Objectgeoriënteerd Programmeren in C++

Harry Broeders
De Haagse Hogeschool
Opleiding Elektrotechniek
27 april 2015
J.Z.M.Broeders@hhs.nl

Voorwoord

Dit dictaat is lang geleden begonnen als een korte introductie van C++ geschreven in WordPerfect en inmiddels uitgegroeid tot een met LaTeX opgemaakt document van meer dan honderd pagina's. In de tussentijd hebben veel studenten en collega's mij feedback gegeven. De opbouwende kritiek van Harm Jongsma, Cor Diependaal, Henk van den Bosch, Dave Stikkelorum, John Visser, Gerard Tuk, Fidelis Theinert, Ben Kuiper, René Theil en vele anderen hebben dit dictaat ontegenzeggelijk beter gemaakt. Desondanks heb ik niet de illusie dat dit dictaat foutvrij is. Op- en aanmerkingen zijn dus nog altijd welkom, mail me op J.Z.M.Broeders@hhs.nl.

Alle programmacode in dit dictaat is getest met Microsoft Visual Studio 2013[1] en GCC[2] versie 4.9.2. Op http://bd.eduweb.hhs.nl/ogoprg/ vind je de source code van alle in dit dictaat besproken programma's. Als je dit dictaat leest op een device met internettoegang, dan kun je op de bestandsnamen in dit dictaat klikken om de programma's te downloaden.

Veel literatuur over C++ en objectgeoriënteerd programmeren is geschreven in het Engels. Om er voor te zorgen dat je deze literatuur na (of tijdens) het lezen van dit dictaat eenvoudig kunt gebruiken heb ik veel van het jargon dat in deze litera-

[1] Microsoft Visual Studio 2013 Express for Windows Desktop is gratis te gebruiken. Zie: http://www.microsoft.com/visualstudio/eng/downloads#d-express-windows-desktop

[2] De GNU Compiler Collection (GCC) bevat een zeer veel gebruikte open source C++ compiler. Zie http://gcc.gnu.org/

tuur wordt gebruikt niet vertaald naar het Nederlands. De keuze om een term wel of niet te vertalen is arbitraal. Zo heb ik bijvoorbeeld het Engelse begrip “return type” niet vertaal in het Nederlandse “terugkeertype” of “retourtype”, maar heb ik het Engelse begrip “parameter type” wel vertaald naar het Nederlandse “parametertype”. De Engelse begrippen “class” en “class template” zijn niet vertaald in “klasse” en “klassetemplate” maar de Engelse begrippen “function” en “function template” zijn wel vertaald in “functie” en “functietemplate”.

Dit dictaat wordt onder andere gebruikt bij de lessen Objectgeoriënteerd Programmeren (OGOPRG) van de opleiding Elektrotechniek aan De Haagse Hogeschool. Af en toe wordt in dit dictaat verwezen naar deze lessen. Als je dit dictaat leest zonder de lessen OGOPRG te volgen, dan kun je deze verwijzingen negeren.

Inhoudsopgave

Inleiding

Dit is het dictaat: “Objectgeoriënteerd Programmeren in C++”. Dit dictaat kan zonder boek gebruikt worden. Als je meer achtergrondinformatie of diepgang zoekt, kun je gebruik maken van het gratis boek: *Thinking in C++, Volume 1: Introduction to Standard C++* van Eckel[2]. Dit boek is gratis te downloaden van `http://www.mindview.net/Books/TICPP/ThinkingInCPP2e.html` maar is echter wel enigszins verouderd dus voor gedetailleerde informatie verwijs ik je naar `http://en.cppreference.com/w/cpp`. Dit dictaat is zoals alle mensenwerk niet foutloos, verbeteringen en suggesties zijn altijd welkom!

Halverwege de jaren ’70 werd steeds duidelijker dat de veel gebruikte software ontwikkelmethode structured design (ook wel functionele decompositie genoemd) niet geschikt is om grote uitbreidbare en onderhoudbare software systemen te ontwikkelen. Ook bleken de onderdelen van een applicatie die met deze methode is ontwikkeld, meestal niet herbruikbaar in een andere applicatie. Het heeft tot het begin van de jaren ’90 geduurd voordat een alternatief voor structured design het zogenoemde, object oriented design (OOD), echt doorbrak. Objectgeoriënteerde programmeertalen bestaan al sinds het begin van de jaren ’70. Deze manier van ontwerpen (OOD) en programmeren (OOP) is echter pas in het begin van de jaren ’90 populair geworden nadat in het midden van de jaren ’80 de programmeertaal C++ door Bjarne Stroustrup was ontwikkeld. Deze taal voegt taalconstructies toe aan de op dat moment in de praktijk meest gebruikte programmeertaal C. Deze objectgeoriënteerde versie van C heeft de naam C++ gekregen en heeft zich ontwikkeld tot één van de meest gebruikte programmeertalen van dit moment. C++

is echter geen pure OO taal (zoals bijvoorbeeld Smalltalk) en kan ook gebruikt worden als procedurele programmeertaal. Dit heeft als voordeel dat de overstap van C naar C++ eenvoudig te maken is maar heeft als nadeel dat C++ gebruikt kan worden als een soort geavanceerd C zonder gebruik te maken van OOP. De taal C++ is nog steeds in ontwikkeling. De laatste versie van de C++ standaard wordt C++14 genoemd [4][3].

Een terugblik op C

We starten dit dictaat met de overgang van gestructureerd programmeren in C naar gestructureerd programmeren in C++. *Ik ga er van uit dat je de taal C redelijk goed beheerst.* Misschien is het nodig om deze kennis op te frissen. Vandaar dat dit dictaat begint met een terugblik op C (wordt verder in de les niet behandeld). We zullen dit doen aan de hand van het voorbeeldprogramma C.c dat een lijst met gewerkte tijden (in uren en minuten) inleest vanaf het toetsenbord en de totaal gewerkte tijd (in uren en minuten) bepaalt en afdrukt.

```
#include <stdio.h>
/* nodig voor gebruik printf (schrijven naar scherm)
                    en scanf  (lezen uit toetsenbord) */

typedef struct { /* Een Tijdsduur bestaat uit:  */
    int uur;     /*    een aantal uren en       */
    int minuten; /*    een aantal minuten.      */
} Tijdsduur;

/* Deze functie drukt een Tijdsduur af */
void drukaf(Tijdsduur td) {
    if (td.uur == 0)
        printf("           %2d minuten\n", td.minuten);
    else
```

[3] Een zogenoemde late draft version van de C++14 standaard is beschikbaar op http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2014/n4296.

```
        printf("%3d uur en %2d minuten\n", td.uur, ↩
         ↪ td.minuten);
}

/* Deze functie drukt een rij met een aantal gewerkte ↩
 ↪ tijden af */
void drukafRij(Tijdsduur rij[], int aantal) {
    int teller;
    for (teller = 0; teller < aantal; teller++)
        drukaf(rij[teller]);
}

/* Deze functie berekent de totaal gewerkte tijd uit een ↩
 ↪ rij met een aantal gewerkte tijden */
Tijdsduur som(Tijdsduur rij[], int aantal) {
    int teller;
    Tijdsduur s = {0, 0};
    for (teller = 0; teller < aantal; teller++) {
        s.uur += rij[teller].uur;
        s.minuten += rij[teller].minuten;
    }
    s.uur += s.minuten / 60;
    s.minuten %= 60;
    return s;
}

#define MAX 5

int main(void) {
    Tijdsduur gewerkteTijdenRij[MAX];
    int aantal = 0, gelezen;
    do {
        printf("Type gewerkte uren en minuten in (of ↩
         ↪ Ctrl-Z): ");
        gelezen = scanf("%d%d", ↩
         ↪ &gewerkteTijdenRij[aantal].uur, ↩
         ↪ &gewerkteTijdenRij[aantal].minuten);
```

```
    }
    while (gelezen == 2 && ++aantal < MAX);
    printf("\n\n");
    drukafRij(gewerkteTijdenRij, aantal);
    printf("De totaal gewerkte tijd is:\n");
    drukaf(som(gewerkteTijdenRij, aantal));
    fflush(stdin); getchar();
    return 0;
}
```

Verklaring:

- In de eerste regel wordt de file `stdio.h` geïnclude. Dit is nodig om gebruik te kunnen maken van functies en types die in de standaard C I/O library zijn opgenomen. In dit programma maak ik gebruik van `printf` (om te schrijven naar het scherm), van `scanf` (om getallen te lezen vanaf het toetsenbord), van `fflush` (om het invoerbuffer te wissen) en van `getchar` (om een karakter te lezen vanaf het toetsenbord).
- Vervolgens is het samengestelde type `Tijdsduur` gedeclareerd. Variabelen van dit type bevatten twee datavelden (Engels: data members) van het type `int`. Deze datavelden heten `uur` en `minuten` en zijn bedoeld voor de opslag van de uren en de minuten van de betreffende tijdsduur.
- Vervolgens worden er drie functies gedefinieerd:
    - `drukaf`. Deze functie drukt de parameter `td` van het type `Tijdsduur` af op het scherm door gebruik te maken van de standaard schrijffunctie `printf`. Het return type van de functie `drukaf` is `void`. Dit betekent dat de functie geen waarde teruggeeft.
    - `drukafRij`. Deze functie drukt een rij met gewerkte tijden af. Deze functie heeft twee parameters. De eerste parameter genaamd `rij` is een array met elementen van het type `Tijdsduur`. De tweede parameter (een integer genaamd `aantal`) geeft aan hoeveel elementen uit de array afgedrukt moeten worden. Tijdens het uitvoeren van de `for`-lus krijgt de lokale integer variabele `teller` achtereenvolgens de waarden

`0` t/m `aantal-1`. Deze teller wordt gebruikt om de elementen uit `rij` één voor één te selecteren. Elk element (een variabele van het type `Tijdsduur`) wordt met de functie `drukaf` afgedrukt.

- `som`. Deze functie berekent de som van een rij met gewerkte tijden. Deze functie heeft dezelfde twee parameters als de functie `drukafRij`. Deze functie heeft ook een lokale variabele genaamd `teller` met dezelfde taak als bij de functie `drukafRij`. De functie `som` definieert de lokale variabele `s` van het type `Tijdsduur` om de som van de rij gewerkte tijden te berekenen. De twee datavelden van de lokale variabele `s` worden eerst gelijk gemaakt aan nul en vervolgens worden de gewerkte tijden uit `rij` hier één voor één bij opgeteld. Twee gewerkte tijden worden bij elkaar opgeteld door de uren bij elkaar op te tellen en ook de minuten bij elkaar op te tellen. Als alle gewerkte tijden op deze manier zijn opgeteld, kan de waarde van `s.minuten` groter dan `59` zijn geworden. Om deze reden wordt de waarde van `s.minuten / 60` opgeteld bij `s.uur`. De waarde van `s.minuten` moet dan gelijk worden aan de resterende minuten. Het aantal resterende minuten kunnen we bereken met `s.minuten = s.minuten % 60`. Of in verkorte notatie `s.minuten %= 60`. Het return type van de functie `som` is van het type `Tijdsduur`. Aan het einde van de functie som wordt de waarde van de lokale variabele `s` teruggegeven (`return s`).

- Vervolgens wordt met de preprocessor directive `#define` de constante `MAX` gedefinieerd met als waarde `5`.
- Tot slot wordt de hoofdfunctie `main` gedefinieerd. Deze functie zal bij het starten van het programma aangeroepen worden. In de functie `main` wordt een array `gewerkteTijdenRij` aangemaakt met `MAX` elementen van het type `Tijdsduur`. Tevens wordt de integer variabele `aantal` gebruikt om het aantal ingelezen gewerkte tijden te tellen. Elke gewerkte tijd bestaat uit twee integers: een aantal uren en een aantal minuten. In de `do while`-lus worden gewerkte tijden vanaf het toetsenbord ingelezen in de array `gewerkteTijdenRij`. De getallen worden ingelezen met de standaard leesfunctie `scanf`. Deze functie bevindt zich in de standaard C library en het

prototype van de functie is gedeclareerd in de headerfile `stdio.h`. De eerste parameter van `scanf` specificeert wat er ingelezen moet worden. In dit geval twee integer getallen. De volgende parameters specificeren de adressen van de variabelen waar de ingelezen integer getallen moeten worden opgeslagen. Met de operator `&` wordt het adres van een variabele opgevraagd. De functie `scanf` geeft een integer terug die aangeeft hoeveel velden ingelezen zijn. Deze return waarde wordt opgeslagen in de variabele `gelezen`. De conditie van de `do while`-lus is zodanig ontworpen dat de `do while`-lus uitgevoerd wordt zo lang de return waarde van `scanf` gelijk is aan `2` én `++aantal<MAX` . De `++` voor `aantal` betekent dat de waarde van `aantal` eerst met één wordt verhoogd en daarna wordt vergeleken met `MAX`. Over deze conditie is echt goed nagedacht. De `&&` wordt namelijk altijd van links naar rechts uitgevoerd. Als de expressie aan de linkerkant van de `&&` niet waar is, dan wordt de expressie aan de rechterkant niet uitgevoerd en wordt de variabele `aantal` dus niet verhoogd. Als het lezen van twee integers met `scanf` niet gelukt is, dan is `gelezen` ongelijk aan `2` en wordt de variabele `aantal` niet opgehoogd. De `do while`-lus wordt dus beëindigd als het inlezen van twee integers niet gelukt is, bijvoorbeeld omdat het einde invoer teken ingetypt is (onder Windows is dit Ctrl+z onder Linux is dit Ctrl+d), óf als de array vol is. Na afloop van de `do while`-lus wordt er voor gezorgd dat de cursor op het begin van de volgende regel komt te staan en dat een regel wordt overgeslagen door de string `"\n\n"` naar het beeldscherm te schrijven met de functie `printf`. Vervolgens wordt de lijst afgedrukt met de functie `drukafRij`. Tot slot wordt de totaal gewerkte tijd (die berekend wordt met de functie `som`) afgedrukt met de functie `drukaf`.

Net voor het einde van de functie `main` wordt eerst de `stdio` functie `fflush (stdin)` en daarna de `stdio` functie `getchar()` aangeroepen. De functie `fflush` maakt het input buffer leeg[4] en de functie `getchar` wacht totdat de

[4] Volgens de C standaard mag fflush niet op een input stream gebruikt worden, zie http://en.cppreference.com/w/c/io/fflush, maar in Microsoft Visual Studio mag dit wel, zie https://msdn.microsoft.com/library/9yky46tz.aspx. Omdat we de `fflush(stdin)` hier gebruiken om een specifiek probleem van Microsoft Visual Studio op te lossen vind ik dit geen probleem.

gebruiker een return intoetst. Dit is nodig omdat de debugger van Microsoft Visual Studio anders het output window meteen sluit als het programma eindigt[5]. Als je gebruik maakt van de GCC compiler in combinatie met de gratis geïntegreerde ontwikkelomgeving (IDE) Code::Blocks (zie http://www.codeblocks.org/) dan kun je de regel met de aanroep naar `fflush` en `getchar` verwijderen omdat deze IDE het output window na afloop van het programma niet meteen zelf sluit. Tot slot geeft de functie `main` de waarde `0` aan het operating system terug. De waarde `0` betekent dat het programma op normale wijze is geëindigd.

Merk op dat dit programma niet meer dan `MAX` gewerkte tijden kan verwerken. Later in dit dictaat (paragraaf 3.8) zul je leren hoe dit probleem is op te lossen door het gebruik van dynamisch geheugen.

De taal C++ bouwt verder op de fundamenten van C. Zorg er dus voor dat jouw kennis en begrip van C voldoende is om daar nu C++ bovenop te bouwen.

Het bestuderen van dit dictaat zonder voldoende kennis en begrip van C is vergelijkbaar met het bouwen van een huis op drijfzand!

[5] Zie voor meer gedetailleerde uitleg: http://bd.eduweb.hhs.nl/gesprg/getchar.htm.

1

Van C naar C++.

De ontwerper van C++ Bjarne Stroustrup geeft op de vraag: Wat is C++? het volgende antwoord[8]:

> C++ is a general-purpose programming language with a bias towards systems programming that
>
> - is a better C,
> - supports data abstraction,
> - supports object-oriented programming, and
> - supports generic programming.

In dit hoofdstuk bespreken we eerst enkele kleine verbeteringen van C++ ten opzichte van zijn voorganger C. Grotere verbeteringen zoals data abstractie, objectgeoriënteerd programmeren en generiek programmeren komen in de volgende hoofdstukken uitgebreid aan de orde. C++ is een zeer uitgebreide taal en dit dictaat moet dan ook zeker niet gezien worden als een cursus C++. In dit dictaat worden slechts de meest belangrijke delen van C++ behandeld. De verbeteringen die in dit hoofdstuk worden besproken zijn slechts kleine verbeteringen. De

meeste worden in de les niet behandeld. Veel van deze in C++ geïntroduceerde verbeteringen zijn later ook in de C standaard opgenomen [4].

1.1 Commentaar tot einde regel

De eerste uitbreiding die we bespreken is niet erg ingewikkeld maar wel erg handig. Je bent gewend om in C programma's commentaar te beginnen met `/*` en te eindigen met `*/`. In C++ en C (vanaf C99) kun je naast de oude methode ook commentaar beginnen met `//` dit commentaar eindigt dan aan het einde van de regel. Dit is handig (minder typewerk) als je commentaar wilt toevoegen aan een programmaregel.

1.2 Binaire getallen

Sinds C++14 kun je in een C++ programma constante getallen niet alleen meer in het decimale, octale[6] en hexadecimale[7] talstelsel invoeren maar ook in het binaire talstelsel.[8] Binaire getallen kun je invoeren door ze vooraf te laten gaan door `0b` of `0B`. Dus de code `int x = 0b10111101;` initialiseert de variabele `i` met de waarde 189 want 10111101_2 is 189_{10}.

1.3 Plaats van variabeledefinities

Zie eventueel [2, blz. 156]: Defining variables on the fly

Je bent gewend om in C programma's, variabelen te definiëren aan het begin van een blok (meteen na {). In C++ en C99 kun je een variabele overal in een blok definiëren. De scope van de variabele loopt van het punt van definitie tot het einde van het blok waarin hij gedefinieerd is. Het is zelfs mogelijk om de

[6] Octale getallen kun je invoeren door ze vooraf te laten gaan door een `0`. Dus de code `int x = 023;` initialiseert de variabele `i` met de waarde 19 want 23_8 is 19_{10}.

[7] Hexadecimale getallen kun je invoeren door ze vooraf te laten gaan door `0x` of `0X`. Dus de code `int x = 0x23;` initialiseert de variabele `i` met de waarde 35 want 23_{16} is 35_{10}.

[8] Dit wordt nog niet ondersteund in Microsoft Visual Studio 2013, maar wel in GCC 4.9.2 met de optie `-std=c++14`.

besturingsvariabele van een `for`-lus in het `for` statement zelf te definiëren[9]. In C++ is het gebruikelijk om een variabele pas te definiëren als de variabele nodig is en deze variabele dan meteen te initialiseren. Dit maakt het programma beter te lezen (je ogen hoeven niet als een jojo op en neer te springen) en de kans dat je een variabele vergeet te initialiseren wordt kleiner.

Voorbeeld van het definiëren en initialiseren van een lokale variabele in een `for`-lus:

```
/* Hier bestaat i nog niet */
for (int i = 0; i < 10; ++i) {
    /* deze code wordt uitgevoerd voor i = 0, 1, ..., ↩
     ↪ 8 en 9 */;
}
/* Hier bestaat i niet meer */
```

1.4 `auto` type

Sinds C++11 is het mogelijk om de compiler zelf het type van een variabele te laten bepalen. Dit doe je door in de variabeledefinitie in plaats van een typenaam het keyword `auto` te gebruiken. In het onderstaande programma wordt een array `kampioensjarenFeyenoord` met integers aangemaakt en verderop in het programma wordt de variabele `eerste` gelijk gemaakt aan het eerste getal uit deze array.

```
int kampioensjarenFeyenoord[] = {1924, 1928, 1936, ↩
 ↪ 1938, 1940, 1961, 1962, 1965, 1969, 1971, 1974, ↩
 ↪ 1984, 1993, 1999};
// ...
int eerste = kampioensjarenFeyenoord[0];
```

Sinds C++11 kunnen we het type van de variabele `eerste` ook door de compiler laten bepalen (Engels: type deduction), zie auto.cpp:

[9] Het is zelfs mogelijk om in de conditie van een `if` of `while` statement een variabele te definiëren. Maar dit wordt maar zelden gebruikt.

```
int kampioensjarenFeyenoord[] = {1924, 1928, 1936, ↩
 ↪ 1938, 1940, 1961, 1962, 1965, 1969, 1971, 1974, ↩
 ↪ 1984, 1993, 1999};

auto eerste = kampioensjarenFeyenoord[0];
```

Dit komt de onderhoudbaarheid van het programma ten goede want als het type van de array nu veranderd wordt van `int` naar `short` dan wordt het type van de variabele `eerste` automatisch aangepast.

1.5 `auto` return type

Sinds C++14 is het mogelijk om de compiler zelf het return type van een functie te laten bepalen.[10] Dit doe je door in de functiedefinitie in plaats van een typenaam het keyword `auto` te gebruiken. In het onderstaande programma wordt een functie `max3` gedefinieerd die het maximum van drie integers bepaald. De functie is geïmplementeerd met behulp van de standaard functie `max` die het maximum van 2 getallen bepaald.

```
int max3(int i1, int i2, int i3) {
    return max(max(i1, i2), i3);
}
```

Sinds C++14 kunnen we het return type van de functie door de compiler laten bepalen (Engels: return type deduction), zie autoreturntype.cpp:

```
auto max3(int i1, int i2, int i3) {
    return max(max(i1, i2), i3);
}
```

Dit komt de onderhoudbaarheid van het programma ten goede want als het type van de parameters nu veranderd wordt van `int` naar `short` dan wordt het return type van de functie automatisch aangepast.

[10] Dit wordt nog niet ondersteund in Microsoft Visual Studio 2013, maar wel in GCC 4.9.2 met de optie `-std=c++14`.

1.6 Range-based for

In C++11 is een nieuwe variant van de `for`-lus, de zogenoemde *range-based for*, geïntroduceerd. Deze for-lus is speciaal bedoeld om de elementen van een array[11] één voor één te benaderen.

Het volgende C programmafragment bepaalt de som van alle elementen in een array:

```
int array[] = {12, 2, 17, 32, 1, 18};
int i, som = 0, aantal = sizeof array / sizeof array[0];
for (i = 0; i < aantal; i++) {
    som += array[i];
}
```

Je kunt in C++11 hetzelfde bereiken met behulp van een range-based for:

```
int array[] = {12, 2, 17, 32, 1, 18};
int som = 0;
for (int element: array) {
    som += element;
}
```

Je ziet dat de syntax van de range-based for heel eenvoudig is.

We kunnen het type van de variabelen `som` en `element` ook door de compiler laten bepalen, zie paragraaf 1.4 (RangeBasedFor.cpp):

```
int array[] = {12, 2, 17, 32, 1, 18};
auto som = 0;
for (auto element: array) {
    som += element;
}
```

[11] De standaard C++ library bevat nog vele andere datastructuren die ook met een range-based for doorlopen kunnen worden zie paragraaf 3.7 en paragraaf 3.8.

Als het type van de variabele `array` gewijzigd wordt, dan wordt het type van de variabelen `som` en `element` automatisch aangepast.

1.7 Constante waarden met `constexpr`

Je bent gewend om in C programma's symbolische constanten[12] te definiëren met de preprocessor directive `#define`. Het is in sinds C++11 mogelijk om een zogenoemde compile time constante te definiëren met behulp van het keyword `constexpr` [13]. Een compile time constante is een constante waarvan de waarde al tijdens het compileren van het programma bepaald kan worden. De grootte van een C-array moet bijvoorbeeld een compile time constante zijn. Een compile time constante kan door de compiler voor een embedded systeem in read-only geheugen worden geplaatst.

Voorbeeld met `#define`:

```
#define aantalRegels 80
```

Hetzelfde voorbeeld met `constexpr`:

```
constexpr int aantalRegels = 80;
```

Omdat je bij het definiëren van een `constexpr` het type moet opgeven kan de compiler meteen controleren of de initialisatie klopt met dit opgegeven type. Bij het gebruik van de preprocessor directive `#define` blijkt dit pas bij het gebruik van de constante en niet bij de definitie. De fout is dan vaak moeilijk te vinden. Een met `constexpr` gedefinieerde symbolische constante is bovendien bekend bij de debugger maar een met `#define` gedefinieerde symbolische constante niet.

[12] Het gebruik van symbolische constanten maakt het programma beter leesbaar en onderhoudbaar.

[13] Het C++11 keyword `constexpr` wordt helaas nog niet ondersteund door Microsoft Visual Studio 2013. Zie paragraaf 1.8 voor een alternatief.

Een compile time constante moet je initialiseren:

```
constexpr int k;
//  Error: (GCC)       uninitialized const 'k'
```

Een compile time constante kun je niet initialiseren met een variabele:

```
int i;
//  ...
constexpr int m = i;
//  Error: (GCC)       the value of 'i' is not usable in a ↩
 ↪ constant expression
```

Een compile time constante mag je (vanzelfsprekend) niet veranderen:

```
aantalRegels = 79;
//  Error: (GCC)       assignment of read-only variable ↩
 ↪ 'aantalRegels'
```

1.8 Read-only variabelen met const

Zie eventueel [2, blz. 165]: Constants

Het is in C++ en C99 mogelijk om een read-only variabele, ook wel een constante genoemd, te definiëren met behulp van een `const` qualifier.

Voorbeeld met `const`:

```
const int aantalRegels = 80;
```

Deze met een `const` qualifier gemarkeerde variabele `aantalRegels` kun je alleen uitlezen en je kunt er dus geen andere waarde in wegschrijven. Een constante moet je initialiseren:

```
const int k;
//  Error: (Microsoft) 'k' : const object must be initialized
//  Error: (GCC)       uninitialized const 'k'
```

Je mag een constante wel met een variabele initialiseren.

```
int i;
//  ...
const int m = i;
```

Als een symbolische constante geïnitialiseerd wordt met een compile time constante, zoals de bovenstaande constante `aantalRegels`, dan is deze symbolische constante ook te gebruiken als compile time constante. Een compile time constante kan door de compiler voor een embedded systeem in read-only geheugen worden geplaatst. Als een symbolische constante geïnitialiseerd wordt met een variabele, zoals de bovenstaande constante `m`, dan is deze symbolische constante *niet* te gebruiken als compile time constante. Deze constante moet dan niet door de compiler in read-write (RAM) geheugen worden geplaatst omdat de constante tijdens het uitvoeren van het programma (tijdens run time) geïnitialiseerd moet worden.

Een constante mag je (vanzelfsprekend) niet veranderen:

```
aantalRegels = 79;
//  Error: (Microsoft) 'aantalRegels' : you cannot assign to ↩
 ↪ a variable that is const
//  Error: (GCC)       assignment of read-only variable ↩
 ↪ 'aantalRegels'
```

Als je een compiler gebruikt die het keyword `constexpr`, zie paragraaf 1.7 nog niet ondersteund, dan kun je `const` ook als alternatief voor een met `#define` gedefinieerde symbolische constante gebruiken.

Omdat je bij een `const` qualifier het type moet opgeven kan de compiler meteen controleren of de initialisatie klopt met dit opgegeven type. Bij het gebruik van de preprocessor directive `#define` blijkt dit pas bij het gebruik van de constante en niet bij de definitie. De fout is dan vaak moeilijk te vinden. In paragraaf 2.9 zul je zien dat het met een `const` qualifier ook mogelijk wordt om constanten (read-only variabelen) te definiëren van zelfgemaakte types.

1.9 Het type bool

Zie eventueel [2, blz. 142]: bool, true, & false

C++ kent in tegenstelling tot C[14] het keyword `bool`. Met dit keyword kun je booleaanse variabelen definiëren van het type `bool`. Een variabele van dit type heeft slechts twee mogelijke waarden: `true` en `false`. In C89 (en ook in oudere versies van C++) wordt voor een booleaanse variabele het type `int` gebruikt met daarbij de afspraak dat een waarde `0` (nul) false en een waarde ongelijk aan nul true betekent. Om C++ zoveel mogelijk compatibel te houden met C wordt een `bool` indien nodig omgezet naar een `int` en vice versa. Het gebruik van het type `bool` maakt meteen duidelijk dat het om een booleaanse variabele gaat.

1.10 Standaard include files

Zie eventueel [2, blz. 94 en 100]: Including headers en Namespaces

De standaard C++ library bevat ook alle functies, types enz. uit de standaard C library. De headerfiles die afkomstig zijn uit de C library beginnen in de C++ library allemaal met de letter `c`. Dus `#include <cmath>` in plaats van `#include <math.h>`.

Om het mogelijk te maken dat de standaard library met andere (oude en nieuwe) libraries in één programma kan worden gebruikt zijn in C++ namespaces opgenomen. Alle symbolen uit de C++ standaard library bevinden zich in een namespace genaamd `std`. Als je een symbool uit de namespace `std` wilt gebruiken, moet je dit symbool laten voorafgaan door `std::`. Als je bijvoorbeeld de som van de sinus en de cosinus van de variabele `x` wilt berekenen in een C++ programma, dan kun je dit als volgt doen:

```
#include <cmath>
//  ...
    y = std::sin(x) + std::cos(x);
```

[14] Vanaf C99 kun je in C ook het type `bool` gebruiken. Je moet dan wel de include file `<stdbool.h>` gebruiken, in C++ is `bool` een keyword en hoef je dus niets te includen.

In plaats van elk symbool uit de C++ standaard library vooraf te laten gaan door `std::` kun je ook na het inluden de volgende regel in het programma opnemen:

```
using namespace std;
```

De som van de sinus en de cosinus van de variabele `x` kun je dus in C++ ook als volgt berekenen:

```
#include <cmath>
using namespace std;
//  ...
    y = sin(x) + cos(x);
```

1.11 Input en output met >> en <<[15]

Zie eventueel [2, blz. 99 en 107]: Using the iostreams class en Reading input

Je bent gewend om in C programma's de functies uit de `stdio` bibliotheek te gebruiken voor input en output. De meest gebruikte functies zijn `printf` en `scanf`. Deze functies zijn echter niet type veilig (*type-save*) omdat de inhoud van het, als eerste argument meegegeven, format (bijvoorbeeld `"%d"`) pas tijdens het uitvoeren van het programma verwerkt wordt. De compiler merkt het dus niet als de type aanduidingen (zoals bijvoorbeeld `%d`) die in het format gebruikt zijn niet overeenkomen met de types van de volgende argumenten. Tevens is het niet mogelijk om een eigen format type aanduiding aan de bestaande toe te voegen. Om deze redenen is in de C++ standaard [4] naast de oude `stdio` bibliotheek (om compatibel te blijven) ook een nieuwe I/O library `iostream` opgenomen. Bij het ontwerpen van nieuwe software kun je het best van deze nieuwe library gebruik maken. De belangrijkste output faciliteiten van deze library zijn de standaard output stream `cout` (vergelijkbaar met `stdout`) en de bijbehorende << operator. De belangrijkste input faciliteiten van deze library zijn de standaard input stream `cin` (vergelijkbaar met `stdin`) en de bijbehorende >> operator.

[15] Deze paragraaf heb je al gelezen als je practicum opgave 1 al hebt gedaan.

Voorbeeld met `stdio`:

```
#include <stdio.h>
//  ...
    double d;
    scanf("%d", d); // deze regel bevat twee fouten!
    printf("d = %lf\n", d);
```

Dit programmadeel bevat twee fouten die niet door de compiler gesignaleerd worden en pas bij executie blijken.

Hetzelfde voorbeeld met `iostream`:

```
#include <iostream>
//  ...
    double d;
    std::cin >> d; // lees d in vanaf toetsenbord
    std::cout << "d = " << d << std::endl; // druk d af op ↩
     ↪ scherm en ga naar het begin van de volgende regel
```

De `iostream` library is zeer uitgebreid. In dit dictaat zullen we daar niet verder op ingaan. Zie eventueel voor verdere informatie [3, hoofdstuk 2 Iostreams] of `http://en.cppreference.com/w/cpp/io`.[16]

1.12 Het type `string`[17]

Zie eventueel [3, blz. 27]: Strings

In C wordt een character string opgeslagen in een variabele van het type `char[]` (character array). De afspraak is dan dat het einde van de character string aangegeven wordt door een null character `'\0'`. In C wordt zo'n variabele aan een functie doorgegeven door middel van een character pointer (`char*`). Deze manier van het opslaan van character strings heeft vele nadelen (waarschijnlijk heb je zelf meerdere malen programma's zien vastlopen door het verkeerd gebruik van deze character strings). In de standaard C++ library is een nieuw type `string` opgeno-

[16] Zie ook practicum opgave 1 voor meer uitleg over input en output in C++.

[17] Deze paragraaf heb je al gelezen als je practicum opgave 1 al hebt gedaan.

men waarin character strings op een veilige manier opgeslagen kunnen worden. Het bewerken van deze `strings` is ook veel eenvoudiger dan strings opgeslagen in character array's. Het type `string` komt in de eerste practicumopdracht uitgebreid aan de orde.

Bijvoorbeeld het vergelijken van character strings in C:

```
#include <stdio.h>
#include <string.h>
//  ...
    char str[100];
    scanf("%100s", str);
    if (strcmp(str, "Hallo") == 0) {
        // invoer is Hallo
    }
```

Het vergelijken van strings gaat in C++ als volgt:

```
#include <iostream>
#include <string>
//  ...
    std::string str;
    std::cin >> str;
    if (str == "Hallo") {
        // invoer is Hallo
    }
```

1.13 Het type `vector` Zie eventueel [2, blz. 112]: Introducing vector

In de C++ standaard library zijn ook de types `array` en `vector` opgenomen. Deze types zijn bedoeld om de oude C-array te vervangen. In paragraaf 3.7 en paragraaf 3.8 komen we hier uitgebreid op terug.

1.14 Function name overloading

Zie eventueel [2, blz. 327]: Function Overloading

In C mag elke functienaam maar één keer gedefinieerd worden. Dit betekent dat twee functies om de absolute waarde te bepalen van variabelen van de types `int` en `double` een verschillende naam moeten hebben. In C++ mag een functienaam meerdere keren gedefinieerd worden (*function name overloading*). De compiler zal aan de hand van de gebruikte argumenten de juiste functie selecteren. Dit maakt deze functies eenvoudiger te gebruiken omdat de gebruiker (de programmeur die deze functies aanroept) slechts één naam hoeft te onthouden.[18] Dit is vergelijkbaar met het gebruik van ingebouwde operator + die zowel gebruikt kan worden om integers als om floating point getallen op te tellen.

