de 'eigen' bitmap meegeven. Speciaal voor dit doel is er het gereserveerde woord `this`. Dit woord duidt een verwijzing naar het object aan, in welks context de methode aangeroepen werd. Het woord `this` mag daarom natuurlijk alleen maar binnen methode-definities gebruikt worden, en niet binnen functie-definities.

Ook de methode `Outline` gebruikt een verschoven kopie van zichzelf. Er wordt nu echter gecom-

```
Bitmap::Bitmap()
{   Clear(); }
void Bitmap::Set(int x, int y)
{   data[x][y] = VOL; }
void Bitmap::Clr(int x, int y)
{   data[x][y] = LEEG; }
char Bitmap::Get(int x, int y)
{   return data[x][y]; }
int  Bitmap::Tst(int x, int y)
{   return data[x][y] == VOL; }

void Bitmap::Clear()
{   for (int x=0; x<SIZE; x++) for (int y=0; y<SIZE; y++)
        data[x][y] = LEEG;
}
void Bitmap::Copy(Bitmap *orig)
{   for (int x=0; x<SIZE; x++) for (int y=0; y<SIZE; y++)
        data[x][y] = orig->Get(x,y);
}
void Bitmap::Xor(Bitmap &other)
{   for (int x=0; x<SIZE; x++) for (int y=0; y<SIZE; y++)
        if (Tst(x,y) ^  other.Tst(x,y)) Set(x,y);
        else                            Clr(x,y);
}
void Bitmap::Or(Bitmap &other)
{   for (int x=0; x<SIZE; x++) for (int y=0; y<SIZE; y++)
        if (Tst(x,y) || other.Tst(x,y)) Set(x,y);
        else                            Clr(x,y);
}

void Bitmap::Read(char *filenaam)
{   if (filenaam[0])
        Read( ifstream(filenaam) );
}
void Bitmap::Write(char *filenaam)
{   if (filenaam[0])
         Write( ofstream(filenaam) );
    else Write( cout );
}
void Bitmap::Read(istream &str)
{   for (int y=0; y<SIZE; y++) for (int x=0; x<SIZE; x++)
        str >> data[x][y];
}
void Bitmap::Write(ostream &str)
{   for (int y=0; y<SIZE; y++)
    {   for (int x=0; x<SIZE; x++)
            str << data[x][y];
        str << endl;
    }
}
```

Listing 34: bmsimpel.cc

bineerd met `Xor` in plaats van met `Or`, waardoor alleen de punten die door het schuiven net buiten de rand van het origineel terecht kwamen overblijven. In figuur 21.2 staat is een voorbeeld te zien van een bitmap, dezelfde bitmap naar links en onderen verschoven, en het resultaat van de `Xor`-operatie op beiden: dit is de gewenste contourlijn.

```
void Bitmap::ShiftLeft()
{   int x, y;
    for (x=1; x<SIZE; x++) for (y=0; y<SIZE; y++) data[x-1][y]    = data[x][y];
    for (y=0; y<SIZE; y++)                         data[SIZE-1][y] = LEEG;
}

void Bitmap::ShiftRight()
{   int x, y;
    for (x=SIZE-1; x>0; x--) for (y=0; y<SIZE; y++) data[x][y] = data[x-1][y];
    for (y=0; y<SIZE; y++)                           data[0][y] = LEEG;
}

void Bitmap::ShiftUp()
{   int x, y;
    for (y=1; y<SIZE; y++) for (x=0; x<SIZE; x++) data[x][y-1]    = data[x][y];
    for (x=0; x<SIZE; x++)                         data[x][SIZE-1] = LEEG;
}

void Bitmap::ShiftDown()
{   int x, y;
    for (y=SIZE-1; y>0; y--) for (x=0; x<SIZE; x++) data[x][y] = data[x][y-1];
    for (x=0; x<SIZE; x++)                           data[x][0] = LEEG;
}
```

Listing 35: bmshift.cc

De methode `Life` berekent een nieuwe generatie in de life-simulatie. Hierbij wordt een kopie gemaakt van de bitmap, zodat de oude generatie nog beschikbaar blijft terwijl de nieuwe generatie ingevuld wordt. In de methode-definitie staat de regel voor het geboren worden en sterven van beestjes. Daarbij wordt gebruik gemaakt van de methode `Neighb`, die het aantal buren van een cel uitrekent. De coördinaten `xl` en `xr` stellen respectievelijk **l**inks en **r**echts van `x` voor. Doordat bij de berekening modulo `SIZE` gerekend wordt, ligt 'links' van de linkerrand weer de uiterste rechterrand. De bitmap wordt dus als een soort cylinder beschouwd. Hetzelfde geldt voor de boven- en onderrand. (Ruimtelijk is de combinatie van een cylinder in twee richtingen een beetje moeilijk voor te stellen, maar voor de simulatie is het het leukste als beestjes niet 'van de rand af' kunnen lopen.)

21.4 De klasse BitEdit

De klasse `BitEdit` heeft een aantal methoden om dingen op het scherm te tekenen, en een aantal (`virtual`) methoden waarmee op invoer van de gebruiker gereageerd kan worden. De teken-methoden staan in listing 37, de invoer-methoden in listing 38.

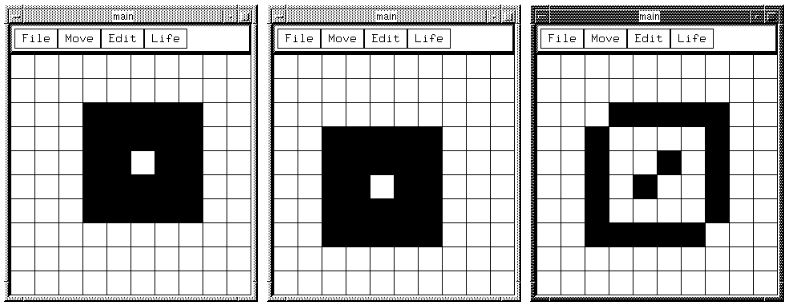
Wat betreft de teken-methoden: `Grid` tekent in mooi rood (kleur 48) horizontale en vertikale lijnen op afstanden zoals aangegeven door `diameter`. De waarde van `diameter` wordt geïnitialiseerd in de constructor-methode `BitEdit`. De methoden `Set` en `Clr` tekenen een rechthoek in respectievelijk zwart (kleur 0) en wit (kleur 63). Door als breedte en hoogte `diameter-1` te gebruiken, worden de rode kaderlijnen net niet uitgewist. De methode `Sync` brengt het scherm weer in overeenstemming met het `Bitmap`-object, door voor alle stippen hetzij `Set` of `Clr` aan te roepen.

De virtuele methoden `Klik` en `Klik3` worden door het systeem aangeroepen als de gebruiker de linker respectievelijk rechter muisknop indrukt. Als reactie roepen we de methode `Set`, respectievelijk `Clr` aan om de stip zwart of wit te maken. Dit gebeurt *zowel* in het `Bitmap`-object *als* in het `BitEdit`-object, zodat die twee consistent met elkaar blijven. Een aanroep van `Sync` is dan ook niet nodig.

In de virtuele methode `Menu` wordt, afhankelijk van de door de gebruiker gekozen operatie, in de meeste gevallen een methode van de `bitmap` aangeroepen. Om het scherm weer in overeenstemming met het resultaat te brengen, wordt daarna de methode `Sync` aangeroepen.

```
void Bitmap::Outline()
{   Bitmap twee;
    twee.Copy(this); twee.ShiftLeft(); twee.ShiftDown(); Xor(twee);
}

void Bitmap::Bold()
{   Bitmap twee;
    twee.Copy(this); twee.ShiftLeft(); Or(twee);
    twee.Copy(this); twee.ShiftDown(); Or(twee);
}

int Bitmap::Neighb(int x, int y)
{   int xl = (x-1); if (xl<0)     xl+=SIZE;
    int yl = (y-1); if (yl<0)     yl+=SIZE;
    int xr = (x+1); if (xr>=SIZE) xr-=SIZE;
    int yr = (y+1); if (yr>=SIZE) yr-=SIZE;

    return Tst(xl,yl) + Tst(x ,yl) + Tst(xr,yl) +
           Tst(xl,y ) +              Tst(xr,y ) +
           Tst(xl,yr) + Tst(x ,yr) + Tst(xr,yr);
}

void Bitmap::Life()
{   Bitmap twee;
    twee.Copy(this);
    for (int x=0; x<SIZE; x++) for (int y=0; y<SIZE; y++)
    {   int n = twee.Neighb(x,y);
        if (twee.Tst(x,y))
        {  if (n==2 || n==3) Set(x,y);
           else              Clr(x,y);
        }
        else
        {  if (n==3) Set(x,y);
           else      Clr(x,y);
        }
    }
}
```

Listing 36: bmoperat.cc



Figuur 21.2: Bitmap, verschoven bitmap, en de Xor-combinatie van die twee

In de virtuele methode `Idle` wordt tenslotte gespecificeerd dat als er verder niets te doen is, een nieuwe life-generatie berekend moet worden. Dit gebeurt echter alleen als de variabele `playlife` de waarde 1 heeft, doordat de gebruiker eerder de menu-optie On heeft gekozen.

```
BitEdit::BitEdit(int n, char**menus) : Grafisch(n,n,menus)
{   diameter = n / SIZE;
    playlife = 0;
    strcpy(filename, "");
    Grid();
}

void BitEdit::Set(int x, int y)
{   ZetKleur(0);
    Rechthoek( x*diameter+1, y*diameter+1, diameter-1, diameter-1 );
}

void BitEdit::Clr(int x, int y)
{   ZetKleur(63);
    Rechthoek( x*diameter+1, y*diameter+1, diameter-1, diameter-1 );
}

void BitEdit::Sync()
{   for (int x=0; x<SIZE; x++)
    for (int y=0; y<SIZE; y++)
        if   (bitmap.Tst(x,y)) Set(x,y);
        else                   Clr(x,y);
}

void BitEdit::Grid()
{   ZetKleur(48);
    for (int x=0; x<=SIZE; x++)
    {   Lijn(0, x*diameter, SIZE*diameter, x*diameter);
        Lijn(x*diameter, 0, x*diameter, SIZE*diameter);
    }
}
```

Listing 37: bmteken.cc

21.5 Het hoofdprogramma

Het hoofdprogramma in listing 39 bestaat voornamelijk uit een opsomming van te gebruiken bibliotheken. Daarvoor wordt hier de constante `SIZE`, die de afmeting van de bitmap bepaalt, gedefinieerd.

Verder is er een klasse `Filenaam`. Met objecten van dat type kan een dialoog gevoerd worden met de gebruiker, net als in hoofdstuk 20. Hierdoor kan de gebruiker een filenaam specificeren. Op de dialoog is een button 'apply' aanwezig. Drukt de gebruiker die in, dan wordt de virtuele methode `apply` aangeroepen, die de dan geldende `waarde` van de ingevulde tekst doorspeelt aan de bitmapeditor.

