Dit document bevat teksten die direct aansluiten op de hoorcolleges van het eerstejaars vak *Programmeermethoden*, Universiteit Leiden, najaar 2016, zie (ook voor de sheets)

`www.liacs.leidenuniv.nl/~kosterswa/pm/`

De hoofdstukken 1 en 2 gaan over computers en over UNIX, en zijn apart verkrijgbaar. Met dank aan allen die aan deze tekst hebben bijgedragen.
Walter A. Kosters, Leiden, 31 augustus 2016.

3 C++: Concepten en programmeervoorbeelden

In dit gedeelte komen de C++-concepten aan bod die bij het vak Programmeermethoden belangrijk zijn en behandeld worden. Ze worden geïllustreerd met korte, duidelijke voorbeeldprogramma's.

3.1 Bibliotheken

Een groot deel van de C++ functionaliteit is opgeslagen in zogeheten *bibliotheken*. Deze bibliotheken bevatten een collectie standaard-instructies op een bepaald gebied. In bijna ieder programma gebruiken we bijvoorbeeld de bibliotheek `iostream`, die we als volgt in het programma opnemen:

```
#include <iostream>
```

In het oude C++ was dit nog:

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

In C++ laten we de lijst van standaard-bibliotheken volgen door:

```
using namespace std
```

De `iostream` bibliotheek bevat instructies op het gebied van de in- en uitvoer. Een veel gebruikt voorbeeld hiervan, `cout`, vonden we al terug in het eerste programma. Hier wordt iets mee op het scherm afgedrukt. Zonder het gebruik van de `iostream` bibliotheek zou deze opdracht niet "herkend" worden door C++ (de compiler eigenlijk).
In veel programma's wordt ook de bibliotheek `cstdlib` gebruikt.

3.2 Commentaar

Het is mogelijk én gebruikelijk C++ programma's te voorzien van stukken tekst die in het programma zelf niets doen, maar commentaar zijn voor de menselijke lezer — de programmeur zelf of anderen.
Commentaar ziet er als volgt uit:

```
// Deze tekst wordt niet "uitgevoerd"
```

Het kan overal aan het einde van een regel toegevoegd worden. Commentaar over meerdere regels of zelfs tussen opdrachten is ook een mogelijkheid en ziet er zo uit:

```
/*Ook deze twee regels aan commentaar bij het programma worden
niet uitgevoerd*/
```

Voor het gebruik van commentaar in de programma's, zoals die voor het vak Programmeermethoden gemaakt moeten worden, zijn richtlijnen opgesteld.

3.3 Variabelen

Aan de basis van de meeste programmeertalen staat het concept: variabele. Een variabele is in feite een klein stukje in het (snelle) geheugen dat gereserveerd wordt voor het opslaan van een bepaald type data. In C++ is het verplicht dit stukje geheugen eerst te reserveren voor een bepaald type, het zogenaamde "declareren van een variabale". Dit gaat bijvoorbeeld als volgt:

```
int zakgeld;
```

Hiermee reserveren we een stuk geheugen voor het opslaan van een geheel getal (`int`). Na de declaratie kunnen we allerlei operaties uitvoeren op de variabele `zakgeld`, bijvoorbeeld:

```
zakgeld = 300; // erg lief kind
```

Waarmee we het getal `300` opslaan op de geheugenlocatie van zakgeld.
Er zijn een heleboel verschillende typen mogelijk, die soms goed en soms minder goed uitwisselbaar zijn. Bij het vak Programmeermethoden zijn dit de belangrijkste:

`int`	Een geheel getal
`double`	Een (benadering van een) reëel getal
`char`	Een karakter
`bool`	Waar of onwaar

Later volgt meer informatie over het werken met de verschillende typen variabelen.

3.4 Condities

Een van de belangrijkste mogelijkheden in een programma is het uitvoeren van verschillende stukken code afhankelijk van een bepaalde conditie. Denk hierbij bijvoorbeeld aan: Als `x` groter is dan `0` dan is `x` positief, anders is `x 0` of negatief. Bovenstaande conditie zou er in C++ bijvoorbeeld zo uit kunnen zien:

```
if ( x > 0 ) {
  cout << "x is positief" << endl;
}  // if
else {
  cout << "x is 0 of negatief" << endl;
}  // else
```

Indien er meer dan een "eis" wordt gesteld dan wordt dat zo gedaan:

```
if ( ( x > 0 ) && ( x < 25 ) )
```

Meervoudige condities als `0.0 <= x <= 1.0` worden in C++niet op die manier geformuleerd. `if ( 0.0 <= x <= 1.0 )` betekent namelijk iets anders. De genoemde conditie wordt dan ook als `if ( 0.0 <= x && x <= 1.0 )` in de C++-code opgenomen. Iets dergelijks geldt ook voor `if ( x and y > 0 )`, waarschijnlijk wordt in dit geval `if ( x > 0 and y > 0 )` bedoeld. Merk tevens op dat bij de regel met `&&` de tweede test niet meer gedaan wordt als de eerste al uitsluitsel geeft. Een constructie als `if ( ( x != 0 ) && ( 1.0 / x == 0.5 ) )` is dus toegestaan. Dit fenomeen heet *shortcircuiting*; ook hoor je hier wel eens de term *lazy evaluation* vallen. In de nieuwste C++-versies mag overigens in plaats van `&&` weer `and` gebruikt worden, en analoog `or` voor `||` en `not` voor `!`.
Laten we nu eens kijken naar een compleet voorbeeld-programma waar in de tot nu toe besproken concepten naar voren komen. De eerste regel van dit programma bevat commentaar voor

```
// Dit is een regel met commentaar ...
#include <iostream>          // moet er in ieder C++-programma bij
using namespace std;
const double pie = 3.14159;  // een constante (beter uit cmath)

int main ( ) {
  double straal;  // straal van de cirkel
  cout << "Geef straal, daarna Enter .. ";
  cin >> straal;
  if ( straal > 0 )
    cout << "Oppervlakte " << pie * straal * straal << endl;
  else
    cout << "Niet zo negatief ..." << endl;
  cout << "Einde van dit programma." << endl;
  return 0;
}  // main
```

Voorbeeldprogramma 1: Straal berekenen

de menselijke lezer. In de tweede en de derde regel wordt verteld dat er instructies uit de bibliotheek `iostream` gebruikt zullen gaan worden. Dan volgt een regel waarin aan de naam `pie` het getal `3.14159` gekoppeld wordt; dit is een constante, een geheugenlocatie die verderop in het programma niet meer gewijzigd kan en mag worden.
Het eigenlijke C++-programma begint bij de regel met `main`. Eén voor één worden dan de regels (of beter *statements*, de opdrachten) uitgevoerd. Eerst wordt een variabele met de naam `straal` aangemaakt; de betreffende geheugenruimte kan een reëel getal (komma-getal) — of beter gezegd een benadering daarvan — bevatten. Dan komt er een tekst op het beeldscherm via de `cout`-regel. Alles wat tussen de `"`'s staat, komt letterlijk op het scherm. Dankzij de `cin`-regel wordt de door de gebruiker ingevoerde waarde gestopt in de variabele `straal`. Dan komt een `if`-statement, dat verscheidene regels beslaat. Indien `straal` een waarde groter dan `0` bevat, wordt de oppervlakte van de betreffende cirkel uitgerekend en afgedrukt (zie Hoofdstuk 3.9.2 voor meer over rekenen). Merk op dat wanneer de `"`'s ontbreken, de waarde van een uitdrukking (expressie) wordt afgedrukt door een `cout`-statement, maar met `"`'s erbij wordt letterlijk wat er tussen de `"`'s staat op het scherm gezet. In één `cout`-statement kunnen ook diverse zaken,

gescheiden door <<'s, afgedrukt worden. Indien straal een waarde kleiner dan of gelijk aan 0 bevat, wordt de tekst Niet zo negatief ... op het scherm gezet, waarbij dankzij de endl de cursor naar een nieuwe regel gaat; de endl zorgt er trouwens ook voor dat de "output-buffer" (alles dat in de wachtrij staat om uitgevoerd te worden) op het beeldscherm geleegd wordt: hij *flusht*. Tot slot wordt nog een tekst afgedrukt, waarna het programma klaar is: met return 0; wordt het beëindigd.
Zou een van de laatste regels uit het programma zo ingesprongen zijn dat de twee cout-regels direct onder elkaar staan, dan maakt dat niets uit voor de werking; een if of een else (of later andere *block*-statements) beslaan standaard alleen de eerstvolgende instructie. Er wordt een ander effect bereikt door een { voor de op een na laatste cout-regel en na de laatste cout-regel te zetten: deze horen dan bij elkaar en worden als één statement, horend bij de else, opgevat. De laatste cout-regel wordt dan niet meer uitgevoerd als straal groter dan 0 is. Meer over de richtlijnen voor inspringende regels is overigens elders te vinden.
Denk er aan dat een enkele = in C++ een toekenning is, en niet een test op gelijkheid. Zo zou in bovenstaand programma de regel if (straal = 0) ongeacht de ingevoerde waarde van straal resulteren in het uitvoeren van de else-situatie. Er wordt namelijk 0 in straal gestopt, wat ook meteen het resultaat van de toekenning is, en 0 is in C++ hetzelfde als false, "niet waar" dus. Testen op gelijkheid gaat in C++ met ==. In veel andere talen gaat een toekenning overigens met een :=.
Een ander bekend probleem is de zogenaamde *hangende else* (dangling else):

```
if ( a > 0 )
  if ( b > 0 )
    cout << "a en b groter dan 0" << endl;
  else
    cout << "???" << endl;
```

Nu is de afspraak dat een else hoort bij de laatste nog openstaande if, dus in ons voorbeeld bij if (b > 0). Wil je hem toch bij de eerste if laten horen, dan moeten er handig accolades worden gezet.
Een schoon scherm/window is overigens als volgt "eenvoudig" te verkrijgen:

```
#include <cstdio>
  ...
  system ("clear");  // op UNIX-systeem, voor DOS/Windows: "cls"
```

3.5 Loops

Het meerdere malen herhalen van een aantal statements is een integraal deel van elke programmeertaal. Herhalingen worden in C++ met behulp van for- en while-loops bewerkstelligd. Hoewel ze in zekere zin equivalent zijn, is het een goede programmeerpraktijk om een while-loop te gebruiken als het aantal herhalingen van te voren onbekend is ("net zolang zeuren totdat"), of lastig te bepalen, en for-loops te reserveren voor situaties waarbij het aantal herhalingen vast ligt ("drie maal bellen"). Enkele elementaire voorbeelden:
Zolang de waarde van i kleiner dan of gelijk aan n is, worden de regels tussen de binnenste accolades herhaald. Omdat we weten hoe vaak de regels tussen de accolades uitgevoerd moeten worden (namelijk n keer), horen we hier eigenlijk een for-loop te gebruiken. Hieronder staat

```
// druk eerste n getallen met hun kwadraat af
void kwadraten1 (int n) {
  int i = 1;  // tellertje, meteen initialiseren
  while ( i <= n ) {
    cout << i << " -- " << i * i << endl;
    i++;
  }  // while
}  // kwadraten1
```

Voorbeeldprogramma 2: Kwadraten 1

```
// druk eerste n getallen met hun kwadraat af
void kwadraten2 (int n) {
  int i;  // tellertje
  for ( i = 1; i <= n ; i++ )
    cout << i << " -- " << i * i << endl;
}  // kwadraten2
```

Voorbeeldprogramma 3: Kwadraten 2

hetzelfde programma waarbij de `while`-loop vervangen wordt door een `for`-loop. Er is meteen te zien dat dit een aantal statements scheelt:
Zelfs mag je zeggen `for ( int i = 1; i <= n ; i++ )`, waarbij `i` een locale variabele voor de `for`-loop wordt, die niet nog eens apart in de functie hoeft te worden aangemaakt. Let er wel op dat `i` dan niet buiten de `for`-loop mag worden gebruikt, iets waar verschillende compilers ook nog wel eens verschillend mee omspringen. Het is verstandig om de test niet `i != n+1` te laten zijn: het levert hier wel hetzelfde resultaat op, maar zou `i` per ongeluk een startwaarde groter dan `n+1` hebben gekregen, dan waren de gevolgen niet prettig. Algemeen geldt dat een `for`-loop een `while`-loop met een standaard tellertje is volgens het volgende schema:

```
for (maken en beginwaarde teller; zo lang gaat de loop door; hoe
wordt de teller veranderd)
```

Let hierbij op de plaats van de punt-komma's (`;`).
De volgende code is typisch geschikt voor een while loop:

```
while ( x != 1 )
  if ( x % 2 == 0 )
    x = x / 2;
  else
    x = 3 * x + 1;
```

Tot op heden is nog niet bekend of deze `while`-loop voor ieder positief geheel begingetal `x` stopt. En indien het stopt, is het nog maar de vraag wat het aantal doorgangen door de test is geweest. Het probleem staat onder meer bekend als het Syracuse-probleem, het Collatz-probleem of het $3x+1$-vermoeden.
Ook bij een programma als:

```
x = 1;
while ( x < 1000 )
  x = 2 * x;
```

is het eenvoudig in te zien dat het stopt en dat na afloop `x` de waarde `1024` heeft, maar het aantal doorgangen is in het algemeen — met `n` in plaats van `1000` — niet voor elke situatie vooraf vast te stellen; een `while`-loop geniet hier dus de voorkeur.
Soms komt de variant

```
do {
  cin >> getal;
} while ( getal <= 0 );
// nu is een positief getal ingelezen
```

goed van pas. Denk eraan dat de body (het stuk tussen accolades) van de loop hier altijd minstens één keer wordt uitgevoerd.