Voorbeeld zonder function name overloading:

```
int abs_int(int i) {
    if (i < 0) return -i; else return i;
}
double abs_double(double f) {
    if (f < 0) return -f; else return f;
}
```

Hetzelfde voorbeeld met function name overloading:

```
int abs(int i) {
    if (i < 0) return -i; else return i;
}
double abs(double f) {
    if (f < 0) return -f; else return f;
}
```

[18] Je kan echter ook zeggen dat overloading het analyseren een programma moeilijker maakt omdat nu niet meer meteen duidelijk is welke functie aangeroepen wordt. Om deze reden is het niet verstandig het gebruik van overloading te overdrijven.

Als je één van deze overloaded functies wilt gebruiken om bijvoorbeeld de absolute waarde van een variabele van het type `double` te berekenen, dan kun je dit als volgt doen:

```
double in;
std::cin >> in; // lees in
std::cout << abs(in) << std::endl; // druk de absolute ↩
 ↪ waarde van in af
```

De compiler bepaalt nu zelf aan de hand van het type van het gebruikte argument (`in`) welk van de twee bovenstaande `abs` functies aangeroepen wordt.

1.15 Default argumenten

Zie eventueel [2, blz. 340]: Default Arguments

Het is in C++ mogelijk om voor de laatste parameters van een functie default argumenten te definiëren. Stel dat we een functie `print` geschreven hebben waarmee we een integer in elk gewenst talstelsel kunnen afdrukken. Het prototype van deze functie is als volgt:

```
void print(int i, int talstelsel);
```

Als we nu bijvoorbeeld de waarde 5 in het binaire (tweetallig) stelsel willen afdrukken, dan kan dit als volgt:

```
print(5, 2);  // uitvoer: 101
```

Als we de waarde 5 in het decimale (tientallig) stelsel willen afdrukken, dan kan dit als volgt:

```
print(5, 10); // uitvoer: 5
```

In dit geval is het handig om bij de laatste parameter een default argument op te geven (in het prototype) zodat we niet steeds het talstelsel 10 als tweede argument hoeven op te geven als we een variabele in het decimale stelsel willen afdrukken.

```
void print(int i, int talstelsel=10);
```

Als de functie nu met maar één argument wordt aangeroepen, wordt als tweede argument `10` gebruikt zodat het getal in het decimale talstelsel wordt afgedrukt. Deze functie kan als volgt worden gebruikt:

```
print(5, 2);  // uitvoer: 101
print(5);     // uitvoer: 5
print(5, 10); // uitvoer: 5
```

Het default argument maakt de `print` functie eenvoudiger te gebruiken omdat de gebruiker (de programmeur die deze functies aanroept) niet steeds het tweede argument hoeft mee te geven als zij/hij getallen decimaal wil afdrukken.[19]

1.16 Naam van een `struct`

In C is de naam van een struct géén typenaam. Stel dat de `struct Tijdsduur` bijvoorbeeld als volgt gedefinieerd is:

```
struct Tijdsduur { /* Een Tijdsduur bestaat uit: */
    int uur;       /*     een aantal uren en     */
    int min;       /*     een aantal minuten.    */
};
```

Variabelen van het type `struct Tijdsduur` kunnen dan als volgt gedefinieerd worden:

```
struct Tijdsduur td1;
```

Het is in C gebruikelijk om met de volgende typedefinitie een typenaam (in dit geval `TTijdsduur`) te declareren voor het type `struct Tijdsduur`:

```
typedef struct Tijdsduur TTijdsduur;
```

[19] Je kan echter ook zeggen dat het default argument het analyseren een programma moeilijker maakt omdat nu niet meer meteen duidelijk is welke waarde als tweede argument wordt meegegeven. Om deze reden is het niet verstandig het gebruik van default argumenten te overdrijven.

Variabelen van dit type kunnen dan als volgt gedefinieerd worden:

```
TTijdsduur td2;
```

In C++ is de naam van een struct meteen een typenaam. Je kunt variabelen van het type `struct Tijdsduur` dus eenvoudig als volgt definiëren:

```
Tijdsduur td3;
```

In practicum opgave 2 zul je zien dat C++ het mogelijk maakt om op een veel betere manier een tijdsduur te definiëren (als abstract data type).

1.17 C++ als een betere C

Als we de verbeteringen die in C zijn doorgevoerd bij de definitie van C++ toepassen in het voorbeeld programma van pagina 11, dan ontstaat het C++ programma C.cpp:

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

struct Tijdsduur { // Een Tijdsduur bestaat uit:
    int uur;       //    een aantal uren en
    int minuten;   //    een aantal minuten.
};

// Deze functie drukt een Tijdsduur af
void drukaf(Tijdsduur td) {
    if (td.uur == 0)
        cout<<"           ";
    else
        cout << setw(3) << td.uur << " uur en ";
    cout << setw(2) << td.minuten << " minuten" << endl;
}
```

```
// Deze functie drukt een rij met een aantal gewerkte ↩
 ↪ tijden af
void drukaf(Tijdsduur rij[], int aantal) {
    for (int teller = 0; teller < aantal; ++teller)
        drukaf(rij[teller]);
}

// Deze functie berekent de totaal gewerkte tijd uit een ↩
 ↪ rij met een aantal gewerkte tijden
Tijdsduur som(Tijdsduur rij[], int aantal) {
    Tijdsduur s = {0, 0};
    for (int teller = 0; teller < aantal; ++teller) {
        s.uur += rij[teller].uur;
        s.minuten += rij[teller].minuten;
    }
    s.uur += s.minuten / 60;
    s.minuten %= 60;
    return s;
}

int main() {
    constexpr int MAX = 5;
    Tijdsduur gewerkteTijdenRij[MAX];
    int aantal = 0;
    do {
        cout << "Type gewerkte uren en minuten in (of ↩
         ↪ Ctrl-Z): ";
        cin >> gewerkteTijdenRij[aantal].uur >> ↩
         ↪ gewerkteTijdenRij[aantal].minuten;
    }
    while (cin && ++aantal < MAX);
    cout << endl << endl;
    drukaf(gewerkteTijdenRij, aantal);
    cout << "De totaal gewerkte tijd is:" << endl;
    drukaf(som(gewerkteTijdenRij, aantal));
    cin.get(); cin.clear(); cin.get();
    return 0;
```

```
}
```

Verklaring van de verschillen:

- Hier is gebruik gemaakt van de standaard I/O library van C++ in plaats van de standaard C I/O library. In de eerste regel wordt de file `iostream` geïnclude. Dit is nodig om gebruik te kunnen maken van functies, objecten en types die in de standaard C++ I/O library zijn opgenomen. In dit programma maak ik gebruik van `cout` (om te schrijven naar het scherm), `endl` (het einde regel teken) en `cin` (om te lezen vanaf het toetsenbord). Om gebruik te kunnen maken van een zogenoemde I/O manipulator wordt in de tweede regel de file `iomanip` geïnclude. In dit programma maak ik gebruik van de manipulator `setw()` waarmee de breedte van een uitvoerveld gespecificeerd kan worden. In de derde regel wordt aangegeven dat de namespace `std` wordt gebuikt. Dit is nodig om de functies, types enz. die in de bovenstaande 2 include files zijn gedefinieerd te kunnen gebruiken zonder er steeds `std::` voor te zetten.
- Voor commentaar aan het einde van de regel is hier `//...` gebruikt in plaats van `/*...*/`.
- De naam van de `struct Tijdsuur` kan meteen als typenaam gebruikt worden. Een `typedef` is hier dus niet nodig.
- Om de datavelden van een tijdsduur af te drukken is hier gebruik gemaakt van `cout` in plaats van `printf`. Merk op dat nu het type van de datavelden *niet* opgegeven hoeft te worden. Bij het gebruik van `printf` werd het type van de datavelden in de format string met `%d` aangegeven. De breedte van het uitvoerveld wordt nu met de I/O manipulator `setw` opgegeven in plaats van in de format string van `printf`.
- De functie om een rij af te drukken heeft hier de naam `drukaf` in plaats van `drukafRij`. De functie om een tijdsduur af te drukken heet ook al `drukaf`, maar in C++ is dit geen probleem. Ik heb hier dus function name overloading toegepast.

- De lokale variabele `teller` is hier in het `for` statement gedefinieerd in plaats van op een aparte regel.
- De compile time constante `MAX` is hier als `constexpr`[20] `int` gedefinieerd in plaats van met `#define`.
- Het inlezen vanaf het toetsenbord is hier gedaan met behulp van `cin` in plaats van `scanf`. Het object `cin` kan gebruikt worden in een conditie om te testen of de vorige leesbewerking gelukt is. De variabele `gelezen` is dus niet meer nodig.
- Als het inlezen op de een of andere manier niet gelukt is, moet het object `cin` gereset worden met behulp van de aanroep `cin.clear();`.

Merk op dat dit programma niet meer dan `MAX` gewerkte tijden kan verwerken. Later in dit dictaat zul je leren hoe dit probleem is op te lossen door gebruik te maken van het type `vector`, zie paragraaf 3.8.

Merk op dat de vernieuwingen die in C++ zijn ingevoerd ten opzichte van C, namelijk het gebruik van abstracte datatypes en objectgeoriënteerde technieken, in dit programma nog *niet* toegepast zijn. In dit programma wordt C++ dus op een C manier gebruikt. Dit is voor kleine programma's geen probleem. Als een programma echter groter is of als het uitbreidbaar of onderhoudbaar moet zijn, kun je beter gebruik maken van de objectgeoriënteerde technieken die C++ biedt.

[20] Het C++11 keyword `constexpr` wordt helaas nog niet ondersteund door Microsoft Visual Studio 2013. Als je een compiler gebruikt die het keyword `constexpr` nog niet ondersteund, dan kun je `constexpr` vervangen door `const`.

2

Objects and classes

In dit hoofdstuk worden de belangrijkste taalconstructies die C++ biedt voor het programmeren van ADT's (*Abstract Data Types*) besproken. Een ADT is een *user-defined* type dat voor een gebruiker niet te onderscheiden is van ingebouwde types (zoals `int`).

2.1 Object Oriented Design (OOD) and Object Oriented Programming (OOP)

In deze paragraaf zullen we eerst ingaan op de achterliggende gedachten van OOD en OOP en pas daarna de implementatie in C++ bespreken. Een van de specifieke problemen bij het ontwerpen van software is dat het nooit echt af is. Door het gebruik van de software ontstaan bij de gebruikers weer ideeën voor uitbreidingen en/of veranderingen. Ook de veranderende omgeving van de software (bijvoorbeeld: operating system en hardware) zorgen ervoor dat software vaak uitgebreid en/of veranderd moet worden. Dit aanpassen van bestaande software wordt *software maintenance* genoemd. Er werken momenteel meer software engineers aan het onderhouden van bestaande software dan aan het ontwikkelen van nieuwe applicaties. Ook is het bij het ontwerpen van (grote) applicaties gebrui-

kelijk dat de klant zijn specificaties halverwege het project aanpast. Je moet er bij het ontwerpen van software dus al op bedacht zijn dat deze software eenvoudig aangepast en uitgebreid kan worden (design for change).

Bij het ontwerpen van hardware is het vanzelfsprekend om gebruik te maken van (vaak zeer complexe) *componenten*. Deze componenten zijn door anderen geproduceerd en je moet er dus voor betalen, maar dat is geen probleem want je kunt het zelf niet (of niet goedkoper) maken. Bij het gebruik zijn alleen de specificaties van de component van belang, de interne werking (implementatie) van de component is voor de gebruiker van de component niet van belang. Het opbouwen van hardware met bestaande componenten maakt het ontwerpen en testen eenvoudiger en goedkoper. Als bovendien voor standaard interfaces wordt gekozen, kan de hardware ook aanpasbaar en uitbreidbaar zijn (denk bijvoorbeeld aan een PC met PCI Express bus). Bij het ontwerpen van software was het, voor het ontstaan van OOP, niet gebruikelijk om gebruik te maken van componenten die door anderen ontwikkeld zijn (en waarvoor je dus moet betalen). Vaak werden wel bestaande datastructuren en algoritmen gekopieerd maar die moesten vaak toch aangepast worden. Het gebruik van functie libraries was wel gebruikelijk maar dit zijn in feite eenvoudige componenten. Een voorbeeld van een complexe component is bijvoorbeeld een editor (inclusief menu opties: openen, opslaan, opslaan als, print, printer set-up, bewerken (knippen, kopiëren, plakken), zoeken en zoek&vervang en inclusief een knoppenbalk). In alle applicaties waarbij je tekst moet kunnen invoeren kun je deze editor dan hergebruiken. Ondanks dat het gebruik van softwarecomponenten, die we kunnen kopen en waarmee we vervolgens zelf applicaties kunnen ontwikkelen, sinds de introductie van OOP is toegenomen blijft het gebruik van softwarecomponenten toch achter bij het gebruik van hardwarecomponenten. Dit heeft volgens mij verschillende redenen:

- Voor een softwarecomponent moet je net als voor een hardwarecomponent betalen. Bij veel hardwarecomponenten komt het niet in je op om ze zelf te maken (een videokaart bijvoorbeeld), maar een softwarecomponent kun je ook altijd zelf (proberen te) maken. Zo hoor je bijvoorbeeld soms zeggen: “Maar dat algoritme staat toch gewoon in het boek van . . . dus dat kunnen we toch zelf wel implementeren”.

- Als je van een bepaalde hardwarecomponent 1000 stuks gebruikt, moet je er ook 1000 maal voor betalen. Bij een softwarecomponent vindt men dit minder normaal en er valt eenvoudiger mee te sjoemelen.
- Programmeurs denken vaak: maar dat kan ik zelf toch veel eenvoudiger en sneller programmeren. Daarbij overschatten ze vaak hun eigen kunnen en onderschatten ze de tijd die nodig is om een component grondig te testen.
- De omgeving (taal, operating system, hardware enz.) van software is divers. Dit maakt het produceren van softwarecomponenten niet eenvoudig. "Heeft u deze component ook in C++ voor Linux in plaats van in C# voor Windows?"[21]

De objectgeoriënteerde benadering is vooral bedoeld om het ontwikkelen van herbruikbare softwarecomponenten mogelijk te maken. In dit dictaat zul je kennismaken met deze objectgeoriënteerde benadering. Wij hebben daarbij gekozen voor de taal C++ omdat dit de meest gebruikte objectgeoriënteerde programmeertaal is voor embedded systems[22]. Het is echter niet de bedoeling dat je na het bestuderen van dit dictaat op de hoogte bent van alle aspecten en details van C++. Wel zul je na het bestuderen van dit dictaat de algemene ideeën achter OOP begrijpen en die toe kunnen passen in C++. De objectgeoriënteerde benadering is een (voor jou) nieuwe manier van denken over hoe je informatie in een computer kunt structureren.

De manier waarop we problemen oplossen met objectgeoriënteerde technieken lijkt op de manier van probleem oplossen die we in het dagelijks leven gebruiken. Een objectgeoriënteerde applicatie is opgebouwd uit deeltjes die we *objecten* noemen. Elk object heeft zijn eigen taak. Een object kan aan een ander object vragen of dit object wat voor hem wil doen, dit gebeurt door het zenden van een boodschap (Engels: *message*) naar dat object. Aan deze message kunnen indien

[21] Er zijn diverse alternatieven ontwikkeld voor het maken van taalonafhankelijke componenten. Microsoft heeft voor dit doel de .NET architectuur ontwikkeld. Deze .NET componenten kunnen in diverse talen gebruikt worden maar zijn wel gebonden aan het Windows platform. De CORBA (Common Object Request Broker Architecture) standaard van de OMG (Object Management Group) maakt het ontwikkelen van taal en platform onafhankelijke componenten mogelijk.

[22] Bron: UBM Tech 2014 Embedded Market Study.

nodig *argumenten* worden meegegeven. Het is de verantwoordelijkheid van het ontvangende object (Engels: the *receiver*) hoe het de message afhandelt. Zo kan dit object de verantwoordelijkheid afschuiven door zelf ook weer een message te versturen naar een ander object. Het algoritme dat de ontvanger gebruikt om de message af te handelen wordt de methode (Engels: *method*) genoemd. Het object dat de message verstuurt is dus niet op de hoogte van de door de ontvanger gebruikte method. Dit wordt *information hiding* genoemd. Het versturen van een message lijkt in eerste instantie op het aanroepen van een functie. Toch zijn er enkele belangrijke verschillen:

- Een message heeft een bepaalde receiver.
- De method die bij de message hoort is afhankelijk van de receiver.
- De receiver van een message kan ook tijdens run time worden bepaald. (Engels: *Late binding between the message (function name) and the method (code)*).

In een programma kunnen zich meerdere objecten van een bepaald object type bevinden, vergelijkbaar met meerdere variabelen van een bepaald variabele type. Zo'n object type wordt klasse (Engels: *class*) genoemd. Elk object behoort tot een bepaalde class, een object wordt ook wel een instantie (Engels: *instance*) van de class genoemd. Bij het definiëren van een nieuwe class hoef je niet vanuit het niets te beginnen maar je kunt deze nieuwe class afleiden (laten overerven) van een al bestaande class. De class waarvan afgeleid wordt, wordt de basisklasse (Engels: *base class*) genoemd en de class die daarvan afgeleid wordt, wordt afgeleide klasse (Engels: *derived class*) genoemd. Als van een derived class weer nieuwe classes worden afgeleid, ontstaat een hiërarchisch netwerk van classes. Een derived class overerft alle eigenschappen van een base class (overerving = Engels: *inheritance*). Als een object een message ontvangt, wordt de bijbehorende method als volgt bepaald (Engels: *method binding*):

- zoek in de class van het receiver object;
- als daar geen method gevonden wordt zoek dan in de base class van de class van de receiver;

- als daar geen method gevonden wordt zoek dan in de base class van de base class van de class van de receiver;
- enz.

Een method in een base class kan dus worden geherdefinieerd (Engels: *overridden*) door een method in de derived class. Naast de hierboven beschreven vorm van hergebruik (inheritance) kunnen objecten ook als onderdeel van een groter object gebruikt worden (compositie = Engels: *composition*). Inheritance wordt ook wel generalisatie (Engels: *generalization*) genoemd en leidt tot een zogenoemde "*is een*" relatie. Je kunt een class `Bakker` bijvoorbeeld afleiden door middel van overerving van een class `Winkelier`. Dit betekent dan dat de class `Bakker` minimaal dezelfde messages ondersteunt als de class `Winkelier` (maar de class `Bakker` kan er wel zijn eigen methods aan koppelen!). We zeggen: "Een `Bakker` is een `Winkelier`" of "Een `Bakker` erft over van `Winkelier`" of "Een `Winkelier` is een generalisatie van `Bakker`". Composition wordt ook wel *containment* of *aggregation* genoemd en leidt tot een zogenoemde "*heeft een*" relatie. Je kunt in de definitie van een class `Auto` vijf objecten van de class `Wiel` opnemen. De `Auto` kan deze `Wielen` dan gebruiken om de aan `Auto` gestuurde messages uit te voeren. We zeggen: "een `Auto` heeft `Wielen`". Welke relatie in een bepaald geval nodig is, is niet altijd eenvoudig te bepalen. We komen daar later nog uitgebreid op terug.

De bij objectgeoriënteerd programmeren gebruikte technieken komen niet uit de lucht vallen maar sluiten aan bij de historische evolutie van programmeertalen. Sinds de introductie van de computer zijn programmeertalen steeds *abstracter* geworden. De volgende abstractie technieken zijn achtereenvolgens ontstaan:

- **Functies en procedures.**
  In talen zoals Basic, Fortran, C, Pascal, Cobol enz. kan een stukjes logisch bij elkaar behorende code geabstraheerd worden tot een functie of procedure. Deze functies of procedures kunnen lokale variabelen bevatten die buiten de functie of procedure niet zichtbaar zijn. Deze lokale variabelen zitten ingekapseld in de functie of procedure, dit wordt *encapsulation* genoemd. Als de specificatie van de functie bekend is, kun je de functie gebruiken zonder dat je de implementatie details hoeft te kennen of te begrijpen. Dit is dus een

vorm van information hiding. Tevens voorkomt het gebruik van functies en procedures het dupliceren van code, waardoor het wijzigen en testen van programma's eenvoudiger wordt. De programmacode wordt korter, maar dit gaat ten koste van de executietijd.

- **Modules.**
  In programmeertalen zoals Modula 2 (en zeer primitief ook in C) kunnen een aantal logisch bij elkaar behorende functies, procedures en variabelen geabstraheerd worden tot een module. Deze functies, procedures en variabelen kunnen voor de gebruikers van de module zichtbaar (=public) of onzichtbaar (=private) gemaakt worden. Functies, procedures en variabelen die private zijn kunnen alleen door functies en procedures van de module zelf gebruikt worden. Deze private functies, procedures en variabelen zitten ingekapseld in de module. Als de specificatie van de public delen van de module bekend is, kun je de module gebruiken zonder dat je de implementatie details hoeft te kennen of te begrijpen. Dit is dus een vorm van data en information hiding.
- **Abstract data types (ADT's).**
  In programmeertalen zoals Ada kan een bepaalde datastructuur met de bij deze datastructuur behorende functies en procedures ingekapseld worden in een ADT. Het verschil tussen een module en een ADT is dat een module alleen gebruikt kan worden en dat een ADT geïnstantieerd kan worden. Dit wil zeggen dat er meerdere variabelen van dit ADT (=type) aangemaakt kunnen worden. Als de specificatie van het ADT bekend is, kun je dit type op dezelfde wijze gebruiken als de ingebouwde types van de programmeertaal (zoals `char`, `int` en `double`) zonder dat je de implementatie details van dit type hoeft te kennen of te begrijpen. Dit is dus weer een vorm van information hiding. Op het begrip ADT komen we nog uitgebreid terug.
- **Generieke functies en datatypes.**
  In C++ kunnen door het gebruik van *templates* generieke functies en datatypes gedefinieerd worden. Een generieke functie is een functie die gebruikt kan worden voor meerdere parametertypes. In C kan om een array met `int`'s te sorteren een functie geschreven worden. Als we vervolgens een ar-

ray met `double`'s willen sorteren, moeten we een nieuwe functie definiëren. In C++ kunnen we met behulp van templates een *generieke functie* definiëren waarmee array's van elk willekeurig type gesorteerd kunnen worden. Een generiek type is een ADT dat met verschillende andere types gecombineerd kan worden. In C++ kunnen we bijvoorbeeld een ADT array definiëren waarin we variabelen van het type `int` kunnen opslaan. Door het gebruik van een template kunnen we een generiek ADT array definiëren waarmee we dan array's van elk willekeurig type kunnen gebruiken. Op generieke functies en datatypes komen we nog uitgebreid terug.

- **Classes.**
  In programmeertalen zoals SmallTalk en C++ kunnen een aantal logisch bij elkaar behorende ADT's van elkaar worden afgeleid door middel van inheritance. Door nu programma's te schrijven in termen van base classes (basisklassen) en de in deze base classes gedefinieerde messages in derived classes (afgeleide klassen) te implementeren kan je deze base classes gebruiken zonder dat je de implementatie van deze classes hoeft te kennen of te begrijpen en kun je eenvoudig van implementatie wisselen. Tevens kun je code die alleen messages aanroept van de base class zonder opnieuw te compileren hergebruiken voor alle direct of indirect van deze base class afgeleide classes. Dit wordt veelvormige of met een moeilijk woord polymorf (Engels: *polymorph*) genoemd. Dit laatste wordt mogelijk gemaakt door de message pas tijdens run time aan een bepaalde method te verbinden en wordt late binding (Engels: *late binding*) genoemd. Op deze begrippen komen we later nog uitgebreid terug.

Toen je leerde programmeren in C heb je geleerd hoe je vanuit een probleemstelling voor een programma de benodigde functies kan vinden. Deze methode bestond uit het opdelen van het probleem in deelproblemen, die op hun beurt weer werden opgedeeld in deelproblemen enz. net zo lang tot de oplossing eenvoudig werd. Deze methode heet functionele decompositie. De software ontwikkelmethoden die hiervan zijn afgeleid worden *structured analyse and design* genoemd. De bekende Yourdon methode is daar een voorbeeld van. Bij het gebruik van objectgeoriënteerde programmeertalen ontstaat al snel de vraag: Hoe vind ik uit-

gaande van de probleemstelling de in dit programma benodigde classes en het verband tussen die classes? Het antwoord op deze vraag wordt bepaald door de gebruikte OOA (Object Oriented Analyse) en OOD (Object Oriented Design) methode.[23]

2.2 ADT's (Abstract Data Types)

De eerste stap op weg naar het objectgeoriënteerde denken en programmeren is het leren programmeren met ADT's. Een ADT[24] is een user-defined type dat voor een gebruiker[25] niet te onderscheiden is van ingebouwde types (zoals `int`). Een ADT koppelt een bepaalde datastructuur (interne variabelen) met de bij deze datastructuur behorende bewerkingen (functies). Deze functies en variabelen kunnen voor de gebruikers van het ADT zichtbaar (*public*) of onzichtbaar (*private*) gemaakt worden. Functies en variabelen die private zijn kunnen alleen door functies van het ADT zelf gebruikt worden. De private functies en variabelen zitten ingekapseld in het ADT. Als de specificatie van het ADT bekend is, kun je dit type op dezelfde wijze gebruiken als de ingebouwde types van de programmeertaal (zoals `char`, `int` en `double`) zonder dat je de implementatie details van dit type hoeft te kennen of te begrijpen. Dit is een vorm van information hiding.

In C is het niet mogelijk om ADT's te definiëren. Als je in C een programma wilt schrijven waarbij je met breuken in plaats van met floating point getallen wilt

[23] Deze onderwerpen komen bij de opleiding Elektrotechniek later in het tweede deel van de OGOPRG lessen aan de orde. Bij de opleiding Technische Informatica zijn deze onderwerpen al aan de orde geweest en komen ze nog uitgebreid bij verschillende andere vakken aan de orde.

[24] De naam ADT wordt ook vaak gebruikt voor een formele wiskundige beschrijving van een type. Ik gebruik het begrip ADT hier alleen in de betekenis van: zelfgedefinieerd (Engels: user-defined) type.

[25] Met gebruiker wordt hier de gebruiker van de programmeertaal bedoeld en niet de gebruiker van het uiteindelijke programma.

werken[26], dan kun je dit (niet in C opgenomen) type `Breuk` alleen maar op de volgende manier zelf definiëren (Breukc.c):

```
typedef struct { /* een breuk bestaat uit: */
    int boven;   /*      een teller en      */
    int onder;   /*      een noemer         */
} Breuk;
```

Een C functie om twee breuken op te tellen kan dan als volgt gedefinieerd worden:

```
Breuk som(Breuk b1, Breuk b2) {
    Breuk s;
    s.boven = b1.boven * b2.onder + b1.onder * b2.boven;
    s.onder = b1.onder * b2.onder;
    return normaliseer(s);
}
```

Het normaliseren[27] van de breuk zorgt ervoor dat ³⁄₈ + ¹⁄₈ als resultaat ½ in plaats van ⁴⁄₈ heeft. Dit zorgt ervoor dat een overflow minder snel optreedt als met het resultaat weer verder wordt gerekend.

Deze manier van werken heeft de volgende nadelen:

- Iedere programmeur die gebruikt maakt van het type `Breuk` kan een waarde toekennen aan de datavelden `boven` en `onder`. Het is in C niet te voorkomen dat het dataveld `onder` op nul gezet wordt. Als een programmeur het dataveld `onder` van een `Breuk` op nul zet, kan dit een fout veroorzaken in code van een andere programmeur die een `Breuk` naar een `double` converteert. Als deze fout geconstateerd wordt, kunnen alle functies waarin breuken gebruikt worden de boosdoeners zijn.

[26] Een belangrijke reden om te werken met breuken in plaats van floating point getallen is het voorkomen van afrondingsproblemen. Een breuk zoals ⅓ moet als floating point getal afgerond worden tot bijvoorbeeld: 0.333333333. Deze afronding kan ervoor zorgen dat een berekening zoals 3 × ⅓ niet zoals verwacht de waarde 1 maar de waarde 0.999999999 oplevert. Ook breuken zoals ¹⁄₁₀ moeten afgerond worden als ze als binair floating point getal worden weergegeven.

[27] Het algoritme voor het normaliseren van een breuk wordt verderop in dit dictaat besproken.

- Iedere programmeur die gebruikt maakt van het type `Breuk` kan er voor kiezen om zelf de code voor het optellen van breuken "uit te vinden" in plaats van gebruik te maken van de functie `som`. Er valt dus niet te garanderen dat alle optellingen van breuken correct en genormaliseerd zullen zijn. Ook niet als we wel kunnen garanderen dat de functies `som` en `normaliseer` correct zijn. Iedere programmeur die breuken gebruikt kan ze namelijk op zijn eigen manier optellen.
- Iedere programmeur die gebruikt maakt van het type `Breuk` zal zelf nieuwe bewerkingen (zoals bijvoorbeeld het vermenigvuldigen van breuken) definiëren. Het zou beter zijn als alleen de programmeur die verantwoordelijk is voor het onderhouden van het type `Breuk` (en de bijbehorende bewerkingen) dit kan doen.

Deze nadelen komen voort uit het feit dat in C de definitie van het type `Breuk` niet gekoppeld is aan de bewerkingen die op dit type uitgevoerd kunnen worden. Tevens is het niet mogelijk om bepaalde datavelden en/of bewerkingen ontoegankelijk te maken voor programmeurs die dit type gebruiken.

In C++ kunnen we door gebruik te maken van een *class* een eigen type `Breuk` als volgt declareren (Breuk0zonderRef.cpp):

```
class Breuk {              // Op een object van de class Breuk
public:                    // kun je de volgende bewerkingen
                           // uitvoeren:
    void leesin();         // -  inlezen vanuit het toetsen-
                           //    bord;
    void drukaf() const[28]; // -  afdrukken op het scherm;
    void plus(Breuk[29] b);  // -  een Breuk erbij optellen.
private:                   // Een object van de class Breuk
                           // heeft privé:
    int boven;             // -  een teller;
    int onder;             // -  een noemer;
    void normaliseer();    // -  een functie normaliseer.
};
```

De class `Breuk` koppelt een bepaalde datastructuur (twee interne integer variabelen genaamd `boven` en `onder`) met de bij deze datastructuur behorende bewerkingen[30] (functies: `leesin`, `drukaf`, `plus` en `normaliseer`). Deze functies en variabelen kunnen voor de gebruikers van de class `Breuk` zichtbaar (`public`) of onzichtbaar (`private`) gemaakt worden. Functies en variabelen die private zijn kunnen alleen door functies van de class `Breuk` zelf gebruikt worden.[31] Deze private functies en variabelen zitten ingekapseld in de class `Breuk`.

Functies die opgenomen zijn in een class worden *memberfuncties* genoemd. De memberfunctie `plus` kan als volgt gedefinieerd worden[32]:

```
void Breuk::plus(const Breuk& b) {
    boven = boven * b.onder + onder * b.boven;
    onder *= b.onder;
    normaliseer();
}
```

Je kunt objecten (variabelen) van de class (het zelfgedefinieerde type) `Breuk` nu als volgt gebruiken:

```
    Breuk a, b; // definieer de objecten a en b van de ↩
     ↪ class Breuk
    a.leesin(); // lees a in
    b.leesin(); // lees b in
    a.plus(b);  // tel b bij a op
```

[28] `const` memberfunctie wordt pas behandeld in paragraaf 2.9.

[29] Het is beter om hier een `const` reference parameter te gebruiken. Dit wordt pas behandeld in paragraaf 2.16.

[30] De verzameling van public functies wordt ook wel de *interface* van de class genoemd. Deze interface definieert hoe je objecten van deze class kunt gebruiken (welke memberfuncties je op de objecten van deze class kan aanroepen).

[31] Variabelen die private zijn kunnen door het gebruik van een pointer en bepaalde type-conversies toch door andere functies benaderd worden. Ze staan namelijk gewoon in het werkgeheugen en ze zijn dus altijd via software toegankelijk. Het gebruik van private variabelen is alleen bedoeld om “onbewust” verkeerd gebruik tegen te gaan.

[32] De definitie van de memberfuncties `leesin`, `drukaf` en `normaliseer` is hier niet gegeven.