In de functie `main` worden de twee interface-elementen (het hoofdwindow en de filenaam-dialoog) gecreëerd door een object van de daarvoor bestemde typen te declareren. Tenslotte wordt de interactie met de gebruiker gestart door aanroep van de methode `Run`. Die methode, gedefinieerd in simwin, zal op zijn tijd de methoden `Klik`, `Menu` etc. aanroepen. Doordat die methoden virtueel zijn en in de klasse `BitEdit` zijn hergedefinieerd, wordt op dat moment de gewenste aktie uitgevoerd.

```
void BitEdit::Klik(int x, int y)
{   x /= diameter;
    y /= diameter;
    bitmap.Set(x,y);
    Set(x,y);
}

void BitEdit::Klik3(int x, int y)
{   x /= diameter;
    y /= diameter;
    bitmap.Clr(x,y);
    Clr(x,y);
}

void BitEdit::SetName(char* s)
{   strcpy(filename, s);
}

void BitEdit::Menu(char *s)
{   if      (!strcmp(s,"Save"))   bitmap.Write     (filename);
    else if (!strcmp(s,"Open"))   bitmap.Read      (filename);
    else if (!strcmp(s,"Left"))   bitmap.ShiftLeft ();
    else if (!strcmp(s,"Right"))  bitmap.ShiftRight();
    else if (!strcmp(s,"Up"))     bitmap.ShiftUp   ();
    else if (!strcmp(s,"Down"))   bitmap.ShiftDown ();
    else if (!strcmp(s,"Clear"))  bitmap.Clear     ();
    else if (!strcmp(s,"Outline"))bitmap.Outline   ();
    else if (!strcmp(s,"Bold"))   bitmap.Bold      ();
    else if (!strcmp(s,"Step"))   bitmap.Life      ();
    else if (!strcmp(s,"On"))     playlife = 1;
    else if (!strcmp(s,"Off"))    playlife = 0;
    else Grafisch::Menu(s);
    Sync();
}

void BitEdit::Idle()
{   if (playlife)
    {   bitmap.Life();
        Sync();
    }
}
```

Listing 38: bminvoer.cc

```
#include <fstream.h>
#include <string.h>
#include "simwin.h"

const int  SIZE   = 10;

#include "bm.h"
#include "bme.h"
#include "bmsimpel.cc"
#include "bmshift.cc"
#include "bmoperat.cc"
#include "bmteken.cc"
#include "bminvoer.cc"

class Filenaam : public StringDialog
{
public:
    Filenaam(Grafisch &g, char*a, char *b) : StringDialog(g,a,b) {};
    virtual void apply() { ((BitEdit*)grafisch)->SetName(waarde); };
};

int main()
{
    static char *menus[] =
    { "#Quit" , "Open"
    //{ "#File" , "Open" , "Save" , "Quit"
    //, "#Move" , "Left" , "Right", "Up"     , "Down"
    //, "#Edit" , "Clear", "Bold" , "Outline"
    //, "#Life" , "Step" , "On"   , "Off"
    , ""
    };

    BitEdit  e(300, menus);
    Filenaam f( e, "Filenaam:", "" );
    e.Run();
}
```

Listing 39: bm.cc

Hoofdstuk 22

Handschriftherkenning, *of:* neurale netwerken

Wie de hoofdstukken tot hier toe heeft gelezen, zou misschien kunnen denken dat alle getoonde voorbeelden typische 'tekstboekvoorbeelden' zijn, en dat programma's 'in het echt' er anders uitzien. Om het tegendeel te bewijzen bekijken we in dit hoofdstuk, als slot van deze cursus, een serieuze toepassing: het herkennen van handgeschreven letters.

Het opvallende is, dat dit programma wel groter is (meer regels, meer files, meer klassen, meer methoden) dan de eerdere voorbeelden, maar dat alle onderdelen op zich er nog steeds hetzelfde uitzien als in de andere hoofdstukken. Voor alle soorten objecten die een rol spelen zijn er klassen, en er zijn een heleboel methoden die de basis-operaties op die klassen implementeren.

Het programma in dit hoofdstuk is ook een goed voorbeeld van het opnieuw gebruiken van eerder geschreven programma's. Zo komt de lijst-bibliotheek uit hoofdstuk 14 weer goed van pas, maar ook delen van de bitmap-editor kunnen opnieuw gebruikt worden.

Voor het oplossen van het probleem gebruiken we een *neuraal netwerk*. In paragraaf 1 t/m 7 zullen we neurale netwerken in het algemeen programmeren. Pas vanaf paragraaf 8 wordt de geschreven programmatuur gebruikt voor de toepassing: het herkennen van handschrift.

22.1 Neurale netwerken

Bij handschriftherkenning is het probleem om bij een gegeven bitmap te bepalen welke letter deze voorstelt. In figuur 22.1 zijn een paar voorbeelden van bitmaps gegeven die als 1, respectievelijk 2 herkend moeten worden. Het probleem is in feite om een functie te vinden van de verzameling van alle mogelijke bitmaps naar de verzameling $\{1 \ldots 10\}$.

Een aantal jaren geleden is ontdekt dat bij veel functies die in de praktijk voorkomen de uitvoer zich goed laten benaderen als *gewogen gemiddelde* van gewogen gemiddeldes van $(\ldots)$ van de invoer. De kunst is om voor een gegeven probleem de wegingsfactoren te vinden. Een functie die op deze manier wordt benaderd heet een *neuraal netwerk*. Het leuke van neurale netwerken is, dat de te gebruike wegingsfactoren experimenteel bepaald kunnen worden, door het netwerk een aantal *voorbeelden* te laten zien met de bijbehorende gewenste antwoorden. Met de aldus bepaalde wegingsfactoren zal het netwerk later voor soortgelijke invoer opnieuw dezelfde uitvoer opleveren.

Een gedetailleerde beschrijving van neurale netwerken kan gevonden worden in hoofdstuk 22 het boek 'Artificial intelligence'. In de rest van dit hoofdstuk zullen we ons vooral concentreren op het programmeren van de benodigde algoritmes. De theorie nemen we over uit 'Artificial intelligence'. De te gebruiken formules zullen we hier niet opnieuw motiveren. Pak dat boek er dus *nu* bij...

22.2 Klassen voor een neuraal netwerk

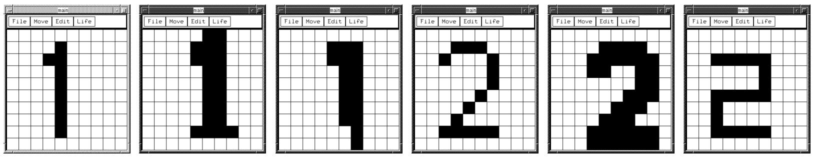
In een neuraal netwerk is er sprake van drie belangrijke soorten objecten: *neuronen*, die zijn gegroepeerd in *layers*, die samen een *netwerk* vormen. Natuurlijk definiëren we voor elk van deze soorten een klasse met daarbijbehorende methoden. De definities voor de klassen `Neuron`, `Layer` en `Net` staan in listing 40.

We bekijken eerst de variabelen die binnen elk soort object een rol spelen. Om te beginnen het *netwerk*. Een netwerk is opgebouwd uit layers. In objecten van de klasse is daarom een array

```
class Layer; class Neuron; class Sample;

class Net
{   Layer   *layer;
    int      nLayer;

    void    beginLearn();
    void    endLearn  (int);
    void    backward  (double[]);
public:
            Net       (int, int[]);
           ~Net       ();
    void    learn     (List<Sample>&);  void  print (ostream &);
    void    forward   (double[]);       void  read  (istream &);
    int     result    ();
};

class Layer
{   friend  Net; friend  Neuron;

    Neuron* neuron;
    int     nNeuron;

            Layer     () {};            void  init      (int,int);
           ~Layer     ();
    void    setValue  (double[]);       void  beginLearn();
    void    calcValue (Layer*);         void  endLearn  (int);
    void    calcBeta  (double[]);       void  print     (ostream&);
    void    calcBeta  (Layer*);         void  read      (istream&);
    void    calcDelta (Layer*);         int   result    ();
};

class Neuron
{   friend Layer;

    double  value;                      double *weight;
    double  beta;                       double *delta;
    int     seqNo;                      int     nInput;

            Neuron    () {};            void    init      (int,int);
           ~Neuron    ();
    double  getValue  ();
    void    setValue  (double);         void    beginLearn();
    void    calcValue (Layer*);         void    endLearn  (int);
    void    calcBeta  (double);         void    print     (ostream&);
    void    calcBeta  (Layer*);         void    read      (istream&);
    void    calcDelta (Layer*);
    double  betaForm  (int);
};
```

Listing 40: nn.h

van layers nodig. Omdat in verschillende netwerken het aantal layers kan verschillen, maken we er een array met dynamisch bepaalde lengte van. Dat wil zeggen: in de klasse wordt een verwijzing een layer gedeclareerd, en in de constructormethode wordt met `new` de eigenlijke array gecreëerd, waar deze verwijzing naar toe gaat wijzen. Verder declareren we in de klasse een variabele die aangeeft hoeveel layers er zijn; deze variabele wordt ook in de constructormethode geïnitialiseerd.



Figuur 22.1: Samples voorstellende de cijfers '1' en '2'

De variabelen in de klasse `Net` zijn dus:

```
class Net
{
    Layer *layer;
    int    nLayer;
};
```

Elke layer bestaat op zijn beurt uit een variabel aantal neuronen. Op dezelfde manier declareren we daarom:

```
class Layer
{
    Neuron *neuron;
    int     nNeuron;
} ;
```

Alle neuronen in een bepaalde layer zijn verbonden met alle neuronen in de volgende layer. Aan de verbindingen in een *gewicht* gehecht. We slaan de gewichten van de verbindingen op in het neuron waar de verbinding *binnenkomt*. Verder kan elk neuron een *waarde* bevatten. De declaratie van de klasse `Neuron` begint daarom als volgt:

```
class
{
    double  value;
    double *weight;
    int     nInput;
```

Het bepalen van de ideale gewichten in het netwerk gebeurt tijdens de *training* ervan. In het boek 'Artificial intelligence' wordt op blz. 452 een algoritme gegeven hoe die training in zijn werk gaat. Bij elke neuron wordt een factor vastgesteld die aangeeft hoe zinvol het is dat het gewicht veranderd. Deze factor heet β. Met behulp van deze factor wordt een verandering van de gewichten Δw uiterekend. Omdat deze waarden een belangrijke rol spelen bij de training van een netwerk, declareren we er variabelen voor in elk neuron: een getal `beta` voor het hele neuron, en een array `delta`, waarvan de elementen corresponderen met de array `weight`. We voegen daarom aan de klassedefinitie toe:

```
    double *delta;
    double  beta;
```

Tenslotte is het tijdens de training nodig dat elk neuron zijn volgnummer in de layer weet:

```
    int seqNo
};
```

22.3 Opbouw van een netwerk

Een netwerk wordt gekarakteriseerd door het aantal layers en de aantallen neuronen daarin. Bij de constructie van een nieuw netwerk kunnen we deze waarden als parameter meegeven. De constructor `Net::Net` heeft daarom een integer parameter die het aantal layers aangeeft, en een array van integers waarin de aantallen neuronen in de afzonderlijke layers staan vermeld. In listing 41 is te zien dat het aantal layers direct wordt gebruikt om de array van layers te creëren.