3.6 Functies

Het komt vaak voor dat een aantal regels code meerdere malen voorkomt in het te ontwerpen programma. Denk hierbij bijvoorbeeld aan het afdrukken van het schaakbord of het berekenen van een faculteit. Het is in de meeste talen, waaronder C++, mogelijk kleine subprogramma's te ontwerpen die deze taken op zich nemen. Deze kleine stukken code worden veelal functies genoemd. Hierboven zagen we er al enkele voorbeelden van (een functie die kwadraten afdrukt). Er bestaan twee soorten van functies: *functies* en *procedures*. Een functie voert handelingen uit en levert een resultaat op, bijvoorbeeld `kwadraat (x)` of `oppervlaktecirkel (straal)`. Een procedure voert slechts een aantal (vaak voorkomende) handelingen uit en heeft verder geen resultaat, zoals bijvoorbeeld `drukafplaatje ( )` of `savefile ("geld.txt");`.
Een functie wordt, net als een variabele, gekoppeld aan een bepaald type (*return-type*) en wordt dus ook gedeclareerd (functie-declaraties staan boven `main ( )`!). Dit gaat als volgt:

```
int kwadraat (int x) {
  int uitkomst;
  // doe hier de berekening
  return uitkomst;
}  // kwadraat
```

de functie wordt vervolgens als volgt in `main ( )` of andere functies gebruikt:

```
int coordinaat1 = kwadraat (10);
int coordinaat2 = kwadraat (8);
```

Procedures moeten ook gedeclareerd worden en hebben het type void:

```
void drukafplaatje ( ) {
  // instructies om het plaatje af te drukken
}  // drukafplaatje
```

en worden als volgt aangeroepen:

```
drukafplaatje ( );
```

Een speciale functie is `main ( )`. Dit is de functie waar het programma altijd mee begint. `main ( )` is een `int` -functie en ziet er (meestal) als volgt uit:

```
int main ( ) {
  // hier staat het hoofdprogramma
  return 0; // het getal 0 heet de exit-code van het programma
}
```

Het is doorgaans verstandig om vanuit functies zo weinig mogelijk uitvoer en invoer te plegen, behalve bij functies die daar speciaal voor bedoeld zijn; zo houd je functies algemeen toepasbaar.
Als je een functie al klaar hebt, en later eens een keer gebruikt, doe je aan *bottom-up* programmeren. Ga je de functie pas schrijven wanneer je hem nodig hebt, dan ben je *top-down* bezig.

3.6.1 Parameters

We zullen de mogelijkheden van een functie nu wat nader bekijken. We gebruiken daarvoor twee (verschillende) functies die getallen verwisselen. De eerste is meteen de “beste”.

```
void wissel (int& a, int& b) {
  int temp;  // ook mag (in een keer):
  temp = a;  // int temp = a;
  a = b;
  b = temp;  // en dus niet: a = b; b = a;
}  // wissel
```

Voorbeeldprogramma 4: Integers wisselen

We zagen net al dat sommige functies informatie nodig hebben (bijvoorbeeld de `x` waar het kwadraat van berekend moet worden). De benodigde informatie voor een functie wordt doorgegeven middels de *parameters*; de variabelen in de functie-declaratie (de *heading*). Deze parameters moeten ook gedeclareerd worden met een type en staan op volgorde. Na het aangeroepen van de functie kunnen de parameters binnen die functie gebruikt worden als gewone variabelen. De bovenstaande functie wisselt de inhoud van twee variabelen om. Het is dan ook logisch dat er twee parameters doorgegeven worden. Wat meteen opvalt is dat de parameters vergezeld zijn van een `&`. Dit brengt ons bij een belangrijk concept met betrekking tot parameters; het verschil tussen *call-by-value* en *call-by-reference*.
Bij het gebruik van bovenstaande functie is het duidelijk dat de programmeur wil dat na de aanroep

```
int x = 1;
int y = 2;
wissel (x,y);
```

de variabele `x` 2 bevat en de variabele `y` 1. Om dit te bewerkstelligen is het noodzakelijk dat alles wat met `a` gebeurd binnen de functie, ook met `x` gebeurd daarbuiten. Het is dus zo dat bij een aanroep als `wissel (x,y)` `a` als synoniem voor `x` functioneert: alles wat met de één

gebeurt, gebeurt met de ander. In feite wordt er bij aanroep met een `&` een referentie, een geheugenadres, doorgegeven. Dit heet call-by-reference. Zouden we in het voorbeeld de twee `&`'s weglaten, dan wordt de huidige waarde van `x` gekopieerd naar de variabele `a`. Zouden we die functie dan aanroepen met bijvoorbeeld `wissel (x,y)`, dan zouden er locale variabelen `a` en `b` worden gecreëerd, die als startwaarde de waarde van `x` respectievelijk `y` zouden krijgen. De twee variabelen zouden binnen de functie worden gewisseld, de functie wordt beëindigd en aan `x` en `y` zou niets veranderd zijn. Om het bovenstaande "wissel-doel" te bereiken zijn de `&`'s (de *en-percenten*) dus noodzakelijk. Bij een functie als:

```
int kwadraat (int x)
```

maakt het niet uit omdat de `x` zelf niet verandert of hoeft te veranderen.
In het geval van call-by-reference mag overigens een aanroep als `wissel (42,88);` niet meer: tussen de haakjes moet een *l-value* staan, iets waaraan je kunt toekennen en wat dus bijvoorbeeld ook links van de toekenning = mag staan. Rechts van toekenningen mogen *r-values* komen: dit kunnen ook expressies zijn als `42` of `m+2`.
Het is handig om de variabelen en parameters met verschillende functies ook anders te noemen. In het hieronder staande voorbeeld is te zien hoe we in het vervolg over de verschillende soorten variabelen en parameters zullen spreken.

```
const int geld = 300; //constante (verandert nooit)
char euro = 'E' ;     //globale variabele (in alle functies te gebruiken)

void maakmerijk (int & zakgeld, //een formele parameter
                                //call-by-reference
                 int bonus) {   //een formele parameter
                                //call-by-value
  int kwadraat = 0;             //een locale variabele
                                //alleen bruikbaar in 'maakmerijk'
  kwadraat = bonus * bonus;
  zakgeld += kwadraat;
}  // maakmerijk

int main ( ) {
  int bonus = 0;                //ook een locale variabale
                                //alleen bruikbaar in 'main'
  cout << "Hoeveel wil je erbij? .. ";
  cin >> bonus;
  euro = 'e';
  zakgeld = geld;
  maakmerijk (zakgeld, bonus);  //zakgeld en bonus zijn actuele parameters
  cout << "ik krijg " << zakgeld << euro << " zakgeld" << endl;
  return 0;
}  // main
```

Voorbeeldprogramma 5: Soorten variabelen

Als er "synoniemen" in het spel zijn, *lijkt* het ingewikkelder:

```
void alias (int r, int& s ) {
  int t;
  t = 3;
  r = r + 2;
  s = s + r + t;
  t = t + 1;
  r = r - 3;
  cout << r << s << t << endl;
}  // alias

int main ( ) {
  int t = 12;
  alias (t,t);
  cout << t << endl;
  return 0;
}  // main
```

De uitvoer van dit programma zal zijn (ga maar eens na):

```
11  29  4 29
```

Als in de eerste regel van de functie een `&` voor `r` wordt toegevoegd krijgen we echter:

```
28  28  4 28
```

Tot slot een wat getructe versie om te laten zien dat het wisselen ook zonder hulpvariabele kan:

```
// Verwissel (swap) de inhoud van a en b.
// Dit maal doen we het met een truc: zonder hulpvariabele.
// Het werkt niet als a en b dezelfde variabele zijn (aliases).
// In de praktijk wordt dit eigenlijk nooit gebruikt.
void wisseltruc (int& a, int& b) {
  a = a + b;  // a = a_oud + b_oud
  b = a - b;  // b = a_oud
  a = a - b;  // a = b_oud
}  // wisseltruc
```

Voorbeeldprogramma 6: Wisseltruc

Nogmaals: dit is alleen een aardigheidje, geen zinvolle programmeer-praktijk. De aanroep `wisseltruc (r,r)` levert ook onverwachte effecten!

3.7 Files

Soms is het nodig dat er informatie uit een bepaald bestand (*file*) gelezen wordt of dat het nodig is er informatie in op te slaan. In C++ is dit redelijk eenvoudig te realiseren. Het nu volgende programma kopieert een invoerfile onveranderd door naar een uitvoerfile, karakter voor karakter. Uiteraard kunnen de namen van de files ook in strings worden ingelezen en zo worden doorgegeven. Let er wel op dat als de filenaam van klasse `string` is, dat er soms naar "ouderwetse" C-stijl moet worden geconverteerd: stel je hebt

```
string naam;
```

gebruik dan

```
invoer.open (naam.c_str ( ),ios::in);
```

```
#include <iostream>
#include <fstream>
#include <cstdlib>   // voor exit
using namespace std;
int main ( ) {
  ifstream invoer;
  ofstream uitvoer;
  char kar;
  invoer.open ("invoer.txt",ios::in);  // koppel invoer aan (echte) file
  if ( ! invoer ) {  // of:  if ( invoer.fail ( ) )
    cout << "File niet geopend" << endl;
    exit (1);  // stopt het programma; return 1 was ook goed
  }  // if
  uitvoer.open ("uitvoer.txt",ios::out);
  kar = invoer.get ( );
  while ( ! invoer.eof ( ) ) {
    uitvoer.put (kar);
    kar = invoer.get ( );
  }  // while
  invoer.close ( );
  uitvoer.close ( );
  return 0;
}  // main
```

Voorbeeldprogramma 7: File kopiëren

Verder zijn er nog veel meer mogelijkheden met file-IO. Let er ook op dat de memberfunctie `eof` pas dan een zinvol resultaat geeft als er een leespoging gedaan is. In het bovenstaande programma schijnt het aantal `get`'s één groter te zijn dan het aantal `put`'s; echter, het afsluitende EOF-symbool wordt door `uitvoer.close ( );` gezet. Het letterlijk intikken van `EOF` gaat overigens soms ook goed (bijvoorbeeld zoiets als `if ( kar == EOF ) ...`); denk er wel aan dat dan `kar` een `int` moet zijn, want `EOF` is doorgaans `-1`!
Door voor de regel `uitvoer.put (kar);` een test als `if ( kar != 'e' )` te zetten worden alle `e`'s "overgeslagen". Er kan bijvoorbeeld ook op regelovergangen (*LineFeed*, `'\n'`) getest worden. In een DOS of Windows-omgeving worden deze doorgaans onmiddellijk voorafgegaan door een *CarriageReturn* (`'\r'`) die alleen in *binary mode* (met `ios::binary`) worden gelezen; bovenstaand programma kopieert ze wel goed, maar handelt met één `get` zowel de `'\r'` als de er meteen na staande `'\n'` af.
Ook bij het van `cin` lezen kunnen deze commando's worden gebruikt. Als `cin >>` en het beruchte `cin.get ( )` of `cin.getline ( )` door elkaar worden gebruikt treden er soms vervelende effecten op: een `\n` wordt schijnbaar soms wel en soms niet gelezen. Het is allemaal te

begrijpen, maar niet altijd even prettig. Zo slaat `cin >> getal` op zijn jacht naar `getal` bijvoorbeeld spaties en regelovergangen over, en heeft een regelovergang nodig om de buffer van `cin` binnen te krijgen.

3.8 Arrays en matrices

Het is in C++ mogelijk om kleine tabelletjes of rijen van getallen, karakters of andere variabelen bij te houden. Dit gebeurt in zogenaamde *arrays*. Een array ziet er typisch als volgt uit:

```
int rijtje[100];
```

Dit is een rijtje van `100` gehele getallen. Het getal dat aangeeft hoe groot het array is (hier dus `100`) moet een constante zijn, zodat de compiler weet hoeveel ruimte hij voor het array in het geheugen moet reserveren. Eventueel mag een array dus ook zo aangemaakt worden:

```
const int zogroot = 100;
int rijtje[zogroot];
```

Na het declareren van het array kunnen de verschillende vakjes als volgt benaderd worden:

```
rijtje[10] = 41;
```

Het getal `10` wordt de *index* genoemd — het geeft aan over welk vakje we het hebben —, het getal `41` is hierbij de (nieuwe) inhoud van dat vakje.
In tegenstelling tot een aantal andere talen loopt de index in C++ van `0...grootte - 1`. Dit wil zeggen dat we bij het bovenstaande array van `100` groot de vakjes van `0` tot `99` kunnen gebruiken. Het is overigens wel mogelijk gegevens naar vakjes boven de `99` te schrijven maar dit levert vaak onverwachte en onwenselijke effecten op. De meeste problemen die ontstaan bij het werken met arrays komen dan ook door het overschrijden van deze *bounds*.
Bij het doorgeven van een array aan een functie is het niet nodig tussen de vierkante haken de grootte aan te geven. Wel is het om bovenstaande reden vaak handig om deze grootte in een aparte variabele mee te geven:

```
void doeiets (int rijtje[ ], int grootte)
```

Arrays zijn gemakkelijk te vullen of uit te lezen met een `for`-loop. In het onderstaande voorbeeld worden alle getallen van een (eerder gevuld) array opgeteld.
Let erop dat de `for`-loop hierbij doorgaat tot en met het laatste getal voor `n`; in dit geval dus (correct) tot en met `99`.
Veelvuldig wordt er gewerkt met dubbele (2-dimensionale) arrays, beter bekend als *matrices*. Zo'n array, zeg

```
int A[n][n];  // n moet const int n = ... zijn
```

verbeeldt een vierkante `n` bij `n` matrix. Niet-vierkante matrices kunnen uiteraard ook worden gedefinieerd. Je kunt zelfs een nieuw type maken via

```
typedef int matrix[n][n];  // n moet const int n = ... zijn
```

```
const int n = 100;
int rijtje[n];
...
int telarrayop (int rijtje[ ], int n){
  int som = 0, i;
  for ( i = 0; i < n; i++) {
    som += rijtje[i];
  } // for
  return som;
}  // telarrayop
```

Voorbeeldprogramma 8: Array-elementen optellen

Het element uit de `i`-de rij en de `j`-de kolom van een variabele `A` van type `matrix` is te vinden in `A[i][j]`. Denk eraan dat rijen en kolommen beginnen te nummeren met `0`, en eindigen met `n-1`.
Net als deze 2-dimensionale arrays is het overigens net zo makkelijk mogelijk om 3- of meerdimensionale arrays te gebruiken.
Het doorgeven van meerdimensionale array's aan functies heeft lastige kanten. De volgende functie, die de som van alle array-elementen van het array `A` berekent, maakt een en ander hopelijk duidelijk:

```
// bepaal som van array-elementen uit eerste rijen van A
int sommatrix (int A[ ][10], int rijen) {
  int i, j, som = 0;
  for ( i = 0; i < rijen; i++ )
    for ( j = 0; j < 10; j++ )
      som += A[i][j];
  return som;
}  // sommatrix
```

Voorbeeldprogramma 9: Matrix-elementen optellen

De value-parameter `rijen` geeft het aantal rijen door, maar het aantal kolommen moet een `const` zijn, bijvoorbeeld 10. Dat moet de compiler weten om bij bijvoorbeeld `A[3][5]` het betreffende adres te kunnen berekenen. Dat adres is hier `A + 3*10+5`, of eigenlijk beter nog `A + (3*10+5)*sizeof (int)`, want bij + wordt met de stapgrootte, dat wil zeggen de grootte van de array-elementen in bytes, rekening gehouden. De waarde van `A` is hier het adres waar het array begint. De rijen van het array liggen vanaf dit adres achter elkaar in het geheugen, en de rijlengte (het aantal kolommen) moet dus van te voren vastliggen. De ontwerpers van C++ hebben uit efficiency-overwegingen voor deze aanpak gekozen. De *unaire* operator `sizeof` geeft de grootte van het betreffende type — of de variabele — in bytes.
Arrays en matrices worden vaak gebruikt voor het zoeken in of het sorteren van gegevens. Voorbeelden hiervan zijn te vinden in Hoofdstuk 4.2.