```
a.drukaf(); // druk a af
```

Een object heeft drie kenmerken:

- **Geheugen** (Engels: *state*);
  Een object kan "iets" onthouden. Wat een object kan onthouden blijkt uit de classdeclaratie. Het object `a` is een instantie van de class `Breuk` en kan, zoals uit de classdeclaratie van `Breuk` blijkt, twee integers onthouden. Elk object heeft (vanzelfsprekend) zijn *eigen* geheugen. Zodat de `Breuk` `a` een andere waarde kan hebben dan de `Breuk` `b`.
- **Gedrag** (Engels: *behaviour*);
  Een object kan "iets" doen. Wat een object kan doen blijkt uit de classdeclaratie. Het object `a` is een instantie van de class `Breuk` en kan, zoals uit de declaratie van `Breuk` blijkt, 4 dingen doen: zichzelf inlezen, zichzelf afdrukken, een `Breuk` bij zichzelf optellen en zichzelf normaliseren. Alle objecten van de class `Breuk` hebben hetzelfde gedrag. Dit betekent dat de code van de memberfuncties *gezamenlijk* gebruikt wordt door alle objecten van de class `Breuk`.
- **Identiteit** (Engels: *identity*).
  Om objecten met dezelfde waarde toch uit elkaar te kunnen houden heeft elk object een eigen identiteit. De twee objecten van de class `Breuk` in het voorgaande voorbeeld hebben bijvoorbeeld elk een eigen naam (`a` en `b`) waarmee ze geïdentificeerd kunnen worden.

Bij de memberfunctie `plus` wordt bij de definitie met de zogenoemde qualifier `Breuk::` aangegeven dat deze functie een member is van de class `Breuk`. De memberfunctie `plus` kan alleen op een object van de class `Breuk` uitgevoerd worden. Dit wordt genoteerd als: `a.plus(b);` Het object `a` wordt de *receiver* (ontvanger) van de memberfunctie genoemd. Als in de definitie van de memberfunctie `plus` de datavelden `boven` en `onder` gebruikt worden, dan zijn dit de datavelden van de receiver (in dit geval object `a`). Als in de definitie van de memberfunctie `plus` de memberfunctie `normaliseer` gebruikt wordt, dan wordt deze memberfunctie op

de receiver uitgevoerd (in dit geval object a). De betekenis van `const` achter de declaratie van de memberfunctie `drukaf` komt later (paragraaf 2.9) aan de orde.

Deze manier van werken heeft de volgende voordelen:

- Een programmeur die gebruik maakt van de class (het type) `Breuk` kan *géén* waarde toekennen aan de private datavelden `boven` en `onder`. Alleen de memberfuncties `leesin`, `drukaf`, `plus` en `normaliseer` kunnen een waarde toekennen aan deze datavelden. Als ergens een fout ontstaat omdat het dataveld `onder` van een `Breuk` op nul is gezet, kunnen alleen de memberfuncties van `Breuk` de boosdoeners zijn.
- Een programmeur die gebruik maakt van de class `Breuk` kan er *niet* voor kiezen om zelf de code voor het optellen van breuken "uit te vinden" in plaats van gebruik te maken van de memberfunctie `plus`. Als we kunnen garanderen dat de functies `plus` en `normaliseer` correct zijn, dan kunnen we dus garanderen dat alle optellingen van breuken correct en genormaliseerd zullen zijn. Iedere programmeur die breuken gebruikt kan ze namelijk alleen optellen door gebruik te maken van de memberfunctie `plus`.
- Iedere programmeur die gebruikt maakt van de class `Breuk` zal *niet* zelf nieuwe bewerkingen (zoals bijvoorbeeld het vermenigvuldigen van breuken) definiëren. Alleen de programmeur die verantwoordelijk is voor het onderhouden van de class `Breuk` (en de bijbehorende bewerkingen) kan dit doen.

Bij het ontwikkelen van kleine programma's zijn deze voordelen niet zo belangrijk maar bij het ontwikkelen van grote programma's zijn deze voordelen wel erg belangrijk. Door gebruik te maken van de `class` `Breuk` met bijbehorende memberfuncties in plaats van de `typedef` `Breuk` met de bijbehorende functies wordt het programma *beter onderhoudbaar* en *eenvoudiger uitbreidbaar*.

De hierboven gedefinieerde class `Breuk` is erg beperkt. In de inleiding van dit hoofdstuk heb ik geschreven: "Een ADT is een user-defined type dat voor een gebruiker niet te onderscheiden is van ingebouwde types (zoals `int`)". Een ADT `Breuk` zal dus als volgt gebruikt moeten kunnen worden:

```
int main() {
    Breuk b1, b2;               // definiëren van variabelen
    cout << "Geef Breuk: ";
    cin >> b1;                  // inlezen met >>
    cout << "Geef nog een Breuk: ";
    cin >> b2;                  // inlezen met >>
    cout << b1 << "+"           // afdrukken met <<
         << b2 << "="
         << (b1 + b2) << endl; // optellen met +
    Breuk b3(18, -9);           // definiëren en initialiseren
    if (b1 != b3) {             // vergelijken met !=
        b3++;                   // verhogen met ++
    }
    cout << b3 << endl;         // afdrukken met <<
    b3 += 5;                    // verhogen met +=
    cout << b3 << endl;         // afdrukken met <<
    if (-2 == b3) {             // vergelijken met een int
        cout << "OK." << endl;
    }
```

Je ziet dat het zelfgedefinieerde type `Breuk` nu op precies dezelfde wijze als het ingebouwde type `int` te gebruiken is. Dit maakt het voor programmeurs die het type `Breuk` gebruiken erg eenvoudig. In het vervolg zal ik bespreken hoe de class `Breuk` stap voor stap uitgebreid en aangepast kan worden zodat uiteindelijk een ADT `Breuk` ontstaat. Dit blijkt nog behoorlijk lastig te zijn en je zou jezelf af kunnen vragen: “Is het al die inspanningen wel waard om een `Breuk` zo te maken dat je hem net zoals een `int` kan gebruiken?” Bedenk dan het volgende: het maken van de class `Breuk` is dan wel een hele inspanning maar iedere programmeur kan vervolgens dit zelfgedefinieerde type, als hij of zij de naam `Breuk` maar kent, als vanzelf (intuïtief) gebruiken. Er is daarbij geen handleiding of helpfunctie nodig, omdat het type `Breuk` zich net zo gedraagt als het ingebouwde type `int`. De class `Breuk` hoeft maar één keer gemaakt te worden, maar zal talloze malen gebruikt worden. Met een beetje geluk zul je als ontwerper van de class `Breuk` later zelf ook gebruiker van de class `Breuk` zijn zodat je zelf de vruchten van je inspanningen kunt plukken.

2.3 Voorbeeld class Breuk (eerste versie)

Dit voorbeeld is een eerste uitbreiding van de op pagina 45 gegeven class Breuk (versie 0). In deze versie zul je leren:

- hoe je een object van de class Breuk kunt initialiseren (door middel van *constructors*);
- hoe je memberfuncties kunt definiëren die ook voor constante objecten van de class Breuk gebruikt kunnen worden;
- hoe je automatische conversie van type X naar het type Breuk kunt laten plaatsvinden (door middel van constructors);
- wat je moet doen om objecten van de class Breuk te kunnen toekennen en kopiëren (niets!).

Ik zal nu eerst de complete broncode presenteren van het programma Breuk1.cpp waarin het type Breuk gedeclareerd, geïmplementeerd en gebruikt wordt. Meteen daarna zal ik één voor één op de bovengenoemde punten ingaan.

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

// Classdeclaratie:

class Breuk {
public:
    // Classinterface. Vertelt:
    // - hoe je een object van deze class kunt maken;
    // - wat je aan een object van deze class kunt vragen;
    // - wat je met een object van de class kunt doen.
    // Constructors zie paragraaf 2.4.
    Breuk();
    Breuk(int t);
    Breuk(int t, int n);
    // Vraag memberfuncties zie pagina 62.
```

```
    int teller() const; // const memberfunctie zie ↩
     ↪ paragraaf 2.9.
    int noemer() const;
    // Doe memberfuncties zie pagina 63.
    void plus(Breuk[33] b);
    void abs();
    // ...
    // Er zijn nog veel uitbreidingen mogelijk.
    // ...
private:
    // Classimplementatie. Vertelt:
    // - wat je nodig hebt om een object van de class te ↩
     ↪ maken.
    // Dit is voor gebruikers van de class niet van belang.
    int boven;
    int onder;
    void normaliseer();
};

// Hulpfunctie: bepaald de grootste gemene deler.

int ggd(int n, int m) {
    if (n == 0) return m;
    if (m == 0) return n;
    if (n < 0) n = -n;
    if (m < 0) m = -m;
    while (m != n)
        if (n > m) n -= m;
        else m -= n;
    return n;
}

// Classdefinitie:
// Vertelt hoe de memberfuncties van de class ↩
 ↪ geïmplenteerd zijn.
// Dit is voor gebruikers van de class niet van belang.
```

```
Breuk::Breuk(): boven(0), onder(1) {
}

Breuk::Breuk(int t): boven(t), onder(1) {
}

Breuk::Breuk(int t, int n): boven(t), onder(n) {
    normaliseer();
}

int Breuk::teller() const {
    return boven;
}

int Breuk::noemer() const {
    return onder;
}

void Breuk::plus(Breuk b) {
    boven = boven * b.onder + onder * b.boven;
    onder *= b.onder;
    normaliseer();
}

void Breuk::abs() {
    if (boven < 0) boven = -boven;
}

void Breuk::normaliseer() {
    assert(onder != 0)[34];
    if (onder < 0) {
        onder = -onder;
        boven = -boven;
    }
    int d(ggd(boven, onder));
    boven /= d;
    onder /= d;
```

```
}

// Hoofdprogramma:

int main() {
    Breuk b1(4);
    cout << "b1(4) = " << b1.teller() << '/' << ↩
     ↪ b1.noemer() << endl;
    Breuk b2(23, -5);
    cout << "b2(23, -5) = " << b2.teller() << '/' << ↩
     ↪ b2.noemer() << endl;
    Breuk b3(b2); // kan dit zomaar? Zie paragraaf 2.7.
    cout << "b3(b2) = " << b3.teller() << '/' << ↩
     ↪ b3.noemer() << endl;
    b3.abs();
    cout << "b3.abs() = " << b3.teller() << '/' << ↩
     ↪ b3.noemer() << endl;
    b3 = b2; // kan dit zomaar? Zie paragraaf 2.8.
    cout << "b3 = b2 = " << b3.teller() << '/' << ↩
     ↪ b3.noemer() << endl;
    b3.plus(5);
    cout << "b3.plus(5) = " << b3.teller() << '/' << ↩
     ↪ b3.noemer() << endl;
    cin.get();
    return 0;
}
```

Uitvoer:

```
b1(4) = 4/1
b2(23, -5) = -23/5
```

[33] Het is beter om hier een `const` reference parameter te gebruiken. Dit wordt pas behandeld op paragraaf 2.16.

[34] De standaard functie `assert` doet niets als de, als argument meegegeven, expressie `true` oplevert maar breekt het programma met een passende foutmelding af als dit niet zo is. Je kunt zogenoemde *assertions* gebruiken om tijdens de ontwikkeling van het programma te controleren of aan bepaalde voorwaarden (waarvan je "zeker" weet dat ze geldig zijn) wordt voldaan.

```
b3(b2) = -23/5
b3.abs() = 23/5
b3 = b2 = -23/5
b3.plus(5) = 2/5
```

2.4 Constructor Breuk

De constructors definiëren hoe een object gemaakt kan worden. De constructors hebben dezelfde naam als de class. Voor de class Breuk heb ik de volgende drie constructors gedeclareerd:

```
Breuk();
Breuk(int t);
Breuk(int t, int n);
```

Als een Breuk bij het aanmaken niet geïnitialiseerd wordt, dan wordt de constructor zonder parameters aangeroepen. In de definitie van deze constructor worden dan de datavelden boven met 0 en onder met 1 geïnitialiseerd. Dit gebeurt in een zogenoemde *initialization list*. Na het prototype van de constructor volgt een : waarna de datavelden één voor één geïnitialiseerd worden door middel van de (...) notatie.

```
Breuk::Breuk(): boven(0), onder(1) {
}
```

In dit geval wordt het dataveld boven geïnitialiseerd met de waarde 0 en wordt het dataveld onder geïnitialiseerd met de waarde 1, verder is er geen code in de constructor opgenomen. Door tussen de { en } bijvoorbeeld een output opdracht op te nemen zou je een melding op het scherm kunnen geven telkens als een Breuk aangemaakt wordt. Deze constructor wordt dus aangeroepen als je een Breuk aanmaakt zonder deze te initialiseren. Na het uitvoeren van de onderstaande code heeft de breuk b1 de waarde 0/1.

```
Breuk b1; // roep constructor zonder parameters aan
```

De overige constructors worden gebruikt als je een `Breuk` bij het aanmaken met één of met twee integers initialiseert. Na afloop van de onderstaande code zal het object `b2` de waarde ½ bevatten (⅜ wordt namelijk in de constructor genormaliseerd tot ½).

```
    Breuk b2(3,6); // roep \USE{constructor} met twee int ↩
     ↪ parameters aan
```

Deze constructor `Breuk(int t, int n);` heb ik als volgt gedefinieerd:

```
Breuk::Breuk(int t, int n): boven(t), onder(n) {
    normaliseer();
}
```

```
    Breuk b3{3,6}; // roep constructor met twee int ↩
     ↪ parameters aan
    Breuk b4 = {3,6}; // roep constructor met twee int ↩
     ↪ parameters aan
```

Sinds C++11[35] is het mogelijk om vanuit een constructor een andere constructor van dezelfde class aan te roepen in de initialization list. De constructor zonder parameters kan dus ook als volgt worden geïmplementeerd (Breuk1ConstructorCallsConstructor.cpp):

```
Breuk::Breuk(): Breuk(0, 1) {
}
```

Door het definiëren van constructors kun je er dus voor zorgen dat elk object bij het aanmaken geïnitialiseerd wordt. Fouten die voortkomen uit het gebruik van een niet geïnitialiseerde variabele worden hiermee voorkomen. Het is dus verstandig om voor elke class één of meer constructors te definiëren[36].

[35] Deze zogenoemde *delegating constructors* zijn beschikbaar in Microsoft Visual Studio vanaf versie 2012 en in GCC vanaf versie 4.7.

[36] Door het slim gebruik van default argumenten kun je het aantal constructors vaak beperken. Alle drie de constructors die ik voor de class `Breuk` gedefinieerd heb zouden door één constructor met default argumenten vervangen kunnen worden: `Breuk::Breuk(int t = 0, int n = 1);`.

Als (ondanks het zo juist gegeven advies) geen enkele constructor gedefinieerd is, dan wordt door de compiler een *default constructor* (= constructor zonder parameters) aangemaakt. Deze default constructor roept voor elk dataveld de default constructor van dit veld aan (Engels: *memberwise construction*).

2.5 Initialization list van de constructor

Het initialiseren van datavelden vanuit de constructor kan op twee manieren geprogrammeerd worden:

- door gebruik te maken van een initialisatielijst (*initialization list*);
- door gebruik te maken van assignments in het code blok van de constructor.

De eerste methode heeft de voorkeur, omdat dit altijd werkt. Een constante dataveld kan bijvoorbeeld niet met een assignment geïnitialiseerd worden.

Dus gebruik:

```
Breuk::Breuk(): boven(0), onder(1) {
}
```

in plaats van:

```
Breuk::Breuk() {
    boven = 0; // Let op: Het is beter om een
    onder = 1; // initialisatielijst te gebruiken!
}
```

2.6 Constructors en type conversies

Zie eventueel [2, blz. 561]: Constructor conversion

In het bovenstaande programma wordt, aan het einde van de functie `main`, de memberfunctie `plus` als volgt aangeroepen:

```
b3.plus(5);
```

Uit de uitvoer blijkt dat dit "gewoon" werkt: `b3` wordt verhoogd met 5/1. Op zich is dit vreemd want de memberfunctie `plus` is gedefinieerd met een `Breuk` als parameter en niet met een `int` als parameter. Als je de memberfunctie `plus` aanroept op een `Breuk` met een `int` als argument, gebeurt het volgende: eerst "kijkt" de compiler of in de class `Breuk` de memberfunctie `plus(int)` gedefinieerd is. Als dit het geval is, wordt deze memberfunctie aangeroepen. Als dit niet het geval is, "kijkt" de compiler of er in de class `Breuk` een memberfunctie is met een parameter van een ander type waarnaar het type `int` omgezet kan worden[37]. In dit geval "ziet" de compiler de memberfunctie `Breuk::plus(Breuk)`. De compiler "vraagt zich nu dus af" hoe een variabele van het type `int` omgezet kan worden naar het type `Breuk`. Of met andere woorden hoe een `Breuk` gemaakt kan worden en geïnitialiseerd kan worden met een `int`. Of met andere woorden of de constructor `Breuk(int)` bestaat. Deze constructor bestaat in dit geval. De compiler maakt nu een tijdelijke naamloze variabele aan van het type `Breuk` (door gebruik te maken van de constructor `Breuk(int)`) en geeft een kopietje van deze variabele door aan de memberfunctie `plus`. De memberfunctie `plus` telt de waarde van deze variabele op bij de receiver. Na afloop van de memberfunctie `plus` wordt de tijdelijke naamloze variabele weer vrijgegeven.

Als je dus een constructor `Breuk(int)` definieert, wordt het type `int` indien nodig automatisch omgezet naar het type `Breuk`. Of algemeen: als je een constructor `X(Y)` definieert, wordt het type `Y` indien nodig automatisch omgezet naar het type `X`[38].

2.7 Default copy constructor

Zie eventueel [2, blz. 493]: Default copy-constructor

[37] Als dit er meerdere zijn, zijn er allerlei regels in de C++ standaard opgenomen om te bepalen welke conversie gekozen wordt. Ik zal dat hier niet bespreken. Een conversie naar een ingebouwd type gaat altijd voor ten opzichte van een conversie naar een zelfgedefinieerd type.

[38] Als je dit niet wilt, kun je het keyword `explicit` voor de constructor plaatsen. De als explicit gedefinieerde constructor wordt dan niet meer automatisch (impliciet) voor type conversie gebruikt.

Een copy constructor wordt gebruikt als een object gekopieerd moet worden. Dit is het geval als:

- een object geïnitialiseerd wordt met een object van dezelfde class;
- een object als argument wordt doorgegeven aan een functie;[39]
- een object als waarde wordt teruggegeven vanuit een functie.[40]

De compiler zal als de programmeur geen copy constructor definieert zelf een *default copy constructor* genereren. Deze default copy constructor kopieert elk deel waaruit de class bestaat vanuit het originele object naar het gekopieerde object (Engels: *memberwise copy*). Het is ook mogelijk om zelf een copy constructor te definiëren (wordt later behandeld, zie eventueel paragraaf 5.7) maar voor de class `Breuk` voldoet de door de compiler gedefinieerde copy constructor prima.

In het programma op pagina 53 wordt de copy constructor van de class `Breuk` als volgt gebruikt:

```
Breuk b3(b2);
```

De door de compiler gegenereerde default copy constructor zal nu een object van de class `Breuk` aanmaken genaamd `b3` en de waarde van de private variabelen `boven` en `onder` uit de `Breuk` genaamd `b2` kopiëren naar `b3`.

De default copy constructor maakt een zogenoemde diepe kopie (Engels: deep copy).[41] Dit wil zeggen dat, als een private variabele zelf ook weer uit onderdelen bestaat, dat deze onderdelen dan ook gekopieerd worden. Stel dat we de class `Rechthoek` definiëren met twee private variabelen van het type `Breuk` genaamd `lengte` en `breedte`.

[39] In paragraaf 2.15 zul je zien dat je in C++ een argument aan een functie ook kunt meegeven zonder dat er een kopie wordt gemaakt.

[40] In paragraaf 2.19 zul je zien dat je in C++ in plaats van de waarde van een object ook het object zelf kunt teruggeven vanuit een functie. Er hoeft in dat geval geen kopie gemaakt te worden.

[41] Met een kopieerapparaat dat in staat is om een diepe kopie te maken, zouden we met één druk op de knop een heel boek kunnen kopiëren. Alle onderdelen waaruit het boek bestaat zouden dan ook automatisch gekopieerd worden. Helaas zijn kopieerapparaten op dit moment alleen nog maar in staat om een zogenoemde oppervlakkige kopie (Engels: shallow copy) te maken.

```
class Rechthoek {
public:
    // ...
private:
    Breuk lengte;
    Breuk breedte;
};
```

De default copy constructor van de class `Rechthoek` zal nu de private variabelen `lengte` en `breedte` kopiëren, die ieder voor zich weer uit private variabelen `boven` en `onder` bestaan, die ook automatisch gekopieerd worden.

Als er in de class `Rechthoek` geen enkele constructor gedefinieerd is dan wordt door de compiler een default constructor gedefinieerd, zie pagina 56. Deze default constructor zal de default constructor van de class `Breuk` aanroepen om de private variabelen `lengte` en `breedte` te initialiseren. Als we dus een object van de class `Rechthoek` aanmaken dan zorgt de default constructor ervoor dat beide private variabelen de waarde ½ krijgen.

2.8 Default assignment operator

Als geen assignment operator (=) gedefinieerd is, wordt door de compiler een *default assignment operator* aangemaakt. Deze default assignment operator roept voor elk dataveld de assignment operator van dit veld aan (Engels: *memberwise assignment*). Het is ook mogelijk om zelf een assignment operator te definiëren (wordt later behandeld, zie eventueel paragraaf 5.8) maar voor de class `Breuk` voldoet de door de compiler gedefinieerde assignment operator prima.

In het programma op pagina 53 wordt de assignment operator van de class `Breuk` als volgt gebruikt:

```
    b3 = b2;
```

De door de compiler gegenereerde default assignment operator zal nu de waarde van de private variabelen `boven` en `onder` uit de `Breuk` genaamd `b2` kopiëren naar de `Breuk` genaamd `b3`.

De default assignment operator zal er voor zorgen dat, als een private variabele zelf ook weer uit onderdelen bestaat, dat deze onderdelen dan ook gekopieerd worden. Stel dat we een object van de hierboven gegeven class `Rechthoek` toekennen aan een ander object van deze class:

```
Rechthoek r1, r2;
// ...
r2 = r1;
```

De default assignment operator van de class `Rechthoek` zal nu de private variabelen `lengte` en `breedte` van `r1` naar `r2` kopiëren. Beide variabelen bestaan voor zich weer uit de private variabelen `boven` en `onder`, die ook automatisch gekopieerd worden.

2.9 const memberfuncties

Zie eventueel [2, blz. 380]: const objects & member functions

Een object (variabele) van de class (het zelfgedefinieerde type) `Breuk` kan ook als read-only object gedefinieerd worden. Bijvoorbeeld:

```
Breuk b(1, 3);           // variabele b met waarde 1/3
const Breuk halve(1, 2); // constante halve met waarde ↩
 ↪ 1/2
```

Een const Breuk mag je (vanzelfsprekend) niet veranderen.

```
    halve = b;
//  Error: (Microsoft) binary '=' : no operator found which ↩
 ↪ operand of type 'const Breuk'
//  Error: (GCC)       no match for 'operator=' in 'halve = b'
```

Stel jezelf nu eens de vraag welke memberfuncties je aan mag roepen op het object halve[42]. De memberfuncties teller en noemer kunnen zonder problemen worden aangeroepen op een read-only breuk omdat ze de waarde van de breuk niet veranderen. De memberfuncties plus en abs mogen echter niet op het object halve worden aangeroepen omdat deze memberfuncties dit object zouden wijzigen en een read-only object mag je vanzelfsprekend niet veranderen. De compiler kan niet (altijd[43]) controleren of het aanroepen van een memberfunctie een verandering in het object tot gevolg heeft. Daarom mag je een memberfunctie alleen aanroepen voor een const object als je expliciet aanduidt dat deze memberfunctie het object niet verandert. Dit doe je door het keyword const achter de memberfunctie te plaatsten. Bijvoorbeeld:

```
    int teller() const;
```

Met deze const memberfunctie kun je de teller van een Breuk opvragen. De implementatie is erg eenvoudig:

```
int Breuk::teller() const {
    return boven;
}
```

Je kunt de memberfunctie teller nu dus ook aanroepen voor read-only breuken:

```
    const Breuk halve(1, 2); // constante halve met waarde ↩
     ↪ 1/2
```

[42] Zie pagina 50 voor de declaratie van de class Breuk.

[43] De compiler kan dit zeker niet als de classdefinitie separaat van de applicatie gecompileerd wordt. Zie paragraaf 2.29.

```
    cout << halve.teller() << endl;
```

Het aanroepen van de memberfunctie `plus` voor de `const` `Breuk` `halve` geeft echter een foutmelding en dat is precies de bedoeling want bij een read-only object moet je niets kunnen optellen:

```
    halve.plus(b);
//  Error: (Microsoft) 'Breuk::plus' : cannot convert 'this'[44] ↩
 ↪ pointer from 'const Breuk' to 'Breuk &'
//  Error: (GCC)       no matching function for call to ↩
 ↪ 'Breuk::plus(Breuk&) const'
```

Als je probeert om in een `const` memberfunctie toch de receiver te veranderen, krijg je de volgende foutmelding:

```
int Breuk::teller() const {
    boven = 1; // teller probeert vals te spelen
//  Error: (Microsoft) 'boven' cannot be modified because it ↩
 ↪ is being accessed through a const object
//  Error: (GCC)       assignment of member 'Breuk::boven' in ↩
 ↪ read-only object
```

Door het toepassen van function name overloading is het mogelijk om 2 functies te definiëren met dezelfde naam en met dezelfde parameters waarbij de ene `const` is en de andere niet, maar dit is erg gedetailleerd en zal ik hier niet behandelen[45].

We kunnen nu de memberfuncties van een class in twee groepen opdelen:

- **Vraag-functies.**
  Deze memberfuncties kunnen gebruikt worden om de toestand van een object op te vragen. Deze memberfuncties hebben over het algemeen wel een return type, geen parameters en zijn const memberfuncties.

[44] Deze foutmelding is niet erg helder. De `this` pointer wordt behandeld op paragraaf 2.13.

[45] Voor degene die echt alles wil weten: als een memberfunctie zowel const als non-const gedefinieerd is dan wordt bij aanroep op een read-only object de const memberfunctie aangeroepen en wordt bij aanroep op een variabel object de non-const memberfunctie aangeroepen. (Logisch nietwaar!)

- **Doe-functies.**
  Deze memberfuncties kunnen gebruikt worden om de toestand van een object te veranderen. Deze memberfuncties hebben over het algemeen geen return type (`void`), wel parameters en zijn non-const memberfuncties.

2.10 Class invariant

In de memberfuncties van de class `Breuk` wordt ervoor gezorgd dat elk object van de class `Breuk` een noemer > 0 heeft en de ggd (grootste gemene deler) van de teller en noemer 1 is. Door de noemer altijd > 0 te maken kan het teken van de `Breuk` eenvoudig bepaald worden (= teken van teller). Door de `Breuk` altijd te normaliseren wordt onnodige overflow van de datavelden `boven` en `onder` voorkomen. Een voorwaarde waaraan elk object van een class voldoet wordt een *class invariant* genoemd. Als je ervoor zorgt dat de class invariant aan het einde van elke public memberfunctie geldig is en als alle datavelden private zijn, dan weet je zeker dat voor elk object van de class `Breuk` deze invariant altijd geldig is.[46] Dit vermindert de kans op het maken van fouten.

2.11 Voorbeeld class `Breuk` (tweede versie)

Dit voorbeeld is een uitbreiding van de in paragraaf 2.3 gegeven class `Breuk` (versie 1). Met behulp van dit voorbeeld zul je leren hoe je een object van de class `Breuk` kunt optellen bij een ander object van de class `Breuk` met de operator `+=` in plaats van met de memberfunctie `plus` (door middel van *operator overloading*). Het optellen van een `Breuk` bij een `Breuk` gaat nu op precies dezelfde wijze als het optellen van een `int` bij een `int`. Dit maakt het type `Breuk` voor programmeurs erg eenvoudig te gebruiken. Ik zal nu eerst de complete broncode presenteren van het programma Breuk2.cpp waarin het type `Breuk` gedeclareerd, geïmple-

[46] Door het definiëren van invarianten (en pre- en postcondities) wordt het zelfs mogelijk om op een formele wiskundige manier te bewijzen dat een ADT correct is. Ik zal dit hier verder niet behandelen.

menteerd en gebruikt wordt. Meteen daarna zal ik op het bovengenoemde punt ingaan.

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

// Classdeclaratie.

class Breuk {
public:
    Breuk(int t, int n);
    int teller() const;
    int noemer() const;
    void operator+=(Breuk47 right);
private:
    int boven;
    int onder;
    void normaliseer();
};

//  ...

void Breuk::operator+=(Breuk right) {
    boven = boven * right.onder + onder * right.boven;
    onder *= right.onder;
    normaliseer();
}

int main() {
    Breuk b1(14, 4);
    cout << "b1(14, 4) = " << b1.teller() << '/' << ↩
     ↪ b1.noemer() << endl;
    Breuk b2(23, -5);
    cout << "b2(23, -5) = " << b2.teller() << '/' << ↩
     ↪ b2.noemer() << endl;
    b1 += b2;
```

```
    cout << "b1 += b2 = " << b1.teller() << '/' << ↩
     ↪ b1.noemer() << endl;
    cin.get();
    return 0;
}
```

Uitvoer:

```
b1(14, 4) = 7/2
b2(23, -5) = -23/5
b1 += b2 = -11/10
```

2.12 Operator overloading

Zie eventueel [2, blz. 511]: Operator overloading

In de taal C++ kun je de betekenis van operatoren (zoals bijvoorbeeld +=) definiëren voor zelfgedefinieerde types. Als het statement:

```
    b1 += b2;
```

vertaald moet worden, waarbij `b1` en `b2` objecten zijn van de class `Breuk`, zal de compiler "kijken" of in de class `Breuk` de `operator+=` memberfunctie gedefinieerd is. Als dit niet het geval is, levert het bovenstaande statement de volgende foutmelding op:

```
    b1 += b2;
//  Error: (Microsoft) binary '+=' : 'Breuk' does not define ↩
 ↪ this operator
//  Error: (GCC)       no match for 'operator+=' in 'b1 += b2'
```

Als de `Breuk::operator+=` memberfunctie wel gedefinieerd is, wordt het bovenstaande statement geïnterpreteerd als:

```
    b1.operator+=(b2);
```

[47] Het is beter om hier een `const` reference parameter te gebruiken. Dit wordt pas behandeld in paragraaf 2.16.

De memberfunctie `operator+=` in deze tweede versie van `Breuk` heeft dezelfde implementatie als de memberfunctie `plus` in de eerste versie van `Breuk` (zie pagina 46).

Je kunt voor een zelfgedefinieerd type alle operatoren zelf definiëren behalve de operator `.` waarmee een member geselecteerd wordt en de operator `?:`[48]. Dus zelfs operatoren zoals `[]` (array index), `->` (pointer dereferentie naar member) en `()` (functie aanroep) kun je zelf definiëren! Je kunt de prioriteit van operatoren niet zelf definiëren. Dit betekent dat `a + b * c` altijd wordt geïnterpreteerd als `a + (b * c)`. Ook de associativiteit van de operatoren met gelijke prioriteit kun je niet zelf definiëren. Dit betekent dat `a + b + c` altijd wordt geïnterpreteerd als `(a + b)+ c`. Ook is het niet mogelijk om de operatoren die al gedefinieerd zijn voor de ingebouwde types zelf te (her)definiëren. Het zelf definiëren van operatoren die in de taal C++ nog niet bestaan bijvoorbeeld `@` of `**` is niet mogelijk. Bij het definiëren van operatoren voor zelfgedefinieerde types moet je er natuurlijk wel voor zorgen dat het gebruik voor de hand liggend is. De compiler heeft er geen enkele moeilijkheid mee als je bijvoorbeeld de memberfunctie `Breuk::operator+=` definieert die de als argument meegegeven `Breuk` van de receiver aftrekt. De programmeurs die de class `Breuk` gebruiken zullen dit minder kunnen waarderen.

Er blijkt nu toch nog een verschil te zijn tussen het gebruik van de operator `+=` voor het zelfgedefinieerde type `Breuk` en het gebruik van de operator `+=` voor het ingebouwde type `int`. Bij het type `int` kun je de operator `+=` als volgt gebruiken: `a += b += c;`. Dit wordt omdat de operator `+=` van rechts naar links geëvalueerd wordt als volgt geïnterpreteerd `a += (b += c)`. Dit betekent dat eerst `c` bij `b` wordt opgeteld en dat het resultaat van deze optelling weer bij `a` wordt opgeteld. Zowel `b` als `a` hebben na de optelling een waarde toegekend gekregen. Als je dit probeert als `a`, `b` en `c` van het type `Breuk` zijn, verschijnt de volgende (niet erg duidelijke) foutmelding:

```
a += b += c;
```

[48] Ook de niet in dit dictaat besproken operatoren `.*` en `sizeof` kun je niet zelf definiëren.

```
//  Error: (Microsoft) binary '+=' : no operator found which ↩
 ↪ takes a right-hand operand of type 'void'
//  Error: (GCC)       no match for 'operator+=' in 'a += ↩
 ↪ b.Breuk::operator+=(c)'
```

Dit komt doordat de `Breuk::operator+=` memberfunctie geen return type heeft (`void`). Het resultaat van de bewerking `b += c` moet dus niet alleen in het object `b` worden opgeslagen maar ook als resultaat worden teruggegeven. De `operator+=` memberfunctie kan dan als volgt gedeclareerd worden:

```
    Breuk[49] operator+=(Breuk right);
```

De definitie is dan als volgt:

```
Breuk Breuk::operator+=(Breuk right) {
    boven = boven * right.onder + onder * right.boven;
    onder = onder * right.onder;
    return DIT_OBJECT; // Pas op: Dit is géén C++ code!
}
```

Ik zal nu eerst bespreken hoe je de receiver (`DIT_OBJECT`) kunt benaderen en daarna zal ik weer op de definitie van de `operator+=` terugkomen.

2.13 `this` pointer

Elke memberfunctie kan beschikken over een impliciete parameter genaamd `this` die het adres bevat van het object waarop de memberfunctie wordt uitgevoerd. Met deze pointer kun je bijvoorbeeld het object waarop een memberfunctie wordt uitgevoerd als return waarde van deze memberfunctie teruggeven. Bijvoorbeeld:

```
Breuk Breuk::operator+=(Breuk right) { ↩
 ↪ // Let op: deze code is niet correct!;
```

[49] Even verderop zal ik bespreken waarom het gebruik van het return type `Breuk` niet juist is en hoe het beter kan.

```
// Zie paragraaf 2.20 voor een correcte implementatie van de ↩
 ↪ operator+=.
    boven = boven * right.onder + onder * right.boven;
    onder = onder * right.onder;
    return *this;
}
```

Omdat `this` een pointer is naar het object waarop de `operator+=` uitgevoerd wordt en omdat dit object teruggeven moet worden, moeten we het object waar de pointer `this` naar wijst teruggeven. We coderen dit, zoals je weet, als `*this`. Omdat het returntype van de functie `Breuk` is, wordt een kopietje van het `Breuk` object waarop de `operator+=` is uitgevoerd teruggegeven.

Als we nu de `Breuk` objecten `a`, `b` en `c` aanmaken en de expressie `a += b += c;` testen, dan zien we dat het werkt zoals verwacht. Object `a` wordt gelijk aan `a + b + c`, object `b` wordt gelijk aan `b + c` en object `c` verandert niet.

Toch is het (nog steeds) niet correct. In de bovenstaande `Breuk::operator+=` memberfunctie zal bij het uitvoeren van het `return` statement een kopie van het huidige object worden teruggegeven. Dit is echter niet correct. Want als we deze operator als volgt gebruiken: `(a += b)+= c;` dan wordt eerst `a += b` uitgerekend en een kopie van `a` teruggegeven, `c` wordt dan bij deze kopie van `a` opgeteld waarna de kopie wordt verwijderd. Dit is natuurlijk niet de bedoeling, het is de bedoeling dat `c` bij `a` wordt opgeteld.

De bedenker van C++, Bjarne Stroustrup, heeft om dit probleem op te lossen een heel nieuw soort variabele aan C++ toegevoegd: de *reference* variabele.

2.14 Reference variabelen

Zie eventueel [2, blz. 475]: References in C++

Een *reference* is niets anders dan een alias (andere naam) voor de variabele waarnaar hij verwijst (of anders gezegd: refereert). Alle operaties die op een variabele zijn toegestaan zijn ook toegestaan op een reference die naar die variabele verwijst.

```
int i{3}[50];
int& j{i}; // een reference moet geïnitialiseerd worden
           // er bestaat nu 1 variabele met 2 namen (i ↩
            ↪ en j)
j = 4;     // i is nu gelijk aan 4
           // een reference is gewoon een "pseudoniem"
```

Een reference lijkt een beetje op een pointer maar er zijn toch belangrijke verschillen:

- Als je de variabele waar een pointer `p` naar wijst wilt benaderen, moet je de notatie `*p` gebruiken maar als je de variabele waar een reference `r` naar verwijst wilt benaderen kun je gewoon `r` gebruiken.
- Een pointer die naar de variabele `v` wijst kun je later naar een andere variabele laten wijzen maar een reference die naar de variabele `v` verwijst kun je later niet naar een andere variabele laten verwijzen.
- Een pointer kan ook naar geen enkele variabele wijzen (de waarde is dan `0`) maar een reference verwijst altijd naar een variabele.

2.15 Reference parameters

In C worden bij het aanroepen van een functie de waarden van de argumenten gekopieerd naar de parameters van de functie. Dit wordt *call by value* (aanroep met waarden) genoemd. Dit betekent dat de argumenten die in een functieaanroep zijn gebruikt na afloop van de functie *niet* veranderd kunnen zijn. Je kunt de parameters in de functiedefinitie dus gewoon als een lokale variabele gebruiken. Een parameter die in een definitie van een functie gebruikt wordt, wordt ook

[50] Tot nu toe is een integer geïnitialiseerd met de uit C overgenomen syntax `int i = 3;`. In C++ mag je ook de syntax `int i(3);` of `int i{3};` of `int i = {3};` gebruiken, alsof het ingebouwde type `int` een constructor heeft die je aanroept. De syntax met accolades heeft de voorkeur omdat deze syntax (uniforme initialisatie) in alle gevallen werkt, bijvoorbeeld ook bij zelfgedefinieerde types (classes) zoals `Breuk`. Toch worden variabelen van de ingebouwde types in de praktijk meestal met een enkel =-teken geïnitialiseerd. Een object `b` van de class `Breuk` kun je met de waarde ⅗ initialiseren met de syntax `Breuk b(3, 7);` of `Breuk b{3, 7};` of `Breuk b = {3, 7};`.

wel een formele parameter genoemd. Een argument dat bij een aanroep van een functie gebruikt wordt, wordt ook wel een actuele parameter genoemd. Bij een functieaanroep wordt dus de *waarde* van de actuele parameter naar de formele parameter gekopieerd[51]. Het is dus geen probleem als de actuele parameter een constante is.