Opvallend in listing 41 is dat er geen constructor van `Layer` is, terwijl we eerder hebben gezien dat in elke layer toch echt een array met neuronen moet worden gecreëerd. De reden hiervoor is, dat constructoren van klassen waarvan we arrays willen maken, geen parameters mogen hebben. Daarom is er een apart methode `init` voorzien, waarmee layers na creatie geïnitialiseerd

```
void Neuron::init(int n, int s)
{  nInput = n;    weight = new double[n];
   seqNo  = s;    delta  = new double[n];
   for (int i=0; i<n; i++)
      weight[i] = (i+1)*(1.0/(n+1)); // any value, as long as they are different
}

Neuron::~Neuron()
{   delete weight;
    delete delta;
}

void Layer::init(int nIn, int nElem)
{   nNeuron = nElem;
    neuron  = new Neuron[nElem];
    for (int i=0; i<nElem; i++)
        neuron[i].init(nIn, i);
}

Layer::~Layer()
{   delete neuron;
}

Net::Net(int nLay, int size[])
{   nLayer = nLay;
    layer  = new Layer[nLay];

    int sizePrev = 0;
    for (int i=0; i<nLay; i++)
    {    layer[i].init( sizePrev, size[i] );
         sizePrev = size[i];
    }
}

Net::~Net()
{   delete layer;
}
```

Listing 41: nnbuild.cc

kunnen worden. Deze methode wordt dan ook bij creatie van het netwerk voor alle layers direct aangeroepen.

Bij initialisatie van een layer is natuurlijk het aantal neuronen in die layer van belang. Dit aantal wordt daarom als parameter meegegeven aan `Layer::init`. Ook het aantal inkomende verbindingen is van belang bij de creatie van de layer, want in de neuronen van die layer moeten de gewichten van de inkomende verbindingen worden opgeslagen. Het aantal inkomende verbindingen is gelijk aan het aantal neuronen in de vorige layer, behalve bij de eerste layer (daar zijn geen inkomende verbindingen). In de definitie van `Net::Net` is te zien dat ook dit aantal wordt meegegeven aan `Layer::init`.

Bij initialisatie van een layer wordt een array van neuronen van de juiste lengte gecreëerd. Direct daarna worden deze neuronen geïnitialiseerd door een aanroep van `Neuron::init`. Daarbij worden arrays voor `weight` en `delta` gecreëerd. De gewichten in de array `weight` krijgen een willekeurige beginwaarde. In 'Artificial intelligence' wordt daarover opgemerkt: 'in fact, just any numbers will do, as long as they differ from one another' (blz. 458).

Voordat we aan de interessante methoden kunnen beginnen, zijn er een aantal methoden nodig van huishoudelijke aard. Deze worden gedefinieerd in listing 42. Zo zijn er voor objecten van type

```
double Neuron::getValue()
{   return value; }

void Neuron::setValue( double v)
{   value = v;    }

void Layer::setValue(double source[])
{   for (int i=0; i<nNeuron; i++) neuron[i].setValue( source[i] );   }

void Neuron::beginLearn()
{   for (int i=0; i<nInput; i++)  delta[i] = 0.0;                    }

void Neuron::endLearn(int nSample)
{   for (int i=0; i<nInput; i++)  weight[i] += delta[i]/nSample;     }

void Layer::beginLearn()
{   for (int i=0; i<nNeuron; i++) neuron[i].beginLearn();            }

void Layer::endLearn(int nSample)
{   for (int i=0; i<nNeuron; i++) neuron[i].endLearn(nSample);       }

void Net::beginLearn()
{   for (int j=0; j<nLayer; j++)  layer[j].beginLearn();             }

void Net::endLearn(int nSample)
{   if (nSample>0)
    for (int j=0; j<nLayer; j++)  layer[j].endLearn(nSample);        }
```

Listing 42: nnset.cc

`Neuron` twee methoden om de waarde op te vragen en te veranderen: `SetValue` en `GetValue` – een ingewikkelde manier om de waarden zelf maar niet publiek te hoeven maken. De klasse `Layer` heeft ook een methode `setValue`, waarmee alle neuronen erin van een waarde kunnen worden voorzien.

22.4 Trainen van een netwerk

Om een netwerk te trainen moeten er een aantal voorbeelden (*samples*) bekend zijn. Voor elk van deze samples wordt bepaald welke output het netwerk ervoor levert. Deze output wordt vergeleken met de gewenste output, en door middel van *backward propagation* wordt een suggestie voor verandering van de gewichten bepaald. Deze suggestie wordt berekend voor *alle* samples, en het gemiddelde ervan is de werkelijke gewicht-verandering.

Tijdens de training verzamelen alle neuronen de som van alle suggesties in de daarvoor gemaakte arrays `delta`. Zijn alle samples geprobeerd, dan kan de wijziging van de gewichten werkelijk worden doorgevoerd.

De neuronen hebben een methode `beginLearn` die moet worden aangeroepen aan het begin van een training. In deze methode worden alle waarden van de array `delta` gelijk aan nul gemaakt. Aan het eind van het proberen van de samples is de som van de suggesties verzameld in de `delta`-arrays; de wijziging van de gewichten met de gemiddelde suggestie wordt uitgevoerd in de methode `Neuron::endLearn`.

Ook voor de klasse `Layer` definiëren we een methode `beginLearn` en `endLearn`. Deze roepen de overeenkomstige methoden aan voor alle neuronen in een layer. Dezelfde methoden zijn er ook voor een `Net`, die het signaal weer doorgeven aan alle layers. Op deze manier kunnen we met één aanroep van `Net::beginLearn` alle `delta`-waarden in alle neuronen initialiseren.

```
void Net::forward( double invoer[] )
{   layer[0].setValue( invoer );
    for (int j=1; j<nLayer; j++)
        layer[j].calcValue( &layer[j-1] );
}

void Net::backward( double uitvoer[] )
{   layer[nLayer-1].calcBeta( uitvoer );
    for (int j=nLayer-1; j>0; j--)
    {   layer[j].calcDelta( &layer[j-1] );
        if (j>1)
            layer[j-1].calcBeta ( &layer[j] );
    }
}

void Net::learn( List<Sample> &samples )
{   int nSample = 0;
    beginLearn();
    for (All<Sample> s(samples); s.valid(); s.next() )
    {   forward ( s.value()->invoer  );
        backward( s.value()->uitvoer );
        nSample++;
    }
    endLearn(nSample);
}
```

Listing 43: nnlearn.cc

22.5 Uitrekenen van de waarde

Om bij een gegeven input de output van het neurale netwerk te berekenen, moet de input in de eerste layer worden gezet, waarna achtereenvolgens de waarden van de volgende layers berekend kunnen worden. In listing 43 staat daartoe een methode `Net::forward` die dit doet: de waarden van de input worden met `Layer::setValue` toegekend aan de eerste layer, en van de volgende layers worden de waarden uitgerekend met `Layer::calcValue`. Daarbij wordt steeds de voorgaande layer als parameter meegegeven. Deze methode roept simpelweg `Neuron::calcValue` aan voor alle neuronen.

De methode `Neuron::calcValue` wordt gedefinieerd in listing 44. Het is de centrale methode bij de berekening van de output: hierin staat gespecificeerd dat *de waarde van een neuron een gewogen gemiddelde is van de neuronen in de vorige layer*:

```
for (int i=0; i<nInput; i++)
    x += weight[i] * previous->neuron[i].getValue();
```

Omdat de waarden van alle neuronen tussen 0 en 1 moeten liggen, wordt het berekende gewogen gemiddelde 'afgevlakt' met de functie `sigmoid`

```
value = sigmoid(x);
```

Dit is de 'squashed S' functie op blz. 450 in 'Artificial intelligence'. Het functievoorschrift daarvan staat op blz. 456:

```
double sigmoid(double x)
{   return 1.0 / (1.0 + exp(-x));
}
```

22.6 Aanpassen van de gewichten

Suggesties voor de aanpassingen van de gewichten Δw kunnen worden berekend met de formules op blz. 453. Daarvoor moeten eerst de getalen β worden berekend. Voor de laatste layer geldt dat

```
double sigmoid(double x)
{   return 1.0 / (1.0 + exp(-x));
}

void Neuron::calcValue( Layer *previous )
{   double x = 0.0;
    for (int i=0; i<nInput; i++)
        x += weight[i] * previous->neuron[i].getValue();
    value = sigmoid(x);
}

void Neuron::calcBeta( double desired )
{   beta = desired - value;
}

void Neuron::calcBeta( Layer *next )
{   double x = 0.0;
    for (int k=0; k<next->nNeuron; k++)
        x += next->neuron[k].betaForm(seqNo);
    beta = x;
}

double Neuron::betaForm(int j)
{   return weight[j] * value * (1-value) * beta;
}

void Neuron::calcDelta( Layer *previous )
{   for (int i=0; i<nInput; i++)
        delta[i] += rate * previous->neuron[i].value * value * (1-value) * beta;
}

void Layer::calcValue( Layer *previous )
{   for (int i=0; i<nNeuron; i++) neuron[i].calcValue( previous );  }

void Layer::calcBeta( double image[] )
{   for (int i=0; i<nNeuron; i++) neuron[i].calcBeta ( image[i] );  }

void Layer::calcBeta( Layer *next )
{   for (int i=0; i<nNeuron; i++) neuron[i].calcBeta ( next );      }

void Layer::calcDelta( Layer *previous )
{   for (int i=0; i<nNeuron; i++) neuron[i].calcDelta( previous );  }
```

Listing 44: nncalc.cc

$\beta = d - o$, waarbij d de 'desired' waarde is, en o de berekende output. Deze formule staat in het programma in de methode `Neuron::calcBeta` in listing 44.

Er is nog een methode `Neuron::calcBeta`, waarmee de formule wordt geïmplementeerd voor de β's in de andere layers:

$$\beta_j \;=\; \sum_k w_{j\to k}\, o_k\, (1 - o_k)\, \beta_k$$

De sommatie is in de listing terug te herkennen in het feit dat de variabele k alle neuronen in de layer `next` doorloopt. De deelformule $w_{j\to k}\, o_k\, (1 - o_k)\, \beta_k$ is terug te vinden in de methode `Neuron::betaForm`, die in de context van het neuron in de 'next' layer wordt uitgerekend, met de waarde van j als parameter. Goed kijken in listing 44! Dit zit subtiel in elkaar, en vormt het hart van het back-propagation algoritme.

```
class Sample
{   friend Net;

    double *invoer;
    double *uitvoer;
public:
            Sample   ( int, double[], int, double[] );
           ~Sample   ();
};
```

Listing 45: nnsample.h

```
Sample::Sample( int nIn, double ins[], int nOut, double outs[] )
{   int i;

    invoer  = new double[nIn];
    uitvoer = new double[nOut];

    for (i=0; i<nIn; i++)
        invoer[i] = ins[i];
    for (i=0; i<nOut; i++)
        uitvoer[i] = outs[i];
}

Sample::~Sample()
{
    delete invoer;
    delete uitvoer;
}
```

Listing 46: nnsample.cc

De rest van listing 44 is weer eenvoudig: De methoden `calcBeta` en `calcDelta` hebben tegenhangers op layer-nivo, die de overeenkomstige methoden voor al hun neuronen aanroepen. De complete back propagation wordt uitgevoerd door de methode `Net::backward` in listing 43.

Met een aanroep van `Net::forward` en `Net::backward` kun je de output van een sample door het netwerk laten berekenen, deze vergelijken met de gewenste output voor dat sample, en an de hand daarvan suggesties doen voor veranderingen van de gewichten.