3.9 Werken met variabelen en getallen

We kijken nu hoe we elementaire bewerkingen kunnen uitvoeren met getallen.

3.9.1 Werken met verschillende types

Aan het converteren, afdrukken en werken met verschillende typen variabelen zitten een aantal haken en ogen. Deze worden hier behandeld.

Gehele getallen
Gehele getallen worden in C++ gerepresenteerd door het type `int` (van integer); meestal zijn dit 4 bytes. Een `int`-variabele kan vrijwel naar alle andere variabelen *gecast* (geconverteerd, gepromoveerd) worden. Stiekem zijn de types `char` (de getalsinhoud staat voor het nummer van het karakter in de *ASCII-tabel*) en `bool` (`0` is `false`, rest is `true`) ook integers, waardoor die drie types makkelijk naar elkaar om te zetten zijn door simpele toekenningen. Tevens kan met al deze typen net als met `int`'s gerekend worden:

```
//van integer-type naar integer-type
bool test = false;
int getal = test;     // getal is nu 0
char karakter = 'A';
getal = karakter;     // getal is nu 65
test = getal;         // test is nu 'true'
getal = 40;
karakter = getal;     // karakter is nu '('
```

Voorbeeldprogramma 10: Integer-types

Een `int` kan overigens ook makkelijk gecast worden naar een reëel getal:

```
// van integer naar double
int zakgeld = 100;
double zakgeldcenten = zakgeld;  // zakgeldcenten is nu 100.000
```

Voorbeeldprogramma 11: Integer naar double

Reële getallen
Voor reële getallen, of liever benaderingen daarvan, heb je in C++ onder meer de types `float` (van floating point) en `double` (van double precision). Meestal gebruikt men de meest precieze van de twee: `double`'s, die typisch 8 bytes geheugen innemen.
Bij het omzetten van reële getallen naar gehele getallen wordt er automatisch gecast:

```
double x = 1.8;
int i;
i = x; //automatische casting
```

Voorbeeldprogramma 12: Double naar integer

Bij een dergelijke automatische cast wordt altijd naar beneden afgerond (de cijfers achter de komma worden weggegooid) en dus niet naar het dichtstbijzijnde gehele getal, zodat bijvoorbeeld als `x` de waarde `1.9999` heeft `i` de waarde `1` krijgt. In zulke gevallen wordt vaak de truc `i = x + 0.5` gebruikt.
Vanwege het feit dat een double slechts een benadering is van een reëel getal volstaat een test als `if ( x == y )` niet om `double`'s `x` en `y` op gelijkheid te testen. Zelfs

```
if ( fabs (x - y) < epsilon )
```

(waarbij `epsilon` bijvoorbeeld `0.00001` is, en `fabs` de absolute waarde is, doorgaans afkomstig uit `cmath`) voldoet niet, en dus moet zoiets als

```
if ( fabs (x - y) < max ( fabs (x), fabs (y) ) * epsilon )
```

komen, waarbij `max (.,.)` een zelfgeschreven maximum-functie is.
Het afdrukken van reële getallen wordt door het volgende voorbeeld hopelijk duidelijk:

```
#include <iomanip>
  ...
  double x = 92.36718;
  cout << "En x is:" << setw (8) << setprecision (2)
       << setiosflags (ios::fixed|ios::showpoint) << x << endl;
```

Voorbeeldprogramma 13: Reële getallen afdrukken

Met `setw (8)` wordt de eerstvolgende af te drukken expressie 8 posities breed rechts uitgelijnd afgedrukt, de "stream manipulator" `setprecision` zorgt er voor dat er voortaan 2 cijfers na de decimale punt-komma komen — het mag ook met de memberfunctie `precision` —, en verder zorgt `fixed` er voor dat een decimale punt wordt gebruikt (in plaats van de "scientific notation" als in `5.1e+012`) en laat `showpoint` een getal als `88.00` zo afdrukken, en niet als `88`. Als het goed is verschijnt er nu ␣␣␣`92.37` op het scherm, waarbij ␣ een spatie aanduidt. Er zijn allerlei varianten als `cout << fixed;` mogelijk.

Boolean
Bij sommige oudere compilers komt het voor dat het type `bool` niet beschikbaar is. Het is echter eenvoudig zelf te maken via:

```
// Type boolean (meestal onder de naam bool al aanwezig!):
enum boolean {False = 0, True = 1};
```

3.9.2 Rekenen

Het rekenen binnen C++ werkt met de bekende rekenkundige operatoren (`*` betekent 'maal'). Bij het gebruik hiervan werken een aantal zaken anders dan je in eerste instantie verwacht. Zo wordt door `9/5` altijd, ook al zeg je `x = 9/5` waarbij `x` een `double` is, `1` opgeleverd (want `5` past `1` maal in `9`). De operator `/` is dus standaard de `integer`-operator "delen door". Wil je dat de berekening een exacter, reëel getal oplevert, gebruik dan casting door middel van `(double)9/5` of `9.0/5`. Het resultaat is dan `1.8000`. Nog mooier is de nieuwe C++-notatie: `static_cast<double>(9)`.

Een speciale `integer`-operator is de `%`. Deze heet de *modulo*-operator en levert de rest bij deling op. `9 % 5` heeft als resultaat dan ook `4`.
Een andere merkwaardigheid binnen C++ is dat bij bijvoorbeeld de operator `+` de "evaluatie-volgorde" niet vastligt: als in een programma ergens `f(x) + g(x)` staat, is niet vastgelegd of eerst `f(x)` of eerst `g(x)` wordt geëvalueerd. Meestal merk je het verschil niet, maar als een functie neveneffecten (*side effects*) heeft kan het resultaat van de volgorde afhangen!
In C++ is een zeer precieze prioriteitenlijst van operatoren vastgesteld. Het is daarom niet altijd nodig — maar wel duidelijk — haakjes te plaatsen.

3.9.3 Random getallen en static variabelen

Vaak is het nodig om willekeurige (*random*) getallen te fabriceren. Daar valt veel over te zeggen, zie bijvoorbeeld de beroemde boeken van Knuth. Daaruit valt te leren dat een willekeurige methode doorgaans geen willekeurige getallen levert.
Een heel eenvoudige methode is de volgende. Stel je hebt een eerste random-getal, zeg x_{oud} (geheel). We berekenen het volgende random-getal x_{nieuw} als $(a * x_{\text{oud}} + c)$ modulo m, waarbij a, c en m zekere parameters zijn. Dit wordt steeds herhaald, waarbij uiteraard telkens de waarde van x_{oud} aan het begin op de laatste x_{nieuw} gezet wordt. Zo krijgen we schijnbaar willekeurige getallen uit $\{0, 1, 2, \ldots, m-1\}$.
Duidelijk is dit niet echt random: door de keuze van a, c, m en de startwaarde van x — ook wel *seed* genoemd —, liggen alle volgende x-waarden vast. We spreken van dan ook van *pseudo-random-getallen*. Voor veel toepassingen is deze methode goed genoeg. Wel is de keuze van de parameters belangrijk. Vaak neemt men $c = 1$, en (als m een macht van 2 is, wat modulo-rekenen eenvoudig maakt) a modulo 8 gelijk aan 5, of (als m een macht van 10 is) a modulo 200 gelijk aan 21. Wordt dit gedaan, dan duurt het lang voordat herhaling optreedt (zodra een x al eerder is geweest, herhaalt zich immers de hele rij!); nu komen alle getallen tussen 0 en $m-1$ aan de beurt. We krijgen dus bijvoorbeeld, met `long` (een grotere `int`) om wat grotere berekeningen toe te staan:

```
// Pseudo-random-getal tussen 0 en 999:
void randomgetal (long& getal) {
  getal = ( 221 * getal + 1 ) % 1000;
}  // randomgetal
```

Voorbeeldprogramma 14: Random getal

De laatste cijfers van de zo gegenereerde getallen worden hier overigens achtereenvolgens $1, 2, 3, 4, 5, \ldots$ — niet zo willekeurig helaas. Iets netter kan de functie als volgt worden geschreven:
De initialisatie van een *static* variabele gebeurt maar één keer, terwijl de waarde behouden blijft tussen twee functie-aanroepen. Je kunt hier ook een functie laten aanroepen die van de tijd gebruik maakt, of die een getal aan de gebruiker van het programma vraagt, en zo de random-generator initialiseren.
Als je een keer zo'n randomgenerator geschreven hebt, kun je deze op allerlei manieren gebruiken. Een voorbeeld: stel je wilt een willekeurige permutatie van de getallen `0` tot en met `n-1` opbergen in het array `int per[n]`. Dan kun je dat als volgt doen:
Let er op dat bij deze methode alle `n!` permutaties evenveel kans hebben om gegenereerd te worden.

```
// Pseudo-random-getal tussen 0 en 999:
long randomgetal ( ) {
  static long getal = 42;
  getal = ( 221 * getal + 1 ) % 1000;
  return getal;
}  // randomgetal
```

Voorbeeldprogramma 15: Random getal beter

```
// stop random permutatie van 0,1,...,n-1 in array A
void maakpermutatie (int A[ ], int n) {
  int i;  // array-index
  int r;  // random array-index
  for ( i = 0; i < n; i++ )
    A[i] = i;
  for ( i = n-1; i >= 0; i-- ) {
    r = randomgetal ( ) % ( i+1 );  // 0 <= r <= i
    wissel (A[i],A[r]);
  }  // for
}  // maakpermutatie
```

Voorbeeldprogramma 16: Random permutatie

In standaardbibliotheken voor C++ zitten meestal verschillende random-generatoren, die overigens doorgaans van het hierboven genoemde type, de *lineaire congruentie methode*, zijn. In `cmath` (vroeger `math.h`) zijn de betreffende functies meestal wel te vinden.

3.10 Recursie

Recursieve functies zijn een speciaal soort functies. Er is sprake van recursie als een functie zichzelf direct of indirect aanroept.
In het algemeen bestaat een recursief proces (functie) uit twee delen:

1. Een kleinste geval (basisgeval), dat eenvoudig genoeg is om direct op te lossen, dus zonder recursie.

2. Een algemene methode die een bepaald geval reduceert tot één of meer kleinere gevallen, waarbij men uiteindelijk bij het basisgeval uitkomt.

De algemene gedaante van een recursieve functie (in een eenvoudige symbolische notatie met een `if ... then ... else ... fi` statement) is dan ook als volgt:

```
if basisgeval then   // test: geval simpel genoeg?
   los op zonder recursie (soms: niets doen);
else
   een of meer recursieve (eenvoudigere) aanroepen;
fi
```

Faculteiten
Het eenvoudigste voorbeeld van het gebruik van recursie is het berekenen van recursief gedefinieerde functies, zoals *faculteiten*, gedefinieerd door:

$$\text{fac}(n) = \begin{cases} 1 & \text{als } n = 0 \\ n * \text{fac}(n-1) & \text{als } n > 0 \end{cases}$$

Hieruit laat zich eenvoudig een programma afleiden:

```
// Berekent n! recursief. Neem aan dat n >= 0.
long faculteit (int n) {
  if ( n == 0 )  // dus niet: if ( n = 0 )
    return 1;
  else
    return n * faculteit (n - 1);
}  // faculteit
```

Voorbeeldprogramma 17: Faculteit berekenen

Het resultaat-type `long` zorgt ervoor dat we iets grotere faculteiten kunnen berekenen dan als er `int` gestaan zou hebben. De body van de functie kan ook geschreven worden als

```
  return ( n ? n * faculteit (n - 1) : 1 );
```

Uiteraard kan deze functie ook eenvoudig niet-recursief (*iteratief*) geschreven worden (Opgave 9).
Analoog, stel dat de functie is:

```
int som (int n) {
  return ( n ? n + som (n - 1) : 0 );
}  // som
```

Nu wordt de som van de getallen `0` tot en met `n` berekend. Dat kan ook heel eenvoudig met een gesloten formule via

```
  return ( n * ( n + 1 ) ) / 2 ;
```

Fibonacci getallen
Een zeer bekend en illustratief voorbeeld van recursie is de berekening van de *Fibonacci-getallen*. Deze zijn (recursief) gedefinieerd door

$$\text{fib}(n) = \begin{cases} 1 & \text{als } n = 0 \text{ of } n = 1 \\ \text{fib}(n-1) + \text{fib}(n-2) & \text{als } n > 1 \end{cases}$$

Dit levert de welbekende rij $1,1,2,3,5,8,13,21,34,55,89,144,233,377,610\ldots$ op. Bovenstaande definitie geeft het n-de getal uit deze rij (beginnend bij 0).
De recursieve definitie geeft onmiddellijk aanleiding tot een recursieve C++-functie:
Deze functie lijdt helaas aan het zogenaamde *waterval-effect* (ook wel *Fibonacci-effect* geheten). De aanroep `fib1 (n)` veroorzaakt een waterval van aanroepen, waarbij aanroepen ook nog eens veel vaker dan één keer plaatsvinden. Zo veroorzaakt de berekening van het 10-de Fibonacci-getal ($n = 10$) bijvoorbeeld vijf maal exact dezelfde berekening van het 6-de

```
long fib1 (int n) {
  if ( ( n == 0 ) || ( n == 1 ) )
    return 1;
  else
    return fib1 (n-1) + fib1 (n-2);
}  // fib1
```