```
void slaRegelsOver(int aantal) {
    while (aantal > 0) {
        cout << endl;
        aantal = aantal - 1;
    }
}
//  ...
    int n = 7;
    slaRegelsOver(n);  // waarde van n wordt gekopieerd ↩
     ↪ naar l
    cout << "n = " << n
         << endl;  // uitvoer: n = 7
    slaRegelsOver(3);  // waarde 3 wordt gekopieerd naar l
```

Als je het argument dat bij de aanroep wordt gebruikt wel wilt aanpassen vanuit de functie, dan kan dit in C alleen door het gebruik van pointers.

```
void swapInts(int* p, int* q) {
    int t = *p;
    *p = *q;
    *q = t;
}
//  ...
    int i = 3;
    int j = 4;
    swapInts(&i, &j);
```

[51] Je kunt in C natuurlijk wel een pointer als parameter definiëren en bij het aanroepen een adres van een variabele meegeven (ook wel *call by reference* genoemd) zodat de variabele in de functie veranderd kan worden. Maar het meegegeven argument zelf (in dit geval het adres van de variabele) kan in de functie niet veranderd worden.

In C++ kun je als alternatief ook een reference als formele parameter gebruiken. Dit maakt het mogelijk om het bij aanroep meegegeven argument (de actuele parameter) vanuit de functie aan te passen. De formele parameter is dan namelijk een pseudoniem (andere naam) voor het aan de functie meegegeven argument.

```
void swapInts(int& p, int& q) {
    int t(p);
    p = q;
    q = t;
}
//  ...
    int i(3);
    int j(4);
    swapInts(i, j);
```

Een reference parameter wordt geïmplementeerd door niet de waarde van het bij aanroep meegegeven argument te kopiëren maar door het adres van het bij aanroep meegegeven argument te kopiëren. Als dan in de functie aan de formele parameter een nieuwe waarde wordt toegekend, dan wordt deze waarde direct op het adres van het bij aanroep meegegeven argument opgeslagen.

Dit betekent dat het vanzelfsprekend niet mogelijk is om als argument een constante te gebruiken omdat aan een constante geen nieuwe waarde toegekend mag worden. Bijvoorbeeld:

```
    swapInts(i, 5);
//  Error: (Microsoft) 'swapInts' : cannot convert parameter ↩
 ↪ 2 from 'int' to 'int &'
//  Error: (GCC)       invalid initialization of non-const ↩
 ↪ reference of type 'int&' from an rvalue of type 'int'
```

Zie Param.cpp voor een compleet voorbeeldprogramma.

2.16 `const` reference parameters

Er is nog een andere reden om een reference parameter in plaats van een "normale" value parameter te gebruiken. Bij een reference parameter wordt zoals we hebben gezien een adres (een vast aantal bytes[52]) gekopieerd terwijl bij een value parameter de waarde (een variabel aantal bytes, afhankelijk van het type) gekopieerd wordt. Als de waarde veel geheugenruimte in beslag neemt, is het om performance redenen beter om een reference parameter te gebruiken.[53] Om er voor te zorgen dat deze functie toch aangeroepen kan worden met een constante kan deze parameter dan het beste als `const` reference parameter gedefinieerd worden. Het wordt dan ook onmogelijk om in de functie een waarde aan de formele parameter toe te kennen.

```
void drukaf(Tijdsduur td) {          // kopieer een Tijdsduur
    // ...
}
void drukaf(const Tijdsduur& td) { // kopieer een adres maar
    // ...                          // voorkom toekenningen aan
}                                   // de inhoud van dit adres
```

Als je probeert om een `const` reference parameter in de functie te veranderen, krijg je de volgende foutmelding:

```
void drukaf(const Tijdsduur& td) {
    td.uur = 23; // drukaf probeert vals te spelen
//  Error: (Microsoft) 'uur' cannot be modified because it is ↩
 ↪ being accessed through a const object
//  Error: (GCC)       assignment of member 'Tijdsduur::uur' ↩
 ↪ in read-only object
```

[52] Op een AVR microcontroller bestaat een adres uit 2 bytes, op een 32-bits systemen uit 4 bytes en op 64-bits systeem uit 8 bytes.

[53] Later zullen we zien dat er nog een veel belangrijke reden is om een reference parameter in plaats van een value parameter te gebruiken, zie paragraaf 4.2.

In alle voorafgaande code waarin een `Breuk` als parameter is gebruikt kan dus beter[54] een `const Breuk&` als parameter worden gebruikt (Bijvoorbeeld op pagina 45, 51 en 64.)

2.17 Parameter FAQ[55]

Q: Wanneer gebruik je een `T&` parameter in plaats van een `T` parameter?

A: Gebruik een `T&` als je wilt dat een toekenning aan de formele parameter (de parameter die gebruikt wordt in de functiedefinitie) ook een verandering van de actuele parameter (het argument dat gebruikt wordt in de functieaanroep) tot gevolg heeft.

Q: Wanneer gebruik je een `const T&` parameter in plaats van een `T` parameter?

A: Gebruik een `const T&` als je in de functie de formele parameter niet verandert en als de geheugenruimte die door een variabele van type `T` wordt ingenomen meer is dan de geheugenruimte die nodig is om een adres op te slaan.

Q: Wanneer gebruik je een `T*` parameter in plaats van een `T&` parameter?

A: Gebruik een `T*` als je ook "niets" door wilt kunnen geven. Een reference *moet* namelijk altijd ergens naar verwijzen. Een pointer kan ook nergens naar wijzen (de waarde van de pointer is dan `0`).

[54] Een breuk bestaat uit 2 `int`s. Op een AVR microcontroller bestaat elke `int` uit 2 bytes en een `Breuk` dus uit $2 \times 2 = 4$ bytes. Een adres bestaat op een AVR uit 2 bytes en kan dus sneller gekopieerd worden dan een `Breuk`. Op een 32-bits systeem bestaat een `int` uit 4 bytes en een `Breuk` dus uit $2 \times 4 = 8$ bytes. Een adres bestaat op een 32-bits systeem uit 4 bytes en kan dus sneller gekopieerd worden dan een `Breuk`. Op een 64-bits systeem bestaat een `int` uit 4 bytes en een `Breuk` dus uit $2 \times 4 = 8$ bytes. Een adres bestaat op een 64-bits systeem uit 8 bytes en kan dus net zo snel gekopieerd worden als een `Breuk`.

[55] FAQ = Frequently Asked Questions (veelgestelde vragen).

2.18 Reference in range-based for

In paragraaf 1.6 heb je kennis gemaakt met de in C++11 geïntroduceerde variant van de `for`-lus, de zogenoemde *range-based for*. Deze for-lus is speciaal bedoeld om de elementen van een array[56] één voor één te benaderen.

De in paragraaf 1.6 gepresenteerde code laat zien hoe je een array element voor element kunt uitlezen:

```
int array[] = {12, 2, 17, 32, 1, 18};
auto som = 0;
for (auto element: array) {
    som += element;
}
```

De variabele `element` krijgt één voor één een waarde uit de array genaamd `array`.

Als je element voor element wilt aanpassen, moet je een `int&` of `auto&` gebruiken in de range-based for:

```
for (auto& element: array) {
    element = 0;
}
```

De reference `element` is nu één voor één een alias voor een element uit de array genaamd `array`. Zie RefRangeBasedFor.cpp voor een compleet voorbeeldprogramma.

2.19 Reference return type

Als een functie een waarde teruggeeft door middel van een `return` statement, wordt de waarde van de expressie achter het `return` statement gekopieerd naar de plaats waar de functie aangeroepen is. Door nu een reference als return type te

[56] De standaard C++ library bevat nog vele andere datastructuren die ook met een range-based for doorlopen kunnen worden zie paragraaf 3.7 en paragraaf 3.8.

gebruiken kun je een alias in plaats van een waarde teruggeven. Deze alias kun je dan gebruiken om een waarde aan toe te kennen. Zie voorbeeldprogramma Refret.cpp:

```
int& max(int& a, int& b) {
    if (a > b) return a;
    else return b;
}

int main() {
    int x(3), y(4);
    max(x, y) = 2; // y is nu 2
    max(x, y)++;   // x is nu 4
```

Pas op dat je geen reference naar een lokale variabele teruggeeft. Na afloop van een functie worden de lokale variabelen uit het geheugen verwijderd. De referentie verwijst dan naar een niet meer bestaande variabele.

```
int& som(int i1, int i2) {
    int s = i1 + i2;
    return s; // Een gemene fout!
}
//  ...
    c = som(a, b);
```

Deze fout (het zogenoemde *dangling reference problem*) is erg gemeen omdat de lokale variabele `s` natuurlijk niet echt uit het geheugen verwijderd wordt. De geheugenplaats wordt alleen vrijgegeven voor hergebruik. Dit heeft tot gevolg dat de bovenstaande aanroep van `som` meestal gewoon werkt[57]. De fout in de definitie van de functie `som` komt pas aan het licht als we de functie bijvoorbeeld als volgt aanroepen:

```
    int& s1 = som(1, 2);
```

[57] De Microsoft Visual Studio C++ compiler geeft de warning: `returning address of local variable or temporary`. De GCC C++ compiler geeft de warning: `reference to local variable s returned`

```
    int& s2 = som(3, 4);
    cout << "1 + 2 = " << s1 << endl;
    cout << "3 + 4 = " << s2 << endl;
```

De uitvoer van dit programma is[58]:

```
1 + 2 = 7
3 + 4 = 2009302084
```

Hetzelfde gevaar is trouwens ook aanwezig als je een pointer (bijvoorbeeld `int*`) als return type kiest (het zogenoemde *dangling pointer problem*).

```
int* som(int i1, int i2) {
    int s = i1 + i2;
    return &s; // Een gemene fout!
}
//  ...
    c = *som(a, b);
```

Je moet er dan voor oppassen dat de pointer niet naar een variabele wijst die na afloop van de functie niet meer bestaat[59].

Een goede manier om de waarde van een lokale variabele terug te geven vanuit een functie is door een "gewoon" type (geen reference en ook geen pointer) als return type te gebruiken.

```
int som(int i1, int i2) {
    int s = i1 + i2;
    return s; // Correct!
}
//  ...
    c = som(a, b);
```

[58] Het programma is gecompileerd met Microsoft Visual Studio 2013

[59] De Microsoft Visual Studio C++ compiler geeft de warning: `returning address of local variable or temporary`. De GCC C++ compiler geeft de warning: `address of local variable s returned`.

Zie Refret.cpp voor een compleet voorbeeldprogramma.

2.20 Reference return type (deel 2)

In de `Breuk::operator+=` memberfunctie die gegeven is op pagina 67 zal bij het uitvoeren van het `return` statement een kopie van het huidige object worden teruggegeven. Dit is echter niet correct. Want als we deze operator als volgt gebruiken: `(a += b)+= c;` dan wordt eerst `a += b` uitgerekend en een kopie van `a` teruggegeven, `c` wordt dan bij deze kopie van `a` opgeteld waarna de kopie wordt verwijderd. Dit is natuurlijk niet de bedoeling, het is de bedoeling dat `c` bij `a` wordt opgeteld. We kunnen dit probleem oplossen door een reference naar het object zelf terug te geven. Hiermee zorgen we er meteen voor dat de expressie `(a += b)+= c` gewoon goed (zoals bij integers) werkt. De `operator+=` memberfunctie kan dan als volgt gedeclareerd worden:

```
Breuk& Breuk::operator+=(const Breuk& right) {
    boven = boven * right.onder + onder * right.boven;
    onder *= right.onder;
    normaliseer();
    return *this;
}
```

2.21 Operator overloading (deel 2)

Behalve de operator += kun je ook andere operatoren overloaden. Voor de class `Breuk` kun je bijvoorbeeld de operator + definiëren, zodat objecten `b1` en `b2` van de class `Breuk` gewoon door middel van `b1 + b2` opgeteld kunnen worden.

```
class Breuk {
public:
    Breuk();
    Breuk(int t);
    Breuk(int t, int n);
    Breuk& operator+=(const Breuk& right);
```

```
    const Breuk operator+(const Breuk& right) const;
    // ...
};

const Breuk Breuk::operator+(const Breuk& right) const {
    Breuk copyLeft(*this); // maak een kopietje van de ↩
     ↪ receiver
    copyLeft += right;     // tel daar het object right ↩
     ↪ bij op
    return copyLeft;       // geef deze waarde terug
}

int main() {
    Breuk b1(2, 3), b2(4, 5), b3;
    b3 = b1 + b2;
//  ...
```

Zoals je ziet heb ik de `operator+` eenvoudig geïmplementeerd door gebruik te maken van de al eerder gedefinieerde `operator+=`. De code kan nog verder vereenvoudigd worden tot:

```
const Breuk Breuk::operator+(const Breuk& right) const {
    return copyLeft(*this) += right;
}
```

Zie operator+Member.cpp voor een compleet voorbeeldprogramma.

2.22 `operator+` FAQ

Q: Waarom gebruik je `const Breuk` in plaats van `Breuk` als return type bij `operator+`?

A: Dit heb ik afgekeken van Scott Meyers [6] en [7]. Als `a`, `b` en `c` van het type `int` zijn, levert de expressie `(a + b) += c` de volgende (onduidelijke) foutmelding op `Error: left operand must be l-value`. Dit is goed omdat het optellen van `c` bij een tijdelijke variabele (de som `a + b` wordt name-

lijk aan een tijdelijke variabele toegekend) zinloos is. De tijdelijke variabele wordt namelijk meteen na het berekenen van de expressie weer verwijderd. Waarschijnlijk heeft de programmeur `(a += b)+= c` bedoeld. Als `a`, `b` en `c` van het type `Breuk` zijn en we kiezen als return type van `Breuk::operator+` het type `Breuk`, dan zal de expressie `(a + b)+= c` zonder foutmelding vertaald worden. Als we echter als return type `const Breuk` gebruiken, dan levert deze expressie wel een foutmelding op omdat de `operator+=` niet op een `const` object uitgevoerd kan worden. Als je het zelfgedefinieerde type `Breuk` zoveel mogelijk op het ingebouwde type `int` wilt laten lijken (en dat wil je), dan moet je dus `const Breuk` in plaats van `Breuk` als return type van de `operator+` gebruiken.[60]

Q: Kun je de `operator+` niet beter een reference return type geven zodat het maken van de kopie bij return voorkomen wordt?

A: Nee! We hebben een lokale kopie aangemaakt waarin de som is berekend. Als we een reference teruggeven naar deze lokale kopie, ontstaat na return een zogenoemde dangling reference (zie pagina 75) omdat de lokale variabele na afloop van de memberfunctie opgeruimd wordt.

Q: Kun je in de `operator+` de benodigde lokale variabele niet met `new`[61] aanmaken zodat we toch een reference kunnen teruggeven? De met `new` aangemaakte variabele blijft immers na afloop van de `operator+` memberfunctie gewoon bestaan.

A: Ja dit kan wel, maar je kunt het beter *niet* doen. De met `new` aangemaakte variabele zal namelijk (zo lang het programma draait) nooit meer vrijgegeven worden. Dit is een voorbeeld van een *memory leak*. Elke keer als er twee breuken opgeteld worden neemt het beschikbare geheugen af!

[60] Dat dit nogal subtiel is blijkt wel uit het feit dat Bjarne Stroustrup (de ontwerper van C++) in een vergelijkbaar voorbeeld [9, blz. 528] geen `const` bij het return type gebruikt.

[61] De operator `new` wordt pas besproken in paragraaf 5.1 van dit dictaat.

Q: Kun je de implementatie van `operator+` niet nog verder vereenvoudigen tot:

```
return *this += right;
```

A: Nee! Na de bewerking `c = a + b;` is dan niet alleen `c` gelijk geworden aan de som van `a` en `b` maar is ook `a` gelijk geworden aan de som van `a` en `b` en dat is natuurlijk niet de bedoeling.

Q: Kun je bij een `Breuk` ook een `int` optellen?

```
Breuk a(1, 2);
Breuk b = a + 1;
```

A: Ja. De expressie `a + 1` wordt geïnterpreteerd als `a.operator+(1)`. De compiler zal dan "kijken" of de memberfunctie `Breuk::operator+(int)` gedefinieerd is. Dit is hier niet het geval. De compiler "ziet" dat er wel een memberfunctie `Breuk::operator+(const Breuk&)` gedefinieerd is en zal vervolgens "kijken" of de `int` omgezet kan worden naar een `Breuk`. Dit is in dit geval mogelijk door gebruik te maken van de constructor `Breuk(int)`. De compiler maakt dan met deze constructor een tijdelijke variabele van het type `Breuk` aan en initialiseert deze variabele met `1`. Vervolgens wordt een reference naar deze tijdelijke variabele als argument aan de `operator+` memberfunctie meegegeven. De tijdelijke variabele wordt na het uitvoeren van de `operator+` memberfunctie weer opgeruimd.

Q: Kun je bij een `int` ook een `Breuk` optellen?

```
Breuk a(1, 2);
Breuk b = 1 + a;
```

A: Nee, nu niet. De expressie `1 + a` wordt geïnterpreteerd als `1.operator+(a)`. De compiler zal dan "kijken" of de het ingebouwde type `int` het optellen met een `Breuk` heeft gedefinieerd. Dit is vanzelfsprekend niet het geval. De compiler "ziet" dat wel gedefinieerd is hoe een `int` bij een `int` opgeteld moet worden en zal vervolgens "kijken" of de `Breuk` omgezet kan worden naar

een `int`. Dit is in dit geval ook niet mogelijk[62]. De Microsoft Visual Studio C++ compiler geeft de volgende foutmelding: `binary '+' : no global operator found which takes type 'Breuk'`. De GCC C++ compiler genereert een iets duidelijkere foutmelding: `No match for 'operator+' in '3 + b'`. Als je echter de `operator+` niet als memberfunctie definieert maar in plaats daarvan de globale `operator+` overloadt, dan wordt het wel mogelijk om een `int` bij een `Breuk` op te tellen.

2.23 Operator overloading (deel 3)

Naast het definiëren van operatoren als memberfuncties van zelfgedefinieerde types kun je ook de globale operatoren overloaden[63]. De globale `operator+` is onder andere gedeclareerd voor het ingebouwde type `int`:

```
const int operator+(int, int);
```

Merk op dat deze globale operator twee parameters heeft, dit in tegenstelling tot de memberfunctie `Breuk::operator+` die slechts één parameter heeft. Bij deze memberfunctie wordt de parameter opgeteld bij de receiver. Een globale operator heeft geen receiver dus zijn voor een optelling twee parameters nodig. Een expressie zoals: `a + b` zal dus, als `a` en `b` beide van het type `int` zijn, geïnterpreteerd worden als `operator+(a, b)`. Je kunt nu door gebruik te maken van operator overloading zelf zoveel globale `operator+` functies definiëren als je maar wilt.

Als je dus een `Breuk` bij een `int` wilt kunnen optellen, kun je de volgende globale `operator+` definiëren:

```
const Breuk operator+(int left, const Breuk& right) {
    return right + left;
}
```

[62] Op paragraaf 2.25 zal ik bespreken hoe je deze type conversie indien gewenst zelf kunt definiëren.

[63] De operatoren `=`, `[]`, `()` en `->` kunnen echter alleen als memberfunctie overloaded worden.

Deze implementatie roept simpelweg de `Breuk::operator+` memberfunctie aan! Optellen is namelijk commutatief, dat wil zeggen dat $a + b$ gelijk is aan $b + a$. Als we van een `int` een `Breuk` willen aftrekken, moet een andere aanpak worden gekozen want aftrekken is niet commutatief, $a - b$ is namelijk ongelijk aan $b - a$.

```
const Breuk operator-(int left, const Breuk& right) {
    Breuk copyLeft(left);
    copyLeft -= right;
    return copyLeft;
}
```

Deze implementatie roept de `Breuk::operator-=` memberfunctie aan. Die moet dan natuurlijk wel gedefinieerd zijn:

```
Breuk& Breuk::operator-=(const Breuk& right) {
    boven = boven * right.onder - onder * right.boven;
    onder *= right.onder;
    normaliseer();
    return *this;
}
```

Zie operator+Global.cpp voor een compleet voorbeeldprogramma.

In plaats van zowel een memberfunctie `Breuk::operator+(const Breuk&)` en een globale `operator+(int, const Breuk&)` te declareren kun je ook één globale `operator+(const Breuk&, const Breuk&)` declareren.

Voorbeeld van het overloaden van de globale `operator+` voor objecten van de class `Breuk`:

```
class Breuk {
public:
    Breuk(int t);
    Breuk& operator+=(const Breuk& right);
    // ...
};
```

```
const Breuk operator+(const Breuk& left, const Breuk& ↩
 ↪ right) {
    Breuk copyLeft(left);
    copyLeft += right;
    return copyLeft;
}
```

Voor de fijnproever: de implementatie van `operator+` kan ook in 1 regel:

```
const Breuk operator+(const Breuk& left, const Breuk& ↩
 ↪ right) {
    return Breuk(left) += right;
}
```

Voor de connaisseur (echte kenner): De implementatie kan ook zo:

```
const Breuk operator+(Breuk left, const Breuk& right) {
    return left += right;
}
```

Doordat de eerste parameter `left` hier als call by value (zie pagina 69) is gedeclareerd wordt bij aanroep “van zelf” een kopietje van het linker argument van de `operator+` gemaakt.

Zie Breuk3.cpp voor een compleet voorbeeldprogramma.

De unary operatoren (dat zijn operatoren met 1 operand, zoals `!`) en de assignment operatoren (zoals `+=`) kunnen het beste als memberfunctie overloaded worden. De overige binaire operatoren (dat zijn operatoren met 2 operanden, zoals `+`) kunnen het beste als gewone functie overloaded worden. In dit geval wordt namelijk (indien nodig) de linker operand of de rechter operand geconverteerd naar het benodigde type.

2.24 Overloaden operator++ en operator--

Bij het overloaden van de operator++ en operator-- ontstaat een probleem omdat beide zowel een *prefix* als een *postfix* operator variant kennen[64]. Dit probleem is opgelost door de postfix versie te voorzien van een (dummy) int parameter.

Voorbeeld van het overloaden van operator++ voor objecten van de class Breuk:

```
class Breuk {
public:
    Breuk(int t);
    Breuk& operator+=(const Breuk& right);
    Breuk& operator++();          // prefix
    const Breuk operator++(int); // postfix
    // ...
};
```

De implementatie van deze memberfuncties wordt in paragraaf 2.26 gegeven.

Het gebruik van het return type Breuk& in plaats van const Breuk& bij de prefix operator++ zorgt ervoor dat de expressie a = ++++b als a en b van het type Breuk zijn gewoon werkt[65] (net zoals bij het type int). Het gebruik van het return type const Breuk in plaats van Breuk bij de postfix operator++ zorgt ervoor dat de expressie b++++ als b van het type Breuk is een error[66] geeft (net zoals bij het type int).

---

[64] Voor wie het niet meer weet: Een postfix operator ++ wordt na afloop van de expressie uitgevoerd dus a = b++; wordt geïnterpreteerd als a = b; b = b + 1;. De prefix operator ++ wordt voorafgaand aan de expressie uitgevoerd dus a = ++b; wordt geïnterpreteerd als b = b + 1; a = b;.

[65] a = ++++b wordt geïnterpreteerd als a = ++(++b). b wordt dus eerst met 1 verhoogd, daarna nogmaals met 1 verhoogd en tenslotte wordt deze waarde aan a toegekend. Oftewel a = ++++b is hetzelfde als b = b + 2; a = b.

[66] a = b++++ wordt geïnterpreteerd als a = (b++)++. De postfix versie van de operator++ maakt een kopietje van b, verhoogd de waarde van b en geeft tot slot de waarde van het kopietje (de *oude* waarde van b) terug. Het is niet mogelijk om de operator ++ op deze returnwaarde uit te voeren. De Microsoft Visual C++ compiler geeft de error: '++' needs l-value. Met l-value wordt een expressie bedoeld die links van een = teken gebruikt kan worden.

2.25 Conversie operatoren

Een constructor van class `Breuk` met één parameter van het type `T` wordt door de compiler gebruikt als een variabele van het type `T` moet worden geconverteerd naar het type `Breuk` (zie paragraaf 2.6). Door het definiëren van een conversie operator voor het type `T` kan de programmeur ervoor zorgen dat objecten van de class `Breuk` door de compiler (indien nodig) omgezet kunnen worden naar het type `T`. Stel dat je wilt dat een `Breuk` die op een plaats wordt gebruikt waar de compiler een `int` verwacht, door de compiler omgezet wordt naar het "gehele deel" van de `Breuk` dan kun je dit als volgt implementeren:

```
class Breuk {
    // ...
    operator int () const;
    // ...
};

Breuk::operator int () const {
    return boven / onder;
}
```

Pas op bij conversies: definieer geen conversie waarbij informatie verloren gaat! Is het dan wel verstandig om een `Breuk` automatisch te laten converteren naar een `int`?[67]

2.26 Voorbeeld class `Breuk` (derde versie)

Dit voorbeeld is een uitbreiding van de in paragraaf 2.11 gegeven class `Breuk` (versie 2). In deze versie zijn de operator == en de operator != toegevoegd. We zullen een nieuwe vriend (*friend*) leren kennen die ons helpt bij het implementeren van deze operatoren. Ook zul je leren hoe je een object van de class `Breuk` kunt wegschrijven en inlezen door middel van de `iostream` library, zodat de ope-

[67] Nee! Het is overigens ook niet verstandig om een `Breuk` automatisch te laten converteren naar een `double`, want ook daarbij gaat informatie verloren (de zogenoemde afrondfout).

rator << voor wegschrijven en de operator >> voor inlezen gebruikt kan worden (door middel van operator overloading). De memberfuncties `teller` en `noemer` zijn nu niet meer nodig. Het wegschrijven van een `Breuk` (bijvoorbeeld op het scherm) en het inlezen van een `Breuk` (bijvoorbeeld van het toetsenbord) gaat nu op precies dezelfde wijze als het wegschrijven en het inlezen van een `int`. Dit maakt het type `Breuk` voor programmeurs erg eenvoudig te gebruiken. Ik zal nu eerst de complete broncode presenteren van het programma Breuk3.cpp waarin het type `Breuk` gedeclareerd, geïmplementeerd en gebruikt wordt. Meteen daarna zal ik op bovengenoemde punten één voor één ingaan.

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

class Breuk {
public:
    Breuk();
    Breuk(int t);
    Breuk(int t, int n);
    Breuk& operator+=(const Breuk& right);
    Breuk& operator++();         // prefix
    const Breuk operator++(int); // postfix
    // ...
    // Er zijn nog veel uitbreidingen mogelijk
    // ...
private:
    int boven;
    int onder;
    void normaliseer();
friend ostream& operator<<(ostream& out, const Breuk& b);
friend bool operator==(const Breuk& left, const Breuk& ↩
 ↪ right);
};

istream& operator>>(istream& in, Breuk& b);
bool operator!=(const Breuk& left, const Breuk& right);
```

```
const Breuk operator+(const Breuk& left, const Breuk& right);
// ...
// Er zijn nog veel uitbreidingen mogelijk
// ...

int ggd(int n, int m) {
    if (n == 0) return m;
    if (m == 0) return n;
    if (n < 0) n = -n;
    if (m < 0) m = -m;
    while (m != n)
        if (n > m) n -= m;
        else m -= n;
    return n;
}

Breuk::Breuk(): boven(0), onder(1) {
}

Breuk::Breuk(int t): boven(t), onder(1) {
}

Breuk::Breuk(int t, int n): boven(t), onder(n) {
    normaliseer();
}

Breuk& Breuk::operator+=(const Breuk& right) {
    boven = boven * right.onder + onder * right.boven;
    onder *= right.onder;
    normaliseer();
    return *this;
}

Breuk& Breuk::operator++() {
    boven += onder;
    return *this;
}
```

```
const Breuk Breuk::operator++(int) {
    Breuk b(*this);
    ++(*this);
    return b;
}

void Breuk::normaliseer() {
    assert(onder != 0);
    if (onder < 0) {
        onder = -onder;
        boven = -boven;
    }
    int d = ggd(boven, onder);
    boven /= d;
    onder /= d;
}

const Breuk operator+(const Breuk& left, const Breuk& ↩
 ↪ right) {
    Breuk copyLeft(left);
    copyLeft += right;
    return copyLeft;
}

ostream& operator<<(ostream& left, const Breuk& right) {
    return left << right.boven << '/' << right.onder;
}

istream& operator>>(istream& left, Breuk& right) {
    int teller;
    if (left >> teller)
        if (left.peek() == '/') {
            left.get();
            int noemer;
            if (left >> noemer) right = Breuk(teller, ↩
             ↪ noemer);
```

```
            else right = Breuk(teller);
        }
        else right = Breuk(teller);
    else right = Breuk();
    return left;
}

bool operator==(const Breuk& left, const Breuk& right) {
    return left.boven == right.boven && left.onder == ↩
     ↪ right.onder;
}

bool operator!=(const Breuk& left, const Breuk& right) {
    return !(left == right);
}

int main() {
    Breuk b1, b2;               // definiëren van variabelen
    cout << "Geef Breuk: ";
    cin >> b1;                  // inlezen met >>
    cout << "Geef nog een Breuk: ";
    cin >> b2;                  // inlezen met >>
    cout << b1 << "+"           // afdrukken met <<
         << b2 << "="
         << (b1 + b2) << endl; // optellen met +
    Breuk b3(18, -9);           // definiëren en initialiseren
    if (b1 != b3) {             // vergelijken met !=
        b3++;                   // verhogen met ++
    }
    cout << b3 << endl;         // afdrukken met <<
    b3 += 5;                    // verhogen met +=
    cout << b3 << endl;         // afdrukken met <<
    if (-2 == b3) {             // vergelijken met een int
        cout << "OK." << endl;
    }
    else {
        cout << "Error." << endl;
```

```
    }

    cin.get();                  // wacht met sluiten console-
    cin.get();                  // window
    return 0;
}
```

Ik heb ervoor gekozen om de globale `operator==` te overloaden in plaats van een memberfunctie `Breuk::operator==` te definiëren. Dit heeft als voordeel dat zowel het linker als het rechter argument indien nodig naar het type `Breuk` geconverteerd kan worden. Dus zowel de expressies `b == 3` als `3 == b` kunnen worden gebruikt als `b` van het type `Breuk` is. De implementatie van deze globale `operator ==` is als volgt:

```
bool operator==(const Breuk& left, const Breuk& right) {
    return left.boven == right.boven && left.onder == ↩
     ↪ right.onder;
}
```

Dit levert bij compilatie echter de volgende fouten op: `cannot access private member declared in class 'Breuk'`.

De globaal gedefinieerde `operator==` is namelijk geen memberfunctie en heeft dus geen toegang tot de private velden van de class `Breuk`. Maar gelukkig is er een vriend die ons hier te hulp komt: de *friend function*.

2.27 `friend` functions

Een functie kan als friend van een class gedeclareerd worden. Deze friend functies hebben dezelfde rechten als memberfuncties van de class. Vanuit een friend functie van een class heb je dus toegang tot de private members van die class. Omdat een friend functie geen memberfunctie is van de class, is er geen receiver object. Je moet dus, om de private members te kunnen gebruiken, zelf in de friend functie aangeven welk object je wilt gebruiken. Ook memberfuncties van een andere class kunnen als friend functies gedeclareerd worden. Als een class

als friend gedeclareerd wordt, betekent dit dat alle memberfuncties van die class friend functies zijn.

De zojuist besproken globale `operator==` kan dus eenvoudig als friend van de class `Breuk` gedeclareerd worden waardoor de compiler errors als sneeuw voor de zon verdwijnen.

```
class Breuk {
public:
    // ...
private:
    // ...
\USE{friend} bool operator==(const Breuk& left, const ↩
 ↪ Breuk& right);
};
```

Het maakt niets uit of een friend declaratie in het private of in het public deel van de class geplaatst wordt. Hoewel een friend functie binnen de class gedeclareerd wordt is een friend functie géén memberfunctie maar een globale (normale) functie. In deze friend functie kun je de private members `boven` en `onder` van de class `Breuk` zonder problemen gebruiken.

In eerste instantie lijkt het er misschien op dat een friend functie in tegenspraak is met het principe van information hiding. Dit is echter niet het geval; een class besluit namelijk zelf wie zijn vrienden zijn. Voor alle overige functies (geen member en geen friend) geldt nog steeds dat de private datavelden en private memberfuncties ontoegankelijk zijn. Net zoals in het gewone leven moet een class zijn vrienden met zorg selecteren. Als je elke functie als friend van elke class declareert, wordt het principe van information hiding natuurlijk wel overboord gegooid.

De globale `operator!=` is als volgt overloaded voor het type `Breuk`:

```
bool operator!=(const Breuk& left, const Breuk& right) {
    return !(left == right);
}
```

Bij de implementatie is gebruik gemaakt van de al eerder voor het type Breuk overloaded globale operator==. Het is dus niet nodig om deze operator!= als friend functie van de class Breuk te definiëren.