22.7 Training

Training van het netwerk kan plaatsvinden door `Net::learn` op een grote lijst samples toe te passen. Een sample bestaat uit twee arrays die de input en de gewenste output voor het netwerk voorstellen. De klasse `Sample` waarin dat wordt uitgedrukt wordt gedefinieerd in listing 45 en listing 46.

We completeren de neurale-netwerkbibliotheek tenslotte met listing 47, waarin methoden worden gedefinieerd om de gewichten in een netwerk af te drukken en in te lezen uit een file. Ook wordt hierin een methode `result` gedefinieerd, waarmee het nummer van de uitvoerneuron met de grootste waarde kan worden bepaald.

22.8 Handschriftherkenning

Hoe is een neuraal netwerk nu te gebruiken voor handschriftherkenning? Een handgeschreven symbool kan beschouwd worden als bitmap. Een bitmap bestaat uit een aantal punten. Deze

```
void Neuron::read(istream &str)
{   for (int k=0; k<nInput;  k++)  str >> weight[k];          }

void Neuron::print(ostream &str)
{   for (int k=0; k<nInput;  k++)  str << weight[k] << " ";
    str << endl;                                              }

void Layer::read(istream &str)
{   for (int i=0; i<nNeuron; i++)  neuron[i].read(str);       }

void Layer::print(ostream &str)
{   for (int i=0; i<nNeuron; i++)  neuron[i].print(str);
    str << endl;                                              }

void Net::read(istream &str)
{   for (int j=0; j<nLayer; j++)   layer[j].read(str);        }

void Net::print(ostream &str)
{   for (int j=0; j<nLayer; j++)   layer[j].print(str);       }

int Layer::result()
{   double max  = -100000;
    int    best = -1;
    for (int i=0; i<nNeuron; i++)
    {   double v = neuron[i].getValue();
        if (v>max)
        {   max  = v;
            best = i;
        }
    }
    return best;
}

int Net::result()
{   return layer[nLayer-1].result();
}
```

Listing 47: nnio.cc

punten gebruiken we als invoer van het neurale netwerk: voor zwarte punten heeft de betreffende invoer de waarde 1, voor witte punten de waarde 0. Voor bitmaps met afmeting 10×10 is dus een netwerk met 100 invoer-neuronen nodig.

De uitvoer van het netwerk bestaat uit evenveel waardes als er te herkennen symbolen zijn. De uitvoer-neuron met de grootste waarde gebruiken we als indicatie voor het herkende symbool. Vergelijk dit met het voorbeeld op bladzijde 458 in 'Artificial intelligence'.

In listing 48 wordt een functie `bitmapSample` gedefinieerd, waarmee een bitmap volgens deze codering omgezet kan worden in een sample voor een neuraal netwerk. De functie zelf is eenvoudig, maar om de typen `Bitmap` en `Sample` te mogen gebruiken, moet een groot aantal bibliotheken worden meegenomen. Hier wordt eerder geïnvesteerd werk geoogst!

22.9 Het trainingsprogramma

Het trainen van het netwerk hoeft maar één keer te gebeuren. Hiervoor maken we een apart programma. De uitvoer hiervan is een file met de berekende gewichten. Deze file kan door de eigenlijke herkenner worden ingelezen. Hiermee wordt bereikt dat het netwerk niet bij elke herkensessie opnieuw getraind hoeft te worden.

```
#include <fstream.h>
#include <string.h>
#include <stdlib.h>
#include <math.h>
#include "lijst.cc"

const double rate = 1.0;
#include "nn.h"
#include "nnsample.h"
#include "nnbuild.cc"
#include "nnset.cc"
#include "nncalc.cc"
#include "nnlearn.cc"
#include "nnio.cc"
#include "nnsample.cc"

const int  SIZE = 10;
#include "bm.h"
#include "bmsimpel.cc"
#include "bmshift.cc"
#include "bmoperat.cc"

const  int NETIN  = SIZE*SIZE;
const  int NETOUT = 10;
const  int LAYERS = 3;
static int NEURONS[LAYERS] = {NETIN, 5, NETOUT };

void bitmapToArray( Bitmap &bitmap, double array[] )
{   for (int i=0; i<NETIN; i++)
        array[i] = bitmap.Tst( i/SIZE, i%SIZE );
}

Sample *bitmapSample( Bitmap &bitmap, int waarde )
{   double invoer [NETIN];
    double uitvoer[NETOUT];

    bitmapToArray( bitmap, invoer );
    for (int i=0; i<NETOUT; i++)
        uitvoer[i] = 0;
    uitvoer[waarde] = 1;

    return new Sample(NETIN, invoer, NETOUT, uitvoer);
}
```

Listing 48: nnbm.cc

In listing 49 staat het trainingsprogramma. Met de bitmap-editor uit hoofdstuk 21 kunnen de voorbeelden worden geprepareerd en in files worden gezet. Het programma `nnprep` uit listing 49 leest filenamen van `cin`, leest de bijbehorende files als bitmap, en maakt daar samples van. Met de aldus gemaakte samples wordt het netwerk getraind. Het resultaat (de bepaalde wegingsfactoren) wordt naar `cout` geschreven.

Dit programma kan het beste worden gebruikt met

```
nnprep <filenamen >gewichten
```

Daarna kan de eigenlijke herkenner worden gebruikt door:

```
nnrecog gewichten
```

```
#include "nnbm.cc"

int main()
{
    const  int learncycles = 1000;

    Net          net ( LAYERS, NEURONS );
    List<Sample> samples;
    Bitmap       bitmap;
    int          waarde;
    char         naam[100];

    while (cin>>waarde>>naam)
    {
        bitmap.Read(naam);
        samples.add( bitmapSample( bitmap, waarde ) );
    }

    for (int a=0; a<learncycles; a++)
        net.learn(samples);

    net.print(cout);
}
```

Listing 49: nnprep.cc

22.10 Het herken-programma

Het programma waarmee de eigenlijke herkenning plaatsvindt, is een aanpassing van bitedit. De tekst van het programma staat in listing 50. In dit programma worden met

```
#include "nnbm.cc"
```

alle relevante neural-netwerk- en bitmap-methoden binnengehaald.

In het programma wordt vervolgens een subklasse Recognizer gemaakt van de klasse BitEdit. Hierin wordt de neiwue functionaliteit gedeinieerd. Er is een menu Recognize toegevoegd. Als de gebruiker dit kiest, wordt de bitmap op het scherm door het neurale netwerk heen gehaald, en wordt het resultaat op cout afgedrukt.

In de constructormethode van Recognizer wordt het te gebruiken neurale netwerk ingelezen. De filenaam hiervoor wordt in het hoofdprogramma van de commandoregel geplukt.

Opgaven

1. (Voor in de zomervakantie:) Prepareer de benodigde samples om het netwerk te trainen de getallen 0 t/m 9 te kunnen herkennen. Raadpleeg blz. 467 van 'Artificial intelligence' voor een schatting van het benodigde aantal samples.

```
#include "nnbm.cc"
#include "simwin.h"
#include "bme.h"
#include "bmteken.cc"
#include "bminvoer.cc"

static char *menus[] =
{ "#File" , "Quit" , "#Move" , "Left" , "Right", "Up"     , "Down",
  "#Edit" , "Clear", "Bold" , "Outline" , "#Recognize", "Recognize" , "" };

class Recognizer : public BitEdit
{
    Net *net;
public:
    Recognizer(char *);
    virtual void Menu(char*);
};

Recognizer::Recognizer(char *netnaam) : BitEdit(300, menus)
{
    net = new Net(LAYERS, NEURONS);
    net->read( ifstream(netnaam) );
}

void Recognizer::Menu(char *s)
{   double invoer[NETIN];

    if (!strcmp(s,"Recognize"))
    {
        bitmapToArray( bitmap, invoer );
        net->forward( invoer );
        cout << "recognized as: " << net->result() << endl;
    }
    else BitEdit::Menu(s);
}

int main(int aantal, char *namen[])
{
    if (aantal!=2)
    {   cout << "doe eerst preprocessing met nnprep" << endl;
        cout << "en gebruik de resultaat-file daarvan als parameter" << endl;
        exit(1);
    }

    Recognizer r(namen[1]);
    r.Run();
}
```

Listing 50: nnrecog.cc

Bijlage A

UNIX-opdrachten

opdracht	parameters	omschrijving
`cal`	[[*maand*] *jaar*]	maak kalender
`cat`	*files*	toon files op scherm
`CC`	*file*`.cc`	C++-compiler
`cd`	*dir*	verander aktieve directory
`cp`	*file1 file2*	file2 wordt copie van file1
`cp`	*files dir*	maak copie van files in dir
`date`		geef datum en tijd
`elm`	*naam*	electronic mail
`grep`	*patroon file*	zoek naar reguliere expressie
`ls`	[*files*]	toon files in aktieve directory
`ls -l`	[*files*]	... met extra informatie
`man`	*opdracht*	handleiding van opdracht
`man -k`	*trefwoord*	... over een trefwoord
`mkdir`	*dir*	maak subdirectory
`more`	*files*	toon files per scherm
`mv`	*file1 file2*	file1 gaat file2 heten
`mv`	*files dir*	verplaats files naar dir
`passwd`		verander je password
`print`	*files*	zet files op papier
`rm`	*files*	verwijder files
`rmdir`	*dir*	verwijder directory
`sort`	*file*	sorteer regels van file
`submit`	`cki` *nr file*	lever prakticumopgave in
`uniq`	*file*	verwijder dubbele regels
`vi`	*file*	editor
`who`		wie werkt op deze computer?
`who -c`		... op het hele systeem?
`xterm &`		maak een nieuw window

Bijlage B

Overzicht vi-functies

cursor verplaatsen (desgewenst voorafgegaan door een aantal)

functie	betekenis
`h`	naar links
`j`	naar omlaag
`l`	naar rechts
`k`	naar omhoog
`$`	naar einde regel
`^`	naar begin regel
`w`	naar volgende woord ('word')
`b`	naar vorige woord ('back')
n`G`	naar regel *n* van file ('goto')
n`H`	naar regel *n* van scherm ('home')
`M`	naar midden scherm
`f`*x*	vooruit naar letter *x* ('find')
`F`*x*	terug naar letter *x*
`/`*tekst*	vooruit naar *tekst*
`?`*tekst*	terug naar *tekst*
`n`	herhaal vorige `/` of `?` ('next')
`N`	idem in andere richting
`%`	zoek bijbehorende haakje
control `D`	half scherm omlaag ('down')
control `U`	half scherm omhoog ('up')
control `F`	heel scherm omlaag ('forward')
control `B`	heel scherm omhoog ('back')
control `G`	toon positie
control `L`	repareer scherm lay-out

veranderingen ongedaan maken

functie	betekenis
`u`	maak laatste verandering ongedaan ('undo')
`U`	maak alle veranderingen op deze regel ongedaan
`:q!`	verlaat `vi` zonder te schrijven
`:e!`	edit de originele file opnieuw

tekst toevoegen (afsluiten met `esc`-toets)

functie	betekenis
`a`	achter de cursor ('append')
`A`	aan het eind van de regel
`i`	voor de cursor ('insert')
`I`	aan het begin van de regel
`o`	op een lege regel hieronder
`O`	op een lege regel hierboven

tekst weghalen (desgewenst voorafgegaan door een aantal)

functie	betekenis
`x`	letter(s) onder en na de cursor
`X`	letter(s) voor de cursor
`dw`	van cursor tot einde woord ('delete word')
`D`	van cursor tot einde regel
`dd`	hele regel
`p`	zet weggehaalde tekst weer terug
`Y`	tekst klaarzetten voor `p` zonder hem weg te halen

tekst veranderen

functie	betekenis
`r`*x*	vervang letter door *x*
`s`*tekst* `esc`	vervang letter door *tekst*
`~`	verander hoofd- en kleinde letter
`J`	plak regels aan elkaar ('join')
`:s/`*aap*`/`*noot*`/c`	vervang op deze regel *aap* door *noot*
`:`*a*`,`*b*`s/`*aap*`/`*noot*`/`	vervang op regel *a* t/m *b* de tekst *aap* door *noot*
`:`*a*`,`*b*`s/`*aap*`/`*noot*`/g`	idem, meerdere keren op één regel
`:`*a*`,`*b*`s/`*aap*`/`*noot*`/c`	idem, vraag steeds om een bevestiging
`&`	herhaal laatste vervanging

stoppen

functie	betekenis
`:q`	verlaat vi, tenzij tekst veranderd is
`:q!`	verlaat vi, zonder veranderingen te bewaren
`:r` *file*	voeg tekst van file toe op de plaats van de cursor
`:w` *file*	schrijf de file
`:w!` *file*	overschrijf de file
`:wq`	schrijf file en verlaat vi
`ZZ`	schrijf file (indien nodig) en verlaat vi
`!`*opdracht*	doe tussendoor een andere UNIX-opdracht