Voorbeeldprogramma 18: Fibonacci getallen

Fibonacci-getal ($n = 6$). In het algemeen vergt de berekening van het n-de Fibonacci-getal zo in totaal maar liefst $2\,\mathrm{fib}(n) - 1$ aanroepen van `fib1` (ga na).
Dit probleem kan verbeterd worden met behulp van *dynamisch programmeren*, waarbij handige tabellen worden bijgehouden. De herhaalde aanroepen van het Fibonacci-effect worden hierbij vermeden door eerder behaalde tussenresultaten te onthouden in een tabel. Verder wordt voordat een aanroep gepleegd wordt, eerst in de tabel gekeken of dit Fibonacci-getal al niet eerder is uitgerekend. Als dat zo is, is de aanroep overbodig. Op deze manier worden alle aanroepen maar één keer gedaan, ten koste van een voldoende groot array. We krijgen:

```
const int MAX = 100;
long memo[MAX];  // stiekem gebruikte globale variabele
                 // deze eerst geheel met nullen vullen

long fib2 (int n) {
  if ( n >= MAX )  // helaas
    return fib2 (n-1) + fib2 (n-2);
  else
    if ( memo[n] > 0 )  // al eerder berekend
      return memo[n];
    else {
      if ( ( n == 0 ) || ( n == 1 ) )
        memo[n] = 1;
      else
        memo[n] = fib2 (n-1) + fib2 (n-2);
      return memo[n];
    }  // else
}  // fib2
```

Voorbeeldprogramma 19: Fibonacci getallen efficiënt

Het kan overigens ook met een eenvoudige iteratieve functie. Deze is veel efficiënter dan de recursieve versie.
Deze iteratieve versie leent zich uitstekend voor — in plaats van `long` — nog grotere getallen: een eigen gemaakt `grootgetal`, waarvoor dan wel onder andere een kopieer-operatie ($=$) en een optelling ($+$, $+1$) gemaakt moeten worden. Overigens kan het binnen de loop ook zonder hulp-variabele:

```
tweede = eerste + tweede;
eerste = tweede - eerste;
```

```
long fib3 (int n) {
  long eerste, tweede, hulp;
  int teller;
  eerste = 1;
  tweede = 1;
  for ( teller = 2; teller <= n; teller++ ) {
  // nu geldt: eerste == fib (teller-2) en tweede == fib (teller-1)
    hulp = tweede;
    tweede = eerste + tweede;
    eerste = hulp;
  }  // for
  return tweede;
}  // fib3
```

Voorbeeldprogramma 20: Fibonacci getallen efficiënt (iteratief)

Maar nu moet je ook nog grote getallen van elkaar aftrekken ...

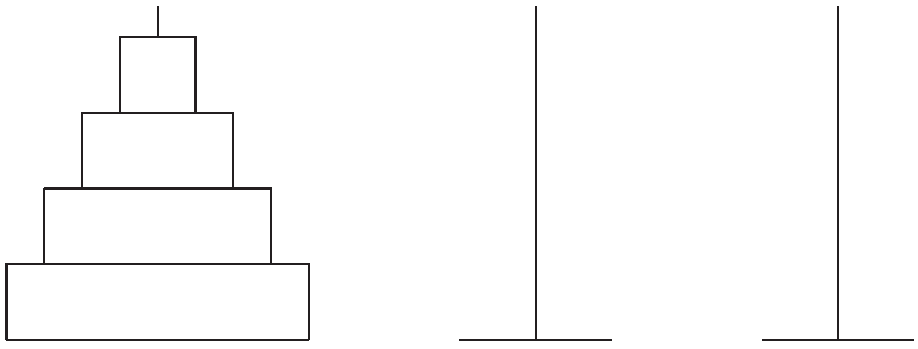
Een mooi, maar wat ingewikkelder voorbeeld van het optreden van recursie vinden we bij de zogenaamde *Ackermann-functie*:

$$\text{Ack}(m,n) = \begin{cases} n+1 & \text{als } m = 0 \text{ en } n \geq 0 \\ \text{Ack}(m-1,1) & \text{als } m > 0 \text{ en } n = 0 \\ \text{Ack}(m-1,\text{Ack}(m,n-1)) & \text{als } m > 0 \text{ en } n > 0 \end{cases}$$

Vele van de onderstaande zoek- en sorteermethoden (zie Hoofdstuk 4.2) kunnen eenvoudig recursief geprogrammeerd worden. We zullen nu enkele andere voorbeelden bekijken.

Torens van Hanoi

Gegeven zijn n ($n \geq 1$) schijven, alle verschillend in grootte. Bij aanvang liggen alle schijven boven op elkaar op één stapel, en wel zo dat de grootste onderop ligt, dan de op een na grootste, ..., en de kleinste bovenop. Kortom: een grotere schijf ligt nooit op een kleinere. Er zijn verder nog twee lege plaatsen beschikbaar. Voor $n = 4$ ziet de beginsituatie er zo uit:



plek 1 plek 2 plek 3

De bedoeling is om de hele toren op een van de twee lege plaatsen te krijgen, waarbij weer de grootste onder en de kleinste boven ligt (en klein op groot). Dit alles moet liefst zo snel mogelijk

gebeuren. We zoeken dus een optimale oplossing. Om dit voor elkaar te krijgen mag men per keer maar één schijf oppakken (en wel alleen de bovenste van een stapel!) en deze bovenop een andere stapel leggen. Een belangrijke restrictie hierbij: er mogen alleen kleinere schijven op grotere gelegd worden, en niet omgekeerd.
De recursieve oplossing hieronder volgt direct uit de volgende observaties. Om de grootste schijf te kunnen verzetten moeten alle $n-1$ schijven erboven weg zijn. Bovendien moet de plek waar de grootste heen moet leeg zijn. Aangezien we een optimale oplossing willen zal dit dan ook meteen de uiteindelijke bestemming van de grootste schijf moeten zijn. Dus: verplaats de bovenste $n-1$ schijven zo snel mogelijk naar de hulpplek (zeg plek 2), zet dan de grootste schijf van plek 1 naar de doelplek (plek 3), en verplaats dan de $n-1$ schijven zo snel mogelijk van de hulpplek naar de doelplek.

```
// torens van Hanoi: recursief
// zet toren van n stuks (optimaal) van a naar b via c
void zet (int n, int a, int b, int c) {
  if ( n > 0 ) {
    zet (n-1,a,c,b);
    cout << "zet van " << a << "naar " << b << endl;  // print zet
    zet (n-1,c,b,a);
  }  // if
}  // zet
```

Voorbeeldprogramma 21: Torens van Hanoi

3.11 Object georiënteerd programmeren

C++ biedt ons een aantal standaard-typen. Het komt echter vaak voor dat een programmeur deze typen wil samenpakken tot één geheel of totaal nieuwe types met eigen mogelijkheden (*funcionaliteit*) wil aanmaken. C++ geeft ons hiervoor genoeg mogelijkheden.
Om eenvoudig te beginnen bekijken we eerst de volgende standaard-declaratie:

```
int zakgeld;
```

Zoals eerder gezegd is `int` hier het type en is `zakgeld` de naam van de variabele. In het object georiënteerd programmeren worden andere termen gebruikt.
Variabelen, pointers, arrays, enz. zijn binnen C++ allemaal *objecten*; je kunt er wat mee doen (er dingen in opslaan, ze wissen, legen, kopiëren, ermee rekenen, etc.). Objecten die zich hetzelfde gedragen worden ondergebracht in eenzelfde *klasse*. Zo zijn bijvoorbeeld alle variabelen die gehele getallen kunnen bevatten en waar je mee kunt rekenen ondergebracht in de klasse `int`. Een klasse kan dus een simpel type variabele zijn — zie het voorbeeld met de `int` —, maar zoals we later zullen zien ook meer. Een klasse is dus eigenlijk een blauwdruk die eigenschappen en functionaliteit vastlegt. Een object dat gedeclareerd wordt volgens deze klasse noemen we een *instantie* van die klasse.
Bij object georiënteerd programmeren draait alles om het ontwerpen van *classes* en het maken van *instances* hiervan. De objecten zijn dan vaak de basis waar mee gewerkt wordt. Voordelen hiervan zijn dat deze manier van programmeren beter aansluit bij de denkwijze van de ”echte wereld” en dat de zelf gedefinieerde classes (klassen) herbruikbaar zijn en ze de modulariteit (zie bij de richtlijnen) verhogen. Kijk maar eens naar onderstaand voorbeeld:

```
int main ( ) {
  schaakbord nieuwbord;
  schaakspel nieuwspel;
  schaakstuk ditstuk;
  nieuwspel.start (nieuwbord);
  ...
  nieuwspel.zetstuk (ditstuk);
  ...
  if ( nieuwspel.afgelopen ( ) ) {
    cout << " einde spel" << endl;
  }  // if
}  // main
```

3.11.1 Struct's

De eenvoudigste (en dus ook de meest beperkte) mogelijkheid bij het maken van nieuwe blauwdrukken is die van de *struct*. Een struct kan opgevat worden als een collectie standaardobjecten die samengevoegd worden, maar ook als een aantal objecten die samen een bepaald concept realiseren. Het onderstaande voorbeeld maakt dit duidelijk.

```
struct MeetkundigeBalk {
  int lengte;
  int breedte;
  int hoogte;
};  // MeetkundigeBalk    let op de ;
```

Het concept van de meetkundige balk wordt vastgelegd door de drie maten lengte, breedte en hoogte. Er kan nu als volgt met de struct gewerkt worden:

```
int main ( ) {
  MeetkundigeBalk balk;
  balk.hoogte = 10;
  balk.lengte = 15;
  balk.breedte = 30;
  int inhoud = balk.hoogte * balk.breedte * balk.lengte;
  cout << inhoud << endl;
  return 0;
}  // main
```

Conceptmatig werken is een kenmerk van object georiënteerd programmeren.
De mogelijkheden van een struct zijn op het eerste gezicht redelijk beperkt. Eigenlijk is een struct slechts een collectie eigenschappen. In de volgende paragraaf zien we hoe we hiernaast ook functionaliteit kunnen toevoegen — en in plaats van `struct` gebruiken we dan `class`.

3.11.2 Classes

Met behulp van een *class* is het mogelijk een heel nieuwe klasse te maken, compleet met een collectie variabelen en functionaliteit. Hier eerst een voorbeeld van zo'n klasse:
Met een ijsje kan je een aantal dingen doen. Je kan een ijsje maken met behulp van bolletjes of je kan een ijsje opeten. Acties die je met of met behulp van een object kan doen vormen diens

```
enum soorten {aardbei, vanille, pistache};
enum groottes {klein, middel, groot);

struct bolletje {
  soorten soort;
  groottes grootte;
  int prijs;
};  // bolletje

class ijsje {
  public:
    void schepopbol (soorten soort, groottes grootte);
    void eetopbol ( );
    int geefprijs ( );
    ijsje ( );
    ~ijsje ( );
    int berekenprijs ( );
    void leegijsje ( );
    bool isleeg ( );
  private:
    int prijs;
    bolletje bolletje1;
    bolletje bolletje2;
};  // ijsje
```

Voorbeeldprogramma 22: Klasse: IJsjes maken

functionaliteit. Functionaliteit wordt in classes gerealiseerd door het declareren van functies in de *klasse-definitie*, zoals hierboven geldt voor de klasse `ijsje`. Functies die binnen zo'n klasse worden gedeclareerd heten *member-functies*. Ze worden als volgt gedefiniëerd:

```
void ijsje::eetopbol ( ) {
  ...
}
```

en zo aangeroepen:

```
  ijsje turbohoorn;
  ...
  turbohoorn.eetopbol ( );
```

De variabelen van een klasse worden *member-variabelen* genoemd of de *eigenschappen* (*properties*) van zo'n klasse.
Member-variabelen en member-functies worden binnen member-functies aangeroepen zonder het object erbij. In plaats hiervan is het mogelijk *this* te gebruiken om het object aan te geven waarvan de member-functie wordt aangeroepen, maar dat is niet gebruikelijk (een enkele keer echter wel nodig):

```
int ijsje::eetopbol ( ) {
```

```
  if ( ! isleeg ( ) ) {
    if ( bolletje2 != -1 ) {
      bolletje2 = -1; // of this->bolletje2 = -1;
    }  // if
    else {
      bolletje1 = -1; // of this->bolletje1 = -1;
    }  // else
  }  // if
}  // ijsje::eetopbol
```

Public versus private
Bij het gebruik van classes is het gebruikelijk de member-functies en member-variabelen op te delen in twee verschillende groepen: *public* en *private*. Zaken die binnen het public gedeelte staan zijn van buitenaf te ”bekijken” of aan te roepen. `eetopbol` hierboven is daar een voorbeeld van. Zaken die in het private gedeelte van de class gedeclareerd worden mogen echter alleen vanuit de eigen member-functies aangeroepen worden. Dit is dus verboden:

```
int main ( ) {
  ijsje turbohoorn;
  int x = turbohoorn.prijs;
}  // main
```

en dit mag wel:

```
int ijsje::geefprijs ( ) {
  return prijs;
}  // ijsje::geefprijs
```

Constructor en destructor
Vaak is het handig om bij het aanmaken van een object alvast een initialisatie functie uit te voeren. Denk hierbij bijvoorbeeld aan het ”leeg maken” van het ijsje. Binnen C++ is dit mogelijk met behulp van de zogeheten *constructor*; een functie die meteen bij het maken van een nieuw object wordt aangeroepen, zoals bijvoorbeeld bij de declaratie `ijsje turbohoorn`. Hierboven valt te zien hoe een constructor gedeclareerd wordt, hieronder staat hoe de definitie eruit ziet.

```
ijsje::ijsje ( ) {
  leegijsje ( );
}  // ijsje::ijsje
```

Merk op dat een constructor geen resultaat-type als `void` of `int` heeft.
Soms is het noodzakelijk dat je aangeeft hoe een bepaalde class ”weggegooid” moet worden, om bijvoorbeeld te voorkomen dat ongebruikt geheugen gereserveerd blijft. Dit kan nodig zijn als tijdens het maken van het object ook andere objecten zijn aangemaakt. In C++ worden de ”weggooi-instructies” uitgevoerd in een *destructor* die net als de constructor geen resultaat-type heeft en er zo uitziet:

```
ijsje::~ijsje ( ) {
  ...
}  // ijsje::~ijsje
```

Een destructor wordt automatisch aangeroepen als de betreffende variabele aan het eind van zijn leven komt.