2.28 Operator overloading (deel 4)

Het object cout dat in de headerfile iostream.h gedeclareerd is, is van het type ostream. Om er voor te zorgen dat je een Breuk b op het scherm kunt afdrukken door middel van: cout << b, moet je, zoals je nu zo langzamerhand wel zult begrijpen, de globale operator<< overloaden[68]:

```
ostream& operator<<(ostream& left, const Breuk& right) {
    left << right.boven << '/' << right.onder;
    return left;
}
```

Deze globale operator gebruikt de private velden van de class Breuk en moet daarom als friend van deze class gedeclareerd worden. Als eerste parameter is een ostream& gebruikt omdat de operator het als argument meegegeven object (in ons geval cout) moet kunnen aanpassen. Als return type is het type ostream& gebruikt. De als parameter meegegeven ostream& wordt ook weer teruggegeven. Dit heeft tot gevolg dat je verschillende << operatoren achter elkaar kunt "rijgen". Bijvoorbeeld:

```
    cout << "De breuk a = " << a << " en de breuk b = " << ↩
     ↪ b << endl;
```

Omdat alle ingebouwde overloaded << operatoren met als eerste parameter een ostream& deze reference ook weer als return waarde teruggeven kun je de bovenstaande operator<< vereenvoudigen tot:

[68] We hebben hier geen keuze omdat het definiëren van de memberfunctie ostream::operator<<(const Breuk) geen reële mogelijkheid is. De class ostream is namelijk in de iostream library gedeclareerd en deze library kunnen (en willen) we natuurlijk niet aanpassen.

```
ostream& operator<<(ostream& left, const Breuk& right) {
    return left << right.boven << '/' << right.onder;
}
```

Op vergelijkbare wijze kun je de globale `operator>>` overloaden zodat de objecten `a` en `b` van de class `Breuk` als volgt ingelezen kunnen worden: `cin >> a >> b`

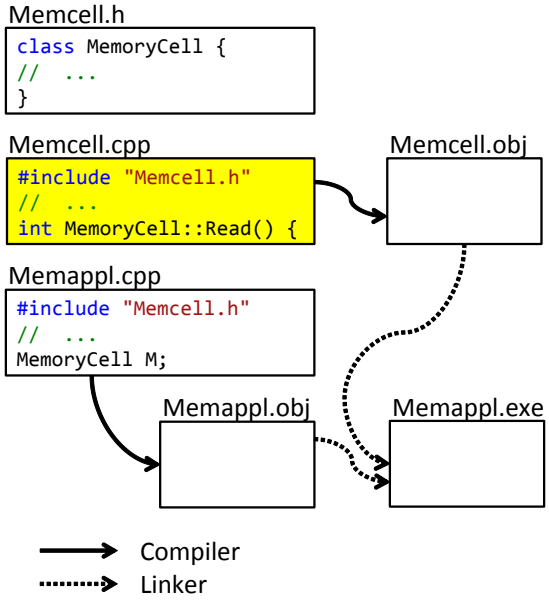
Voor het type van de eerste parameter en het return type moet je in dit geval `istream` gebruiken.

Als je deze laatste versie van `Breuk` vergelijkt met versie 0 op pagina 45, dan is eigenlijk het enige belangrijke verschil dat het zelfgedefinieerde type `Breuk` nu op precies dezelfde wijze als de ingebouwde types te gebruiken is. Je hebt er 48 pagina's gedetailleerde informatie voor door moeten worstelen om dit te bereiken. Dat dit toch de moeite waard is heb ik op een van de eerste van die 48 pagina's al een keer benadrukt, maar het zou kunnen zijn dat je door alle tussenliggende details het uiteindelijke nut van alle inspanning vergeten bent. Dus zal ik hier nogmaals antwoord geven op de vraag: "Is het al die inspanningen wel waard om een `Breuk` zo te maken dat je hem net zoals een ingebouwd type kan gebruiken?" Ja! Het maken van de class `Breuk` is dan wel een hele inspanning maar iedere programmeur kan vervolgens dit zelfgedefinieerde type, als hij of zij de naam `Breuk` maar kent, als vanzelf (intuïtief) gebruiken. Er is daarbij geen handleiding of helpfunctie nodig is, omdat het type `Breuk` zich net zo gedraagt als een ingebouwd type. De class `Breuk` hoeft maar één keer gemaakt te worden, maar zal talloze malen gebruikt worden.

2.29 Voorbeeld separate compilation van class `MemoryCell`

Je kunt de *declaratie* en *definitie* (implementatie) van een class splitsen in een `.h` en een `.cpp` file. Dit heeft als voordeel dat je de implementatie afzonderlijk kunt

compileren tot een zogenoemde object file[69]. De gebruiker van de class kan dan de `.h` file gebruiken in zijn eigen programma en vervolgens de `.obj` file met zijn eigen programma mee "linken". Zie figuur 2.1. De gebruiker hoeft dan dus niet te beschikken over de implementatie.



Figuur 2.1: Separate compilation.

Voor de class `MemoryCell` zien de files Memcell.h, Memcell.cpp en Memappl.cpp er als volgt uit:

```
// Dit is file Memcell.h
// Prevent multiple inclusion.[70]
#ifndef _Memcell_
#define _Memcell_
class MemoryCell {
public:
    int Read() const;
```

[69] Een object file bevat machine code en heeft onder Microsoft Windows de extensie `.obj` en onder Linux de extensie `.o`

```
    void Write(int x);
private:
    int StoredValue;
};
#endif

// Dit is file Memcell.cpp
#include "Memcell.h"

int MemoryCell::Read() const {
    return StoredValue;
}

void MemoryCell::Write(int x) {
    StoredValue = x;
}

// Dit is file Memappl.cpp
#include <iostream>
#include "Memcell.h"
using namespace std;

int main() {
    MemoryCell M;
    M.Write(5);
    cout << "Cell contents are " << M.Read() << endl;
//  ...
```

[70] Door middel van de preprocessor directives `#ifndef` enz. worden compilatiefouten voorkomen als de gebruiker de file `Memcell.h` per ongeluk meerdere malen geïnclude heeft. De eerste keer dat de file geïnclude wordt, wordt het symbool `_Memcell_` gedefinieerd. Als de file daarna opnieuw geïnclude wordt, wordt in de `#ifndef` "gezien" dat het symbool `_Memcell_` al bestaat en wordt pas bij de `#endif` weer verder gegaan met vertalen.

3

Templates

Een van de belangrijkste doelen van C++ is het ondersteunen van het hergebruik van code. In dit hoofdstuk wordt één van de taalconstructies die C++ biedt om hergebruik van code mogelijk te maken, de *template*, besproken. De template maakt het mogelijk om functies of classes te schrijven die werken met een nog onbepaald type. We noemen dit *generiek* programmeren. Pas tijdens het (her)gebruik van de functie of class moeten we specificeren welk type gebruikt moet worden.

3.1 Template functies

Op pagina 70 heb ik de functie `swapInts` besproken waarmee twee `int` variabelen verwisseld kunnen worden:

```
void swapInts(int& p, int& q) {
    int t(p);
    p = q;
    q = t;
}
```

Als je twee `double` variabelen wilt verwisselen, kun je deze functie niet rechtstreeks gebruiken. Waarschijnlijk ben je op dit moment gewend om de functie `swapInts` op de volgende wijze te "hergebruiken":

- Maak een kopie van de functie met behulp van de editor functies "knippen" en "plakken".
- Vervang het type `int` door het type `double` met behulp van de editor functie "zoek en vervang".

Deze vorm van hergebruik heeft de volgende nadelen:

- Telkens als je zelf een nieuw type definieert (bijvoorbeeld `Tijdsduur`) waarvoor je de functie `swap` ook wilt kunnen gebruiken zul je opnieuw moeten knippen, plakken, zoeken en vervangen.
- Bij een wat ingewikkelder algoritme, bijvoorbeeld sorteren, is het niet altijd duidelijk welke `int` je wel en welke `int` je niet moet vervangen in `double` als je in plaats van een array met elementen van het type `int` een array met elementen van het type `double` wilt sorteren. Hierdoor kunnen in een goed getest algoritme toch weer fouten opduiken.
- Als er zich in het algoritme een logische fout bevindt of als je het algoritme wilt vervangen door een efficiëntere versie, dan moet je de benodigde wijzigingen in elke gekopieerde versie aanbrengen.

Door middel van een functietemplate kun je de handelingen (knippen, plakken, zoeken en vervangen) die nodig zijn om een functie geschikt te maken voor een ander datatype automatiseren. Je definieert dan een zogenoemde *generieke* functie, een functie die je als het ware voor verschillende datatypes kunt gebruiken.

De definitie van de functietemplate voor `swap` ziet er als volgt uit (SwapTemplate.cpp):

```
template <typename T> void swap(T& p, T& q) {
    T t(p);
    p = q;
    q = t;
```

```
}
```

Na het keyword `template` volgt een lijst van template-parameters tussen < en >. Een template-parameter zal meestal een type zijn[71]. Dit wordt aangegeven door het keyword `typename`[72] gevolgd door een naam voor de parameter. Ik heb hier de naam `T` gebruikt maar ik had net zo goed de naam `VulMaarIn` kunnen gebruiken. De template-parameter moet[73] in de parameterlijst van de functie gebruikt worden. De definitie van deze functietemplate genereert nog geen enkele machinecode instructie. Het is alleen een "mal" waarmee (automatisch) functies aangemaakt kunnen worden.

Als je nu de functie `swap` aanroept, zal de compiler zelf afhankelijk van het type van de gebruikte argumenten de benodigde "versie" van `swap` genereren[74] door voor het template-parameter (in dit geval `T`) het betreffende type in te vullen.

Dus de aanroep:

```
int n(2);
int m(3);
swap(n, m);
```

heeft tot gevolg dat de volgende functie gegenereerd wordt[75]:

[71] Een template kan ook "normale" parameters hebben. Zie paragraaf 3.4.

[72] In plaats van `typename` mag ook `class` gebruikt worden. Omdat het keyword `typename` in een eerdere versie van C++ nog niet aanwezig was, gebruiken veel C++ programmeurs en C++ boeken in plaats van `typename` nog steeds het "verouderde" `class`. Het gebruik van `typename` is op zich duidelijker omdat bij het gebruik van de template zowel zelfgedefinieerde types (classes) als ingebouwde types (zoals `int`) gebruikt kunnen worden.

[73] Dit is niet helemaal waar. Zie de volgende voetnoot.

[74] Zo'n gegenereerde functie wordt een *template instantiation* genoemd. Je ziet nu ook waarom de template-parameter in de parameterlijst van de functiedefinitie gebruikt moet worden. De compiler moet namelijk aan de hand van de gebruikte argumenten kunnen bepalen welke functie gegenereerd en/of aangeroepen moet worden. Als de template-parameter niet in de parameterlijst voorkomt, moet deze parameter bij het gebruik van de functie (tussen < en > na de naam van de functie) expliciet opgegeven worden.

[75] Als je zelf deze functie al gedefinieerd hebt, zal de compiler geen functie genereren maar de al gedefinieerde functie gebruiken.

```
void swap(int& p, int& q) {
    int t(p);
    p = q;
    q = t;
}
```

Als de functie swap daarna opnieuw met twee int's als argumenten aangeroepen wordt, dan wordt gewoon de al gegenereerde functie aangeroepen. Als echter de functie swap ook als volgt aangeroepen wordt:

```
    Breuk b(1, 2);
    Breuk c(3, 4);
    swap(b, c);
```

dan heeft dit tot gevolg dat een tweede functie swap gegenereerd wordt[76]:

```
void swap(Breuk& p, Breuk& q) {
    Breuk t(p);
    p = q;
    q = t;
}
```

Het gebruik van een functietemplate heeft de volgende voordelen:

- Telkens als je zelf een nieuw type definieert (bijvoorbeeld Tijdsduur) kun je daarvoor de functie swap ook gebruiken. Natuurlijk moet het type Tijdsduur dan wel de operaties ondersteunen die in de template op het type T uitgevoerd worden. In dit geval kopiëren (copy constructor) en assignment (operator=).
- Als er zich in het algoritme een logische fout bevindt of als je het algoritme wilt vervangen door een efficiëntere versie, dan hoef je de benodigde wijzigingen alleen in de functietemplate aan te brengen en het programma opnieuw te compileren.

[76] Hier blijkt duidelijk het belang van function name overloading (zie paragraaf 1.14).

Het gebruik van een functietemplate heeft echter ook het volgende nadeel:

- Doordat de compiler de volledige functietemplate definitie nodig heeft om een functie aanroep te kunnen vertalen moet de definitie van de functietemplate in een headerfile (`.h` file) opgenomen worden en geïnclude worden in elke `.cpp` file waarin de functietemplate gebruikt wordt. Het is niet mogelijk om de functietemplate afzonderlijk te compileren tot een object file en deze later aan de rest van de code te "linken" zoals dit met een "gewone" functie wel kan.

Bij het ontwikkelen van kleine programma's zijn deze voordelen misschien niet zo belangrijk maar bij het ontwikkelen van grote programma's zijn deze voordelen wel erg belangrijk. Door gebruik te maken van een functietemplate in plaats van "met de hand" verschillende versies van een functie aan te maken wordt een programma *beter onderhoudbaar* en *eenvoudiger uitbreidbaar*.

3.2 Class Templates

In de vorige paragraaf hebben we gezien dat je een template van een functie kunt maken waardoor de functie generiek wordt. Een generieke functie kun je voor verschillende datatypes gebruiken. Als je de generieke functie aanroept, zal de compiler zelf afhankelijk van het type van de gebruikte argumenten de benodigde "versie" van de generieke functie genereren door voor de template-parameter het betreffende type in te vullen. Natuurlijk is het ook mogelijk om een memberfunctie als template te definiëren, waarmee je dus generieke memberfuncties kunt maken. Het is in C++ zelfs mogelijk een hele class generiek te maken door deze class als template te definiëren. Om te laten zien hoe dit werkt maken we eerste een class `Dozijn` waarin je twaalf integers kunt opslaan. Daarna maken we een template van deze class zodat we deze class niet alleen kunnen gebruiken voor het opslaan van twaalf integers maar voor het opslaan van twaalf elementen van *elk* type.

De class `Dozijn` waarin twaalf integers kunnen worden opgeslagen kan als volgt gedeclareerd worden (Dozijn.cpp):

```
class Dozijn {
public:
    void zetIn(int index, int waarde);
    int leesUit(int index) const;
private:
    int data[12];
};
```

In een `Dozijn` kunnen dus twaalf integers worden opgeslagen in de private array genaamd `data`. De programmeur die een object van de class `Dozijn` gebruikt kan een integer in dit object "zetten" door de boodschap `zetIn` naar dit object te sturen. De plaats waar de integer moet worden "weggezet" wordt als argument aan dit bericht meegegeven. De plaatsen zijn genummerd van `0` t/m `11`. De waarde `13` kan, bijvoorbeeld, als volgt op plaats nummer `3` worden weggezet.

```
    Dozijn d;
    d.zetIn(3,13);
```

Deze waarde kan uit het object worden "gelezen" door de boodschap `leesUit` naar het object te sturen. Bijvoorbeeld:

```
    cout << "De plaats nummer 3 in d bevat de waarde: " << ↩
     ↪ d.leesUit(3) << endl;
```

De implementatie van memberfuncties van de class `Dozijn` is als volgt:

```
void Dozijn::zetIn(int index, int waarde) {
    if (index >= 0 && index < 12)
        data[index] = waarde;
}

int Dozijn::leesUit(int index) const {
    if (index >= 0 && index < 12)
        return data[index];
    return 0; /* ik weet niets beters[77] */
}
```

De als argument meegegeven `index` wordt bij beide memberfuncties gecontroleerd.

Om de inhoud van een object van de class `Dozijn` eenvoudig te kunnen afdrukken is de `operator`<< als volgt overloaded:

```
ostream& operator<<(ostream& o, const Dozijn& d) {
    o << d.leesUit(0);
    for (int i = 1; i < 12; ++i)
        o << ", " << d.leesUit(i);
    return o;
}
```

Je kunt de class `Dozijn` bijvoorbeeld als volgt gebruiken:

```
int main() {
    Dozijn kwadraten;
    for (int j = 0; j < 12; ++j)
        kwadraten.zetIn(j, j * j);
    cout << "kwadraten = " << kwadraten << endl;
    cin.get();
    return 0;
}
```

Dit programma geeft de volgende uitvoer:

```
kwadraten = 0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100, 121
```

In objecten van de class `Dozijn` kunnen twaalf elementen van het type `int` worden opgeslagen. Als je een `Dozijn` met elementen van het type `double` nodig hebt, kun je natuurlijk gaan kopiëren, plakken, zoeken en vervangen maar daar zitten weer de in de vorige paragraaf besproken nadelen aan. Als je verschillende versies van `Dozijn` "met de hand" genereert, moet je bovendien elke versie een andere naam geven omdat een class naam uniek moet zijn. In plaats daarvan kun je ook

[77] Het is in C++ mogelijk om een functie te verlaten zonder dat een returnwaarde wordt teruggegeven door een zogenoemde exception te gooien. Exceptions worden in paragraaf 6.10 behandeld.

het template mechanisme gebruiken om een `Dozijn` met elementen van het type `T` te definiëren, waarbij het type `T` pas bij het gebruik van de class template `Dozijn` wordt bepaald. Bij het gebruik van de class template `Dozijn` kan de compiler niet zelf bepalen wat het type `T` moet zijn. Vandaar dat je dit bij het gebruik van de class template `Dozijn` zelf moet specificeren. Bijvoorbeeld:

```
Dozijn<Breuk> db; // een dozijn breuken
```

3.3 Voorbeeld class template `Dozijn`

De *class template* `Dozijn` kan als volgt worden gedeclareerd (DozijnTemplate.cpp):

```
template<typename T> class Dozijn {
public:
    void zetIn(int index, const T& waarde);
    const T& leesUit(int index) const;
private:
    T data[12];
};
```

Merk op dat de parameter `index` nog steeds van het type `int` is. De "plaatsen" in het `Dozijn` zijn altijd genummerd van `0` t/m `11` onafhankelijk van het type van de elementen die in het `Dozijn` opgeslagen worden. Het type van de private array `data` is nu van het type `T` en dit type is een parameter van de template. De tweede parameter van de memberfunctie `zetIn` en het return type van `leesUit` zijn van het type `const T&` in plaats van het type `T` om onnodige kopietjes te voorkomen. Alle memberfuncties van de *template* class zijn vanzelfsprekend ook *templates*. De memberfuncties van de tempate class `Dozijn` kunnen als volgt gedefinieerd worden:

```
template<typename T> void Dozijn<T>::zetIn(int index, ↩
 ↪ const T& waarde) {
    if (index >= 0 && index < 12)
        data[index] = waarde;
}
```

```
template<typename T> const T& Dozijn<T>::leesUit(int ↩
 ↪ index) const {
    if (index < 0)
        index = 0;
    if (index > 11)
        index = 11;
    return data[index];
}
```

De memberfunctie `leesUit` geeft nu de waarde van plaats `0` terug als de `index <= 0` is en geeft de waarde van plaats `11` terug als de `index >= 11` is. Je kunt in dit geval namelijk niet de waarde `0` teruggeven zoals we bij het `Dozijn` met integers (op pagina 101) gedaan hebben omdat we niet weten of het type `T` wel een waarde `0` kan hebben.

Om de inhoud van een `Dozijn` van een willekeurig type eenvoudig te kunnen afdrukken is de `operator<<` met behulp van een template als volgt overloaded:

```
template<typename T> ostream& operator<<(ostream& o, const ↩
 ↪ Dozijn<T>& d) {
    o << d.leesUit(0);
    for (int i = 1; i < 12; ++i)
        o << ", " << d.leesUit(i);
    return o;
}
```

Deze class template `Dozijn` kan nu als volgt gebruikt worden:

```
int main() {
    Dozijn<int> kwadraten;
    for (int j = 0; j < 12; ++j)
        kwadraten.zetIn(j, j * j);
    cout << "kwadraten = " << kwadraten << endl;
    Dozijn<string> provincies;
    provincies.zetIn(0, "Drenthe");
    provincies.zetIn(1, "Flevoland");
```

```
    provincies.zetIn(2, "Friesland");
    provincies.zetIn(3, "Gelderland");
    provincies.zetIn(4, "Groningen");
    provincies.zetIn(5, "Limburg");
    provincies.zetIn(6, "Noord-Brabant");
    provincies.zetIn(7, "Noord-Holland");
    provincies.zetIn(8, "Overijssel");
    provincies.zetIn(9, "Utrecht");
    provincies.zetIn(10, "Zeeland");
    provincies.zetIn(11, "Zuid-Holland");
    cout << "provincies = " << provincies << endl;
    cin.get();
    return 0;
}
```

Bij het gebruik van de class template `Dozijn` kan de compiler niet zelf bepalen wat het type `T` moet zijn. Vandaar dat je dit bij het gebruik van de class template `Dozijn` zelf moet specificeren.

De uitvoer van het bovenstaande programma is:

```
kwadraten = 0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100, 121
provincies = Drenthe, Flevoland, Friesland, Gelderland, ↩
 ↪ Groningen, Limburg, Noord-Brabant, Noord-Holland, ↩
 ↪ Overijssel, Utrecht, Zeeland, Zuid-Holland
```

Je ziet dat je de class template `Dozijn` kunt gebruiken om twaalf elementen van het type `int` in op te slaan maar ook om twaalf elementen van het type `string` in op te slaan.

3.4 Voorbeeld class template `Rij`

In de, in de vorige paragraaf beschreven, class template `Dozijn` kunnen we twaalf elementen van een type naar keuze opslaan. Als we meer dan twaalf elementen willen opslaan, kunnen we deze template niet gebruiken. We zouden nu bijvoorbeeld een class template `Gros` kunnen definiëren waar honderdvierenveertig ele-

menten in passen door de class template `Dozijn` te kopiëren en overal de waarde 12 te vervangen door de waarde 144. Het kopiëren van code is echter, zoals je weet, slecht voor de onderhoudbaarheid. We kunnen beter een class template `Rij` definiëren waarbij we het gewenste aantal elementen als (tweede) template-parameter kunnen opgeven.

De class template `Rij` kan als volgt worden gedeclareerd (Rij.cpp):

```
template<typename T, int N> class Rij {
public:
    void zetIn(int index, const T& waarde);
    const T& leesUit(int index) const;
    int aantalPlaatsen() const;
private:
    T data[N];
};
```

De memberfuncties van de tempate class `Rij` kunnen als volgt gedefinieerd worden:

```
template<typename T, int N> void Rij<T, N>::zetIn(int ↩
 ↪ index, const T& waarde) {
    if (index >= 0 && index < N) data[index] = waarde;
}

template<typename T, int N> const T& Rij<T, ↩
 ↪ N>::leesUit(int index) const {
    if (index < 0) index = 0;
    if (index > N - 1) index = N - 1;
    return data[index];
}

template<typename T, int N> int Rij<T, ↩
 ↪ N>::aantalPlaatsen() const {
    return N;
}
```

Om de inhoud van een `Rij` eenvoudig te kunnen afdrukken is de `operator<<` met behulp van een template als volgt overloaded:

```
template<typename T, int N>
ostream& operator<<(ostream& o, const Rij<T, N>& r) {
    o << r.leesUit(0);
    for (int i = 1; i < N; ++i)
        o << ", " << r.leesUit(i);
    return o;
}
```

Deze class template `Rij` kan nu als volgt gebruikt worden:

```
int main() {
    Rij<int, 10> kwad;
    for (int i = 0; i < kwad.aantalPlaatsen(); ++i)
        kwad.zetIn(i, i * i);
    cout << "kwad = " << kwad << endl;

    Rij<char, 26> alfabet;
    for (int i = 0; i < alfabet.aantalPlaatsen(); ++i)
        alfabet.zetIn(i, 'A' + i);
    cout << "alfabet = " << alfabet << endl;

    cin.get();
    return 0;
}
```

De uitvoer van het bovenstaande programma is:

```
kwad = 0, 1, 4, 9, 16, 25, 36, 49, 64, 81
alfabet = A, B, C, D, E, F, G, H, I, J, K, L, M, N, O, P, ↩
 ↪ Q, R, S, T, U, V, W, X, Y, Z
```

Je ziet dat je de class template `Rij` kunt gebruiken om 10 elementen van het type `int` in op te slaan maar ook om 26 elementen van het type `char` in op te slaan.

3.5 Template details

Over templates valt nog veel meer te vertellen:

- Template specialisation;
  Een speciale versie van een template die alleen voor een bepaald type (bijvoorbeeld `int`) of voor bepaalde types (bijvoorbeeld `T*`) wordt gebruikt. De laatste vorm wordt *partial* specialisation genoemd.
- Template memberfuncties;
  Een class (die zelf geen template hoeft te zijn) kan memberfuncties hebben die als template gedefinieerd zijn.
- Default waarden voor template-parameters.
  Net zoals voor gewone functie parameters kun je voor template-parameters ook default waarden (of types) specificeren.

Voor al deze details verwijs ik je naar [3, hoofdstuk 3] en [9, hoofdstuk 23 t/m 29].

3.6 Standaard Templates

In de C++ standaard [4] is een groot aantal standaard templates voor datastructuren en algoritmen opgenomen. De basis voor deze verzameling templates is in de jaren '90 gelegd door Alex Stepanov en Meng Lee in de zogenoemde STL (Standard Template Library). In de standaard library zijn ook een generiek type `array` en een generiek type `vector` opgenomen.

3.7 `std::array`

De class template `array` uit de standaard C++ library vervangt de C-array. Deze `array` ondersteunt net zoals de C-array de `operator[]`. De voordelen van `array` in vergelijking met een C-array:

- Een `array` is in tegenstelling tot een C-array een "echt" object.

- Je kunt een `array` "gewoon" vergelijken, toekennen en kopiëren.
- Het aantal elementen van een `array` moet bij het compileren bekend zijn (net zoals bij een C-array) maar kan wel tijdens het uitvoeren van het programma opgevraagd worden. Met andere woorden: een `array` weet zelf hoe groot hij is.
- Een `array` heeft memberfuncties:
  - `size()` geef het aantal elementen in de `array`.
  - `at(...)` geef element op de opgegeven positie. Bijna gelijk aan `operator[]` maar `at` controleert of de index geldig is en gooit een exception[78] als dit niet zo is.
  - ...[79]

De elementen van een `array` kunnen, net zoals de elementen van een C-array, één voor één gelezen of bewerkt worden met behulp van een range-based for (zie paragraaf 1.6 en paragraaf 2.18).

Voorbeeld van een programma met een `array` (StdArray.cpp):

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

int main() {
    // definieer array van 15 integers
    array<int, 15> a;
    // vul met kwadraten
    int i = 0;
    for (auto& e: a) {
        e = i * i;
        ++i;
    }
```

[78] Een exception is een manier om een fout vanuit een component door te geven aan de code die deze component gebruikt. Exceptions worden in paragraaf 6.10 behandeld.

[79] Zie http://en.cppreference.com/w/cpp/container/array

```
    // druk af
    for (auto e: a) {
        cout << e << " ";
    }
    cout << endl;

    // kopiëren van de een array
    auto b = a;
    for (auto e: b) {
        cout << e << " ";
    }
    cout << endl;
    // vergelijken van array's
    if (a != b)
        cout << "DIT KAN NIET!" << endl;

    // a[100] = 12;
    // ongeldige index ==> crash (als je geluk hebt!)
    // a.at(100) = 12;
    // ongeldige index ==> foutmelding (exception)

    cin.get();
    return 0;
}
```

Uitvoer:

```
0 1 4 9 16 25 36 49 64 81 100 121 144 169 196
0 1 4 9 16 25 36 49 64 81 100 121 144 169 196
```

3.8 std::vector

De class template `vector` uit de standaard C++ library wordt in C++ vaak gebruikt op plaatsen waar je een array wilt gebruiken maar nog niet tijdens het compileren weet hoeveel elementen de array moet bevatten. Een `vector` kan tijdens het uitvoeren van het programma *groeien* en *krimpen*. Deze vector ondersteunt ook

net zoals de C-array de `operator[]`. De voordelen van `vector` in vergelijking met een C-array:

- Een `vector` is in tegenstelling tot een C-array een "echt" object.
- Je kunt een `vector` "gewoon" vergelijken, toekennen en kopiëren.
- Een `vector` kan groeien en krimpen.
- Een `vector` heeft memberfuncties:
  - `size()` geef het aantal elementen in de vector.
  - `at(...)` geef element op de opgegeven positie. Bijna gelijk aan `operator[]` maar `at` controleert of de index geldig is en gooit een exception[80] als dit niet zo is.
  - `capacity()` geef het aantal elementen waarvoor geheugen gereserveerd is. Als de `vector` groter groeit, worden automatisch meer elementen gereserveerd. De capaciteit wordt dan telkens verdubbeld.
  - `push_back(...)` voeg een element toe aan de `vector`. Na afloop is de `size` van de `vector` dus met 1 toegenomen.
  - `resize(...)` Verander de `size` van de `vector`.
  - ...[81]

De elementen van een `vector` kunnen, net zoals de elementen van een C-array, één voor één gelezen of bewerkt worden met behulp van een range-based for (zie paragraaf 1.6 en paragraaf 2.18).

Voorbeeld van een programma met een `vector` (StdVector1.cpp):

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

[80] Een exception is een manier om een fout vanuit een component door te geven aan de code die deze component gebruikt. Exceptions worden in paragraaf 6.10 behandeld.

[81] Zie `http://en.cppreference.com/w/cpp/container/vector`

```
int main() {
    // definieer vector van integers
    vector<int> v;
    // vul met kwadraten
    for (int i = 0; i < 15; ++i) {
        v.push_back(i * i);
    }
    // druk af
    for (auto e: v) {
        cout << e << " ";
    }
    cout << endl;

    // kopiëren van een vector
    auto w = v;
    for (auto e: w) {
        cout << e << " ";
    }
    cout << endl;
    // vergelijken van vectoren
    if (v != w)
        cout << "DIT KAN NIET!" << endl;

    // v[100] = 12;
    // ongeldige index ==> crash (als je geluk hebt!)
    // v.at(100) = 12;
    // ongeldige index ==> foutmelding (exception)

    cin.get();
    return 0;
}
```

Uitvoer:

```
0 1 4 9 16 25 36 49 64 81 100 121 144 169 196
0 1 4 9 16 25 36 49 64 81 100 121 144 169 196
```

Voorbeeldprogramma om een onbekend aantal elementen in een `vector` in te lezen (StdVector2.cpp):

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

void leesInts(vector<int>& vec) {
    // gooi huidige inhoud vec weg
    vec.resize(0);
    int i;
    cout << "Voer een willekeurig aantal integers in, ";
    cout << "sluit af met een letter:" << endl;
    while (cin >> i) {
        vec.push_back(i);
    }
    // zorg dat cin na de "onjuiste" invoer weer gebruikt ↩
     ↪ kan worden
    cin.clear();
    cin.get();
}

int main() {
    // definieer vector
    vector<int> v;
    // vul deze vector
    leesInts(v);
    // druk af
    for (auto e: v) {
        cout << e << " ";
    }
    cout << endl;
```

```
    cin.get();
    cin.get();
    return 0;
}
```

4

Inheritance

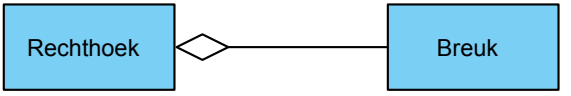
Een van de belangrijkste doelen van C++ is het ondersteunen van het hergebruik van code. In het vorige hoofdstuk werd één van de taalconstructies die C++ biedt om hergebruik van code mogelijk te maken, de *template*, besproken. In dit hoofdstuk worden de twee belangrijkste manieren besproken waarop een herbruikbare softwarecomponent gebruikt kan worden om een nieuwe (ook weer herbruikbare) softwarecomponent te maken. Deze twee vormen van hergebruik worden *composition* en *inheritance* genoemd. Composition ken je al, maar inheritance is (voor jou) nieuw en vormt de kern van OOP.

De in hoofdstuk 2 gedefinieerde class `Breuk` lijkt op het eerste gezicht al een prima herbruikbare softwarecomponent. Als je echter een variant van deze class `Breuk` wilt definiëren, dan heb je op dit moment geen andere keuze dan de class `Breuk` te kopiëren, te voorzien van een andere naam, bijvoorbeeld `MyBreuk`, en de benodigde (kleine) wijzigingen in deze kopie aan te brengen. Deze manier van genereren van varianten produceert, zoals je al weet, een minder goed onderhoudbaar programma. Je zult in dit hoofdstuk leren hoe je door het toepassen van een objectgeoriënteerde techniek (inheritance) een onderhoudbare variant van een herbruikbare softwarecomponent kan maken. Door middel van deze techniek kun je dus softwarecomponenten niet alleen hergebruiken op de manier zoals de ont-

werper van de component dat bedoeld heeft, maar kun je de softwarecomponent ook naar je eigen wensen omvormen.

Hergebruik door middel van *composition* is niet specifiek voor OOP. Ook bij de gestructureerde programmeermethode paste je deze vorm van hergebruik al toe. Het hergebruik van een softwarecomponent door middel van composition is niets anders als het gebruiken van deze component als onderdeel van een andere (nieuwe) softwarecomponent. Als je bijvoorbeeld rechthoeken wilt gaan gebruiken waarbij de lengte en de breedte als breuk moeten worden weergegeven, dan kun je het ADT `Breuk` als volgt (her)gebruiken:

```
class Rechthoek {
public:
    // ...
private:
    Breuk lengte;
    Breuk breedte;
};
```



Figuur 4.1: Schematische weergave van composition.

We zeggen dan dat de class `Rechthoek` een HAS-A (heeft een) relatie heeft met de class `Breuk`. Schematisch kan dit zoals getekend in figuur 4.1 worden weergegeven. De hier gebruikte tekennotatie heet UML (Unified Modelling Language) en is een standaard notatie die veel bij objectgeoriënteerd ontwerpen wordt gebruikt. Een UML diagram waarin de relaties tussen classes worden weergegeven wordt een UML *class diagram* genoemd.

Bij gestructureerd programmeren is dit de enige relatie die softwarecomponenten met elkaar kunnen hebben. Bij objectgeoriënteerd programmeren bestaat ook de zogenoemde IS-A relatie waarop ik nu uitvoerig zal ingaan.

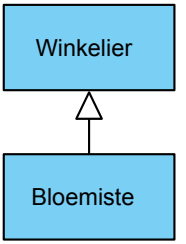
Om de verschillende begrippen te introduceren zal ik niet meteen gebruik maken van een praktisch voorbeeld. Nadat ik de verschillende begrippen geïntroduceerd heb zal ik in een uitgebreid praktisch voorbeeld uit de elektrotechniek laten zien hoe deze begrippen in de praktijk kunnen worden toegepast, zie paragraaf 4.7.1. Ook zal ik dan bespreken wat de voordelen van een objectgeoriënteerde benadering zijn ten opzichte van een gestructureerde of ADT benadering.

4.1 De syntax van inheritance

Door middel van *overerving* (Engels: *inheritance*) kun je een "nieuwe variant" van een bestaande class definiëren zonder dat je de bestaande class hoeft te wijzigen en zonder dat er code gekopieerd wordt. De class die als uitgangspunt gebruikt wordt, wordt de *base* class genoemd (of ook wel parent class of superclass). De class die hiervan afgeleid (derived) wordt, wordt de *derived* class genoemd (of ook wel child class of subclass). In UML kan dit schematisch worden weergegeven zoals in figuur 4.2 gegeven is.

In C++ code wordt dit als volgt gedeclareerd:

```
class Winkelier {
//  ...
};

class Bloemiste: public[82] Winkelier {
//  Bloemiste is afgeleid van Winkelier
//  ...
};
```



Figuur 4.2: Schematische weergave van inheritance.