Bijlage C

Praktikumopgaven

C.1 Opgave 1

Probleemstelling

Schrijf een programma dat een kalender afdrukt van een in te lezen maand en jaar (vergelijkbaar met de Unix-opdracht `cal`).

Specificatie

Het programma moet interactief te gebruiken zijn. Het moet de gebruiker daarom vragen om twee getallen in te tikken: een maandnummer en een jaartal. De invoer kan bijvoorbeeld zijn:

```
6 1994
```

De uitvoer moet er dan als volgt uitzien:

```
juni 1994
ma di wo do vr za zo
       1  2  3  4  5
 6  7  8  9 10 11 12
13 14 15 16 17 18 19
20 21 22 23 24 25 26
27 28 29 30
```

Het programma hoeft niet te werken voor het jaar 1582 en eerder. In dat jaar vond in Nederland (althans in Holland en Zeeland) de hervorming naar de Gregoriaanse kalender plaats. (In Engeland vond deze hervorming pas plaats in 1752; dat jaartal wordt ook door Unix' `cal` aangehouden. Ons programma moet echter voor Nederland werken!)

Het programma moet ook werken voor jaren in de toekomst. Natuurlijk moet het programma ook rekening houden met schrikkeljaren.

Het programma moet met een passende foutmelding worden afgebroken bij foutieve invoer (maandnummer ligt niet tussen 1 en 12, of jaartal voor 1583).

Methode

In dit programma spelen twee problemen een rol: het bepalen op welke dag van de week de eerste van de maand valt, en de lay-out van de uitvoer.

Voor het eerste probleem kun je gebruik maken van de volgende observatie: $365 = 52 \times 7 + 1$. Het begin van elk jaar verschuift dus elk jaar één dag: 1 januari 1991 valt op een dinsdag, en 1 januari 1992 op een woensdag. Geef de dagen van de week een nummer (0=maandag, 1=dinsdag, ...). Neem het dagnummer van 1 januari in een ver verleden, tel er het aantal verstreken jaren bij op, en tel er dan nog de lengtes van de reeds verstreken maanden bij. Verwaarloos veelvouden van 7 in het resultaat, en je weet het dagnummer van de eerste van de gevraagde maand.

Het wordt iets gecompliceerder omdat je ook rekening moet houden met schrikkeljaren: voor elke reeds verstreken maand februari van schrikkeljaren moet je 1 extra bij het resultaat tellen. Voor het bepalen of een jaar een schrikkeljaar is geldt de volgende regel:

- een jaar is een schrikkeljaar als het deelbaar is door 4 (dus 1992 is een schrikkeljaar);
- uitzondering: eeuwjaren zijn geen schrikkeljaar (dus 1900 is geen schrikkeljaar);

- uitzondering op de uitzondering: 400-tallen zijn tòch een schrikkeljaar (dus 2000 is wèl een schrikkeljaar).

Heb je het dagnummer van de eerste van de maand gevonden, dan kun je de kalender afdrukken. In het ongunstigste geval beslaat een maand zes weken (bijvoorbeeld mei 1994). Je moet dus zes regels met getallen afdrukken. Het is het makkelijkste om op de maandag vóór de eerste van de maand te beginnen, en zes weken door te gaan. Juni 1994 begint dus op de '−1e' juni en eindigt op de '40e', en juli 1994 begint op de '−3e' en eindigt op de '38e'. Bepaal een formule die aangeeft hoe dit beginpunt afhangt van het dagnummer van de eerste van de maand. Voor datums kleiner dan 1 en groter dan de lengte van de maand druk je natuurlijk geen getal af, maar drie spaties. Ga op de juiste momenten over op een nieuwe regel.

Voor de tekst in de kop (zoals `juni 1994`) kun je een `switch`-opdracht of een array met teksten gebruiken.

C.2 Opgave 2

Probleemstelling

Beschouw het programma in hoofdstuk 16 van het diktaat. Met dit programma kan een zoeknetwerk worden opgebouwd, en kunnen paden tussen twee 'nodes' in dit netwerk worden gevonden. Het programma heeft echter een paar tekortkomingen:

1. Alle invoer komt van de stream `cin`. De gebruiker moet dus het hele netwerk intikken, voordat er interessante zoekopdrachten gegeven kunnen worden. Weliswaar kan met *input-redirection* (zie hoofdstuk 0.16) de invoer uit een file gehaald worden, maar dan moeten in die file ook de zoek-opdrachten staan. Handiger zou het zijn als de netwerk-opbouwopdrachten in een file staan, en de zoekopdrachten van het toetsenbord worden gelezen.

2. Het zoeken gebeurt in het wilde weg. Wie met de trein van Utrecht naar Lunteren wil, zal naar treinen in de richting Amersfoort kijken, en misschien ook de route via Ede proberen, maar in ieder geval niet eerst een reisroute via Rotterdam zoeken. Het programma is hier wel toe in staat.

3. Bij elke 'link' in het netwerk hoort een getal dat de kosten van de link voorstelt. In het programma wordt van deze getallen nergens gebruik van gemaakt: het programma vindt het allang mooi als het een weg tussen start- en goal-node vindt, en bekommert zich niet om de kosten die daaraan verbonden zijn.

Specificatie

Verander het programma `netmain` uit hoofdstuk 16, zodat deze tekortkomingen worden verholpen. Dat moet op de volgende manier gebeuren:

1. Het programma moet gestart kunnen worden met op de commandoregel nul of meer filenamen achter de naam van het programma. Het programma moet de inhoud van deze files eerst lezen, voordat het begint aan het lezen van `cin`. In de files staan hetzefde soort regels dat ook als `cin`-inover wordt geaccepteerd.

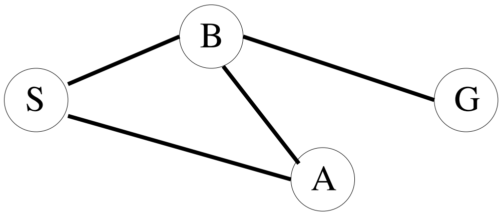
2. Het programma moet een 'heuristically informed method' gebruiken, zoals beschreven op blz. 70 van 'Artificial Intelligence'. Daartoe veranderen we de invoer-specificatie van het programma. Bij het toevoegen van een stad moeten ook de coördinaten van de stad opgegeven worden. De invoer kan dus bijvoorbeeld zijn:

```
stad Utrecht   136 456
stad Amsterdam 122 487
stad Hilversum 141 471
stad Haarlem   104 489
weg  Utrecht   Amsterdam 40
weg  Utrecht   Hilversum 20
weg  Hilversum Amsterdam 30
weg  Amsterdam Haarlem   20
zoek Utrecht   Haarlem
```

Bij het zoeken van een pad van Utrecht naar Haarlem moet het programma de link Utrecht–Amsterdam eerder proberen van de link Utrecht–Hilversum, omdat Amsterdam dichter bij Haarlem ligt dan Hilversum: de afstand Amsterdam–Haarlem is $\sqrt{(122-104)^2+(487-489)^2} = 18.1$ km, terwijl Hilversum–Haarlem $\sqrt{(141-104)^2+(471-489)^2} = 41.1$ km is. Merk op dat deze hemelsbrede afstanden ('as the crow flies', zegt Winston) niet hetzelfde zijn als de 'kosten' van de weg.

De coördinaat-informatie moet natuurlijk ook in de nodes van het netwerk worden opgeslagen.

3. De gretige manier waarop het programma door de heuristische methode uit (2) op zijn doel afstevent, levert niet altijd direct de beste route. Bijvoorbeeld in dit netwerk:



zal het programma de weg SA proberen vóór SB, omdat A dichter bij G ligt; de route die het eerst gevonden wordt is dan SABG, terwijl SAG natuurlijk goedkoper is.

Het programma moet dus niet de eerste oplossing opleveren die hij vindt, maar door blijven zoeken naar eventuele betere oplossingen. De beste oplossing moet uiteindelijk worden opgeleverd. Er moet een extra controle worden ingebouwd om hopeloze pogingen af te breken: als een nog niet compleet pad al langer is dan de beste-oplossing-tot-nu-toe, dan zal het met dat pad nooit meer wat worden. Op deze manier wordt voorkomen dat het programma na het vinden van het pad Utrecht–Amsterdam–Haarlem gaat proberen of het via Rotterdam–Eindhoven–Maastricht misschien goedkoper kan.

Aanwijzing: het is handig om in de *treenodes* de totale lengte van het pad dat erdoor wordt beschreven op te slaan.

Invoer

Het programma moet als invoer regels accepteren met het volgende formaat:

`stad` *naam getal getal*
`weg` *naam naam getal*
`zoek` *naam naam*
`probeer` *naam naam*
`test` *naam naam*

Ale getallen zijn integers tussen 1 en 1000. Het commando `zoek` moet de goedkoopste route tussen twee steden afdrukken. Het commando `probeer` moet hetzelfde doen, maar daarbij ook de (steeds beter wordende) eerdere oplossingen afdrukken. Het commando `test` doet hetzelfde, maar drukt *alle* paden (complete en niet-complete) paden af die gedurende het zoekproces in overweging zijn genomen. Bij elk pad wat afgedrukt wordt moeten ook de kosten ervan worden afgedrukt. Je mag er zonder controle van uitgaan dat de invoer correct is, en dat namen van steden hoogstens 19 letters hebben.