3.12 Pointers

Pointers zijn speciale variabelen binnen C++ . Ze kunnen het beste gezien worden als een ”pijl” naar een andere variabele (of object). In werkelijkheid zijn het de adressen van de variabelen waar ze naartoe ”wijzen”. Laten we beginnen met een eenvoudig voorbeeld. Stel we hebben

```
int* p;
int* q;
```

Dan zijn p en q pointers naar integers, oftewel adressen van integers. Voorlopig wijzen ze nog nergens naar, of liever: ze zijn onbepaald. Het volgende programma-fragment laat elementaire handelingen zien:

```
p = new int;  // maak een nieuwe int en stop diens adres in p
*p = 42;      // die int wordt 42
q = p;        // kopieer zijn adres in q
*q = 37;      // ook via q kun je de int wijzigen
p = NULL;     // nu wijst p ECHT nergens naar
p = q;        // en nu weer naar die oude int
delete q;     // die we tot slot via q weggooien
```

Voorbeeldprogramma 23: Werken met pointers

Samengevat: p is een pointer, een adres, terwijl *p datgene is waar die pointer naar wijst. En zodra je zegt int* p bestaat de pointer p, maar waar hij naar toe wijst weet je dan nog niet. Maar zeg je p = NULL, dan weet je dat wel: nergens naar. Denk eraan dat delete q de integer waar q naar wijst weggooit, en niet q zelf. Overigens is

```
delete q;
q = NULL;
```

netjes, maar

```
q = NULL;
delete q;
```

niet. Dit laatste gaat niet fout — er gebeurt gewoon niets. Wat overigens nergens op slaat is iets als

```
q = new int;
q = p;
```

Eerst een nieuwe int maken, en dan diens adres overschrijven, is nogal onzinnig.
We geven hier een enkel voorbeeld om met pointers te leren omgaan. Stel we hebben een lijst met karakters nodig, waar “willekeurig” lange series karakters in kunnen worden opgeslagen. We definiëren:

```
class vakje {  // een struct mag ook
  public:
    char info;
    vakje* volgende;
};  // vakje
```

```
vakje* hulp = ingang;
while ( hulp != NULL ) {
  cout << hulp->info;  // hulp->info betekent: (*hulp).info
  hulp = hulp->volgende;
}  // while
```

Voorbeeldprogramma 24: Pointerlijst: doorlopen

Nu kunnen we zo'n pointerlijst, als die ten minste eenmaal is opgebouwd, met ingang `ingang` van type `vakje*`, eenvoudig aflopen met
Laten we eerst eens met de hand één voor één vakjes bouwen en aan elkaar vast maken. Om te beginnen een vakje met alleen een `'l'` erin:

```
vakje* ingang;
ingang = new vakje;
ingang->info = 'l';
ingang->volgende = NULL;
```

Als we nu een vakje met `'c'` erin hier *voor* willen zetten hebben we een hulppointer, zeg `p`, nodig:

```
vakje* p;
p = new vakje;
p->info = 'c';
p->volgende = ingang;
ingang = p;
```

Stel dat we *tussen* de vakjes met `'c'` en `'l'` een vakje met `'a'` erin willen krijgen. Dat gaat als volgt:

```
vakje* p;
p = new vakje;
p->info = 'a';
p->volgende = ingang->volgende;
ingang->volgende = p;
```

Een nieuw vakje vooraan een bestaande lijst toevoegen gaat algemener met:

```
void zetervoor (char letter, vakje*& ingang) {  // let op de &
  vakje* p;
  p = new vakje;
  p->info = letter;
  p->volgende = ingang;
  ingang = p;
}  // zetervoor
```

Voorbeeldprogramma 25: Pointerlijst: vooraan toevoegen

Deze functie werkt zelfs goed als `ingang` `NULL` was. Aan het eind van de functie, vlak voor de afsluitende accolade, moet NIET `delete p;` staan; `p` wordt immers als locale variabele vanzelf

weer vrijgegeven, terwijl dat waar p naar wijst uiteraard moet blijven bestaan!
We kunnen dezelfde lijst als boven nu opbouwen met:

```
vakje* ingang = NULL;
zetervoor ('l',ingang);
zetervoor ('a',ingang);
zetervoor ('c',ingang);
```

Maar het is wellicht slim om *achteraan* de lijst toe te voegen. Dat kan met de volgende functie, waarbij de pointer `laatste` steeds naar het laatste vakje van een lijst wijst:

```
void zeterachter (char letter, vakje*& laatste) {  // let op de &
  vakje* p = new vakje;
  p->info = letter;
  p->volgende = NULL;
  if ( laatste != NULL)
    laatste->volgende = p;
  laatste = p;
}  // zetervoor
```

Voorbeeldprogramma 26: Pointerlijst: achteraan toevoegen

De prijs die je betaalt is dat je naast de ingangspointer ook een pointer naar het laatste vakje moet onthouden. Dezelfde lijst als boven wordt nu opgebouwd met:

```
vakje* ingang = NULL;
vakje* laatste = NULL;
zeterachter ('c',laatste); // of: zeterachter ('c',ingang);
ingang = laatste;          // laatste = ingang;
zeterachter ('a',laatste);
zeterachter ('l',laatste);
```

Call-by-reference en call-by-value spelen bij pointers een belangrijke rol. Stel we hebben de functie

```
void demo (char letter, vakje*& ingang) {
  if ( ingang != NULL )
    ingang->info = letter;
}  // demo
```

Laten we eens achtereenvolgens uitvoeren `ingang = NULL; zetervoor ('A',ingang);`, en daarna `demo ('B',ingang);`. Of nu de `&` al of niet in de heading van `demo` voorkomt, de letter `B` komt in het door `ingang` aangewezen vakje terecht. Immers, de *pointer* `ingang` verandert niet — en dat is de parameter waar het over gaat —, maar wel datgene waar deze pointer naar wijst. Vergelijk dit met

```
void demo2 (vakje*& ingang) {
  ingang = NULL;
}  // demo2
```

```
// Vindt eerste vakje met letter erin (uit lijst met ingang),
// als zo'n vakje bestaat; anders NULL
vakje* vind (char letter, vakje* ingang) {
  bool gevonden = false;
  vakje* hulp = ingang;  // NIET eerst hulp = new vakje
  while ( ( hulp != NULL ) && !gevonden )
    if ( letter == hulp->info )
      gevonden = true;
    else
      hulp = hulp->volgende;
  return hulp;
}  // vind
```

Voorbeeldprogramma 27: Pointerlijst: vakje vinden

Nu zal, als de `&` wordt weggelaten, een *kopie* van de actuele parameter `NULL` gemaakt worden! De echte actuele parameter verandert niet, wat wel gebeurt als `&` er weer bij staat.
Een pointerlijst doorzoeken verloopt via:
Dit gaat analoog aan *lineair zoeken*. Zie hiervoor Hoofdstuk 4.2. Het kan natuurlijk ook recursief:

```
// Als vind, maar nu recursief
vakje* vindrecursief (char letter, vakje* ingang) {
  if ( ingang == NULL )
    return NULL;
  else
    if ( letter == ingang->info )
      return ingang;
    else  // komt letter voor in de rest van de lijst?
      return vindrecurief (letter,ingang->volgende);
}  // vindrecursief
```

Voorbeeldprogramma 28: Pointerlijst: recursief vakje vinden

3.13 Strings

Vaak wordt in programma's met een rijtje karakters, een *string*, gewerkt. Vooral bij in- en uitvoer worden strings gebruikt. In `string.h` zitten allerlei handige functies om strings te manipuleren. Ze werken met variabelen van type `char*`, bijvoorbeeld ook `char tekst[100]`: de variabele `tekst` is hier eigenlijk een pointer naar een `char`. Let er op dat je zelf in principe dit geheugen moet beheren, dus `new` en `delete` moet gebruiken. Een "leuk" voorbeeld is de functie `strcpy`, die ongeveer als volgt is geïmplementeerd:

```
void strcpy (char* s, char* t) {  // kopieer t naar s
  while ( *s++ = *t++ )  ;
}  // strcpy
```

Hier wordt dus in één regeltje een string gekopieerd!
Een eenvoudiger voorbeeld is de functie `strcmp` die een getal kleiner dan 0 oplevert als de string `s` alfabetisch voorafgaat aan de string `t`, 0 als ze gelijk zijn, en in de overige gevallen een getal groter dan 0. Hierbij wordt dankbaar de afsluitende `'\0'` die elke string hoort te hebben gebruikt.

```
int strcmp (char* s, char* t) {  // komt s voor t?
  int i;
  for ( i = 0; s[i] == t[i]; i++ )
    if ( s[i] == '\0' )
      return 0;
  return s[i] - t[i];
}  // strcmp
```

Voorbeeldprogramma 29: String compare

Overigens wordt een `"` in een string voorgesteld door `\"`. Veel mensen gebruiken in plaats van `endl` ook binnen de `" "` een `\n`, en schrijven `cout << "Hij zei: \"Leuk!\"\n";`. Denk er wel aan dat een `\n` niet "flusht": de uitvoer wordt gebufferd weggeschreven.
Tot slot: in de nieuwere C++-versies wordt in plaats van de "oude C-stijl" `string.h` de nieuwe verbeterde `string` gebruikt. Daar kun je "gewoon" strings kopiëren. Onder het oude bewind komt `getline` soms van pas:

```
char zin[80];
cin.getline (zin,80);
```

zorgt ervoor dat er maximaal 79 karakters plus de afsluitende `\0` in de variabele `zin` terecht komen: er wordt tot dit aantal of tot (en met) de Enter gelezen. Let er op dat de afsluitende Enter alleen wordt weggelezen als hij bereikt wordt. Die lastige buffers maken het leven niet eenvoudiger ...

3.14 De Standard Template Library — STL

In de Standard Template Library, afgekort STL, zijn allerlei fraaie datastructuren, zoals rijen en verzamelingen (zie Hoofdstuk 5), reeds aanwezig. In bijvoorbeeld het boek van Deitel en Deitel of in dat van Ammeraal is hier volop informatie over te vinden.
We bekijken hier kort één voorbeeld: de klasse `vector`. Met behulp van de uit het voorafgaande bekende array-operator `[ ]` kunnen als gebruikelijk elementen benaderd worden, maar "vectoren" kunnen ook dynamisch groeien en krimpen! Bovenaan het programma moet `#include <vector>` staan. Een vector `v` met gehele getallen wordt gedeclareerd door

```
vector< int>v;
```

Met `v.push_back (1215);` wordt `1215` achteraan `v` toegevoegd, en de grootte (`v.size ( )`) van `v` groeit automatisch met één. Eigenlijk is de klasse `vector` een *container*.

3.15 C++-standaards

In 1998 zijn allerlei standaards vastgesteld voor C++. Ook zijn diverse aanvullingen doorgevoerd. Voor meer details raadplege men bijvoorbeeld het uitgebreide boek van Deitel en Deitel. We geven hier slechts enkele voorbeelden. Oudere compilers — en zelfs nieuwe — voldoen helaas niet altijd aan alle vernieuwingen.
Allereerst zijn zogeheten *namespaces* geïntroduceerd. Hiermee kan voorkomen worden dat dingen die per ongeluk hetzelfde heten door elkaar worden gehaald. Zo kan bijvoorbeeld `cout` een andere betekenis toebedeeld krijgen — maar wie zou dat nu willen? Bovenaan een programma komt in plaats van de ook nog dikwijls gebruikte ene regel

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

nu:

```
#include <iostream>
using namespace std;
```

Ook eigen namespaces kunnen worden gefabriceerd. In bovenstaand voorbeeld mag je in plaats van het gewone `cout << ...` nu ook zeggen `std::cout << ...` en zo naamsconflicten voorkomen.
In het voorbeeld is ook te zien dat de `.h` bij de `include`-regel verdwenen is. Dat geldt voor de standaard includefiles; eigengemaakte behouden hun `.h`. Bij enkele standaard includefiles is de naam iets gewijzigd: `<cmath>`, `<climits>`, `<cstring>` en `<ctime>` vervangen respectievelijk `<math.h>`, `<limits.h>`, `<string.h>` en `<time.h>`. Er zijn ook mooie nieuwe gemaakt, zoals `<stack>`, `<string>` (!), `<limits>` (!) en `<set>`, zie Hoofdstuk 3.14 over de Standard Template Library.
Als je compiler erop berekend is kunnen ook *templates* gebruikt worden; meer hierover bij vervolgcolleges als Datastructuren. Een enkel voorbeeld, waarin het maximum van drie variabelen van klasse `Tiep` wordt bepaald:

```
template <class Tiep> Tiep maximum (Tiep x, Tiep y, Tiep z) {
  Tiep max = x;
  if ( y > max )
    max = y;
  if ( z > max )
    max = z;
  return max;
}  // maximum
```

Mits de klasse `Tiep` maar kopiëren met = toestaat en objecten van klasse `Tiep` zich met > laten vergelijken, hebben we hier een algemeen werkend sjabloon gemaakt.

4 Algoritmen

Bij programmeren draait bijna alles om het vervaardigen van stappenplannen ofwel *algoritmen*. In dit hoofdstuk bekijken we een aantal illustratieve en/of veel gebruikte algoritmen(soorten). In het vervolgvak Algoritmiek wordt meer aandacht besteed aan handigheidjes, nieuwe algoritmen, en zaken als efficiëntie- en ordevergelijkingen ervan.

4.1 Rekenalgoritmen

Algoritmen draaien doorgaans om het berekenen van een of ander gegeven. Vele hiervan komen voort uit de wiskunde. Hieronder behandelen we er een paar.