Je kunt van een base class meerdere derived classes afleiden. Je kunt een derived class ook weer als base class voor een nieuwe afleiding gebruiken.

Een derived class heeft (minimaal) dezelfde datavelden en (minimaal) dezelfde public memberfuncties als de base class waarvan hij is afgeleid. Om deze reden mag je een object van de derived class ook gebruiken als de compiler een object van de base class verwacht[83]. De relatie tussen de derived class en de base class wordt een IS-A (is een) relatie genoemd. De derived class is een (speciaal geval van de) base class. Omdat een derived class datavelden en memberfuncties kan toevoegen aan de base class is het omgekeerde niet waar. Je kunt een object

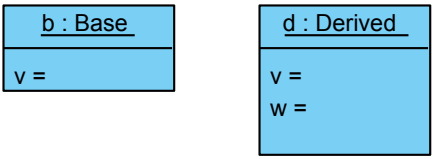
[82] Er bestaat ook private inheritance maar dit wordt in de praktijk niet veel gebruikt en ik zal het hier dan ook niet bespreken. Wij gebruiken altijd public inheritance (niet vergeten om het keyword `public` achter de : te typen anders krijg je per default private inheritance).

[83] Maar pas op voor het slicing probleem dat ik later (paragraaf 4.12) zal bespreken.

van de base class niet gebruiken als de compiler een object van de derived class verwacht. Een base class IS-NOT-A derived class.

De derived class erft alle datavelden van de base class over. Dit wil zeggen dat een object van een derived class (minimaal) dezelfde datavelden heeft als een object van de base class. Private datavelden uit de base class zijn in objecten van de derived class wel aanwezig maar kunnen vanuit memberfuncties van de derived class *niet* rechtstreeks bereikt worden. In de derived class kun je bovendien "extra" datavelden toevoegen. Als de objecten `b` en `d` op onderstaande wijze gedefinieerd zijn, kan de structuur van deze objecten in UML weergegeven worden zoals getekend is in figuur 4.3. Een UML diagram waarin de structuur en inhoud van objecten wordt weergegeven wordt een UML *object diagram* genoemd. Het object `b` bevat alleen een dataveld `v` en een object `d` bevat zowel een dataveld `v` als een dataveld `w`. Het dataveld `v` is alleen toegankelijk vanuit de memberfuncties van de class `Base` en het dataveld `w` is alleen toegankelijk vanuit de memberfuncties van de class `Derived`.

```
class Base {
//  ...
private:
    int v;
};

class Derived: public Base {
//  ...
private:
    int w;
};

//  ...
    Base b;
    Derived d;
```



Figuur 4.3: Schematische weergave van de objecten `b` en `d`.

Ook erft de derived class alle memberfuncties van de base class over. Dit wil zeggen dat op een object van een derived class (minimaal) dezelfde memberfuncties

uitgevoerd kunnen worden als op een object van de base class. Private memberfuncties uit de base class zijn in de derived class wel aanwezig maar kunnen vanuit de derived class *niet* rechtstreeks aangeroepen worden. In de derived class kun je bovendien "extra" memberfuncties toevoegen.

Als de objecten `b` en `d` op onderstaande wijze gedefinieerd zijn, kan de memberfunctie `getV` zowel op het object `b` als op het object `d` uitgevoerd worden. De memberfunctie `getW` is vanzelfsprekend alleen op het object `d` uit te voeren. De private memberfunctie `setV` is alleen aan te roepen vanuit de andere memberfuncties van de class `Base` en de private memberfunctie `setW` is alleen aan te roepen vanuit de andere memberfuncties van de class `Derived`.

```
class Base {
public:
    // ...
    int getV() const { return v; }[84]
private:
    void setV(int i) { v = i; }
    int v;
};

class Derived: public Base {
public:
    // ...
    int getW() const { return w; }
private:
    void setW(int i) { w = i; }
    int w;
};

//  ...
    Base b;
    Derived d;
```

4.2 Polymorfisme

Als de classes `Base` en `Derived` zoals hierboven gedefinieerd zijn, kan een pointer van het type `Base*` niet alleen wijzen naar een object van de class `Base` maar ook naar een object van de class `Derived`. Want een `Derived` IS-A (is een) `Base`. Omdat een pointer van het type `Base*` naar objecten van verschillende classes kan wijzen wordt zo'n pointer een *polymorfe* (veelvormige) pointer genoemd. Evenzo kan een reference van het type `Base&` niet alleen verwijzen naar een object van de class `Base` maar ook naar een object van de class `Derived`. Want een `Derived` IS-A (is een) `Base`. Omdat een reference van het type `Base&` naar objecten van verschillende classes kan verwijzen wordt zo'n reference een *polymorfe* (veelvormige) reference genoemd. Later in dit hoofdstuk zal blijken dat *polymorfisme* de kern is waar het bij OOP om draait[85]. Door het toepassen van polymorfisme kun je software maken die eenvoudig aangepast, uitgebreid en hergebruikt kan worden. (Zie het uitgebreide voorbeeld op paragraaf 4.7.1.)

Als ik nu de volgende functie definieer:

```
void drukVaf(const Base& p) {
    cout << p.getV();
}
```

dan kun je deze functie dus niet alleen gebruiken voor een object van de class `Base` maar ook voor een object van de class `Derived`. Dit kan omdat de parameter `p` van de functie polymorf is. De functie wordt daardoor dus zelf ook polymorf (veelvormig).

[84] Alle memberfuncties zijn in dit voorbeeld in de class zelf gedefinieerd. Dit zogenaamd *inline* definiëren (zie ook paragraaf 6.7) heeft als voordeel dat ik iets minder hoef te typen, maar het nadeel is dat de class niet meer afzonderlijk gecompileerd kan worden (zie paragraaf 2.29).

[85] Inheritance, dat vaak als de kern van OOP wordt genoemd is mijn inziens alleen een middel om polymorfisme te implementeren.

4.3 Memberfunctie overriding

Een derived class kan, zoals we al hebben gezien, datavelden en memberfuncties toevoegen aan de base class. Een derived class kan bovendien memberfuncties die in de base class geïmplementeerd zijn in de derived class een andere implementatie geven (*overridden*). Dit kan alleen als de base class de memberfunctie *virtual* heeft gedeclareerd:[86]

```
class Base {
public:
    virtual void printName() const {
        cout << "Ik ben een Base." << endl;
    }
};
class Derived: public Base {
public:
    virtual[87] void printName() const [88] {
        cout << "Ik ben een Derived." << endl;
    }
};
```

Als via een polymorfe pointer een memberfunctie wordt aangeroepen, wordt de memberfunctie van de class van het object waar de pointer naar wijst aangeroepen. Omdat een polymorfe pointer tijdens het uitvoeren van het programma naar objecten van verschillende classes kan wijzen, kan de keuze van de memberfunctie pas tijdens het uitvoeren van het programma worden bepaald. Dit wordt *late*

[86] Dit is niet waar. Maar als je een non-virtual memberfunctie uit de base class in de derived class toch opnieuw implementeert dan wordt de functie in de base class niet overridden maar overloaded. Overloading geeft onverwachte (en meestal ongewenste) effecten zoals je later (paragraaf 4.8) zult zien.

[87] Het keyword `virtual` kan hier weggelaten worden. Als een memberfunctie eenmaal virtual gedeclareerd is, blijft hij namelijk virtual. Het is echter mijn inziens duidelijker het keyword `virtual` ook in de derived class op te nemen.

[88] Sinds C++11 is het mogelijk om expliciet met behulp van het woord `override` aan te geven dat deze memberfunctie een, reeds in de base class gedeclareerde memberfunctie, override. Zie paragraaf 4.9.

binding of ook wel *dynamic binding* genoemd. Op soortgelijke wijze kan een polymorfe reference verwijzen naar een object van verschillende classes. Als via deze polymorfe reference een memberfunctie wordt aangeroepen, wordt de memberfunctie van de class van het object waar de reference naar verwijst aangeroepen. Ook in dit geval is er sprake van *late binding*.

Voorbeeld van het gebruik van polymorfisme:

```
Base b;
Derived d;
Base* bp1(&b);
Base* bp2(&d);
bp1->printName();
bp2->printName();
```

Uitvoer:

```
Ik ben een Base.
Ik ben een Derived.
```

Het is ook mogelijk om vanuit de in de derived class overridden memberfunctie de originele functie in de base class aan te roepen. Als ik het feit dat een `Derived` IS-A `Base` in het bovenstaande programma verwerk, ontstaat het volgende programma:

```
class Base {
public:
    virtual void printName() const {
        cout << "Ik ben een Base." << endl;
    }
};

class Derived: public Base {
public:
    virtual void printName() const {
        cout << "Ik ben een Derived en ";
        Base::printName();
    }
```

```
};
```

Voorbeeld van het gebruik van polymorfisme:

```
Base b;
Derived d;
Base* bp1(&b);
Base* bp2(&d);
bp1->printName();
bp2->printName();
```

Uitvoer:

```
Ik ben een Base.
Ik ben een Derived en Ik ben een Base.
```

Aan het herdefiniëren van memberfuncties moeten bepaalde voorwaarden gesteld worden, zoals geformuleerd door Liskov[5] in het zogenoemde Liskov Substitution Principle (LSP), om er voor te zorgen dat er geen problemen ontstaan bij polymorf gebruik van de class. Simpel gesteld luidt het LSP: "Een object van de derived class moet op alle plaatsen waar een object van de base class verwacht wordt gebruikt kunnen worden.". Het is belangrijk om goed te begrijpen wanneer inheritance wel/niet gebruikt moet worden. Bedenk dat overerving altijd een *type-relatie* oplevert. Als class `Derived` overerft van class `Base`, geldt "`Derived` is een `Base`". Dat wil zeggen dat elke bewerking die op een object (variabele) van class (type) `Base` uitgevoerd kan worden ook op een object (variabele) van class (type) `Derived` uitgevoerd moet kunnen worden. In de class `Derived` moet je alleen datavelden en memberfuncties toevoegen en/of virtual memberfuncties overridden, maar nooit memberfuncties van de class `Base` overloaden. Het verkeerd gebruik van inheritance is een van de meest voorkomende fouten bij OOP. Een leuke vraag op dit gebied is: is een struisvogel een vogel? (Oftewel mag een class `struisvogel` overerven van class `vogel`?) Het antwoord is afhankelijk van de declaratie van `vogel`. Als een `vogel` een (non-virtual) memberfunctie `vlieg()` heeft waarmee het dataveld `hoogte > 0` wordt, dan niet! In dit geval

heeft de ontwerper van de class `vogel` een fout gemaakt (door te denken dat alle vogels kunnen vliegen).

4.4 Abstract base class

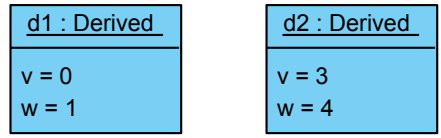
Het is mogelijk om een virtual memberfunctie in een base class alleen maar te declareren en nog niet te implementeren. Dit wordt dan een *pure virtual* memberfunctie genoemd en de betreffende base class wordt een *Abstract Base Class (ABC)* genoemd. Een virtual memberfunctie kan pure virtual gemaakt worden door de declaratie af te sluiten met `= 0;`. Er kunnen geen objecten (variabelen) van een ABC gedefinieerd worden. Elke concrete derived class die van de ABC overerft is "verplicht" om alle pure virtual memberfuncties uit de base class te overridden.

4.5 Constructors bij inheritance

Als een constructor van de derived class aangeroepen wordt, wordt automatisch *eerst* de constructor (zonder parameters) van de base class aangeroepen. Als je in plaats van de constructor zonder parameters een andere constructor van de base class wilt aanroepen vanuit de constructor van de derived class, dan kan dit door deze aanroep in de initialization list van de constructor van de derived class op te nemen. De base class constructor wordt altijd als *eerste* uitgevoerd (onafhankelijk van zijn positie in de initialization list).

Als de objecten `d1` en `d2` op onderstaande wijze gedefinieerd zijn, dan zijn deze objecten geïnitialiseerd zoals getekend is in het UML object diagram dat gegeven is in figuur 4.4.

```
class Base {
public:
    Base(): v(0) { }
    Base(int i): v(i) { }
private:
    int v;
};
```



Figuur 4.4: Schematische weergave van de objecten d1 en d2.

```
class Derived: public Base {
public:
    Derived(): w(1) { } // roept automatisch Base() aan.
    Derived(int i, int j): Base(i), w(j) { }
private:
    int w;
};

//  ...
    Derived d1;
    Derived d2(3, 4);
```

4.6 protected members

Het is in C++ ook mogelijk om in een base class datavelden en memberfuncties te definiëren die niet toegankelijk zijn voor gebruikers van objecten van deze class, maar die wel vanuit de van deze base class afgeleide classes bereikbaar zijn. Dit wordt in de classdeclaratie aangegeven door het keyword `protected` te gebruiken.

```
class Toegang {
public:
    // via een object van de class Toegang (voor iedereen) ↩
     ↪ toegankelijk.
protected:
    // alleen toegankelijk vanuit classes die direct of ↩
     ↪ indirect afgeleid zijn van de class Toegang en ↩
     ↪ vanuit de class Toegang zelf.
```

```
private:
    // alleen toegankelijk vanuit de class Toegang zelf.
};
```

Het definiëren van protected datavelden wordt afgeraden omdat dit slecht is voor de onderhoudbaarheid van de code. Als een protected dataveld een illegale waarde krijgt, moet alle broncode worden doorzocht om de plaatsen te vinden waar dit dataveld veranderd wordt. In elke derived class is het protected dataveld namelijk te veranderen. Het is soms wel zinvol om protected memberfuncties te definiëren. Deze protected functies kunnen dan niet door gebruikers van de objecten van deze class worden aangeroepen maar wel in de memberfuncties van derived classes.

4.7 Voorbeeld: ADC kaarten

Ik zal nu een praktisch programmeerprobleem beschrijven. Vervolgens zal ik een gestructureerde oplossing, een oplossing door middel van een ADT en een objectgeoriënteerde oplossing bespreken.[89] Daarna zal ik deze oplossingen met elkaar vergelijken (nu mag je één keer raden welke oplossing de beste zal blijken te zijn).

4.7.1 Probleemdefinitie

In een programma om een machine te besturen moeten bepaalde signalen via een ADC kaart (ADC = Analoog Digitaal Converter) ingelezen worden. Het programma moet met 2 verschillende types ADC kaarten kunnen werken. Deze kaarten hebben de typenamen AD178 en NI323. Deze kaarten zijn functioneel gelijk en hebben beide een 8 kanaals 16 bits ADC met instelbare voorversterker. Het initialiseren van de kaarten, het selecteren van een kanaal, het uitlezen van de "sampled" waarde en het instellen van de versterkingsfactor moet echter bij elke kaart op een andere wijze gebeuren (andere adressen, andere bits en/of andere procedures). In de applicatie moeten meerdere ADC kaarten van verschillende types gebruikt kunnen worden. In de applicatie is alleen de spanning in volts van

[89] In de theorielessen zal ik een ander (minder praktisch, maar mijns inziens wel leuker) voorbeeld bespreken.

de verschillende signalen van belang. Voor beide 16 bits ADC kaarten geldt dat deze spanning U als volgt berekend kan worden: $U = S * F/6553.5[V]$. S is de "sampled" 16 bits waarde (two's complement) en F is de ingestelde versterkingsfactor. Hoe kun je in het programma nu het beste met de verschillen tussen de kaarten omgaan?

4.7.2 Een gestructureerde oplossing

Eerst zal ik bespreken hoe je dit probleem op een gestructureerde manier oplost. Met de methode van functionele decompositie deel ik het probleem op in een aantal deelproblemen. Voor elk deelprobleem definieer ik vervolgens een functie:

- `initCard` voor het initialiseren van de kaart;
- `selectChannel` voor het selecteren van een kanaal;
- `getChannel` voor het opvragen van het op dit moment geselecteerde kanaal;
- `setAmplifier` voor het instellen van de versterkingsfactor;
- `sampleCard` voor het uitlezen van een sample;
- `readCard` voor het uitlezen van de spanning in volts.

Voor elke kaart die in het programma gebruikt wordt moeten een aantal gegevens bijgehouden worden zoals: het kaarttype, de ingestelde versterkingsfactor en het geselecteerde kanaal. Om deze gegevens per kaart netjes bij elkaar te houden heb ik de struct `ADCCard` gedeclareerd. Voor elke kaart die in het programma gebruikt wordt, wordt dan een variabele van dit struct type aangemaakt. Aan elk van de eerder genoemde functies wordt de te gebruiken kaart dan als parameter van het type `ADCCard` doorgegeven. Zie Kaart0.cpp:

```
enum CardType {AD178, NI323};

struct ADCCard {
    CardType type;
    double amplifyingFactor;
    int selectedChannel;
};
```

```
void initCard(ADCCard& card, CardType type) {
    card.type = type;
    card.amplifyingFactor = 1.0;
    card.selectedChannel = 1;
    // ... eventueel voor alle kaarten benodigde code
    switch (card.type) {
        case AD178:
            // ... de specifieke voor de AD178 benodigde code
            break;
        case NI323:
            // ... de specifieke voor de NI323 benodigde code
            break;
    }
}

void selectChannel(ADCCard& card, int channel) {
    card.selectedChannel = channel;
    // ... eventueel voor alle kaarten benodigde code
    switch (card.type) {
        case AD178:
            // ... de specifieke voor de AD178 benodigde code
            break;
        case NI323:
            // ... de specifieke voor de NI323 benodigde code
            break;
    }
}

int getChannel(const ADCCard& card) {
    return card.selectedChannel;
}

void setAmplifier(ADCCard& card, double factor) {
    card.amplifyingFactor = factor;
    // ... eventueel voor alle kaarten benodigde code
    switch (card.type) {
```

```
        case AD178:
            // ... de specifieke voor de AD178 benodigde code
            break;
        case NI323:
            // ... de specifieke voor de NI323 benodigde code
            break;
        }
}

int sampleCard(const ADCCard& card) {
    int sample;
    // ... eventueel voor alle kaarten benodigde code
    switch (card.type) {
        case AD178:
            // ... de specifieke voor de AD178 benodigde code
            break;
        case NI323:
            // ... de specifieke voor de NI323 benodigde code
            break;
    }
    return sample;
}

double readCard(const ADCCard& card) {
    return sampleCard(card) * card.amplifyingFactor / 6553.5;
}

int main() {
    ADCCard c1;
    initCard(c1, AD178);
    setAmplifier(c1, 10);
    selectChannel(c1, 3);
    cout << "Kanaal " << getChannel(c1) << " van kaart c1 ↩
     ↪ = " << readCard(c1) << " V." << endl;

    ADCCard c2;
    initCard(c2, NI323);
```

```
setAmplifier(c2, 5);
selectChannel(c2, 4);
cout << "Kanaal " << getChannel(c2) << " van kaart c2 ↩
 ↪ = " << readCard(c2) << " V." << endl;
```

Hopelijk heb je allang zelf de nadelen van deze aanpak bedacht:

- Iedere programmeur die gebruikt maakt van het type `struct ADCCard` kan een waarde toekennen aan de datavelden. Zo zou een programmeur die de struct `ADCCard` gebruikt in plaats van de functie `selectChannel` het statement `++adc.c;` kunnen bedenken om het geselecteerde kanaal met 1 te verhogen. Dit werkt natuurlijk niet omdat dan alleen het in de struct opgeslagen kanaalnummer verhoogd wordt terwijl in werkelijkheid geen ander kanaal geselecteerd wordt.
- Iedere programmeur die gebruikt maakt van het type `struct ADCCard` kan er voor kiezen om zelf de code voor het inlezen van een spanning in volts "uit te vinden" in plaats van gebruik te maken van de functie `readCard`. Er valt dus niet te garanderen dat altijd de juiste formule wordt gebruikt. Ook niet als we wel kunnen garanderen dat de functies `readCard` en `sampleCard` correct zijn.
- Iedere programmeur die gebruikt maakt van het type `struct ADCCard` zal zelf nieuwe bewerkingen (zoals bijvoorbeeld het opvragen van de ingestelde versterkingsfactor) definiëren. Het zou beter zijn als alleen de programmeur die verantwoordelijk is voor het onderhouden van het type `struct ADCCard` (en de bijbehorende bewerkingen) dit kan doen.
- Als een nieuw type 8 kanaals 16 bits ADC met instelbare voorversterker (typenaam BB647) in het programma ook gebruikt moet kunnen worden, dan moet ten eerste het enumeratie type `CardName` uitgebreid worden. Bovendien moeten alle functies, waarin door middel van een `switch` afhankelijk van het kaarttype verschillende code wordt uitgevoerd, gewijzigd en opnieuw gecompileerd worden.

Ook de oplossing voor (een deel van) deze problemen heb je vast en zeker al bedacht.

4.7.3 Een oplossing door middel van een ADT

Deze problemen kunnen voorkomen worden als een ADT gebruikt wordt, waarin zowel de data van een kaart als de functies die op een kaart uitgevoerd kunnen worden, ingekapseld zijn[90]. Zie Kaart1.cpp:

```
enum CardType {AD178, NI323};

class ADCCard {
public:
    ADCCard(CardType name);
    void selectChannel(int channel);
    int getChannel() const;
    void setAmplifier(double factor);
    double read() const;
private:
    CardType type;
    double amplifyingFactor;
    int selectedChannel;
    int sample() const;
};

ADCCard::ADCCard(CardType name): type(name), ↩
 ↪ amplifyingFactor(1.0), selectedChannel(1) {
    // ... eventueel voor alle kaarten benodigde code
    switch (type) {
        case AD178:
            // ... de specifieke voor de AD178 benodigde code
            break;
        case NI323:
            // ... de specifieke voor de NI323 benodigde code
            break;
    }
```

[90] De I/O registers van de ADC kaart zelf zijn helaas niet in te kapselen. We kunnen dus niet voorkomen dat een programmeur in plaats van de ADT `ADCCard` te gebruiken rechtstreeks de kaart aanspreekt.

```
}

void ADCCard::selectChannel(int channel) {
    selectedChannel = channel;
    // ... eventueel voor alle kaarten benodigde code
    switch (type) {
        case AD178:
            // ... de specifieke voor de AD178 benodigde code
            break;
        case NI323:
            // ... de specifieke voor de NI323 benodigde code
            break;
    }
}

int ADCCard::getChannel() const {
    return selectedChannel;
}

void ADCCard::setAmplifier(double factor) {
    amplifyingFactor = factor;
    // ... eventueel voor alle kaarten benodigde code
    switch (type) {
        case AD178:
            // ... de specifieke voor de AD178 benodigde code
            break;
        case NI323:
            // ... de specifieke voor de NI323 benodigde code
            break;
    }
}

int ADCCard::sample() const {
    int sample;
    // ... eventueel voor alle kaarten benodigde code
    switch (type) {
        case AD178:
```

```
            // ... de specifieke voor de AD178 benodigde code
            break;
        case NI323:
            // ... de specifieke voor de NI323 benodigde code
            break;
    }
    return sample;
}

double ADCCard::read() const {
    return sample() * amplifyingFactor / 6553.5;
}

int main() {
    ADCCard k1(AD178);
    k1.setAmplifier(10);
    k1.selectChannel(3);
    cout << "Kanaal " << k1.getChannel() << " van kaart k1 ↩
     ↪ = " << k1.read() << " V." << endl;

    ADCCard k2(NI323);
    k2.setAmplifier(5);
    k2.selectChannel(4);
    cout << "Kanaal " << k2.getChannel() << " van kaart k2 ↩
     ↪ = " << k2.read() << " V." << endl;
```

Aan deze oplossingsmethode zit echter nog steeds het volgende nadeel:

- Als een nieuw type 8 kanaals 16 bits ADC met instelbare voorversterker (typenaam BB647) in het programma ook gebruikt moet kunnen worden, dan moet ten eerste het enumeratie type `CardName` uitgebreid worden. Bovendien moeten alle memberfuncties, waarin door middel van een `switch` afhankelijk van het kaarttype verschillende code wordt uitgevoerd, gewijzigd en opnieuw gecompileerd worden.

Op zich is dit nadeel bij deze oplossing minder groot dan bij de gestructureerde oplossing omdat in dit geval alleen de ADT `ADCCard` aangepast hoeft te worden,

terwijl in de gestructureerde oplossing de gehele applicatie (kan enkele miljoenen regels code zijn) doorzocht moet worden op het gebruik van het betreffende `switch` statement. Toch kan ook de oplossing door middel van een ADT tot een probleem voor wat betreft de uitbreidbaarheid leiden.

Stel dat ik niet zelf het ADT `ADCCard` heb ontwikkeld, maar dat ik deze herbruikbare ADT heb ingekocht. Als een geschikte ADT te koop is, heeft kopen de voorkeur boven zelf maken om een aantal redenen:

- De prijs van de ADT zal waarschijnlijk zodanig zijn dat zelf maken al snel duurder wordt. Als de prijs van de bovenstaande ADT 100 Euro is (een redelijke schatting), betekent dit dat je zelf (als beginnende professionele programmeur) de ADT in minder dan één dag moet ontwerpen, implementeren, testen en documenteren om goedkoper uit te zijn.
- De gekochte ADT is hoogst waarschijnlijk uitgebreid getest. Zeker als het product al enige tijd bestaat zullen de meeste bugs er inmiddels uit zijn. De kans dat de applicatie plotseling niet meer werkt als de kaarten in een andere PC geprikt worden is met een zelfontwikkelde ADT groter dan met een gekochte ADT.
- Als de leverancier van de AD178 kaart een hardware bug oplost waardoor ook de software aansturing gewijzigd moet worden, dan zal de leverancier van de ADT (hopelijk) ook een nieuwe versie uitbrengen.

Het is waarschijnlijk dat de leverancier van de ADT alleen de `adccard.h` file en de `adccard.obj` file aan ons levert maar de broncode `adccard.cpp` file niet aan ons beschikbaar stelt[91]. Het toevoegen van de nieuwe ADC kaart (BB647) is dan niet mogelijk.

De (gekochte) ADT `ADCCard` is een herbruikbare softwarecomponent die echter niet door de gebruiker uit te breiden is. Je zult zien dat je door het toepassen van de objectgeoriënteerde technieken inheritance en polymorfisme wel een softwa-

[91] Ook als de broncode wel beschikbaar is dan willen we deze code, om redenen van onderhoudbaarheid, liever niet wijzigen. Want als de leverancier dan met een nieuwe versie van de code komt, dan moeten we ook daarin al onze wijzigingen weer doorvoeren.

recomponent kan maken die door de gebruiker uitgebreid kan worden zonder dat de gebruiker de broncode van de originele component hoeft te wijzigen.

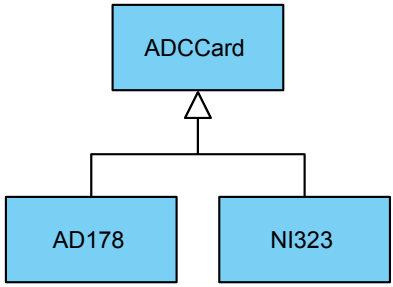
4.7.4 Een objectgeoriënteerde oplossing

Bij het toepassen van de objectgeoriënteerde benadering constateer ik dat in het programma twee types ADC kaarten gebruikt worden. Beide kaarten hebben dezelfde functionaliteit en kunnen dus worden afgeleid van dezelfde base class[92]. Dit is schematisch weergeven in het UML class diagram dat gegeven is in figuur 4.5. Ik heb de base class als volgt gedeclareerd:

```
class ADCCard {
public:
    ADCCard();
    virtual ~ADCCard() = default;[93]
    virtual void selectChannel(int channel) = 0;
    int getChannel() const;
    virtual void setAmplifier(double factor) = 0;
    double read() const;
protected:
    void rememberChannel(int channel);
    void rememberAmplifier(double factor);
private:
    double amplifyingFactor;
    int selectedChannel;
    virtual int sample() const = 0;
};
```

[92] Het bepalen van de benodigde classes en hun onderlinge relaties is bij grotere programma's niet zo eenvoudig. Het bepalen van de benodigde classes en hun relaties wordt OOA (Object Oriented Analyse) en OOD (Object Oriented Design) genoemd. Hoe je dit moet doen wordt voor Elektrotechniek studenten later in het tweede deel van OGOPRG behandeld. Bij de opleiding Technische Informatica is dit al behandeld en komt het later in andere vakken nog uitgebreid aan de orde.

[93] Dit is een zogenoemde *virtual destructor*, waarom het nodig is om een virtual destructor te definiëren wordt verderop in dit dictaat besproken. Zie paragraaf 5.4.



Figuur 4.5: Overerving bij ADC kaarten.

De memberfuncties `selectChannel`, `setAmplifier` en `sample` heb ik *pure virtual* (zie paragraaf 4.4) gedeclareerd omdat de implementatie per kaarttype verschillend is. Dit maakt de class `ADCCard` abstract. Je kunt dus geen objecten van dit type definiëren, alleen references en pointers. Elke afgeleide concrete class (elk kaarttype) moet deze functies "overridden". De memberfuncties `getChannel` en `read` heb ik non-virtual gedeclareerd omdat het niet mijn bedoeling is dat een derived class deze functies "override". Wat er gebeurt als je dit toch probeert zal ik in paragraaf 4.8 bespreken. De memberfuncties `rememberChannel` en `rememberAmplifier` heb ik protected gedeclareerd om er voor te zorgen dat ze vanuit de derived classes bereikbaar zijn. De memberfunctie `sample` heb ik private gedefinieerd omdat deze functie alleen vanuit de memberfunctie `read` gebruikt hoeft te kunnen worden. derived classes moeten deze memberfunctie dus wel definiëren maar ze mogen hem zelf niet aanroepen! Ook gebruikers van deze derived classes hebben mijn inziens de memberfunctie `sample` niet nodig. Ze worden dus door mij verplicht de memberfunctie `read` te gebruiken zodat de returnwaarde altijd in volts is (ik hoop hiermee fouten bij het gebruik te voorkomen).

Van de abstracte base class `ADCCard` heb ik vervolgens de concrete classes `AD178` en `NI323` afgeleid. Zie Kaart2.cpp:

```
class AD178: public ADCCard {
public:
    AD178();
```

```
    virtual void selectChannel(int channel);
    virtual void setAmplifier(double factor);
private:
    virtual int sample() const;
};

class NI323: public ADCCard {
public:
    NI323();
    virtual void selectChannel(int channel);
    virtual void setAmplifier(double factor);
private:
    virtual int sample() const;
};
```

De diverse classes heb ik als volgt geïmplementeerd:

```
ADCCard::ADCCard(): amplifyingFactor(1.0), ↩
 ↪ selectedChannel(1) {
    // ... voor alle kaarten benodigde code
}
int ADCCard::getChannel() const {
    return selectedChannel;
}
double ADCCard::read() const {
    return sample() * amplifyingFactor / 6553.5;
}
void ADCCard::rememberChannel(int channel) {
    selectedChannel = channel;
}
void ADCCard::rememberAmplifier(double factor) {
    amplifyingFactor = factor;
}

AD178::AD178() {
    // ... de specifieke voor de AD178 benodigde code
}
```

```
void AD178::selectChannel(int channel) {
    rememberChannel(channel);
    // ... de specifieke voor de AD178 benodigde code
}
void AD178::setAmplifier(double factor) {
    rememberAmplifier(factor);
    // ... de specifieke voor de AD178 benodigde code
}
int AD178::sample() const {
    // ... de specifieke voor de AD178 benodigde code
}

NI323::NI323() {
    // ... de specifieke voor de NI323 benodigde code
}
void NI323::selectChannel(int channel) {
    rememberChannel(channel);
    // ... de specifieke voor de NI323 benodigde code
}
void NI323::setAmplifier(double factor) {
    rememberAmplifier(factor);
    // ... de specifieke voor de NI323 benodigde code
}
int NI323::sample() const {
    // ... de specifieke voor de NI323 benodigde code
}
```

De onderstaande functie heb ik een polymorfe parameter gegeven zodat hij met elk type ADC kaart gebruikt kan worden.

```
    selectedChannel.selectChannel(3);
    cout << "Kanaal " << selectedChannel.getChannel() << " ↩
     ↪ = " << selectedChannel.read() << " V." << endl;
}

int main() {
```

Deze functie kan ik dan als volgt gebruiken:

```
    cout.setf(std::ios_base::fixed, ↩
     ↪ std::ios_base::floatfield);
    NI323 card2;
    doIt(card2);

    cin.get();
```

Merk op dat de functie `doIt` de in de base class `ADCCard` gedefinieerde memberfunctie `read` aanroept die op zijn beurt de in de *derived* class gedefinieerde memberfunctie `sample` aanroept.

4.7.5 Een kaart toevoegen

Als het programma nu aangepast moet worden zodat een nieuwe kaart (typenaam BB647) ook gebruikt kan worden, dan kan dit heel eenvoudig door de volgende class te declareren (Bb647.h):

```
class BB647: public ADCCard {
public:
    BB647();
    virtual void selectChannel(int channel);
    virtual void setAmplifier(double factor);
private:
    virtual int sample() const;
};
```

Met de volgende implementatie (Bb647.cpp):

```
BB647::BB647() {
    // ... de specifieke voor de BB647 benodigde code
}
void BB647::selectChannel(int channel) {
    rememberChannel(channel);
    // ... de specifieke voor de BB647 benodigde code
```

```
}
void BB647::setAmplifier(double factor) {
    rememberAmplifier(factor);
    // ... de specifieke voor de BB647 benodigde code
}
int BB647::sample() const {
    // ... de specifieke voor de BB647 benodigde code
}
```

Het programma `main` kan nu als volgt aangepast worden (Kaart4.cpp):

```
int main() {
    AD178 card1;
    doIt(card1);
    NI323 card2;
    doIt(card2);
    BB647 card3;    // new!
    doIt(card3);    // new!
```

Als alle functie- en classdeclaraties in aparte `.h` en alle functie- en classdefinities in aparte `.cpp` files opgenomen zijn, dan hoeft alleen de nieuwe class en de nieuwe `main` functie opnieuw vertaald te worden. De rest van het programma kan dan eenvoudig (zonder hercompilatie) meegelinkt worden. Dit voordeel komt voort uit het feit dat de functie `doIt` polymorf is. Aan de parameter die gedefinieerd is als een `ADCCard&` kun je objecten van elke van deze class afgeleide classes (`AD178`, `NI323` of `BB647`) gebruiken. Je ziet dat de objectgeoriënteerde oplossing een zeer goed *onderhoudbaar* en *uitbreidbaar* programma oplevert.

4.8 Overloading en overriding van memberfuncties

Het onderscheid tussen memberfunctie *overloading* en memberfunctie *overriding* is van groot belang. Op paragraaf 1.14 heb je gezien dat een functienaam meerdere keren gebruikt (overloaded) kan worden. De compiler zal aan de hand van de gebruikte argumenten de juiste functie selecteren. Dit maakt deze functies eenvoudiger te gebruiken omdat de gebruiker (de programmeur die deze func-

ties aanroept) slechts één naam hoeft te onthouden. Elke functienaam, dus ook een memberfunctienaam, kan overloaded worden. Een memberfunctie die een andere memberfunctie *overload* heeft dus dezelfde naam. Bij een aanroep van de memberfunctienaam wordt de juiste memberfunctie door de compiler aangeroepen door naar de argumenten bij aanroep te kijken.