C.3 Opgave 3

Probleemstelling

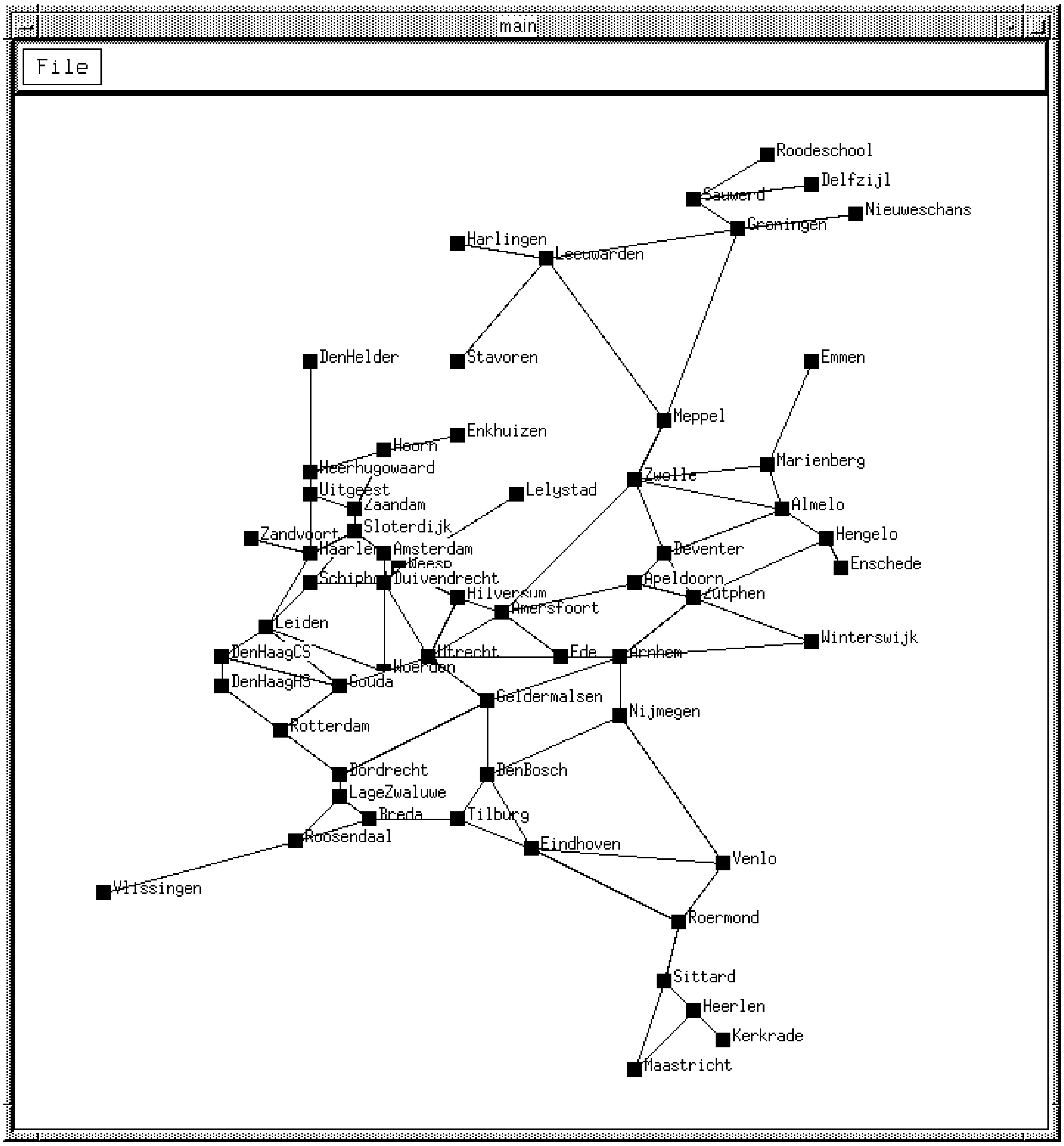
Het resultaat van opgave 2 is een programma dat de beste weg kan zoeken in een zoeknetwerk. De gebruikersvriendelijkheid van het programma laat nog enigzins te wensen over: de gebruiker moet startpunt en doel intikken, bijvoorbeeld

```
zoek DenHelder Hilversum
```

en krijgt dan de beste route te zien in de vorm van een lijst van tussenliggende plaatsen:

```
DenHelder - Heerhugowaard - Uitgeest - Zaandam -
Sloterdijk - Amsterdam - Weesp - Hilversum
```

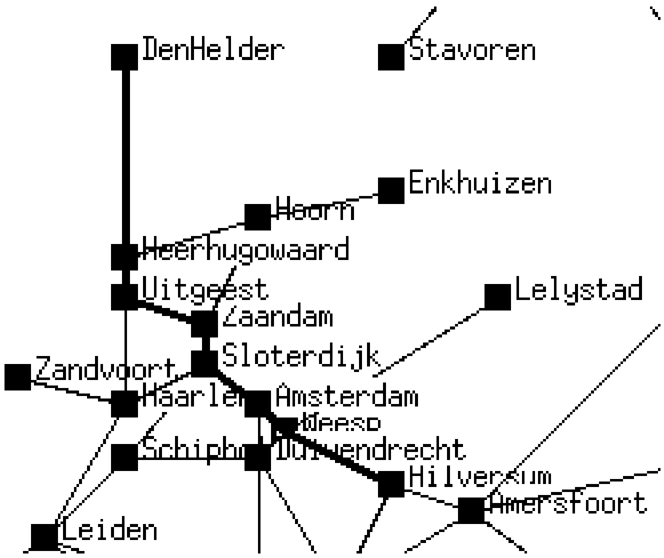
Er is een handiger interface denkbaar, waarin de gebruiker op een grafische weergave van het zoeknetwerk het startpunt en doel kan aanklikken met de muis, waarna de beste route ingetekend wordt. Een voorbeeld van zo'n interface staat hieronder:



Specificatie

Het programma moet een netwerk inlezen uit een file die hij als parameter krijgt, net als in opgave 2. In de file komen alleen regels voor die beginnen met `stad` en `weg`.

Het netwerk moet getekend worden in een window van 700×700 pixels: steden als vierkantjes van 10×10 met bijschrift, wegen als lijnen. Vervolgens moet de gebruiker twee steden kunnen aanklikken met de muis. Het programma moet de beste route tussen die twee steden met een dikke lijn (dikte 3) intekenen. Een voorbeeld hiervan staat hieronder:



Bovenin het window moeten verder nog de totale kosten van het getekende pad geschreven worden. Daarna mag de gebruiker weer twee nieuwe steden aanklikken, waarna *in plaats van* de eerder getekende route de route tussen die steden wordt getoond, enzovoorts.

Aanwijzingen

Uiteraard neem je het programma uit opgave 2 als uitgangspunt. De grafische interface kun je bouwen met de simwin-bibliotheek. De enige invoer die het programma nodig heeft (behalve zijn file-parameter) zijn muiskliks. Je moet dus een subklasse van `Grafisch` definiëren, waarin je de methode `Klik` herdefinieert. In het hoofdprogramma declareer je een object van dat nieuwe type.

In de methode `Klik` krijg je de coördinaten van het aangeklikte punt binnen. Je zult moeten bepalen welke stad daarbij hoort, op eenzelfde soort manier als in de methode `Net::findnode`. Bij een klik zul je moeten bepalen of het het begin- of eindpunt is. Is er nog niet eerder iets aangeklikt, dan was het blijkbaar een beginpunt. De bijbehorende (verwijzing naar een) stad moet je dan alleen maar opslaan. Daarvoor moet er dus een variabele in de klasse gedeclareerd worden.

Tref je in die variabele al een stad aan, dan is dit een tweede aangeklikte stad. Tussen dit start- en eindpunt kun je dan een route gaan zoeken, met de methode uit opgave 2. Dit levert je een pad op, dat getekend moet worden.

Voor het tekenwerk kunje de `print` methodes van de klassen `Net`, `NetNode`, `NetLink` en `TreeNode` aanpassen. In plaats van een `ostream&` waarop de uitvoer geschreven moet worden, krijgen deze methodes nu een `Grafisch&` parameter, waarop de uitvoer getekend moet worden. Vanuit je hergedefinieerde methode `Klik` kun je de print-methodes dan aanroepen met `*this` als parameter.

Gebruik op de juiste plaats de methode `Schoon`, zodat elk pad op een schoon kaartje getekend wordt.

Een complicatie is nog dat de methode `Klik` moet zoeken in het netwerk. Maar waar moet hij dat netwerk vandaan halen? Er moet dus een verwijzing naar het te gebruiken netwerk worden opgenomen in objecten van je nieuwe klasse.

Bijlage D

Voorbeeldtentamen

1. **Algemeen**

   (a) Wat is het verschil tussen een *opdracht* (Engels: *statement*) en een *expressie*?

   (b) Een expressie kan ook als opdracht gebruikt worden. Hoe? Geef een voorbeeld waarin dit nuttig is.

   (c) Waarvoor wordt het gereserveerde woord `friend` gebruikt?

   (d) Iemand schrijft de volgende `for`-opdracht:

   ```
   for (i=0; i!=10; i++)
   {   if (a[i]==x)
       {   cout << "gevonden";
           i = 10;
       }
   }
   ```

   De bedoeling is dat een array wordt doorzocht; op het moment dat de waarde `x` in de array wordt aangetroffen, wordt dit gemeld, en wordt de `for`-opdracht afgebroken. Helaas werkt dit programma niet zoals bedoeld was. Wat gaat er fout?

2. **Arrays**

   (a) Schrijf een functie met als parameters een niet-lege tekst en een array van teksten. Het einde van de array wordt aangegeven met een lege tekst (`""`). De functie moet opleveren hoe vaak de tekst in de array voorkomt.

   (b) Er zijn twee manieren om een array te declareren, die verschillen in het moment waarop de lengte van de array wordt vastgelegd. Geef van beide manieren een voorbeeld; wijs aan op welke plaats in het programma de lengte wordt gespecificeerd, en leg uit op welk moment die lengte wordt vastgelegd.

   (c) Beschrijf in woorden het *binary search* algoritme om snel te zoeken op welke plaats een element in een array voorkomt.
   Onder welke voorwaarde kan het algoritme worden gebruikt?
   Hoe kan de complexiteit van het binary-searchalgoritme worden omschreven?

3. **Lijsten**

   In het diktaat worden klassen `List`, `ListElem` en `All` gedefinieerd, waarmee lijsten van integers kunnen worden gemaakt. Dat gebeurt ongeveer zo:

   ```
   class ListElem
   {public:
       int       data;
       ListElem *next;
                 ListElem (int a, ListElem *b) {data=a; next=b; };
   };
   class List
   {public:
       ListElem    *first;
                    List  ()      {first=NULL; };
       int          empty ()      {return first==NULL; };
       void         add   (int x) {first = new ListElem(x, first); };
   ```

```
};
class All   // enzovoort
```

De methode `add` voegt een lijstelement toe aan het begin van een lijst.

(a) Stel dat de elementen in een lijst in opklimmende volgorde staan, dat wil zeggen het eerste element is het kleinste, het tweede element is groter (of gelijk) aan het eerste, enzovoort. In de klasse-definitie `List` voegen we een methode-declaratie toe:

```
void insert(int x);
```

Deze methode moet het element `x` op de juiste plaats in een lijst invoegen, dus zodat de opklimmende volgorde gehandhaafd blijft. Omdat de tekst van deze methode wat langer is dan die van `add`, wordt deze methode niet direct in de klasse-definitie gedefinieerd, maar daarbuiten.

Opgave: schrijf de definitie van de methode `insert`.