4.1.1 Grootste gemeenschappelijke deler

Allereerst een functie die de *grootste gemeenschappelijke (gemene) deler*, de *ggd*, van twee gegeven gehele getallen berekent.

```
// Bepaal grootste gemene deler (ggd) van gehele, positieve x en y
// met behulp van het algoritme van Euclides.
int ggd (int x, int y) {
  int rest;
  while ( y != 0 ) {
    rest = x % y;  // rest bij deling van x door y (x modulo y)
    x = y;
    y = rest;
  }  // while
  return x;
}  // ggd
```

Voorbeeldprogramma 30: GGD berekenen

Verstokte C-liefhebbers zullen als test overigens schrijven `while ( y ) { ... }`, maar of dat de duidelijkheid bevordert is nog maar de vraag. De ggd-functie kan bijvoorbeeld gebruikt worden om breuken te vereenvoudigen, zie verderop.

```
// Vereenvoudig de breuk teller/noemer zoveel mogelijk.
// Neem aan dat teller >= 0 en noemer > 0.
void vereenvoudig (int& teller, int& noemer) {
  int deler = ggd (teller,noemer);
  if ( deler > 1 ) {
    teller = teller / deler;
    noemer = noemer / deler;
  }  // if
}  // vereenvoudig
```

Voorbeeldprogramma 31: Breuken vereenvoudigen

De test `if ( deler > 1 )` hoeft er overigens niet bij. Wel moet er van een hulpvariabele `deler` gebruik gemaakt worden: wordt er geprobeerd twee maal door de grootste gemene deler

van teller en noemer te delen (dus tweemaal "dezelfde" functieaanroep, wat op zich ook al niet zo snugger is), dan zal bij de tweede deling de helaas gewijzigde waarde van `teller` gebruikt worden. De noemer van de breuk zal dan niet veranderen, wat — als de ggd niet 1 is — toch in de bedoeling ligt!
De functie voor de ggd kan ook eenvoudig recursief geschreven worden, zie onder. In feite gebeurt hier hetzelfde als bij de niet-recursieve versie: beide zijn implementaties van het *algoritme van Euclides*.

```
// Als ggd, maar nu recursief.
int ggdrecursief (int x, int y) {
  if ( y == 0 )
    return x;  // want ggd (x,0) = x als x niet 0 is
  else
    return ggdrecursief (y, x % y);
}  // ggdrecursief
```

Voorbeeldprogramma 32: GGD recursief berekenen

4.1.2 Priemgetallen

Er zijn vele manieren om te bepalen of een gegeven geheel getal groter dan 1 een *priemgetal* is, dat wil zeggen geen delers heeft behalve 1 en zichzelf. Het meest voor de hand liggende algoritme — maar lang niet het snelste — staat verderop.

```
// Voor sqrt (worteltrekken):
#include <cmath>
// Levert true precies als getal een priemgetal is.
bool priem (int getal) {
  int deler = 2;
  double wortel = sqrt (getal);
  bool geendelers = true;
  while ( ( deler <= wortel ) && geendelers ) {
    if ( ( getal % deler ) == 0 )  // deler gevonden: getal niet priem
      geendelers = false;
    deler++;
  }  // while
  return geendelers;
}  // priem
```

Voorbeeldprogramma 33: Priemgetal?

De extra hulpvariabele `wortel` voorkomt het steeds opnieuw uitrekenen van de wortel uit het oorspronkelijke getal. Het is duidelijk dat je niet meer voorbij deze wortel hoeft te kijken: als daar een deler van het oorspronkelijke getal zou zitten, zou er ergens voor die wortel ook een moeten zijn.
De `while`-loop kan overigens ook als `for`-loop opgeschreven worden. Doorgaans zullen wij, zoals al eerder opgemerkt, alleen een `for`-loop gebruiken als het aantal keren dat de body doorlopen moet worden van te voren duidelijk bekend is (we moeten iets bijvoorbeeld 17 keer, of `n`

keer, doen). In andere gevallen geven we zoals eerder gezegd de voorkeur aan een `while`-loop (we moeten dan iets doen net zolang als ...).
Een vergelijkbaar algoritme is de *zeef van Erathosthenes*. Hier wordt in een array bijgehouden welke getallen, kleiner dan een zekere bovengrens, nu wel of niet priem zijn. We geven dit algoritme als een "hoofdprogramma", `main` dus.

```
#include <iostream>
using namespace std;
const int MAX = 100;
int main ( ) {
  bool zeef[MAX];  // zijn getallen 0,1,...,MAX-1 priem (zeef[i] true)?
  int getal;
  int veelvoud;
  double wortel = sqrt (MAX);
  zeef[0] = false;
  zeef[1] = false;
  for ( getal = 2; getal < MAX; getal++ )
    zeef[getal] = true;  // ... tot het tegendeel bewezen is
  for ( getal = 2; getal < wortel; getal++ )
    if ( zeef[getal] ) {
      veelvoud = 2 * getal;  // getal + getal als * "te duur" is
      while ( veelvoud < MAX ) {
        zeef[veelvoud] = false;
        veelvoud = veelvoud + getal;  // oftewel veelvoud += getal;
      }  // while
    }  // if
  for ( getal = 2; getal < MAX; getal++ ) {
    if ( zeef[getal] )
      cout << getal << "  ";
  }  // for
  return 0;  // sommige compilers waarschuwen als deze regel ontbreekt
}  // main
```

Voorbeeldprogramma 34: De zeef van Erathosthenes

4.1.3 Driehoek van Pascal

Nog een voorbeeld uit de wereld van de gehele getallen betreft de welbekende driehoek van Pascal. Elk getal in de driehoek is de som van de twee getallen erboven. De getallen heten ook wel *binomiaalcoëfficiënten*. We lopen op een handige manier door de matrix heen en vullen rij voor rij de array-elementen in. We maken gebruik van de bekende identiteit

$$\binom{i}{j} = \binom{i-1}{j-1} + \binom{i-1}{j}$$

en krijgen zo het volgende programma:
We vullen alleen die array-elementen die we echt nodig hebben:

```
void pascaldriehoek ( ) {
  int i, j;
  int Pascal[n][n];  // in i-de rij, j-de kolom komt "i boven j"
  for ( i = 0; i < n; i++ )
    Pascal[i][0] = 1;  // de eerste kolom bevat enen
  Pascal[0][1] = 0;
  for ( i = 1; i < n; i++ ) {
    cout << endl << Pascal[i][0] << "  ";  // 1 op een nieuwe regel
    for ( j = 1; j <= i ; j++ ) {
      Pascal[i][j] = Pascal[i-1][j-1] + Pascal[i-1][j];
      cout << Pascal[i][j] << "  ";
    }  // for
    if ( i != n - 1 )
      Pascal[i][i+1] = 0;
  }  // for
}  // pascaldriehoek
```

Voorbeeldprogramma 35: Driehoek van Pascal

```
1  0  ... 1  1  0  ... 1  2  1  0  ... 1  3  3  1  0  ...
```

Het is overigens ook mogelijk om met een enkel array te werken. In de `i`-de slag bevat dat array dan de getallen uit de `i`-de rij van de driehoek van Pascal. Nu moet elk getal de som van de twee getallen erboven worden, wat in het enkele array betekent de som van het getal links van jezelf en jezelf. Loop je nu van links naar rechts door het array, dan wordt te vroeg de waarde van de array-elementen gewijzigd, namelijk terwijl je hun oude waarde nog nodig hebt. Door van rechts naar links te lopen, en op te merken dat de driehoek toch symmetrisch is, ontstaat een eenvoudig programma.

```
void pascaldriehoekbeter ( ) {
  int i, j;
  int rij[n];
  for ( j = 1; j < n; j++ )
    rij[j] = 0;
  rij[0] = 1;
  for ( i = 1; i < n; i++ ) {
    for ( j = i; j > 0 ; j-- ) {
      rij[j] = rij[j-1] + rij[j];
      cout << rij[j] << "  ";
    }  // for
    cout << rij[0] << endl;  // een 1 erachter
  }  // for
}  // pascaldriehoekbeter
```

Voorbeeldprogramma 36: Driehoek van Pascal efficiënt

4.1.4 Matrixvermenigvuldiging

Matrices willen graag met elkaar vermenigvuldigd worden. (Voor diegenen die nog niet weten wat dat is: geen nood.) Het ligt voor de hand om een klasse `matrix` te maken, waarbij memberfuncties voor bijvoorbeeld inverteren, determinant nemen, vermenigvuldigen, kopiëren (altijd mee oppassen: met pointers in het spel kan het fout aflopen). In de Standard Template Library (STL, zie Hoofdstuk 3.14) is zo'n type overigens al aanwezig. Wij zullen hier "gewoon" twee vierkante matrices, zeg A en B, vermenigvuldigen, met als resultaat C. De definitie is dat

$$C_{ij} = \sum_{k=0}^{n-1} A_{ik} B_{kj}, \quad 0 \leq i, j < n.$$

De getallen uit de i-de rij van A en de j-de kolom van B worden paarsgewijs vermenigvuldigd en de resultaten opgeteld ten einde het matrixelement C_{ij} op te leveren.

```
// C wordt A * B (matrixvermenigvuldiging).
void vermenigvuldigen (int A[n][n], int B[n][n], int C[n][n]) {
  int i, j, k;
  for ( i = 0; i < n; i++ )
    for ( j = 0; j < n; j++ ) {
      C[i][j] = 0;
      for ( k = 0; k < n; k++ )
        C[i][j] += A[i][k] * B[k][j];
    }  // for
}  // vermenigvuldigen
```

Voorbeeldprogramma 37: Matrixvermenigvuldiging

Hierbij moeten aanroepen als `vermenigvuldigen (A,A,A)` vermeden worden!
Zo te zien kost matrixvermenigvuldigen van twee n bij n matrices overigens n^3 vermenigvuldigingen van array-elementen; men zegt wel: een $O(n^3)$-algoritme (zie het college Algoritmiek voor deze notatie). Het kan ook sneller, zie het college Complexiteit.

4.1.5 Verzamelingen

We gaan nu uit een vaas met verschillend genummerde balletjes een aantal balletjes trekken zonder teruglegging. Welke getallen-volgordes zijn er mogelijk?
Een vaas is eigenlijk wat in de wiskunde een *verzameling* genoemd wordt. We zullen dan ook een C++-klasse `verzameling` maken, en vervolgens ons probleem oplossen. Deze klasse kunnen we vast wel vaker gebruiken. Wat we dan gedaan hebben is in feite het maken van een *abstract datatype* (*ADT*); andere voorbeelden hiervan zijn stapels en rijen (zie Hoofdstuk 5). We beginnen met:
In principe kun je met *templates* ook verzamelingen fabriceren waarvan de elementen niet noodzakelijk getallen zijn, maar van een algemeen type; dat zullen we nu maar niet doen. We moeten dus nog wel de verschillende member-functies schrijven.

```
verzameling::verzameling ( ) {
  int i;
```

```
// Grootte van het universum:
const int n = 20;
// En de klasse verzameling voor deelverzamelingen van {0,1,2,...,n-1}:
class verzameling {
  public:
    // default constructor (de lege verzameling)
    verzameling ( );
    // destructor: niks doen in dit geval
    ~verzameling ( ) { }  // destructor verzameling
    void drukaf ( );
    void erbij (int i);
    void eruit (int i);
    bool ziterin (int i);
    // geef alle volgordes om hoeveel getallen
    // uit de verzameling te trekken;
    // diepte geeft de recursie-diepte aan
    void trekking (int hoeveel, int diepte);
  private:
    // i-de element true betekent dat i in de verzameling zit
    bool inhoud[n];
};  // verzameling
```

Voorbeeldprogramma 38: Verzameling

```
  for ( i = 0; i < n; i++ )
    inhoud[i] = false;
}  // default constructor verzameling

void verzameling::drukaf ( ) {
  int i;
  cout << endl;
  for ( i = 0; i < n; i++ )
    if ( ziterin (i) )
      cout << i << "  ";
  cout << endl;
}  // verzameling::drukaf

void verzameling::erbij (int i) {
  inhoud[i] = true;
}  // verzameling::erbij

void verzameling::eruit (int i) {
  inhoud[i] = false;
}  // verzameling::eruit

bool verzameling::ziterin (int i) {
  return inhoud[i];
```

```
}  // verzameling::ziterin
```

Resteert nog `trekking` te schrijven, want daar ging het om:

```
// We moeten uit de verzameling hoeveel getallen trekken (zonder
// teruglegging). Als hoeveel 0 is hoeven we niets te doen.
// In diepte houden we de recursie-diepte bij.
void verzameling::trekking (int hoeveel, int diepte) {
  int knikker;
  if ( hoeveel > 0 ) {
    for ( knikker = 0; knikker < n; knikker++ )
      if ( ziterin (knikker) ) {
        spaties (2*diepte);
        cout << knikker << endl;
        eruit (knikker);
        trekking (hoeveel-1,diepte+1);
        erbij (knikker);
      }  // if
  }  // if
}  // verzameling::trekking
```

Voorbeeldprogramma 39: Trekking uit een verzameling

Hierbij wordt een functie `void spaties (int aantal)` gebruikt die `aantal` spaties afdrukt (opgave: schrijf deze zelf). Het doel hiervan is dat uit de uitvoer met enige moeite af te lezen valt wat we willen weten. Zo zal

```
3
  5
  9
5
  3
...
```

betekenen dat de getallenrijtjes 3 5, 3 9, 5 3, ... geproduceerd zijn. Probeer zelf eens de uitvoer wat mooier te krijgen (Opgave 29). De — overigens niet gebruikte — member-functie `drukaf` levert ook niet zulke mooie resultaten op het beeldscherm. Probeer zelf ook eens de functie zo te verbeteren dat bijvoorbeeld netjes komma's en accolades toegevoegd worden, zoals past bij een verzameling.
Uiteindelijk kan de functie `trekking` als volgt worden aangeroepen:

```
// Voorbeeldaanroep
int main ( ) {
  verzameling vaas;  // aanroep default constructor ==> geen ( )
  vaas.erbij (3);
  vaas.erbij (5);
  vaas.erbij (9);
  vaas.trekking (2,0);
  return 0;
}  // main
```

Het is ook mogelijk een verzameling als een enkelverbonden pointerlijst te implementeren. Dit is met name verstandig wanneer het kleine deelverzamelingen van een groot universum betreft: er zouden anders arrays met veel elementen `false` nodig zijn. Dankzij de bovenstaande opzet hoeven gelukkig de functies `drukaf` en `trekking` niet gewijzigd te worden; ze maken alleen via andere member-functies gebruik van het array of de pointerlijst.

4.2 Zoeken en Sorteren

Van oudsher hebben algoritmes voor sorteren en zoeken in de belangstelling gestaan. Bijvoorbeeld voor het zoeken van een telefoonnummer of het sorteren van een leden-database. Gezien de vaak grote hoeveelheden gegevens is het belangrijk dat dit enigszins effciënt gaat. Er zijn dan ook veel verschillende algoritmen bedacht. We zullen er hier enkele kort behandelen. Steeds gebruiken we een array van het volgende type:

```
// Arraygrootte:
const int n = 20;
// Een array A:
int A[n];
```

Eigenlijk moeten we de array-grootte `n` steeds als parameter doorgeven. Let erop de arraygrenzen niet te overschrijden! Laten we als voorbeeld eens het kleinste getal uit een array opsporen (zie verderop).

```
// Geef kleinste getal uit array A (dat n elementen heeft)
int minimum (const int A[ ], int n) {
  int klein = A[0];
  int i;
  for ( i = 1; i < n; i++ )
    if ( A[i] < klein )  // kleinere gevonden
      klein = A[i];
  return klein;
}  // minimum
```

Voorbeeldprogramma 40: Zoek kleinste getal

Arrays en pointers hebben in C++ een nauwe band. Wij zullen proberen de twee zaken niet te verwarren, en ons niet bezighouden met zaken als de gelijkheid van `&A[0]` en `A` voor een variabele `A` van type `array`. Zo mag bijvoorbeeld in de heading van de functie ook `const int * A` staan. Eigenlijk staat hier voor de compiler de volgende informatie: op deze plek staat een serie `int`'s; hoeveel dat er zijn is "niet van belang". De tweede parameter, `n`, wordt gebruikt om dit aantal door te geven. De `const` zorgt er hier voor dat je de array-elementen niet mag wijzigen. In bovenstaande functie zou `A[0] = klein;` een waarschuwing van de compiler opleveren.