Voorbeeld[94] van het gebruik van overloading van memberfuncties (Overload.cpp):

```
class Class {
public:
    void f() const {
        cout << "Ik ben f()" << endl;
    }
    void f(int i) const { // overload f()
        cout << "Ik ben f(int)" << endl;
    }
};

int main() {
    Class object;
    object.f();  // de compiler kiest zelf de juiste functie
    object.f(3); // de compiler kiest zelf de juiste functie
//  ...
```

Uitvoer:

```
Ik ben f()
Ik ben f(int)
```

Overloading en overerving gaan echter *niet* goed samen. Het is niet goed mogelijk om een memberfunctie uit een base class in een derived class te overloaden. Als dit toch geprobeerd wordt, maakt dit alle functies van de base class met dezelfde naam onzichtbaar (*hiding rule*).

[94] In dit voorbeeld (en ook in enkele volgende voorbeelden) heb ik de memberfunctie in de class zelf gedefinieerd. Dit zogenaamd *inline* definiëren (zie ook paragraaf 6.7) heeft als voordeel dat ik iets minder hoef te typen, maar het nadeel is dat de class niet meer afzonderlijk gecompileerd kan worden (zie paragraaf 2.29).

Voorbeeld van het verkeerd gebruik van overloading en de hiding rule (Hide.cpp):

```
// Dit voorbeeld laat zien hoe het NIET moet!
// Je moet overloading en overerving NIET combineren!

class Base {
public:
    void f() const {
        cout << "Ik ben f()" << endl;
    }
};

class Derived: public Base {
public:
    void f(int i) const[95] { // Verberg f() !! Geen goed ↩
     ↪ idee !!!
        cout << "Ik ben f(int)" << endl;
    }
};

void func(Base& b) {
    b.f();
}

int main() {
    Base b;
    Derived d;

    b.f();
    d.f(3);

    d.f();
//  Error: (Microsoft) 'Derived::f' : function does not take ↩
 ↪ 0 arguments[96]
//  Error: (GCC)       no matching function for call to ↩
 ↪ 'Derived::f()'
    d.Base::f();[97]
```

```
    // ...
```

Uitvoer (nadat code `d.f();` is verwijderd):

```
Ik ben f()
Ik ben f(int)
Ik ben f()
```

De hiding rule vergroot de onderhoudbaarheid van een programma. Stel dat programmeur Bas een base class `Base` heeft geschreven waarin *geen* memberfunctie met de naam `f` voorkomt. Een andere programmeur, Dewi, heeft een class `Derived` geschreven die overerft van de class `Base`. In de class `Derived` is de memberfunctie `f(double)` gedefinieerd. In het hoofdprogramma wordt deze memberfunctie aangeroepen met een `int` als argument. Deze `int` wordt door de conversie regels van C++ automatisch omgezet in een `double`. Zie HideReden.cpp:

```
// Code van Bas
class Base {
public:
    // geen f(...)
};

// Code van Dewi
class Derived: public Base {
public:
    void f(double d) const {
        cout << "Ik ben f(double)" << endl;
    }
};

int main() {
```

[95] De memberfunctie `Derived::f(int)` verbergt (Engels: hides) de memberfunctie `Base::f()`.

[96] De functie `Base::f()` wordt verborgen (Engels: hidden) door de functies `Derived::f(int)`.

[97] Voor degene die echt alles wil weten: De hidden memberfunctie kan nog *wel* aangeroepen worden door gebruik te maken van zijn zogenoemde *qualified name*: `base_class_name::member_function_name`.

```
    Derived d;
    d.f(3);
    // ...
```

Uitvoer:

```
Ik ben f(double)
```

Bas besluit nu om zijn class `Base` uit te breiden en voegt een functie `f` toe:

```
// Aangepaste code van Bas
class Base {
public:
    // ...
    void f(int i) const {
        cout << "Ik ben f(int)" << endl;
    }
};
```

Deze aanpassing van Bas heeft dankzij de hiding rule *geen* invloed op de code van Dewi. De uitvoer van het `main` programma wijzigt niet! Als de hiding rule niet zou bestaan, zou de uitvoer van `main` wel zijn veranderd. De hiding rule zorgt er dus voor dat een toevoeging in een base class geen invloed heeft op code in een derived class. Dit vergroot de onderhoudbaarheid.

De hiding rule zorgt dus voor een betere onderhoudbaarheid maar tegelijkertijd zorgt deze regel ervoor dat overloading en overerving niet goed samengaan. Bij het gebruik van overerving moet je er dus altijd voor zorgen dat je geen functienamen gebruikt die al gebruikt zijn in de classes waar je van overerft.

In paragraaf 4.3 heb je gezien dat een *virtual* gedefinieerde memberfunctie in een derived class overridden kan worden. Een memberfunctie die in een derived class een memberfunctie uit de base class *override* moet dezelfde naam en dezelfde parameters hebben[98]. Alleen memberfuncties van een “ouder” of “voorouder” class

[98] Als de virtual memberfunctie in de base class `const` is, moet de memberfunctie in de derived class die deze memberfunctie uit de base class override ook `const` zijn.

kunnen overridden worden in de “kind” class. Als een overridden memberfunctie via een polymorfe pointer of reference (zie paragraaf 4.2) aangeroepen wordt, dan wordt tijdens het uitvoeren van het programma bepaald naar welk type object de pointer wijst (of de reference verwijst). Pas daarna wordt de in deze class gedefinieerde memberfunctie aangeroepen.

Als er dus twee memberfuncties zijn met dezelfde naam en dezelfde parameters in een base en in een derived class, dan wordt bij een aanroep van de memberfunctienaam de juiste memberfunctie:

- door de compiler aangeroepen door naar het statische type van het object waarop de memberfunctie wordt uitgevoerd te kijken als de memberfuncties niet virtual zijn (er is dan sprake van overloading);
- tijdens run time aangeroepen door naar het dynamische type van het object waarop de memberfunctie wordt uitgevoerd te kijken als de memberfunctie wel virtual zijn (er is dan sprake van overriding).[99]

Voorbeeld van het gebruik van overriding en **verkeerd** gebruik van overloading. De functie `f` wordt overloaded en de functie g wordt overridden (Override.cpp):

```
class Base {
public:
    void f(int i) const {
        cout << "Base::f(int) called." << endl;
    }
    virtual void g(int i) const {
        cout << "Base::g(int) called." << endl;
    }
// ...
};
```

[99] Voor degene die echt alles wil weten: de overridden memberfunctie kan wel aangeroepen worden door gebruik te maken van zijn zogenoemde qualified name: `base_class_name::member_function_name`. Voor degene die echt alles wil begrijpen: een functie met dezelfde parameters in twee classes zonder overervingsrelatie zijn overloaded omdat, bij aanroep, de receivers verschillend zijn.

```
class Derived: public Base {
public:
    void f(int i) const { // f is overloaded!
        cout << "Derived::f(int) called." << endl;
    }
    virtual void g(int i) const { // g is overridden
        cout << "Derived::g(int) called." << endl;
    }
// ...
};

int main() {
    Base b;
    Derived d;
    Base* pb = &d;
    b.f(3);
    d.f(3);
    pb->f(3);
    b.g(3);
    d.g(3);
    pb->g(3);
    pb->Base::g(3);
```

Uitvoer:

```
Base::f(int) called.
Derived::f(int) called.
Base::f(int) called.
Base::g(int) called.
Derived::g(int) called.
Derived::g(int) called.
Base::g(int) called.
```

4.9 Expliciet overridden van memberfuncties

Sinds C++11 kun je *expliciet* aangeven dat je een memberfunctie wilt overridden. Dit voorkomt dat je een functie “per ongeluk” overloadt.

Stel dat de volgende base class is gegeven (AccidentalOverload.cpp):

```
class Base {
public:
    void f(int i) const {
        cout << "Base::f(int) called." << endl;
    }
    virtual void g(int i) const {
        cout << "Base::g(int) called." << endl;
    }
};
```

Een (niet zo snuggere) programmeur wil nu van deze base class een class afleiden en daarbij zowel de functie `f` als de functie g *overridden*:

```
class Derived: public Base {
public:
    void f(int i) const {
        cout << "Derived::f(int) called." << endl;
    }
    virtual void g(int i) {
        cout << "Derived::g(int) called." << endl;
    }
};
```

De programmeur heeft het volgende testprogramma geschreven:

```
int main() {
    Derived d;
    Base* pb = &d;
    pb->f(3);
    pb->g(3);
```

```
    // ...
```

De programmeur, die denkt zowel `f` als `g` overridden te hebben, verwachtte de volgende uitvoer:

```
Derived::f(int) called.
Derived::g(int) called.
```

De werkelijke uitvoer is echter:

```
Base::f(int) called.
Base::g(int) called.
```

Dit komt omdat de programmeur beide functies `f` en `g` *overloaded* heeft in plaats van overridden. De functie `f` is in de base class niet virtual gedefinieerd en kan dus niet overridden worden. De functie `g` in de derived class heeft een andere definitie dan in de basis class (in de basis class is `g` een `const` memberfunctie maar in de derived class ontbreekt het woordje `const`). De functie `g` wordt dus overloaded in plaats van overridden.

Sinds C++11 kunnen we deze problemen voorkomen door expliciet aan te geven dat we een functie willen overridden. Dit kun je doen door het woord `override` achter de declaratie van de memberfunctie te plaatsen.

De class `Derived` kan dan als volgt gedefinieerd worden (ExplicitOverride.cpp):

```
class Derived: public Base {
public:
    void f(int i) const override {
        cout << "Derived::f(int) called." << endl;
    }
    virtual void g(int i) override {
        cout << "Derived::g(int) called." << endl;
    }
};
```

De programmeur geeft hier expliciet aan dat het de bedoeling is om `f` en `g` te overridden. Als deze class vertaald wordt, zal de compiler een foutmelding geven:

```
class Derived: public Base {
public:
    void f(int i) const override {
//  Error: (Microsoft) 'Derived::f' : method with override ↩
 ↪ specifier 'override' did not override any base class ↩
 ↪ methods
//  Error: (GCC)       'void Derived::f(int) const' marked ↩
 ↪ override, but does not override
        cout << "Derived::f(int) called." << endl;
    }
    virtual void g(int i) override {
//  Error: (Microsoft) 'Derived::g' : method with override ↩
 ↪ specifier 'override' did not override any base class ↩
 ↪ methods
//  Error: (GCC)       'virtual void Derived::g(int)' marked ↩
 ↪ override, but does not override
        cout << "Derived::g(int) called." << endl;
    }
};
```

4.10 `final` overriding van memberfuncties

Een functie die in een derived class overridden is kan, in een class die weer afgeleid is van deze class, opnieuw overridden worden. Normaal is dit geen probleem, maar in sommige gevallen is dit ongewenst. Sinds C++11 kunnen we expliciet aangeven dat een functie die overridden is niet verder overridden mag worden. Stel dat we in een bibliotheekapplicatie een basis class `UitleenbaarItem` hebben met een puur virtuele memberfunctie `id` waarmee het id van het betreffende item kan worden opgevraagd. Van deze basis class zijn verschillende classes zoals `Boek` en `DVD` afgeleid elk met een overridden memberfunctie `id`. In de class `Boek` gebruiken we het ISBN nummer als id. Stel dat we niet willen dat deze implementatie van de functie `id` in classes die afgeleid worden van `Boek` overridden

wordt dan kunnen we de functie in de class Boek met het woord final markeren. Het is dan niet meer toegestaan om deze functie (verder) te overridden. Zie FinalOverride.cpp:

```
class UitleenbaarItem {
public:
    virtual string id() const = 0;
};

class DVD: public UitleenbaarItem {
public:
    virtual string id() const {
        // ...
    }
};

class Boek: public UitleenbaarItem {
public:
    virtual string id() const final {
        return ISBN;
    }
private:
    string ISBN;
};
```

Als je nu toch probeert om in een derived class van Boek de memberfunctie id te overridden, dan geeft de compiler een foutmelding:

```
class Reisgids: public Boek {
    virtual string id() const {
//  Error: (Microsoft) 'Boek::id': function declared as ↩
 ↪ 'final' cannot be overridden by 'Reisgids::id'
//  Error: (GCC)       virtual function 'virtual std::string ↩
 ↪ Reisgids::id() const' overriding final function
    }
};
```

4.11 `final` overerving

Soms is het niet de bedoeling dat een class als base class gebruikt wordt. Sinds C++11 kunnen we expliciet aangeven dat je niet kunt overerven van een bepaalde class door in de class declaratie meteen na de naam het woord `final` te gebruiken. Stel dat we in een bibliotheekapplicatie een basis class `UitleenbaarItem` hebben gedefinieerd waar de class `DVD` van afgeleid is. Stel dat we niet willen dat deze class als basis class gebruikt wordt, dan kunnen we de class `DVD` met het woord `final` markeren. Het is dan niet meer toegestaan om van deze class te overerven.

Zie FinalInheritance.cpp:

```
class UitleenbaarItem {
public:
    virtual string id() const = 0;
};

class DVD final: public UitleenbaarItem {
public:
    virtual string id() const {
        string s;
        // ...
        return s;
    }
};
```

Als je nu toch probeert om van de class `DVD` te overerven, dan geeft de compiler een foutmelding:

```
class DisneyDVD: public DVD {
//  Error: (Microsoft) 'DisneyDVD': cannot inherit form 'DVD' ↩
 ↪ as it has been declared as 'final'
//  Error: (GCC)       cannot derive from 'final' base 'DVD' ↩
 ↪ in derived type 'DisneyDVD'
};
```

4.12 Slicing problem

Een object d van een class `Derived`, die public overerft van class `Base`, mag worden toegekend aan een object b van de class `Base` met de toekenning `b = d;`. Er geldt immers: een `Derived` is een `Base`. Evenzo mag een object b van de class `Base` worden geïnitialiseerd met een object d van de class `Derived` door het aanroepen van de copy constructor `Base b(d);`. Deze toekenning en initialisatie leveren problemen op als het object d van de class `Derived` meer geheugenruimte inneemt dan een object b van de class `Base`. Dit is het geval als in class `Derived` (extra) datavelden zijn opgenomen. Deze extra datavelden kunnen niet aan het object van class `Base` toegekend worden omdat het object van class `Base` hier geen ruimte voor heeft. Dit probleem wordt het *slicing problem* genoemd. Het is dus aan te raden om nooit een object van een derived class toe te kennen aan een object van de base class.

Voorbeeld van slicing (Slice.cpp):

```
class Mens {
public:
    Mens(const string& n): name(n) {
    }
    string naam() const {
        return name;
    }
    virtual string soort() const {
        return "mens";
    }
    virtual unsigned int salaris() const {
        return 0;
    }
// ...
private:
    const string name;
};

class Docent: public Mens {
```

```
public:
    Docent(const string& n, unsigned short s): Mens(n), ↩
     ↪ sal(s) {
    }
    virtual string soort() const {
        return "docent";
    }
    virtual unsigned int salaris() const {
        return sal;
    }
    virtual void verhoogSalarisMet(unsigned short v) {
        sal += v;
    }
// ...
private:
    unsigned short sal;
};

int main() {
    Docent bd("Harry", 30000);
    cout << bd.soort() << " " << bd.naam() << " verdient " ↩
     ↪ << bd.salaris() << endl;
    bd.verhoogSalarisMet(10000);
    cout << bd.soort() << " " << bd.naam() << " verdient " ↩
     ↪ << bd.salaris() << endl;

    Mens m(bd); // Waar blijft het salaris?
    cout << m.soort() << " " << m.naam() << " verdient " ↩
     ↪ << m.salaris() << endl;

    Mens& mr(bd);
    cout << mr.soort() << " " << mr.naam() << " verdient " ↩
     ↪ << mr.salaris() << endl;

    Mens* mp(&bd);
    cout << mp->soort() << " " << mp->naam() << " verdient ↩
     ↪ " << mp->salaris() << endl;
```

```
//  ...
```

Uitvoer:

```
docent Harry verdient 30000
docent Harry verdient 40000
mens Harry verdient 0
docent Harry verdient 40000
docent Harry verdient 40000
```

4.13 Voorbeeld: Opslaan van polymorfe objecten in een `vector`

In het voorgaande programma heb je gezien dat alleen pointers en references polymorf kunnen zijn. Als we dus polymorfe objecten willen opslaan, kan dit alleen door pointers of references naar deze objecten op te slaan. Het opslaan van references in een `vector` is echter niet mogelijk omdat een reference altijd moet verwijzen naar een object dat al bestaat en bij het aanmaken van de `vector` weten we nog niet welke objecten in de `vector` moeten worden opgeslagen. De enige mogelijkheid is dus het opslaan van polymorfe pointers naar objecten. Hier volgt een voorbeeld van het gebruik van een `vector` en polymorfisme (Fruitmand.cpp):

```
class Fruit {
public:
    virtual string soort() const = 0;
// ...
};

class Appel: public Fruit {
public:
    virtual string soort() const {
        return "Appel";
    }
// ...
};
```

```
class Peer: public Fruit {
public:
    virtual string soort() const {
        return "Peer";
    }
// ...
};

class FruitMand {
public:
    void voegToe(Fruit& p) {
        fp.push_back(&p);
    }
    void printInhoud() const {
        cout << "De fruitmand bevat:" << endl;
        for (Fruit* e: fp)
            cout << e->soort() << endl;
    }
private:
    vector<Fruit*> fp;
};

int main() {
    FruitMand m;
    Appel a1, a2;
    Peer p1;
    m.voegToe(a1);
    m.voegToe(p1);
    m.voegToe(a2);
    m.printInhoud();
//  ...
```

De uitvoer van dit programma is als volgt:

```
De fruitmand bevat:
Appel
```

```
Peer
Appel
```

Omdat de `vector` nu pointers naar de objecten bevat, moeten we zelf goed opletten dat deze objecten niet verwijderd worden voordat de `vector` wordt verwijderd.

4.14 Voorbeeld: impedantie calculator

In dit uitgebreide praktijkvoorbeeld kun je zien hoe de OOP technieken die je tot nu toe hebt geleerd kunnen worden toegepast bij het maken van een elektrotechnische applicatie.

4.14.1 Weerstand, spoel en condensator

Passieve elektrische componenten (hierna componenten genoemd) hebben een complexe[100] impedantie Z. Deze impedantie is een functie van de frequentie f. Er bestaan 3 soorten (basis)componenten:

- R (weerstand): heeft een weerstandswaarde r uitgedrukt in Ohm. $Z(f) = r$.
- L (spoel): heeft een zelfinductie l uitgedrukt in Henry. $Z(f) = \mathrm{j} \cdot 2 \cdot \pi \cdot f \cdot l$.
- C (condensator): heeft een capaciteit c uitgedrukt in Farad. $Z(f) = -\mathrm{j}/(2 \cdot \pi \cdot f \cdot c)$.

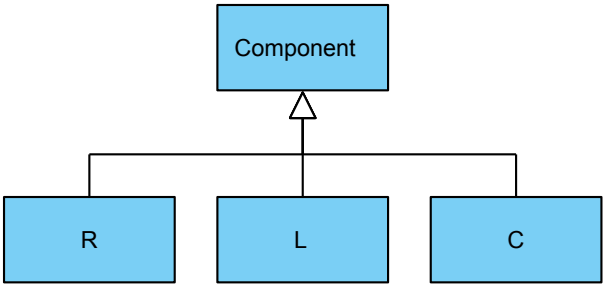
De classes `Component`, `R`, `L` en `C` hebben de volgende relaties (zie ook figuur 4.6):

- een `R` *is een* `Component`;
- een `L` *is een* `Component`;
- een `C` *is een* `Component`.

[100] Voor wie niet weet wat complexe getallen zijn: http://nl.wikipedia.org/wiki/Complex_getal.

We willen een programma maken waarin gebruik gemaakt kan worden van passieve elektrische componenten. De ABC (Abstract Base Class) `Component` kan dan als volgt gedefinieerd worden:

```
class Component {
public:
    virtual ~Component() = default;[101]
    virtual complex<double> Z(double f) const = 0;
    virtual void print(ostream& o) const = 0;
};
```



Figuur 4.6: R, L en C zijn afgeleid van Component

Het type `complex` is opgenomen in de standaard C++ library. De functie `Z` moet in een van `Component` afgeleide class de impedantie berekenen (een complex getal) bij de als parameter gegeven frequentie `f`. De functie `print` moet in een van `Component` afgeleide class het type en de waarde afdrukken op de als parameter gegeven output stream `o`. Bijvoorbeeld: `L(1E-3)` voor een spoel van 1mH.

Als we componenten ook met behulp van de `operator<<` willen afdrukken, moeten we deze operator als volgt overloaden (Netwerk0.cpp):

```
ostream& operator<<(ostream& o, const Component& c) {
    c.print(o);
    return o;
}
```

[101] Dit is een zogenoemde *virtual destructor*, waarom het nodig is om een virtual destructor te definiëren wordt verderop in dit dictaat besproken. Zie paragraaf 5.4.

De classes `R`, `L` en `C` kunnen dan als volgt gebruikt worden:

```
void printImpedanceTable(const Component& c) {
    cout << "Impedantie tabel voor: " << c << endl << endl;
    cout << setw(10) << "freq" << setw(20) << "Z" << endl;
    for (double freq(10); freq < 10E6; freq *= 10)
        cout << setw(10) << freq << setw(20) << c.Z(freq) ↩
         ↪ << endl;
    cout << endl;
}

int main() {
    R r(1E2);
    printImpedanceTable(r);
    cin.get();
    C c(1E-5);
    printImpedanceTable(c);
    cin.get();
    L l(1E-3);
    printImpedanceTable(l);
//  ...
```

Merk op dat de functie `printImpedanceTable` niet “weet” welke `Component` gebruikt wordt. Dit betekent dat deze polymorfe functie voor alle huidige “soorten” componenten te gebruiken is. De functie is zelfs ook voor toekomstige “soorten” componenten bruikbaar. Dit maakt het programma eenvoudig uitbreidbaar.

De uitvoer van het bovenstaande programma is:

```
Impedantie tabel voor: R(100)

      freq                   Z
        10           (100,0)
       100           (100,0)
      1000           (100,0)
     10000           (100,0)
```

```
    100000             (100,0)
    1e+006             (100,0)

Impedantie tabel voor: C(1e-005)

      freq                   Z
        10        (0,-1591.55)
       100        (0,-159.155)
      1000        (0,-15.9155)
     10000        (0,-1.59155)
    100000       (0,-0.159155)
    1e+006      (0,-0.0159155)

Impedantie tabel voor: L(0.001)

      freq                   Z
        10       (0,0.0628319)
       100        (0,0.628319)
      1000         (0,6.28319)
     10000         (0,62.8319)
    100000         (0,628.319)
    1e+006         (0,6283.19)
```

Vraag:

Implementeer nu zelf de classes `R`, `C` en `L`.

Antwoord:

```
class R: public Component { // R = Weerstand
public:
    R(double r): r(r) {
    }
```

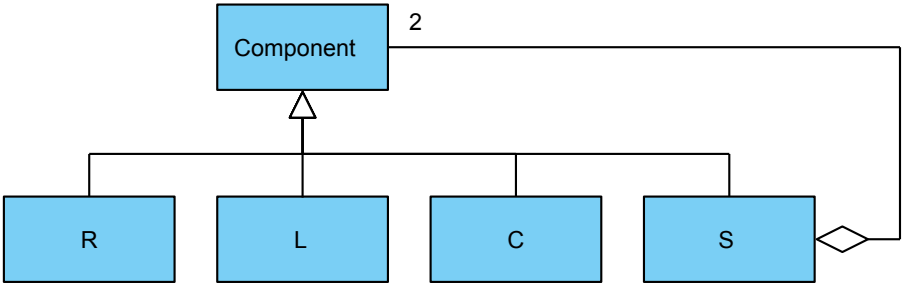
```
    virtual complex<double> Z(double) const {
        return r;
    }
    virtual void print(ostream& o) const {
        o << "R(" << r << ")";
    }
private:
    double r;
};

class L: public Component { // L = Spoel
public:
    L(double l): l(l) {
    }
    virtual complex<double> Z(double f) const {
        return complex<double>(0, 2 * PI * f * l);
    }
    virtual void print(ostream& o) const {
        o << "L(" << l << ")";
    }
private:
    double l;
};

class C: public Component { // C = Condensator
public:
    C(double c): c(c) {
    }
    virtual complex<double> Z(double f) const {
        return complex<double>(0, -1 / (2 * PI * f * c));
    }
    virtual void print(ostream& o) const {
        o << "C(" << c << ")";
    }
private:
    double c;
};
```

4.14.2 Serie- en parallelschakeling

Natuurlijk wil je het programma nu uitbreiden zodat je ook de impedantie van serieschakelingen kunt berekenen. Je moet jezelf dan de volgende vragen stellen:

- Is een serieschakeling een component?
- Heeft een serieschakeling een (of meer) component(en)?

Dat een serieschakeling *bestaat uit* componenten zal voor iedereen duidelijk zijn. Het antwoord op de eerste vraag is misschien moeilijker. De ABC `Component` is gedefinieerd als "iets" dat een impedantie heeft en dat geprint kan worden. Als je bedenkt dat een serieschakeling ook een impedantie heeft en ook geprint kan worden, zal duidelijk zijn dat een serieschakeling een soort component *is*. De class `S` (serieschakeling) moet dus van de class `Component` afgeleid worden maar bestaat ook uit andere componenten. Zie figuur 4.7. Dit heeft als bijkomend voordeel dat je het aantal componenten waaruit een serieschakeling bestaat tot 2 kunt beperken. Als je bijvoorbeeld een serieschakeling wilt doorrekenen van een `R`, `L` en `C`, dan maak je eerst van de `R` en `L` een serieschakeling, die je vervolgens met de `C` combineert tot een tweede serieschakeling. Op soortgelijke wijze kun je het programma uitbreiden met parallelschakelingen. Met dit programma kun je dan van elk passief elektrisch netwerk de impedantie berekenen.



Figuur 4.7: `S` is afgeleid van `Component` en `S` heeft 2 `Components`.

De classes `S` en `P` kunnen dan als volgt gebruikt worden (Netwerk1.cpp):

```
int main() {
```

```
    R r1(1E2);
    C c1(1E-6);
    L l1(3E-2);
    S s1(r1, c1);
    S s2(r1, l1);
    P p(s1, s2);
    printImpedanceTable(p);
    // ...
```

Je ziet dat je de al bestaande polymorfe functie `printImpedanceTable` ook voor objecten van de nieuwe classes `S` en `P` kunt gebruiken!

De uitvoer van het bovenstaande programma is:

```
Impedantie tabel voor: ((R(100)+C(1e-006))//(R(100)+L(0.03)))

      freq                  Z
        10   (100.016,1.25659)
       100   (101.591,12.5146)
      1000  (197.893,-14.3612)
     10000  (101.132,-10.5795)
    100000    (100.011,-1.061)
    1e+006     (100,-0.106103)
```

Vraag:

Implementeer nu zelf de classes `S` en `P`.

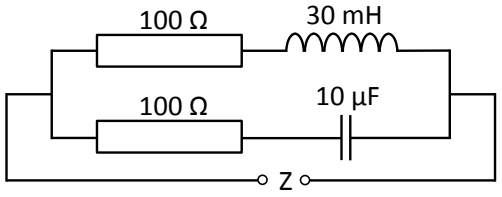
Antwoord:

```
class S: public Component { // S = Serie schakeling van ↩
 ↪ twee componenten
public:
    S(const Component& c1, const Component& c2): c1(c1), ↩
     ↪ c2(c2) {
    }
```

```
    virtual complex<double> Z(double f) const {
        return c1.Z(f) + c2.Z(f);
    }
    virtual void print(ostream& o) const {
        o << "(" << c1 << "+" << c2 << ")";
    }
private:
    const Component& c1;
    const Component& c2;
};

class P: public Component { // P = Parallel schakeling van ↩
 ↪ twee componenten
public:
    P(const Component& c1, const Component& c2): c1(c1), ↩
     ↪ c2(c2) {
    }
    virtual complex<double> Z(double f) const {
        return (c1.Z(f) * c2.Z(f)) / (c1.Z(f) + c2.Z(f));
    }
    virtual void print(ostream& o) const {
        o << "(" << c1 << "//" << c2 << ")";
    }
private:
    const Component& c1;
    const Component& c2;
};
```

4.14.3 Een grafische impedantie calculator

Op http://bd.eduweb.hhs.nl/sopx2/prog/impcalc vind je een Windows applicatie waarmee de absolute waarde van de impedantie van een in te voeren netwerk als functie van de frequentie grafisch weergegeven kan worden. Dit programma (her)gebruikt de hierboven gedefinieerde classes `R`, `L`, `C`, `S` en `P`. Op de website http://bd.eduweb.hhs.nl/ogoprg/pract/opdr5.htm kun je, als je dat leuk vindt, zelf leren hoe je grafische applicaties kunt maken.



Figuur 4.8: Een passief netwerk.

Het netwerk dat gegeven is in figuur 4.8 moet als volgt worden ingevoerd:

```
(R(100)+L(3E-2))//(R(100)+C(1E-6))
```

4.15 Inheritance details

Over inheritance valt nog veel meer te vertellen:

- Private en protected inheritance. Een manier van overerven waarbij de *is-een* relatie niet geldt. Kan meestal vervangen worden door composition (*heeft-een*).
- Multiple inheritance. Overerven van meerdere classes tegelijk.
- Virtual inheritance. Speciale vorm van inheritance nodig om bepaalde problemen bij multiple inheritance op te kunnen lossen.

Voor al deze details verwijs ik je naar [2, hoofdstuk 14 en hoofdstuk 15] en [9, hoofdstuk 20 en 21].

5

Dynamic memory allocation en destructors

In dit hoofdstuk worden de volgende onderwerpen besproken:

- Dynamic memory allocation;
- Destructor.

Deze technieken worden vervolgens toegepast bij het maken van een ADT `Array`. Aan het einde van het hoofdstuk zal ik laten zien hoe je er voor kunt zorgen dat je deze ADT ook met een range-based for kunt doorlopen en met een initialisatielijst kunt initialiseren.

5.1 Dynamische geheugen allocatie (`new` en `delete`)

Je bent gewend om variabelen globaal of lokaal te definiëren. De geheugenruimte voor globale variabelen wordt gereserveerd zodra het programma start en pas weer vrijgegeven bij het beëindigen van het programma. De geheugenruimte voor een lokale variabele wordt, zodra die variabele gedefinieerd wordt, gereserveerd op de stack. Als het blok (compound statement), waarin de variabele gedefini-

eerd is, wordt beëindigd, dan wordt de gereserveerde ruimte weer vrijgegeven. Vaak wil je zelf bepalen wanneer ruimte voor een variabele gereserveerd wordt en wanneer deze ruimte weer vrijgegeven wordt. In een programma met een GUI (grafische gebruikers interface) wil je bijvoorbeeld een variabele (object) aanmaken voor elk window dat de gebruiker opent. Deze variabele kan weer worden vrijgegeven zodra de gebruiker dit window sluit. Het geheugen dat bij het openen van het window was gereserveerd kan dan (her)gebruikt worden bij het openen van een (ander) window.

Dit kan in C++ met de operatoren `new` en `delete`. Voor het dynamisch aanmaken en verwijderen van array's (waarvan de grootte dus tijdens het uitvoeren van het programma bepaald kan worden) beschikt C++ over de operatoren `new[]` en `delete[]`. De operatoren `new` en `new[]` geven een pointer naar het nieuw gereserveerde geheugen terug. Deze pointer kan dan gebruikt worden om dit geheugen te gebruiken. Als dit geheugen niet meer nodig is, kan dit worden vrijgegeven door de operator `delete` of `delete[]` uit te voeren op de pointer die bij `new` of respectievelijk `new[]` is teruggegeven. De met `new` aangemaakte variabelen bevinden zich in een speciaal geheugengebied *heap* genaamd. Tegenover het voordeel van dynamische geheugenallocatie, een grotere flexibiliteit, staat het gevaar van een geheugenlek (memory leak). Een geheugenlek ontstaat als een programmeur vergeet een met `new` aangemaakte variabele weer met `delete` te verwijderen.

In C werden de functies `malloc` en `free` gebruikt om geheugenruimte op de heap te reserveren en weer vrij te geven. Deze functies zijn echter niet type-safe (type veilig) omdat het return type van `malloc` een `void*` is die vervolgens door de gebruiker naar het gewenste type moet worden omgezet. De compiler merkt het dus niet als de gebruikte type aanduidingen niet overeenkomen. Om deze reden zijn in C++ nieuwe memory allocatie operatoren (`new` en `delete`) toegevoegd.

Voorbeeld met `new` en `delete`:

```
double* dp(new double); // reserveer een double
int i; cin >> i;
double* drij(new double[i]); // reserveer een array ↩
 ↪ met i doubles
```

```
    // ...
    delete dp; // geef de door dp aangewezen ↩
     ↪ geheugenruimte vrij
    delete[] drij; // idem voor de door drij aangewezen array
```

In paragraaf 3.8 heb je gezien hoe, door het gebruik van een `std::vector` in plaats van het gebruik van een statische array, geen grens gesteld hoeft te worden aan het aantal elementen in een array. De enige grens is dan de grootte van het beschikbare (virtuele) werkgeheugen. De `std::vector` is geïmplementeerd met behulp van van dynamische geheugenallocatie.

5.2 Destructor ~Breuk

Een class kan naast een aantal constructors (zie paragraaf 2.4) ook één *destructor* hebben. De destructor heeft als naam, de naam van de class voorafgegaan door het teken ~. Als programmeur hoef je niet zelf de destructor aan te roepen. De compiler zorgt ervoor dat de destructor aangeroepen wordt net voordat het object opgeruimd wordt. Dit is:

- aan het einde van het blok waarin de variabele gedefinieerd is voor een lokale variabele;
- aan het einde van het programma voor globale variabele;
- bij het aanroepen van `delete` voor dynamische variabelen.