Hint: behandel apart de gevallen dat het element helemaal aan het begin moet worden toegevoegd, of elders in de lijst. Denk er aan dat de lijst ook leeg kan zijn.

(b) Wat moet er aan de definitie van de methode `insert` gewijzigd worden als de elementen van de lijst, in plaats van integers, verwijzingen naar objecten van een nader te bepalen type zijn? Je mag aannemen dat de operator `<` gedefinieerd is tussen twee objecten die zijn opgeslagen in de lijst.

(c) Om grote hoeveelheden gegevens op te slaan, kun je twee verschillende datastructuren gebruiken: arrays, of structuren die met verwijzingen aan elkaar hangen (zoals lijsten). Wat is het voordeel van lijst-achtige structuren? En het nadeel?

4. **Object-georiënteerd programmeren**

In deze opgave staat het volgende probleem centraal: een aantal kinderboeken moeten geordend worden op moeilijkheidsgraad. Zo kan bepaald worden in welke volgorde de boeken door basisschoolkinderen gelezen kunnen worden.

De moeilijkheidsgraad wordt uitgedrukt door een getal. Dit getal wordt berekend door het gemiddeld aantal woorden in de zinnen van het boek op te tellen bij de gemiddelde lengte van de woorden.

Woorden bestaan uitsluitend uit letters (hoofdletters en kleine letters). Zinnen worden afgesloten met een punt. Alle andere symbolen (leestekens, spaties, regelovergangen) zijn voor de analyse niet van belang, en dienen uitsluitend om woorden van elkaar te scheiden.

(a) Definieer een klasse `Boek` waarmee de relevante gegevens van een boek beschreven kunnen worden. Elk boek heeft een titel, die bij constructie van een object uit de klasse wordt gespecificeerd. Verder moeten er minstens twee methoden zijn: een methode `lees` waarmee het boek uit een te specificeren stream gelezen wordt, en een methode `graad` waarmee de moeilijkheidsgraad van de tot nu toe gelezen hoofdstukken bepaald wordt. Het is mogelijk dat `lees` meerdere malen achtereen wordt aangeroepen, bijvoorbeeld als verschillende hoofdstukken uit verschillende streams gelezen moeten worden.

(b) Definieer de constructormethode van `Boek` en de methodes `lees` en `graad`.

(c) Definieer de operator `<` met twee boeken als parameter, die 1 oplevert als het eerste boek simpeler is dan het tweede, en 0 anders.

Aanwijzing: een operator kan op precies dezelfde manier worden gedefinieerd als een functie. Schrijf daarbij echter in plaats van de naam van de functie: `operator <`.

(d) Bij gebruik van het programma moeten op de commandoregel een aantal titels kunnen worden opgegeven. De titels bestaan uit één woord, en zijn tevens de naam van een file waarin de tekst van het betreffende boek staat. De uitvoer van het programma bestaat uit de titels van de boeken op volgorde van moeilijkheid.

Opgave: Schrijf dit programma.

Hint: je mag gebruik maken van de klasse `List` en zijn vrienden zoals in het diktaat gedefinieerd met als elementen objecten van een willekeurige klasse, en bovendien van de uitbreidingen uit opgave 3.

5. **Window I/O**

(a) Wat is het verschil tussen een gewone methode en een `virtual` methode?

(b) Waartoe dient de methode `Idle`, die in de window-bibliotheek simwin wordt gedefinieerd, en waarom is er in niet-window programma's minder behoefte aan zo'n methode?

(c) Schrijf een programma dat een window van 400×400 pixels aanmaakt. Gecentreerd rond de plaats waar de gebruiker met de muis klikt moet een zwart vierkantje getekend worden.

Als de gebruiker op de 4-toets drukt, krijgen alle vierkantjes die daarna getekend worden diameter 4. Zo kan de gebruiker vierkantjes tekenen met diameters tussen 1 en 9.

In het window moet ook een menu File aanwezig zijn, met daarin keuzemogelijkheden Save en Quit. Bij keuze Save moeten van alle reeds getekende vierkantjes de coordinaten van de linkerbovenhoek en rechteronderhoek naar `cout` geschreven worden. Bij de keuze Quit moet de default-aktie van simwin hiervoor uitgevoerd worden.

Bijlage E

Lettercode-tabel

	0x0.		0x1.		0x2.		0x3.		0x4.		0x5.		0x6.		0x7.	
0x.0	0	nul	16	dle	32	sp	48	Γ"30	64	Γ"40	80	Γ"50	96	Γ"60	112	Γ"70
0x.1	1	soh	17	dc1	33	Γ"21	49	Γ"31	65	Γ"41	81	Γ"51	97	Γ"61	113	Γ"71
0x.2	2	stx	18	dc2	34	Γ"22	50	Γ"32	66	Γ"42	82	Γ"52	98	Γ"62	114	Γ"72
0x.3	3	etx	19	dc3	35	Γ"23	51	Γ"33	67	Γ"43	83	Γ"53	99	Γ"63	115	Γ"73
0x.4	4	eot	20	dc4	36	Γ"24	52	Γ"34	68	Γ"44	84	Γ"54	100	Γ"64	116	Γ"74
0x.5	5	enq	21	nak	37	Γ"25	53	Γ"35	69	Γ"45	85	Γ"55	101	Γ"65	117	Γ"75
0x.6	6	ack	22	syn	38	Γ"26	54	Γ"36	70	Γ"46	86	Γ"56	102	Γ"66	118	Γ"76
0x.7	7	bel	23	etb	39	Γ"27	55	Γ"37	71	Γ"47	87	Γ"57	103	Γ"67	119	Γ"77
0x.8	8	bs	24	can	40	Γ"28	56	Γ"38	72	Γ"48	88	Γ"58	104	Γ"68	120	Γ"78
0x.9	9	ht	25	em	41	Γ"29	57	Γ"39	73	Γ"49	89	Γ"59	105	Γ"69	121	Γ"79
0x.A	10	lf	26	sub	42	Γ"2A	58	Γ"3A	74	Γ"4A	90	Γ"5A	106	Γ"6A	122	Γ"7A
0x.B	11	vt	27	esc	43	Γ"2B	59	Γ"3B	75	Γ"4B	91	Γ"5B	107	Γ"6B	123	Γ"7B
0x.C	12	ff	28	fs	44	Γ"2C	60	Γ"3C	76	Γ"4C	92	Γ"5C	108	Γ"6C	124	Γ"7C
0x.D	13	cr	29	gs	45	Γ"2D	61	Γ"3D	77	Γ"4D	93	Γ"5D	109	Γ"6D	125	Γ"7D
0x.E	14	so	30	rs	46	Γ"2E	62	Γ"3E	78	Γ"4E	94	Γ"5E	110	Γ"6E	126	Γ"7E
0x.F	15	si	31	us	47	Γ"2F	63	Γ"3F	79	Γ"4F	95	Γ"5F	111	Γ"6F	127	del

Bijlage F

Belangrijke begrippen

Gereserveerde woorden

81	**break**	opdracht: afbreken **switch**-opdracht
81	**case**	punt waar een **switch**-opdracht begint
50	**char**	type: symbolen
51	**class**	begin van klassedefinitie
50	**const**	constante-definitie
9	**double**	type: floating-point getal met dubbele precisie
10	**else**	gedeelte van **if**-opdracht
9	**float**	type: floating-point getal
19	**for**	opdracht: speciale vorm van herhaling
61	**friend**	deel van een klasse-definitie
10	**if**	opdracht: voorwaardelijk uitgevoerde opdracht
6	**int**	type: gehele getallen
43	**new**	expressie: maak nieuw geheugen
61	**public**	deel van een klasse-definitie
22	**return**	opdracht: retourneren functieresultaat
81	**switch**	opdracht: keuze uit meerdere alternatieven
67	**template**	klassedefinitie met typevariabelen
89	**virtual**	herdefinieerbare functie
27	**void**	type: resultaat van functie zonder resultaat
14	**while**	opdracht: herhaling

Andere belangrijke namen

3	**#include**	toevoegen van bibliotheek
38	**<fstream.h>**	bibliotheek: files
2	**<iostream.h>**	bibliotheek: invoer- en uitvoer
11	**<math.h>**	bibliotheek: wiskunde-functies
42	**<string.h>**	bibliotheek: string-functies
2	**main**	functie: hoofdprogramma
5	**cin**	object: invoerstream
2	**cout**	object: uitvoerstream
38	**ifstream**	class: type van streams
42	**strcpy**	functie: string kopiëren
42	**strcmp**	functie: strings vergelijken
42	**strlen**	functie: lengte van een string
23	**width**	methode van streams: breedte van uivoerveld
49	**get**	methode van streams: leen eerste karakter
2	**endl**	constante: einde van de regel
11	**sqrt**	functie: vierkantswortel

Bijlage G

Uitwerking werkkollegeopgaven

3.1
```
if (a==0.0)
{   // vergelijking  bx+c=0
    if (b==0.0)
    {
        if (c==0.0)
            cout << "elke x is een oplossing";
        else
            cout << "geen oplossing";
    }
    else
        cout << "Een oplossing " << -c/b ;
}
else if (discr<0.0)
```
rest van het programma blijft ongewijzigd.

3.2 Een waarde ligt dichtbij een andere waarde als de absolute waarde van het verschil klein is. De functie `abs` is beschikbaar in de bibliotheek `<math.h>`.

```
#include <iostream.h>
#include <math.h>
int main()
{   float x, y;
    cin >> x;
    y = 1;    //  geen goede benadering, maar in ieder geval iets!
    while (abs(y*y-x)>1e-6)
        y = (y+x/y) / 2;
    cout << y;
}
```

3.3 Er staat een puntkomma achter `if (getal<0)`. De voorwaardelijk uitgevoerde opdracht is de lege opdracht. De volgende opdracht is, onvoorwaardelijk, `cout<<"negatief";`.
'Hé, wat doet die `else` daar?'

4.1 Hier is een vorm van herhaling, of eventeueel functies, nodig; met alleen `if` lukt het niet.

```
for (x=0; x<2; x++)
    if (x==1) A;
    else      B;
```

4.2 Net zolang delen door 10 totdat er nul uitkomt. Even uitkijken voor het getal 0 zelf.

```
cin >> getal;
aantal = 0;
if (getal<0) getal = -getal;
if (getal==0) aantal++;
while (getal>0)
{
    getal /= 10;
    aantal ++;
}
```

5.1 Na een paar stappen zie je de regelmaat. (Met volledige inductie kun je dit desgewenst ook bewijzen).

n	0	1	2	3	4	5	100	n
a	0	1	8	27	65	125	1000000	n^3
b	1	7	19	37	61	91	30301	$3n^2+3n+1$
c	6	12	18	24	30	36	606	$6n+6$

6.1 Bij het berekenen van de absolute waarde moet bij negatieve getallen het teken worden omgedraaid; andere getallen veranderen niet.

```
double abs( double x )
{   if (x<0) return -x;
    return x;
}
```

Voor de tekst `return x` kun je desgewenst `else` zetten. Maar deze opdracht wordt toch al niet uitgevoerd als `x<0`, want met de `return`-opdracht wordt de verwerking van de functie afgebroken.