4.2.1 Lineair zoeken

We zoeken in een (ongesorteerd) array naar een getal. Voor de hand ligt het volgende. Vooraan (of achteraan) beginnen en element voor element vergelijken tot we het gezochte element gevonden hebben of tot alle elementen geweest zijn, en het getal klaarblijkelijk niet voorkomt.

```
// Zoek getal in array A. Lineair zoeken.
// Geeft index met A[index] = getal, als getal ten minste
// voorkomt; zo niet: resultaat wordt -1.
int lineairzoeken (int A[n], int getal) {
  int index = 0;
  bool gevonden = false;
  while ( ! gevonden && ( index < n ) ) {
    if ( getal == A[index] )
      gevonden = true;  // of meteen return index;
    else
      index++;
  }  // while
  if ( gevonden )
    return index;
  else
    return -1;
}  // lineairzoeken
```

Voorbeeldprogramma 41: Lineair zoeken

Als je pech hebt, bijvoorbeeld als het gezochte getal niet voorkomt, kost je dat voor een array met n elementen n vergelijkingen; kortom: $O(n)$ (lineaire orde); zie Opgave 51. De methode heet *lineair zoeken*.

4.2.2 Binair zoeken

Als het array gesorteerd is kunnen we wat slimmer zijn. In een telefoonboek bijvoorbeeld zal iedereen die een naam zoekt (om het bijbehorende telefoonnummer te krijgen) gebruik maken van de alfabetische ordening. Zoek je overigens op nummer (om te weten wie dat nummer heeft), dan zit er — tenzij je bij de telefoondienst werkt — niets anders op dan met lineair zoeken het hele telefoonboek door te nemen.
Binair zoeken is een voor de hand liggende methode om een getal op te sporen in een gesorteerd array. Kijk allereerst of het middelste element het gezochte element is. Zo niet, bepaal dan op grond van vergelijken met dat middelste element of het zoekproces voortgezet moet worden in de linker helft of juist in de rechter helft van het array en herhaal dit. Stop zodra het element gevonden is, danwel het te onderzoeken array leeg is. Als het aantal elementen even is: kies één van de twee middelste.
Een aanroep ziet eruit als `index = binairzoeken (A,n,getal)`. In het slechtste geval ben je hier, als er bijvoorbeeld $n = 2^k - 1$ (met $k > 0$ geheel) elementen zijn, k vergelijkingen aan kwijt; men zegt wel: $O(\log n)$ (logaritmische orde). De methode binair zoeken kan ook recursief worden opgeschreven. Zonder verder commentaar:

4.2.3 Een eenvoudige sorteermethode

Nu willen we een array (rij) op grootte oplopend sorteren. Een heel eenvoudige methode is de volgende. Beginsituatie: het gesorteerde stuk is leeg en het ongesorteerde stuk is de hele rij. Zoek nu eerst het kleinste element in het ongesorteerde stuk en verwissel dat met het voorste

```
// Zoek getal in GESORTEERD array A met n elementen.
// Binair zoeken.
// Geeft index met A[index] = getal, als getal ten minste
// voorkomt; zo niet: resultaat wordt -1.
int binairzoeken (int A[n], int n, int getal) {
  int links = 0, int rechts = n-1;  // zoek tussen links en rechts
  int midden;
  bool gevonden = false;
  while ( ! gevonden && ( links <= rechts ) ) {
    midden = ( links + rechts ) / 2;
    if ( getal == A[midden] )
      gevonden = true;  // of meteen return midden;
    else
      if ( getal > A[midden] )
        links = midden + 1;
      else
        rechts = midden - 1;
  }  // while
  if ( gevonden )
    return midden;
  else
    return -1;
}  // binairzoeken
```

Voorbeeldprogramma 42: Binair zoeken iteratief

```
// zoek in een oplopend gesorteerd array A, tussen links en rechts,
// de index met A[index] == getal; -1 als getal niet in A voorkomt
int binairzoeken (int A[ ], int n, int links, int rechts, int getal) {
  int midden;
  if ( links > rechts )    // basisgeval
    return -1;
  else {   // recursieve aanroepen
    midden = (links + rechts)/2;
    if ( getal == A[midden] )   // gevonden!
      return midden;
    else   // verder zoeken: recursieve aanroepen
      if ( getal > A[midden] )   // rechts hetzelfde doen
        return binairzoeken (A, n, midden+1, rechts, getal);
      else   // links hetzelfde doen
        return binairzoeken (A, n, links, midden-1, getal);
  } // else recursieve aanroepen
} // binairzoeken
```

Voorbeeldprogramma 43: Binair zoeken recursief

element van dat stuk. Het gesorteerde stuk is nu één element groter en het ongesorteerde stuk is één element kleiner. Herhaal dit totdat het ongesorteerde stuk leeg is. De methode heet ook wel *selection sort*.
We zullen het nu schrijven als ware array een klasse, waarvan we een member-functie aan het maken zijn. We gebruiken een (private) member-variabele `inhoud`, waarin uiteraard het array zit opgeborgen:

```
class array {
  private:
    int inhoud[n];
  public:
    void simpelsort ( );
    ...
};  // array
```

```
 void array::simpelsort ( ) {
  int voorste, kleinste, plaatskleinste, k;
  for ( voorste = 0; voorste < n; voorste++ ) {
    plaatskleinste = voorste;
    kleinste = inhoud[voorste];
    for ( k = voorste + 1; k < n; k++ )
      if ( inhoud[k] < kleinste ) {
        kleinste = inhoud[k];
        plaatskleinste = k;
      }  // if
    if ( plaatskleinste > voorste )
      wissel (inhoud[plaatskleinste],inhoud[voorste]);
  }  // for
}  // array::simpelsort
```

Voorbeeldprogramma 44: Simpel sorteren

Hierbij geeft de "binary scope resolution operator" `::` aan dat de functie `simpelsort` hoort bij de klasse `array`. De heading zou hier (zonder OOP) ook mogen luiden:

```
void simpelsort (int inhoud[ ], int n) {
```

Nogmaals: de eerste parameter is een array, wat in C++ hetzelfde is als een pointer naar — in dit geval — een `int`. De *actuele* lengte van dit array wordt als tweede parameter meegegeven; zelfs als de eerste parameter `int inhoud[n]` was geweest — waar de C++-compiler overigens niets speciaals mee doet — moeten we nog steeds goed op de array-grenzen letten!

4.2.4 Bubblesort

Een eenvoudige variant hierop is de methode *bubblesort*. De C++-code is bijzonder compact, maar bubblesort is helaas niet zo'n goede sorteermethode. Het sorteren van een rijtje met n getallen kost altijd $\frac{1}{2}n(n-1)$ vergelijkingen; vaak zegt men: $O(n^2)$ (kwadratische orde). De volgende methodes, Shellsort en quicksort, doen dat soms of vaak (misschien zelfs altijd) sneller.

```
void bubblesort (int A[ ], int n) {
  int ronde, j;
  for ( ronde = 1; ronde < n; ronde++ )
    for ( j = 0; j < n - ronde; j++ )
      if ( A[j] > A[j+1] )
        wissel (A[j],A[j+1]);
}  // bubblesort
```

Voorbeeldprogramma 45: Bubblesort

Een array, `A` geheten, wordt als volgt op grootte gesorteerd. In de eerste ronde worden `A[0]` en `A[1]` vergeleken en indien nodig (namelijk als `A[0]` groter is dan `A[1]`) wordt hun inhoud verwisseld; daarna `A[1]` en `A[2]`, ..., `A[n-2]` en `A[n-1]`; het is duidelijk dat het grootste element nu achteraan staat. In de tweede ronde worden `A[0]` en `A[1]` vergeleken, ..., `A[n-3]` en `A[n-2]`; er vindt dus één vergelijking minder plaats. Zo gaat dit verder; in de laatste (de `(n-1)`ste) ronde hoeven alleen nog maar `A[0]` en `A[1]` vergeleken te worden. De grote getallen "borrelen" als het ware naar achteren, vandaar de naam bubblesort.

4.2.5 Invoegsorteer

Als een rij gesorteerd is en men wil een element toevoegen, dan kan *invoegsorteer*, oftewel *insertion sort*, gebruikt worden: zoek de plaats op waar het element moet komen en voeg het element op die plaats ertussen (bij een array betekent dat, dat het achterstuk één plaats naar achter moet schuiven).
Een ongesorteerde rij kan men met invoegsorteer sorteren door element voor element aan een oorspronkelijke lege rij (die is namelijk "per definitie" gesorteerd) toe te voegen zoals beschreven. De complexiteit is vergelijkbaar met die van bubblesort. De C++-code laten we als opgave: Opgave 54.

4.2.6 Shellsort

De methode *Shellsort* sorteert bijzonder snel. Het gaat als volgt: we zetten eerst de getallen in het array `A` die op zekere afstand, zeg `h`, van elkaar liggen goed ten opzichte van elkaar; dit noemen we `h`-sorteren. Er geldt dan `A[i] <= A[i+h]` voor alle `i` waarvoor `i` en `i+h` geldige array-indices zijn. Verander (bijvoorbeeld halveer, maar er zijn betere keuzes) die afstand in `h'`, en ga dan `h'`-sorteren. Herhaal dit tot (en met) de afstand `1` is. Het sorteren van de deelrijen gaat met een of andere "snelle" methode. Wij gebruiken hier een variant op bubblesort, waarbij de deelrijen ingenieus vervlochten worden. Het aardige is dat een `h`-gesorteerde rij na het `h'`-sorteren nog steeds `h`-gesorteerd is. De complexiteit van Shellsort hangt sterk af van de keuze van de `h`'s. Zie verder Opgave 54.

4.2.7 Quicksort

Het sorteren van een rij getallen kunnen we ook recursief doen. We krijgen dan (symbolisch genoteerd) zoiets als:

```
sorteer (rij) :
  if ( rij heeft meer dan 1 element ) {
```

```
void array::shellsort ( ) {
  int i, j, sprong = n;
  bool klaar;
  while ( sprong > 1 ) {  // inhoud is nu sprong-gesorteerd
    sprong = sprong / 2;  // er bestaan betere keuzes dan / 2
    klaar = false;
    while ( ! klaar ) {
      klaar = true;
      for ( i = 0; i + sprong < n; i++ ) {
        j = i + sprong;
        if ( inhoud[i] > inhoud[j] ) {
          wissel (inhoud[i],inhoud[j]);
          klaar = false;
        }  // if
      }  // for
    } // while
  }  // while
}  // array::shellsort
```

Voorbeeldprogramma 46: Shellsort

```
    verdeel rij in linkerrij en rechterrij;
    sorteer (linkerrij);
    sorteer (rechterrij);
    combineer (linkerrij, rechterrij);
  }
```

De methode quicksort heeft deze vorm. Bij quicksort wordt al het werk gestoken in de verdeelstap, waardoor het combineren niets doen wordt. Bij de recursieve methode mergesort daarentegen is het verdelen makkelijk (in tweeën hakken), en zit al het werk in de combineerstap (*ritsen*). Als men een rij van n stuks verdeelt in een linkerrij van $n-1$ elementen en een rechterrij van 1, en combineren wordt invoegen, dan krijgen we de methode insertion sort terug.
Het sorteeralgoritme *quicksort*, dat gebruikt wordt om een array `A` dat `n` gehele getallen bevat te sorteren, werkt als volgt. Zij `K` gelijk aan `A[0]`. Alle getallen kleiner dan `K` worden vooraan gezet, alle getallen groter dan `K` achteraan, en `K` zelf ertussen. Vervolgens worden begin- en eindstuk (recursief) met quicksort gesorteerd.
De C++-code laten we liggen voor het college Algoritmiek. Het handig in het array zelf manipuleren (met `K`) blijkt nog niet zo eenvoudig te zijn. Hoe het ook zij, de methode doet zijn naam eer aan: mits zorgvuldig toegepast, wordt er snel gesorteerd. Zie Opgave 55.
Op internet zijn mooie visualisaties van sorteermethoden te bewonderen.

4.2.8 Indexbestanden

Vaak wordt een groot bestand niet fysiek gesorteerd, maar met behulp van zogeheten *indexbestanden* — ook wel *index-arrays* genoemd, let op het verschil met array-indices (in het Engels *subscripts*) — toegankelijk gemaakt. Dit heeft verschillende oorzaken. Het bestand kan te groot zijn, of het is al op een andere *sleutel* gesorteerd.

Een indexbestand is een array met gehele getallen die aangeven in welke volgorde het oorspronkelijke bestand gesorteerd moet worden doorlopen. Zo is het eerste element uit het indexbestand de array-index van het “kleinste” element uit het oorspronkelijke bestand. Als opgave: schrijf een C++-programma dat dit proces implementeert, zie ook Opgave 50.

5 Abstracte Datastructuren

Soms zijn de door C++ standaard aangeleverde datatypen niet toereikend; bijvoorbeeld omdat je algoritme minder goed werkt, je niet-standaard gegevens wilt gebruiken of de opslagcapaciteit te klein is. Met behulp van de in C++ aangeleverde mechanismen als classes en pointers is het goed mogelijk zelf wat ingewikkeldere en meer specialistische *datastructuren* te creëren. De kenmerken hiervan hangen meestal sterk samen met het algoritme waar ze voor ontwikkeld worden. Bij vakken als Algoritmiek en Datastructuren zal dieper ingegaan worden op de verschillende mogelijkheden.