6.2 De waarde waarvan de wortel bepaald moet worden is nu parameter, en wordt dus niet meer ingelezen. De hulpvariabele `y` moeten we nog apart declareren. Aan het eind van de functie staat geen uitvoeropdracht, maar een `return`-opdracht. Het resultaat van de functie zou immers nodig kunnen zijn in een grotere expressie; deze functie moet dan geen tussenresultaten op het scherm schrijven. Gebruikers van de functie die het resultaat toch op het scherm willen zetten, moeten dat zelf maar doen.

```
double wortel( double x )
{   double y;
    y = 1;    //  geen goede benadering, maar in ieder geval iets!
    while (abs(y*y-x)>1e-6)
        y = (y+x/y) / 2;
    return y;
}
```

6.3 We proberen gewoon alle delingen. Gaat er een deling op, dan is het zeker geen priemgetal, en kunnen we stoppen. Overleeft de functie al deze tests, dan is het wel een priemgetal.

```
int priem( int x )
{   int d;
    for (d=2; d<x; d++)
        if (x%d==0)
            return 0;
    return 1;
}
```

7.1 Deze functie drukt iets af, en heeft dus resultaattype `void`. Met een aantal `if`-opdrachten zorgen we ervoor dat er geen ongewenste `1` wordt afgedrukt, de exponent alleen als dat nodig is, enzovoort.

```
void polynoom( int n )
{   int c, k;
    for (k=n; k>=0; k--)
    {   c = boven(n,k);
        if (c>1 || k==0) cout << c;
        if (k>0) cout << "x";
        if (k>1) cout << "^" << k;
        if (k>0) cout << " + ";
    }
    cout << endl;
}
```

7.2 De functie moet blijvend resultaat hebben op zijn parameters, dus die worden als 'veranderbare' parameters gedeclareerd. In de functie is een hulpvariabele nodig, omdat de waardes niet gelijktijdig veranderd kunnen worden.

```
void verwissel( int &x, int &y )
{  int z;
   z = x;
   x = y;
   y = z;
```

```
}
```

8.1 Een hek met 10 segmenten heeft 11 palen nodig. Van jou weet ik het niet, maar ik ben 30, en dus met mijn 31e jaar bezig. Het is 1994, en toch al de 20e eeuw. Jezus schijnt achteraf gezien in het jaar 4 geboren te zijn, omdat de paus die in de 5e eeuw de jaartelling bedacht het ook niet precies meer wist. Maar afgezien daarvan: het jaar 1 is het eerste levensjaar van Jezus. Er was geen jaar 0. De 1e eeuw loopt dus van 1 t/m 100, de 2e eeuw van 101 t/m 200, en de 20e van 1901 t/m 2000. Het volgende millenium begin dus op 1 januari 2001. Maar op 1 januari 2000 is het ook feest, hoor. Je ziet wel weer hoe erg je moet oppassen met rangtelwoorden, net als met rangnummers van array-elementen.

8.2 Het relevante fragment:

```
float gemiddeld, somafwijking;
gemiddeld = (float)totaal/aantal;
somafwijking = 0;
for (getal=0; getal<11; getal++)
    somafwijking += turf[getal] * abs(getal-gemiddeld);
cout << "spreiding: " << somafwijking/aantal;
```

8.3 De eerste rij is gemakkelijk te initialiseren. Elke volgende rij berekenen we uit de vorige. Het getal 'er recht boven' zit al in het betreffende arrayelement; dit moet dus worden opgehoogd met het getal 'er schuin linksboven'. Dit getal moet nog de 'oude' waarde zijn, daarom lopen we de array van rechts naar links langs.

```
int a[11];
// initialisatie van de "nulde rij"
a[0] = 1;
for (x=0; x<10; x++)
{
   // afdrukken van de rij
   for (y=0; y<=x; y++)
       cout << a[y];
   cout << endl;
   // uitrekenen volgende rij
   a[x+1] = 1;
   for (y=x; y>0; y--)
      a[y] += a[y-1];
}
```

9.1 De invoerfile kan meer dan 1000 artikelen bevatten, waardoor de array buiten zijn grenzen wordt aangesproken. Een controle moet het ophogen van aantal verhinderen:

```
if (aantal<1000)
    aantal++;
else
    cout << "maximaal aantal artikelen (1000) overschreden";
```

Een verkeerde verbetering is de assignments aan `codes[aantal]` en `bedragen[aantal]` te verbeteren; dan wordt namelijk nog wel `aantal` opgehoogd, waardoor `zoek` de array op verboden plaatsen bekijkt.

9.2 Je kunt de array-elementen direct inlezen:

```
aantal = 0;
while (invoer >> codes[aantal] >> bedragen[aantal])
    aantal++;
```

of nog compacter:

```
for (aantal=0; invoer>>codes[aantal]>>bedragen[aantal]; aantal++) ;
```

10.1 De functie `strlen` krijgt een char array als parameter, en inspecteert net zoveel elementen totdat de eindmarkering wordt aangetroffen:

```
int strlen( char t[] )
```

```
{  int n;
   n = 0;
   while (t[n] != 0)
      n++;
   return n;
}
```

Iets dergelijks geldt voor de functie `strcpy`. Je moet alleen nu niet vergeten ook de kopie van een eindmarkering te voorzien:

```
void strcpy( char kopie[], char orig[] )
{  int n;
   n = 0;'
   while (orig[n] != 0)
   {  kopie[n] = orig[n];
      n++;
   }
   kopie[n] = 0;
}
```

11.1 Woordenboek met 50000 woorden kost gemiddeld 25000 vergelijkingen met de lineaire methode, en 15 met de binaire methode. Op een pagina staan circa 400 woorden; in een boek dus 40000. Totale opzoektijd met de lineaire methode is 40000 maal 25000 microseconden is ruim een kwartier. Met de binaire methode 40000 maal 15 microseconden is ruim een halve seconde. Tel uit je winst!

11.2 Bijvoorbeeld: van=6, tot=11, lengte=5 (oneven); het is immers 'tot en zonder'. Dan geldt `(tot+van)/2` is 17/2 is 8 (naar beneden afgerond). Dit is inderdaad het midden, want ervoor liggen elementen 6 en 7, en erna elementen 9 en 10.

Een interval met even lengte: van=6 en tot=10, lengte=4 (even). Het gemiddelde is nu 16/2 is ook 8, dat wil zeggen het element iets rechts van het midden. Dat geldt ook voor andere even lengtes.

11.3 In sommige gevallen blijft het algoritme dan steeds hetzelfde interval ter lengte 1 proberen, zonder dat het zoek-interval nog kleiner wordt. Het programma blijft dan oneindig lang doorgaan.

11.4 Dan wordt in sommige (de meeste) gevallen ten onrechte besloten dat de gezochte waarde niet in de array voorkomt.

12.1

```
int main(int aw, char *w[])
{
   int n=0; char c;
   ifstream invoer(w[1]);
   while (invoer.get(c))
      if (c==10) n++;
   cout << n << " regels";
}
```

12.2

```
class String
{
  char *s;
public:
  int  len   ();
  void hoofd ();
       String(char*);
}
String::String(char*x) {s=x;}
int String::len()      { return strlen(s); }
void String::hoofd()
{  int i;
   int n = len();
   for (i=0; i<len; i++)
      if ('a'<=s[i] && s[i]<='z')
          s[i] = s[i] - 'a' + 'A';
}
```

13.1 De array `turf` wordt twee-dimensionaal. Er is geen reden om de rijen met `new` te creëren, zoals in woordenboek, want de tabel is vierkant. Dus:

```
int turf[MAX][MAX];
```

De lees-methode wordt aangepast:

```
void Freqtab::lees( istream &s )
{
   char c1, c2;
   c1 = 0;
   while ( s.get(c2) )
   {  verhoog(c1,c2);
      c1 = c2;
   }
}
```

Verhoog krijgt natuurlijk ook twee parameters. Of je combinaties met leestekens ook wilt meerekenen mag je zelf bedenken.

13.3 Niet geweldig goed. Maar het kan wel de Bildzeitung van de New York Times onderscheiden.

13.4 Natuurlijk niet! Want dan zou je iemand te grazen kunnen nemen door zijn stijl te imiteren. Natuurlijk wel! Cijfers liegen immers niet?

16.1 Het algoritme is depth-first. Het wordt breadth-first door in **netzoek.cc** in regel 29 **addAtBegin** te vervangen door **addAtEnd**.

16.2 De nieuw te proberen paden worden aan de queue toegevoegd. Vlak daarvoor kun je ze dus uitprinten, door voor regel 29 in te voegen: **ntn->print(cout);**. Het is zeer instructief om dit daadwerkelijk uit te proberen; je ziet dan goed het verschil tussen depth-first en breadth-first.

17.1 De klasse **ifstream** (input-file-stream) is een subklasse van **istream** (input-niet-noodzakelijk-file-stream). In regel 41 op blz 77 geven we inderdaad een **ifstream** mee op de plaats waar een **istream** wordt verwacht.

18.1 Herdefinieer de methode **Toets**:

```
void Lijnen::Toets(char c)
{
   switch(c)
   {
    case 'B': ZetKleur(1); break;
    case 'G': ZetKleur(4); break;
    case 'R': ZetKleur(16); break;
    case 'Y': ZetKleur(20); break;
    case 'M': ZetKleur(17); break;
    case 'C': ZetKleur(5); break;
    case 'Z': ZetKleur(0); break;
    default:  if (c>'0' && c<='9')
              LijnDikte(c-'0');
   }
}
```

18.2 Maak een nieuwe klasse, waarmeen één lijn wordt beschreven, door x- en y-coordinaat en eventuele kleur en dikte. Voeg een variabele toe aan de klasse **Lijnen**, namelijk een lijst van objecten van die nieuwe klasse. Herdefinieer de methode **Menu**: hierin moet de hele lijst langsgelopen worden, waarbij elk element wordt geprint. Daarna moet de oude **Menu** worden aangeroepen door **Grafisch::Menu()**, zodat het programma inderdaad gestopt wordt.

19.1 Voeg twee nieuwe int-variabelen toe aan de klasse, die het nieuwe punt linksboven aanduiden, zeg **xlb** en **ylb**. Initialiseer deze met de waarde **-1**. Herdefinieer nu de methode **Klik** (je moet hem dus ook al aankondigen in de klasse-definitie). In de functie kijk je of er al eerder een punt was aangeklikt. Zo nee, dan krijgen **xlb** en **ylb** hun waarde. Zo ja, dan wordt het nieuwe middelpunt uitgerekend, en het tekenen gestart met een aanroep van **Opnieuw**.

```
void Mandel::Klik(int x, int y)
{
    if (xlb==-1)
    {
        xlb = x;
        ylb = y;
    }
    else
    {
        x0 = xlb;
        y0 = ylb;
        xlb = -1;
        br = x - xlb;
        h  = y - ylb;
        if (br<h) br = h;
        interval = interval * br / breedte;
    }
}
```

Het inlezen van `interval` in `Opnieuw` kan nu achterwege blijven.

Voor het afvangen van toetsaanslagen herdefiniëren we `Toets`:

```
void Mandel::Toets(char c)
{
    switch(c)
    {
     case '+': makk++; break;
     case '-': makk--; break;
    }
}
```

19.2 De essentieële regels in de `Kleur`-functie worden:

```
newa = a*a*a - 3*a*b*b + x;
newb = 3*a*a*b - b*b*b + y;
```

Wordt ook mooi. Probeer maar.