5.1 Stapels en rijen

Stapels en rijen zijn, evenals de reeds genoemde verzamelingen, een fraai voorbeeld van abstracte datatypen. Stapels en rijen zijn datastructuren waarmee een aantal problemen op inzichtelijke wijze kan worden opgelost. Ze zijn niet standaard in C++ aanwezig, dus als men er gebruik van wil maken zal men eerst een implementatie moeten maken. Nu kan een rij of stapel zowel met behulp van array's als met behulp van pointers worden gefabriceerd. Beide methoden hebben hun eigen verdiensten.
Een *stapel* (*stack*, denk aan een stapel borden) is een reeks elementen van hetzelfde type met de volgende toegestane operaties:

- men kan een lege stapel aanmaken,
- men kan zien of de stapel leeg is,
- men kan er een element aan toevoegen (*push*),
- men kan er het laatst-toegevoegde element weer uithalen (*pop*).

Een stapel heeft dus de *LIFO*-eigenschap: LIFO = Last In First Out.
Een *rij* (*queue*, denk aan een rij voor een kassa) is een reeks elementen van hetzelfde type met de volgende toegestane operaties:

- men kan een lege rij aanmaken,
- men kan zien of de rij leeg is,
- men kan er een element aan toevoegen,
- men kan er het eerst-toegevoegde element weer uithalen.

Een rij heeft dus de *FIFO*-eigenschap: FIFO = First In First Out.
Bij implementaties van stapels/rijen moet men soms nog een functie toevoegen:

- kijk of de stapel/rij vol is.

Als we alleen naar deze eigenschappen kijken vatten we stapels en rijen op als zogenaamde abstracte datatypen. Deze blijken een zeer belangrijke rol te vervullen binnen de informatica; we zullen ze nog vaak tegenkomen, met name bij vakken als Algoritmiek en Datastructuren. De taal C++ leent er zich bijzonder goed voor om abstracte datatypen te maken. Dat komt ook omdat het begrip abstract datatype in de object-georiënteerde wereld zit ingebakken.
Een voorbeeld. Stel dat we uit een stapel met gehele getallen alle optredens van het grootste getal willen verwijderen; verder moet de stapel onderanderd blijven. Als de stapel leeg was moet er niets gebeuren.

```
void haalgrootstegetaluitstapel (stapel& S) {
  stapel hulp;
  int x, max;
  if ( ! S.isstapelleeg ( ) ) {
     S.haalvanstapel (max);
     hulp.zetopstapel (max);
     while ( ! S.isstapelleeg ( ) ) {
       S.haalvanstapel (x);
       if ( x > max ) max = x;
       hulp.zetopstapel (x);
     }  // while
     while ( ! hulp.isstapelleeg ( ) ) {
       hulp.haalvanstapel (x);
       if ( x != max ) S.zetopstapel (x);
     }  // while
  }  // if
}  //  haalgrootstegetaluitstapel
```

Voorbeeldprogramma 47: Stapel: Verwijder grootste

Let er op dat klassen in C++ doorgaans call-by-reference worden doorgegeven, met een `&` dus. Dan wordt er namelijk geen lokale kopie gemaakt, wat anders soms nare gevolgen heeft. Verder nemen we aan dat er een constructor is, die ervoor zorgt dat `stapel hulp;` meteen `hulp` de lege stapel maakt.
Als voorbeeld zullen we nu een pointer-implementatie maken van een stapel voor gehele getallen. Als opgave: geef een array-implementatie. Met behulp van *templates* kunnen we zelfs stapels voor "willekeurige" types fabriceren.
We hebben allereerst een overigens al bekend extra type nodig:

```
class vakje {  // een struct mag ook
  public:
    // constructor (een destructor hoeft misschien niet)
    vakje ( ) {
      info = 0; volgende = NULL; }  // constructor vakje
    int info;
    vakje* volgende;
};  // vakje
```

Voorbeeldprogramma 48: Een Stapelvakje

De vakjes waaruit de pointerlijst straks bestaat, zijn opgebouwd uit een veld `info` voor een geheel getal en een veld `volgende` voor de rest van de stapel. En nu de klasse stapel:

```
// Klasse stapel:
class stapel {
  public:
    // constructor
    stapel ( ) {
      bovenste = NULL; }  // constructor stapel
    // destructor
    ~stapel ( );
    // zet element bovenop stapel: push
    void zetopstapel (int);
    // haal element van stapel: pop
    void haalvanstapel (int&);
    // is stapel leeg?
    bool isstapelleeg ( ) {
      return ( ( bovenste == NULL ) ? true : false );
      // return !bovenste; doet het ook, of if ( bovenste == NULL ) ...
    }  // isstapelleeg
    // afdrukken stapel: soms handig
    void drukaf ( );
  private:
    vakje *bovenste;
};  // stapel
```

Voorbeeldprogramma 49: De klasse stapel

De member-functies van de klasse `stapel` zijn:

```
stapel::~stapel ( ) {
  int getal;
  while ( ! isstapelleeg ( ) ) haalvanstapel (getal);
}  // stapel::~stapel

void stapel::zetopstapel (int getal) {
  vakje *temp;
  temp = new vakje;  // constructor voor vakje wordt aangeroepen!
  temp->info = getal;
  temp->volgende = bovenste;
  bovenste = temp;
}  // stapel::zetopstapel

void stapel::haalvanstapel (int& getal) {
  vakje *temp;
  getal = bovenste->info;
  temp = bovenste;
  bovenste = bovenste->volgende;
  delete temp;
}  // stapel::haalvanstapel

void stapel::drukaf ( ) {
```

```
  vakje *temp = bovenste;
  cout << "Stapel bevat:" << endl;
  while ( temp != NULL ) {  // while ( temp ) mag ook: NULL == 0!
    cout << temp->info << "  ";
    temp = temp ->volgende;
  }  // while
  cout << endl;
}  // stapel::drukaf
```

Tot slot een `main` die dit gebruikt:

```
main ( ) {
  stapel S;
  int getal = 0;
  while ( getal >= 0 ) {
    S.drukaf ( );
    cout << "Zet getal op stapel; 0 haalt van stapel, < 0 stopt" << endl;
    cin >> getal;
    if ( getal > 0 ) S.zetopstapel (getal);
    else if ( ( getal == 0 ) && ( ! S.isstapelleeg ( ) ) ) {
      S.haalvanstapel (getal);
      cout << getal << " van stapel gehaald " << endl;
    }  // if
  }  // while
  return 0;
}  // main
```

Voor een rij gaat het analoog:

```
class rij {
  private:
    int aantal; // is niet nodig, misschien wel handig
    vakje* voorste;
    vakje* achterste;
  public:
    rij ( ) {
      aantal = 0; voorste = NULL; achterste = NULL; }  // constructor rij
    ~rij ( );
    bool isrijleeg ( ) {
      return ( voorste == NULL ); }  // isrijleeg
    void zetinrij (int getal );
    void haaluitrij (int& getal );
};  // rij
```

Voorbeeldprogramma 50: De klasse rij

Bij de implementatie moet erop gelet worden aan welke kant verwijderd en aan welke kant toegevoegd moet worden. Het blijkt lastig om te verwijderen bij `achterste` (als tenminste de enkelverbonden lijst bij `voorste` begint), dus verwijderen we bij `voorste`. We krijgen:

```
rij::~rij ( ) {
  int getal;
  while ( ! isrijleeg ( ) ) haaluitrij (getal);
}  // rij::~rij

void rij::zetinrij (int getal) {
  vakje* hulp = new vakje;
  aantal++;
  hulp->info = getal;
  hulp->volgende = NULL;
  if ( achterste != NULL ) achterste->volgende = hulp;
  else voorste = hulp;
  achterste = hulp;
}  // rij::zetinrij

void rij::haaluitrij (int& getal) {
  vakje* hulp = voorste;
  aantal--;
  getal = hulp->info;
  voorste = voorste->volgende;
  if ( voorste == NULL ) achterste = NULL;
  delete hulp;
}  // rij::haaluitrij
```

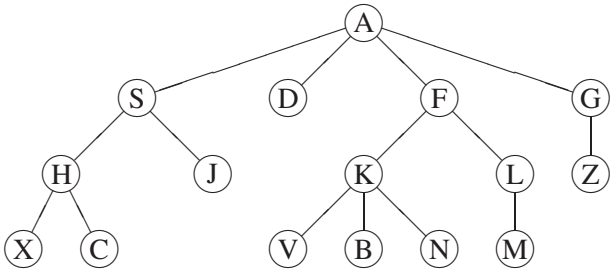
Een rij kan ook met behulp van een array worden gemaakt, zie verderop. Hierbij is `voor` de array-index van het voorste getal uit de rij, en `achter` de array-index van het laatste getal uit de rij. Omdat een volle rij van een lege rij moet kunnen worden onderscheiden, blijft er altijd minstens één array-element ongebruikt. In de praktijk wordt dit vaak anders opgelost: introduceer een extra membervariabele `aantal` die het actuele aantal elementen in de rij bijhoudt. Een destructor mag hier weer achterwege blijven.

5.2 Binaire bomen

Een boom wordt in de discrete wiskunde gedefinieerd als een samenhangende (dat wil zeggen bestaande uit één stuk) graaf zonder cykels (= kringen). Als we nu een speciale knoop aanwijzen, de *wortel*, en we tekenen de paden vanaf de wortel naar de andere knopen naar beneden, dan krijgen we een hiërarchische structuur die lijkt op een stamboom. We gebruiken voor dit soort bomen dan ook eenzelfde terminologie: kind, afstammeling, ouder, voorouder. In een boom hebben we dus ouder-kind relaties tussen knopen, en men kan sommige andere knopen alleen via de wortel bereiken. Hieronder een voorbeeld met karakters in de knopen.

```
const int MAX = 100; class rij {  // maximaal MAX-1 elementen
  public:
    rij ( ) {
      voor = 1; achter = 0; }  // constructor rij
    bool isrijleeg ( ) {
      return ( ( achter + 1 ) % MAX == voor ); }  // isrijleeg
    bool isrijvol ( ) {
      return ( ( achter + 2 ) % MAX == voor ); }  // isrijvol
    void zetinrij (int getal) {
      achter = ( achter + 1 ) % MAX;
      inhoud[achter] = getal; }  // zetintrij
    void haaluitrij (int& getal) {
      getal = inhoud[voor];
      voor = ( voor + 1 ) % MAX; }  // haaluitrij
  private:
    int inhoud[MAX];
    int voor, achter;
};  // rij
```

Voorbeeldprogramma 51: Een alternatieve rij



Een *binaire boom* is nu een boom waarin elke knoop ofwel nul, ofwel één ofwel twee kinderen heeft. In dat laatste geval noemen we het ene kind het linkerkind, en het andere het rechterkind. Als een knoop één kind heeft, dan is dat ofwel een linkerkind ofwel een rechterkind.
Een betere definitie van binaire boom is de volgende. Hierin wordt een binaire boom recursief gedefinieerd.

Definitie: een binaire boom is een eindige verzameling knopen die ofwel leeg is, ofwel bestaat uit een knoop (de wortel) en twee disjuncte verzamelingen knopen die beide ook weer een binaire boom zijn, genaamd linkersubboom en rechtersubboom.

We zullen een binaire boom gebruiken om informatie in op te slaan. Die informatie, bijvoorbeeld gehele getallen of letters, staat in de knopen. De structuur is slechts toegankelijk via de wortel. Een voor de hand liggende implementatie van een binaire boom is met pointers, en staat hieronder. We gebruiken hier even geen aparte klasse binaire boom om het niet nodeloos ingewikkeld te maken. Allereerst definiëren we een klasse knoop, vervolgens de boom als een

pointer naar een knoop. Het gaat hier om een binaire boom die gehele getallen bevat. De knopen zijn opgebouwd uit een veld `info` voor een geheel getal, een veld `links` voor de linkersubboom en een veld `rechts` voor de rechtersubboom.

```
class knoop {  // een struct mag ook
  public:
    knoop ( ) {
      info = 0; links = NULL; rechts = NULL; }  // constructor knoop
    int info;
    knoop* links;
    knoop* rechts;
};  // knoop
```

Voorbeeldprogramma 52: Knoop van een binaire boom

```
knoop* wortel; // de ingang tot de binaire boom
```

Aangezien een binaire boom recursief gedefinieerd is, is het niet verwonderlijk dat veel functies die iets met bomen doen ook recursief zijn. We geven wat voorbeelden.

1. *Wandelingen* Vaak is het nodig om alle knopen van de boom te bezoeken. Bijvoorbeeld om de volledige inhoud van de boom achter elkaar af te drukken. De drie meest gebruikte wandelingen zijn de *preorde* wandeling (WLR: Wortel-Links-Rechts), de *symmetrische* wandeling (LWR) en de *postorde* wandeling (LRW). Als voorbeeld bekijken we de preorde wandeling. Hierbij wordt eerst de wortel van de boom bezocht, en vervolgens de linkersubboom (op preorde manier) en tenslotte de rechtersubboom (op preorde manier). Het basisgeval is de lege boom (niets doen!). Dit leidt tot de volgende recursieve C++-functie :

```
void preorde (knoop* wortel) {
  if ( wortel != NULL ) {
    cout << wortel->info << endl;
    preorde (wortel->links);
    preorde (wortel->rechts);
  }  // if
}  // preorde
```

Voorbeeldprogramma 53: Preorde wandeling

2. *Knopen tellen* Om het aantal knopen van de hele boom te bepalen moet je het aantal knopen van de linkersubboom en het aantal knopen van de rechtersubboom bij elkaar optellen, plus nog 1 voor de wortel. De lege boom heeft uiteraard 0 knopen. We krijgen in C++:

3. *Boom afbreken* We willen nu de boom netjes helemaal afbreken, dus alle knopen met delete weghalen. Na afloop is de boom dus leeg (NULL!). Merk op dat we hier een postorde wandeling gebruiken.

```
int aantal (knoop* wortel) {
  if ( wortel == NULL )
    return 0;
  else
    return ( 1 + aantal (wortel->links) + aantal (wortel->rechts) );
}  // aantal
```

Voorbeeldprogramma 54: Boomknopen tellen

```
void breekaf (knoop* & wortel) {
  if ( wortel != NULL ) {
    breekaf (wortel->links);
    breekaf (wortel->rechts);
    delete wortel;
    wortel = NULL;
  }  // if
}  // breekaf
```

Voorbeeldprogramma 55: Binaire boom vellen

Er zijn vele soorten binaire bomen, elk met hun eigen toepassingen. We noemen hier even de *binaire zoekboom*. Dit is een binaire boom waarin voor de waarden in de knopen geldt: de waarde in de knoop zelf is groter dan de waarden in zijn linkersubboom, en kleiner dan de waarden in zijn rechtersubboom. In deze bomen kunnen we efficiënt zoeken. Zie verder de vakken Algoritmiek en Datastructuren.