Als voorbeeld zullen we aan de eerste versie van de class `Breuk` (zie paragraaf 2.3) een destructor toevoegen:

```
class Breuk {
public:
    Breuk();
    Breuk(int t);
    Breuk(int t, int n);
    ~Breuk();
    // rest van de class Breuk is niet gewijzigd.
};
```

```
Breuk::~Breuk() {
    cout << "Een breuk met de waarde " << boven << "/" << ↩
     ↪ onder
         << " is verwijderd uit het geheugen." << endl
         << "Druk op enter om verder te gaan..." << endl;
    cin.get();
}
```

Het hoofdprogramma om deze versie van Breuk te testen is (zie pagina 53):

```
int main() {
    Breuk b1(4);
    cout << "b1(4) = " << b1.teller() << '/' << ↩
     ↪ b1.noemer() << endl;
    Breuk b2(23, -5);
    cout << "b2(23, -5) = " << b2.teller() << '/' << ↩
     ↪ b2.noemer() << endl;
    Breuk b3(b2);
    cout << "b3(b2) = " << b3.teller() << '/' << ↩
     ↪ b3.noemer() << endl;
    b3.abs();
    cout << "b3.abs() = " << b3.teller() << '/' << ↩
     ↪ b3.noemer() << endl;
    b3 = b2;
    cout << "b3 = b2 = " << b3.teller() << '/' << ↩
     ↪ b3.noemer() << endl;
    b3.plus(5);
    cout << "b3.plus(5) = " << b3.teller() << '/' << ↩
     ↪ b3.noemer() << endl;
    cin.get();
    return 0;
}
```

De uitvoer van dit programma (Breuk1_destructor.cpp) is nu:

```
b1(4) = 4/1
b2(23, -5) = -23/5
b3(b2) = -23/5
b3.abs() = 23/5
b3 = b2 = -23/5
Een breuk met de waarde 5/1 is verwijderd uit het geheugen.
Druk op enter om verder te gaan...

b3.plus(5) = 2/5

Een breuk met de waarde 2/5 is verwijderd uit het geheugen.
Druk op enter om verder te gaan...

Een breuk met de waarde -23/5 is verwijderd uit het geheugen.
Druk op enter om verder te gaan...

Een breuk met de waarde 4/1 is verwijderd uit het geheugen.
Druk op enter om verder te gaan...
```

Zoals je ziet worden de in de functie `main` aangemaakte lokale objecten `b1`, `b2` en `b3` aan het einde van de functie `main` uit het geheugen verwijderd. De volgorde van verwijderen is *omgekeerd* ten opzichte van de volgorde van aanmaken. Dit is niet toevallig maar hier is door de ontwerper van C++ goed over nagedacht. Stel dat we aan het begin van een functie 4 objecten van de class `Wiel` aanmaken en vervolgens een object van de class `Auto` aanmaken waarbij we aan de constructor van `Auto` de 4 `Wiel` objecten meegeven. Bij het verlaten van deze functie worden de objecten dan in omgekeerde volgorde uit het geheugen verwijderd[102]. In de destructor `~Auto` (die wordt aangeroepen net voordat de geheugenruimte van het `Auto` object wordt vrijgegeven) kunnen we het `Auto` object nog gewoon naar de

[102] Bij het uitvoeren van een programma worden de lokale variabelen aangemaakt op de stack. Omdat een stack de LIFO (Last In First Out) volgorde gebruikt worden de laatst aangemaakte lokale variabelen weer als eerste van de stack verwijderd.

autosloperij rijden. We weten zeker dat de `Wiel` objecten nog niet zijn verwijderd omdat het `Auto` object later is aangemaakt, en dus eerder wordt verwijderd. Iets uit elkaar halen gaat ook altijd in de omgekeerde volgorde dan iets in elkaar zetten dus het is logisch dat het opruimen van objecten in de omgekeerde volgorde van het aanmaken gaat.

We zien dat halverwege de functie `main` ook nog een `Breuk` object wordt verwijderd. Dit lijkt op het eerste gezicht wat vreemd. Als we goed kijken, zien we dat deze destructor wordt aangeroepen bij het uitvoeren van de volgende regel uit `main`:

```
b3.plus(5);
```

Snap je welk `Breuk` object aan het einde van deze regel wordt verwijderd? Lees indien nodig paragraaf 2.6 nog eens door. De memberfunctie `plus` verwacht een `Breuk` object als argument. De integer `5` wordt met de constructor `Breuk(5)` omgezet naar een `Breuk` object. Dit impliciet door de compiler aangemaakte object wordt dus aan het einde van de regel (nadat de memberfunctie `plus` is aangeroepen en teruggekeerd) weer uit het geheugen verwijderd.

Als geen destructor gedefinieerd is, wordt door de compiler een *default destructor* aangemaakt. Deze default destructor roept voor elk dataveld de destructor van dit veld aan (*memberwise destruction*). In dit geval heb ik de destructor `~Breuk` een melding op het scherm laten afdrukken. Dit is in feite nutteloos en ik had net zo goed de door de compiler gegenereerde default destructor kunnen gebruiken.

5.3 Destructors bij inheritance

In paragraaf 4.5 hebben we gezien dat als een constructor van de derived class aangeroepen wordt, automatisch *eerst* de constructor (zonder parameters) van de base class wordt aangeroepen. Als de destructor van de derived class automatisch (door de compiler) wordt aangeroepen, dan wordt *daarna* automatisch de

destructor van de base class aangeroepen. De base class destructor wordt altijd als *laatste* uitgevoerd. Snap je waarom?[103]

5.4 Virtual destructor

Als een class nu of in de toekomst als base class gebruikt wordt, moet de destructor virtual zijn zodat van deze class afgeleide classes via een polymorfe pointer "deleted" kunnen worden.

Hier volgt een voorbeeld van het gebruik van een ABC en polymorfisme (Fruitmand_destructor.cpp):

```
class Fruit {
public:
    virtual ~Fruit() {
        cout << "Er is een stuk Fruit verwijderd." << endl;
    }
    virtual string soort() const = 0;
// ...
};

class Appel: public Fruit {
public:
    virtual ~Appel() {
        cout << "Er is een Appel verwijderd." << endl;
    }
    virtual string soort() const {
        return "Appel";
    }
// ...
};

class Peer: public Fruit {
public:
```

[103] Omdat geheugenruimte in de omgekeerde volgorde van aanmaken moet worden vrijgegeven.

```
    virtual ~Peer() {
        cout << "Er is een Peer verwijderd." << endl;
    }
    virtual string soort() const {
        return "Peer";
    }
// ...
};

class FruitMand {
public:
    ~FruitMand() {
        for (Fruit* e: fp)
            delete e;
    }
    void voegToe(Fruit* p) {
        fp.push_back(p);
    }
    void printInhoud() const {
        cout << "De fruitmand bevat:" << endl;
        for (Fruit* e: fp)
            cout << e->soort() << endl;
    }
private:
    vector<Fruit*> fp;
};

int main() {
    {
        FruitMand m;
        m.voegToe(new Appel);
        m.voegToe(new Peer);
        m.voegToe(new Appel);
        m.printInhoud();
    } // hier wordt de Fruitmand m verwijderd!
    // ...
```

De uitvoer van dit programma is als volgt:

```
De fruitmand bevat:
Appel
Peer
Appel
Er is een Appel verwijderd.
Er is een stuk Fruit verwijderd.
Er is een Peer verwijderd.
Er is een stuk Fruit verwijderd.
Er is een Appel verwijderd.
Er is een stuk Fruit verwijderd.
```

Als in de base class `Fruit` *geen* virtual destructor gedefinieerd wordt maar een gewone (non-virtual) destructor, wordt de uitvoer als volgt:

```
De fruitmand bevat:
Appel
Peer
Appel
Er is een stuk Fruit verwijderd.
Er is een stuk Fruit verwijderd.
Er is een stuk Fruit verwijderd.
```

Dit komt doordat de destructor via een polymorfe pointer (zie paragraaf 4.2) aangeroepen wordt. Als de destructor virtual gedefinieerd is, wordt tijdens het uitvoeren van het programma bepaald naar welk type object (een appel of een peer) deze pointer wijst. Vervolgens wordt de destructor van deze class (`Appel` of `Peer`) aangeroepen[104]. Omdat de destructor van een derived class ook altijd de destructor van zijn base class aanroept (zie paragraaf 5.3) wordt de destructor van `Fruit` ook aangeroepen. Als de destructor niet virtual gedefinieerd is, wordt tijdens het compileren van het programma bepaald van welk type de pointer is. Vervolgens

[104] De destructor is dan dus *overridden*.

wordt de destructor van deze class (`Fruit`) aangeroepen[105]. In dit geval wordt dus alleen de destructor van de base class aangeroepen.

Let op! Als we een class aanmaken zonder destructor, zal de compiler zelf een zogenoemde *default destructor* aanmaken. Deze automatisch aangemaakte destructor is echter *niet* virtual. In elke class die nu of in de toekomst als base class gebruikt wordt moeten we dus zelf een virtual destructor declareren. Sinds C++11 is het niet meer nodig om deze destructor ook zelf te definiëren, achter de declaratie kan de code `= default` gebruikt worden om aan te geven dat de default implementatie van de destructor, zie pagina 170 gebruikt moet worden.

5.5 Voorbeeld class `Array`

Het in C ingebouwde array type heeft een aantal nadelen en beperkingen. De belangrijkste daarvan zijn:

- De grootte van de array moet bij het compileren van het programma bekend zijn. In de praktijk komt het vaak voor dat pas bij het uitvoeren van het programma bepaald kan worden hoe groot de array moet zijn.
- Er vindt bij het gebruiken van de array geen controle plaats op de gebruikte index. Als je element N benadert uit een array die $N-1$ elementen heeft, krijg je geen foutmelding maar wordt de geheugenplaats "achter" het einde van de array benaderd. Dit is vaak de oorzaak van fouten die lang onopgemerkt kunnen blijven en dan (volgens de wet van Murphy op het moment dat het het slechtst uitkomt) plotseling voor de dag kunnen komen. Deze fouten zijn vaak heel moeilijk te vinden omdat de oorzaak van de fout niets met het gevolg van de fout te maken heeft.

In dit voorbeeld zul je zien hoe ik zelf een eigen array type genaamd `Array` heb gedefinieerd[106] waarbij:

[105] De destructor is dan dus *overloaded*.

[106] In de standaard C++ library is sinds C++11 ook een type `std::array` opgenomen, zie paragraaf 3.7. Dit type is enigszins vergelijkbaar met mijn zelfgemaakte type `Array`. Het grootste verschil is dat de size van de array bij de `std::array` als template argument wordt meegegeven

- de grootte pas tijdens het uitvoeren van het programma bepaald kan worden;
- bij het indexeren de index wordt gecontroleerd en een foutmelding wordt gegeven als de index zich buiten de grenzen van de `Array` bevindt.

De door de compiler gegenereerde copy constructor, destructor en `operator=` blijken voor de class `Array` niet correct te werken. Ik heb daarom voor deze class zelf een copy constructor, destructor en `operator=` gedefinieerd. Na het voorbeeldprogramma Array.cpp worden de belangrijkste aspecten toegelicht en worden verwijzingen naar verdere literatuur gegeven.

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

class Array {
public:
    explicit Array(int s);
    Array(const Array& r);
    Array& operator=(const Array& r);
    ~Array();
    int& operator[](int index);
    const int& operator[](int index) const[107];
    int length() const;
    bool operator==(const Array& r) const;
    bool operator!=(const Array& r) const;
private:
    int size;
    int* data;
```

en bij mijn `Array` als constructor argument wordt meegegeven. De grootte (Engels: size) van een `std::array` moet dus tijdens het compileren van het programma (compile time) bekend zijn maar de size van een `Array` kan tijdens het uitvoeren van het programma (run time) bepaald worden. Wat dat betreft lijkt mijn `Array` dus meer op een `std::vector` die in paragraaf 3.8 is behandeld. Het verschil tussen een `std::vector` en mijn `Array` is dat een `std::vector` nadat hij is aangemaakt nog kan groeien en krimpen. Mijn `Array` kan dat niet.

```
};

Array::Array(int s): size(s), data(new int[s]) {
}

Array::Array(const Array& r): size(r.size), data(new ↩
 ↪ int[r.size]) {
    for (int i = 0; i < size; ++i)
        data[i] = r.data[i];
}

Array& Array::operator=(const Array& r) {
    Array t(r);
    std::swap(data, t.data);
    std::swap(size, t.size);
    return *this;
}

Array::~Array() {
    delete[] data;
}

int& Array::operator[](int index) {
    assert(index >= 0 && index < size);
    return data[index];
}

const int& Array::operator[](int index) const {
    assert(index >= 0 && index < size);
    return data[index];
}

int Array::length() const {
    return size;
}

bool Array::operator==(const Array& r) const {
```

```
    if (size != r.size)
    return false;
    for (int i = 0; i < size; ++i)
        if (data[i] != r.data[i])
            return false;
    return true;
}

bool Array::operator!=(const Array& r) const {
    return !(*this == r);
}

int main() {
    cout << "Hoeveel elementen moet de Array bevatten? ";
    int i;
    cin >> i;
    if (i > 0) {
        Array a(i);
        for (int j = 0; j < a.length(); ++j)
            a[j] = j * j; // vul a met kwadraten
        Array b(a);
        cout << "b = " << b << endl;
        cout << "a[12] = " << a[12] << endl;
        cout << "b[12] = " << b[12] << endl;
        a[0] = 4;
        cout << "a[0] = " << a[0] << endl;

        if (a == b)
            cout << "a is nu gelijk aan b." << endl;
        else
            cout << "a is nu ongelijk aan b." << endl;

        b = a;
        cout << "b = a is uitgevoerd." << endl;

        if (a != b)
            cout << "a is nu ongelijk aan b." << endl;
```

```
        else
            cout << "a is nu gelijk aan b." << endl;
    }
    else
        cout << "Doe niet zo negatief!" << endl;
    cin.get();
    cin.get();
    return 0;
}
```

5.6 `explicit` constructor

De constructor `Array(int)` is `explicit` gedeclareerd om te voorkomen dat de compiler deze constructor gebruikt om een `int` automatisch om te zetten naar een `Array`. Zie paragraaf 2.6.

5.7 Copy constructor en default copy constructor

Een copy constructor wordt gebruikt als een object gekopieerd moet worden. Dit is het geval als:

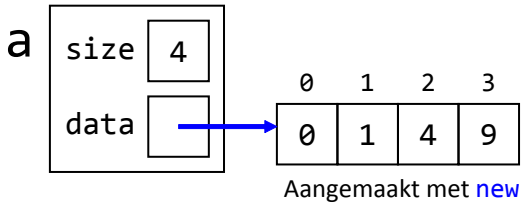
- een object geïnitialiseerd wordt met een object van dezelfde class;

[107] Het overloaden van de `operator[]` is noodzakelijk omdat ik vanuit deze operator een reference terug wil geven zodat met deze reference in het `Array` object geschreven kan worden bijvoorbeeld `v[12] = 144;`. Als ik deze operator als `const` zou hebben gedefinieerd (wat in eerste instantie logisch lijkt, de `operator[]` verandert immers niets in het `Array` object), dan zou deze operator ook gebruikt kunnen worden voor een `const Array` object. Met de teruggegeven reference kun je dan in een `const Array` object schrijven en dat is natuurlijk niet de bedoeling. Om deze reden heb ik de `operator[]` niet als `const` gedefinieerd. Dit heeft tot gevolg dat de `operator[]` niet meer gebruikt kan worden voor `const Array` objecten. Dit is weer teveel van het goede want nu kun je ook niet meer lezen met behulp van `operator[]` uit een `const Array` object bijvoorbeeld `i = v[12];`. Om het lezen uit een `const Array` object toch weer mogelijk te maken heb ik naast de non-const `operator[]` nog een `const operator[]` gedefinieerd. Deze `const operator[]` geeft een `const` reference terug en zoals je weet kan een `const` reference alleen gebruikt worden om te lezen. (Als je deze voetnoot na één keer lezen begrijpt, is er waarschijnlijk iets niet helemaal in orde). Zie Array_operator[].cpp voor een voorbeeldprogramma.

- een object als argument wordt doorgegeven aan een value parameter van een functie;
- een object als waarde wordt teruggegeven vanuit een functie.

De compiler zal als de programmeur geen copy constructor definieert zelf een default copy constructor genereren. Deze default copy constructor kopieert elk deel waaruit de class bestaat vanuit de een naar de andere (= memberwise copy). Naast de default copy constructor genereert de compiler ook (indien niet door de programmeur gedefinieerd) een default assignment operator en een default destructor. De default assignment operator doet een memberwise assignment en de default destructor doet een memberwise destruction.

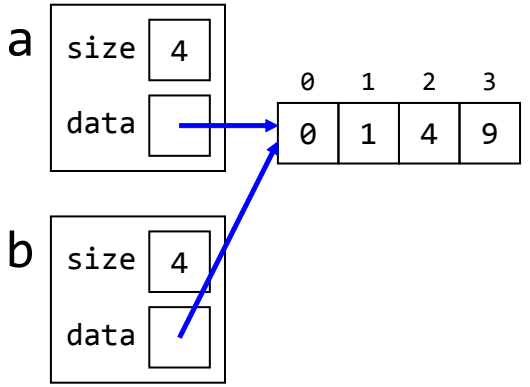
Dat je voor de class `Array` zelf een destructor moet definiëren waarin je het in de constructor met `new` gereserveerde geheugen met `delete` weer vrij moet geven zal niemand verbazen. Dat je voor de class `Array` zelf een copy constructor en `operator=` moet definiëren ligt misschien minder voor de hand. Ik zal eerst bespreken wat het probleem is bij de door de compiler gedefinieerde default copy constructor en `operator=`. Daarna zal ik bespreken hoe we zelf een copy constructor en een operator kunnen declareren en implementeren. Een `Array` a met vier elementen gevuld met kwadraten is schematisch weergegeven in figuur 5.1.



Figuur 5.1: Object a van de class `Array`.

De door de compiler gegenereerde copy constructor zal een memberwise copy uitvoeren. De datavelden `size` en `data` worden dus gekopieerd. Als je de `Array` a naar de `Array` b kopieert door middel van het statement `Array b(a);`[108], ontstaat de in figuur 5.2 weergegeven situatie.

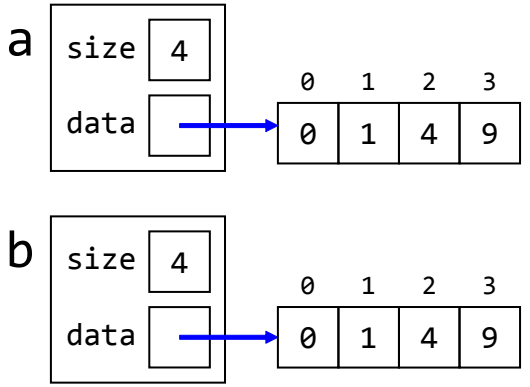
[108] Dit statement kan ook als volgt geschreven worden `Array b = a;`. Ook in dit geval wordt de copy constructor van de class `Array` aangeroepen en dus niet de `operator=` memberfunctie. Het



Figuur 5.2: Het effect van de default copy constructor.

Dit is niet goed omdat als je nu de kopie wijzigt (bijvoorbeeld `b[2] = 8;`) dan zal ook het origineel (`a[2]`) gewijzigd zijn en dat is natuurlijk niet de bedoeling.

De gewenste situatie na het kopiëren van de `Array` `a` naar de `Array` `b` is gegeven in figuur 5.3. Om deze situatie te bereiken moeten je zelf de copy constructor van de class `Array` declareren: `Array::Array(const Array&);`



Figuur 5.3: Het gewenste effect van de copy constructor.

gebruik van het = teken bij een initialisatie is verwarrend omdat het lijkt alsof er een assignment wordt gedaan terwijl in werkelijkheid de copy constructor wordt aangeroepen. Om deze reden raad ik je aan om bij initialisatie altijd de notatie `Array w(v);` of `Array w{v};` te gebruiken.

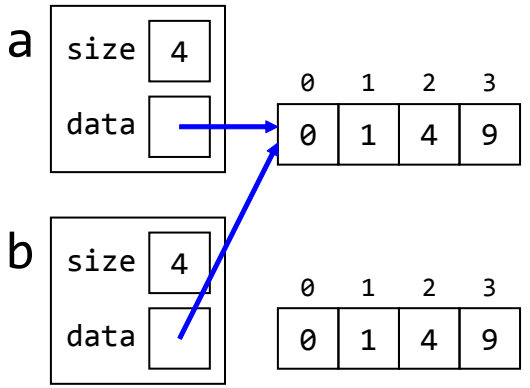
De copy constructor kan eenvoudig als volgt gedefinieerd worden:

```
Array::Array(const Array& r): size(r.size), data(new ↩
 ↪ int[r.size]) {
    for (int i = 0; i < size; ++i)
        data[i] = r.data[i];
}
```

De parameter van de copy constructor moet een `Array&` zijn en kan geen `Array` zijn. Als je namelijk een `Array` als (call by value) parameter gebruikt, moet een kopietje worden gemaakt bij aanroep van de copy constructor, maar daarvoor moet de copy constructor worden aangeroepen, waarvoor een kopietje moet worden gemaakt, maar daarvoor moet de copy constructor . . . enz.

5.8 Overloading `operator=`

Voor de door de compiler gegenereerde assignment operator geldt ongeveer hetzelfde verhaal als voor de door de compiler gegenereerde copy constructor. Na het statement `b = a;` zal de situatie zoals gegeven in figuur 5.4 ontstaan. Je ziet dat de door de compiler gegenereerde assignment operator niet alleen onjuist werkt maar bovendien een memory leak veroorzaakt.



Figuur 5.4: Het effect van de default assignment operator.

Als je de assignment operator voor de class `Array` zelf wilt definiëren, moet je de memberfunctie `operator=` declareren.

Deze memberfunctie kun je dan als volgt implementeren:

```
Array& Array::operator=(const Array& r) {
    Array t(r);
    std::swap(data, t.data);
    std::swap(size, t.size);
    return *this;
}
```

Voor het return type van `operator=` heb ik `Array&` gebruikt. Dit zorgt ervoor dat assignment operatoren achter elkaar "geregen" kunnen worden, bijvoorbeeld: `c = b = a` zie paragraaf 2.19. De implementatie van de `operator=` moet de `Array r` toekennen aan de receiver. In de implementatie van de `operator=` is gebruik gemaakt van de standaard functie `swap` die als volgt gedefinieerd is in de include file `<algorithm>`:

```
template <class T>
void swap (T& a, T& b) {
    T tmp(a);
    a = b;
    b = tmp;
}
```

De `operator=` begint met het maken van een kopie van de als argument meegegeven `Array r`. Vervolgens wordt de `data` (pointer) en de `size` van de receiver en `t` verwisseld. De nieuwe waarde van de receiver wordt dus gelijk aan `r`. Na het uitvoeren van het `return` statement wordt het object `t` verwijderd. Hierdoor wordt de oude waarde van de receiver opgeruimd. Bedenk dat `t` door de verwisseling de oude waarde van de receiver bevat! Waarschijnlijk moet je deze paragraaf een paar keer lezen om het helemaal te vatten.

Voor de liefhebbers:

Vraag:

Is de onderstaande implementatie van `operator=` correct? Verklaar je antwoord!

```
Array& Array::operator=(Array r) {
    std::swap(data, r.data);
    std::swap(size, r.size);
    return *this;
}
```

Antwoord:

Ja! De copy constructor wordt nu impliciet aangeroepen bij het doorgeven van het bij aanroep gebruikte argument aan de parameter `r` (call by value).

5.9 Wanneer moet je zelf een destructor, copy constructor en `operator=` definiëren?

Een class moet een *zelf* gedefinieerde copy constructor, `operator=` en destructor bevatten als:

- die class een pointer bevat en;
- als bij het kopiëren van een object van de class niet de pointer, maar de data waar de pointer naar wijst moet worden gekopieerd en;
- als bij een toekenning aan een object van de class niet de pointer, maar de data waar de pointer naar wijst moet worden toegekend en;
- als bij het vrijgeven van de geheugenruimte van een object van de class niet de pointer, maar de data waar de pointer naar wijst moet worden vrijgegeven (door middel van `delete`).

Dit betekent dat de class `Breuk` geen zelfgedefinieerde assignment operator, geen zelfgedefinieerde copy constructor en ook geen zelfgedefinieerde destructor nodig heeft. De class `Array` heeft wel een zelfgedefinieerde assignment operator, een zelfgedefinieerde copy constructor en ook een zelfgedefinieerde destructor nodig.

In de vorige paragraaf heb ik het zelfgedefinieerde type `Array` besproken. Dit type heeft een aantal voordelen ten opzichte van het ingebouwde array type. De belangrijkste zijn dat het aantal elementen pas tijdens het uitvoeren van het programma bepaald wordt en dat bij het gebruik van de `operator[]` de index wordt gecontroleerd. Het zelfgedefinieerde type `Array` heeft ten opzichte van het ingebouwde array type als nadeel dat de elementen alleen maar van het type `int` kunnen zijn. Als je een `Array` met elementen van het type `double` nodig hebt, kun je natuurlijk gaan kopiëren, plakken, zoeken en vervangen maar daar zitten weer de bekende nadelen aan (elke keer als je code kopieert wordt de onderhoudbaarheid van die code slechter). Als je verschillende versies van `Array` met de hand genereert, moet je bovendien elke versie een andere naam geven omdat een class naam uniek moet zijn. In plaats daarvan kun je ook het template mechanisme gebruiken om een `Array` met elementen van het type `T` te definiëren, waarbij het type `T` pas bij het gebruik van de class template `Array` wordt bepaald. Bij het gebruik van de class template `Array` kan de compiler niet zelf bepalen wat het type `T` moet zijn. Vandaar dat je dit bij het gebruik van de class template `Array` zelf moet specificeren.

Bijvoorbeeld:

```
//  ...
    Array<Breuk> vb(300); // een Array met 300 breuken.
```

5.10 Voorbeeld class template `Array`

Zie Array_template.cpp:

```
#include <iostream>
#include <cmath>
```

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

template <typename T> class Array {
public:
    explicit Array(int s);
    Array(const Array<T>& v);
    Array<T>& operator=(const Array<T>& r);
    ~Array();
    T& operator[](int index);
    const T& operator[](int index) const;
    int length() const;
    bool operator==(const Array<T>& r) const;
    bool operator!=(const Array<T>& r) const;
private:
    int size;
    T* data;
};

template <typename T> Array<T>::Array(int s): size(s), ↩
 ↪ data(new T[s]) {
}

// ... enz. ...

int main() {
    cout << "Hoeveel elementen moet de Array bevatten? ";
    int i; cin >> i;
    if (i > 0) {
        Array<double> v(i);
        for (int j = 0; j < v.length(); ++j)
            v[j] = sqrt(double(j)); // Vul v met wortels
        cout << "v[12] = " << v[12] << endl;
        Array<int> w(i);
        for (int t = 0; t < w.length(); ++t)
            w[t] = t * t;           // Vul w met kwadraten
        cout << "w[12] = " << w[12] << endl;
```

```
    }
    else
        cout << "Doe niet zo negatief!" << endl;
    // ...
```

Een template kan ook meerdere parameters hebben. Een template-parameter kan in plaats van een typename parameter ook een normale parameter zijn. Zo zou je ook de volgende class template kunnen definiëren:

```
template <typename T, int size> class FixedArray[109] {
    // ...
private:
    T data[size];
}
```

Deze class template kan dan als volgt gebruikt worden:

```
//  ...
    FixedArray<Breuk, 300> vb; // Een Array met 300 breuken.
```

Er zijn grote verschillen tussen deze class template `FixedArray` en de eerder gedefinieerde class template `Array`:

- De `size` moet bij de laatste template tijdens het compileren bekend zijn. De compiler genereert (instantieert) namelijk de benodigde versie van de class template en vult daarbij de opgegeven template argumenten in.
- De compiler genereert een nieuwe versie van de class telkens als deze class gebruikt wordt met andere template argumenten. Dit betekent dat `FixedArray<int, 3> a` en `FixedArray<int, 4> b` verschillende types zijn. Dus expressies zoals `a != b` en `a = b` enz. zijn dan niet toegestaan. Telkens als je een `FixedArray` met een andere `size` definieert, wordt er weer een nieuw type met bijbehorende machinecode voor alle memberfuncties gege-

[109] Deze `FixedArray` lijkt erg veel op de sinds C++11 in de standaard opgenomen `std::array`, zie paragraaf 3.7.

nereerd[110]. Bij de class template `Array` wordt maar één versie aangemaakt als de variabelen `Array<int> a(3)` en `Array<int> b(4)` gedefinieerd worden. De expressies `a != b` en `a = b` enz. zijn dan wel toegestaan.

5.11 Ondersteuning range-based for voor class template `Array`

De in paragraaf 5.10 gegeven class template Array kan niet met behulp van een range-based for (zie paragraaf 1.6) doorlopen worden. Als we dit toch proberen dan krijgen we een foutmelding:

```
for (auto e: v) {
//  Error: (Microsoft) no callable 'begin' function found ↩
 ↪ for type 'Array<int>'
//  Error: (GCC)       'begin' was not declared in this ↩
 ↪ scope
```

Als we dit willen laten werken moeten we de functies `begin`, `cbegin`, `end` en `cend` voor de class template `Array` definiëren. De functies `begin` en `end` moet overloaded worden voor read-only (`const`) objecten van de class `Array` en voor non-`const` objecten van de class `Array`. De functies `cbegin` en `cend` moeten enkel gedefinieerd worden voor read-only (`const`) objecten van de class `Array`. De functies `begin` en `cbegin` moeten een pointer[111] teruggeven naar het eerste element van de `Array`. De functies `end` en `cend` moeten een pointer[111] teruggeven net na het laatste element van de `Array`.

Zie Array_template_rbf.cpp:

```
template <typename T> class Array {
```

[110] Dit is niet de waarheid. Een memberfunctie van een class template wordt niet gegenereerd als de template geïnstantieerd wordt maar pas als de compiler daadwerkelijk een aanroep naar de betreffende memberfunctie moet vertalen. Dit is hier echter niet van belang.

[111] In het algemeen moeten deze functies een zogenoemde iterator teruggeven, zie voor meer informatie het dictaat *Algoritmen en datastructuren*[1, hoofdstuk 6].

```
public:
    explicit Array(int s);

// ... enz. ...

    T* begin();
    const T* begin() const;
    const T* cbegin() const;
    T* end();
    const T* end() const;
    const T* cend() const;

// ... enz. ...

};

template <typename T> T* Array<T>::begin() {
    return data;
}

template <typename T> const T* Array<T>::begin() const {
    return data;
}

template <typename T> const T* Array<T>::cbegin() const {
    return data;
}

template <typename T> T* Array<T>::end() {
    return data + size;
}

template <typename T> const T* Array<T>::end() const {
    return data + size;
}
```

```
template <typename T> const T* Array<T>::cend() const {
    return data + size;
}
```

5.12 Ondersteuning initialisatielijst voor class template Array

De in paragraaf 5.10 gegeven class template Array kan niet met behulp van een initialisatielijst geïnitialiseerd worden. Als we dit toch proberen dan krijgen we een foutmelding:

```
    Array<int> v = {1, 2, 3};
//  Error: (Microsoft) 'initializing': cannot convert ↩
    ↪ from 'initializer-list' to 'Array<int>'
//  Error: (GCC)       could not convert '{1, 2, 3}' from ↩
    ↪ '<brace-enclosed initializer list>' to 'Array<int>'
```

Als we dit willen laten werken moeten we een extra constructor toevoegen aan de class Array.

Zie Array_template_rbf.cpp:

```
template <typename T> class Array {
public:
    explicit Array(int s);
    Array(initializer_list<T> list);

// ... enz. ...

};

template <typename T> Array<T>::Array(initializer_list<T> ↩
 ↪ list): size(list.size()), data(new T[size]) {
    auto listIter = list.begin();
    for (int i = 0; i < size; ++i) {
```

```
        data[i] = *listIter++;
    }
}
```

6

Losse flodders

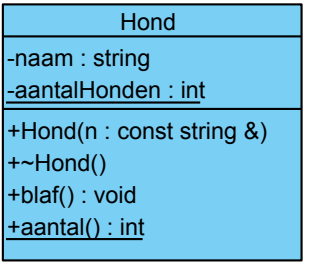
In dit hoofdstuk worden nog enkele onderwerpen besproken die niets met elkaar te maken hebben en die ik niet in een van de voorafgaande hoofdstukken wilde opnemen.

6.1 `static` class members

Je hebt geleerd dat elk object zijn eigen datavelden heeft terwijl de memberfuncties door alle objecten van een bepaalde class gedeeld worden. Stel nu dat je wilt tellen hoeveel objecten van een bepaalde class "levend" zijn. Dit zou kunnen door een globale teller te definiëren die in de constructor van de class met 1 wordt verhoogd en in de destructor weer met 1 wordt verlaagd. Het gebruik van een globale variabele maakt het programma echter slecht onderhoudbaar.

Een `static` dataveld is een onderdeel van de class en wordt door alle objecten van de class gedeeld. Z'n `static` dataveld kan bijvoorbeeld gebruikt worden om het aantal "levende" objecten van een class te tellen. In UML worden `static` datavelden en `static` memberfuncties onderstreept weergegeven zoals in figuur 6.1 te zien is. Zie HondenTeller.cpp:

```
#include <iostream>
#include <string>

using namespace std;

class Hond {
public:
    Hond(const string& n);
    ~Hond();
    void blaf() const;
    static int aantal();
private:
    string naam;
    static int aantalHonden;
};

int Hond::aantalHonden = 0;

Hond::Hond(const string& n): naam(n) {
    ++aantalHonden;
}
Hond::~Hond() {
    --aantalHonden;
}
int Hond::aantal() {
    return aantalHonden;
}
void Hond::blaf() const {
    cout << naam << " zegt: WOEF" << endl;
}

int main() {
    cout << "Er zijn nu " << Hond::aantal() << " honden." ↩
     ↪ << endl;
    {
        Hond h1("Boris");
```



Figuur 6.1: Class Hond met `static` members.

```
        h1.blaf();
        cout << "Er zijn nu " << Hond::aantal() << " ↩
         ↪ honden." << endl;
        Hond h2("Fikkie");
        h2.blaf();
        cout << "Er zijn nu " << Hond::aantal() << " ↩
         ↪ honden." << endl;
    }
    cout << "Er zijn nu " << Hond::aantal() << " honden." ↩
     ↪ << endl;
    cin.get();
    return 0;
}
```

Uitvoer:

```
Er zijn nu 0 honden.
Boris zegt: WOEF
Er zijn nu 1 honden.
Fikkie zegt: WOEF
Er zijn nu 2 honden.
Er zijn nu 0 honden.
```

Je kunt een `static` memberfunctie op twee manieren aanroepen:

- via een object van de class: `object_naam.memberfunctie_naam(parameters);`
- via de classnaam: `class_naam::memberfunctie_naam(parameters)`.

De laatste methode heeft de voorkeur omdat die ook gebruikt kan worden als er nog geen objecten zijn.

Een `static` memberfunctie heeft ten opzichte van een “normale” memberfunctie de volgende beperkingen:

- Een `static` memberfunctie heeft geen receiver (ook niet als hij via een object aangeroepen wordt).

- Een `static` memberfunctie heeft dus geen `this` pointer.
- Een `static` memberfunctie kan dus geen "gewone" memberfuncties aanroepen en ook geen "gewone" datavelden gebruiken.

6.2 Namespaces

Bij grote programma's kunnen verschillende classes "per ongeluk" dezelfde naam krijgen. In C++ kun je classes (en functies etc.) groeperen in zogenoemde *namespaces*:

```
namespace Bd {
   void f(int);
   double sin(double x);
}

// andere file zelfde namespace:
namespace Bd {
   class string {
   //  ...
   };
}

// andere namespace:
namespace Vi {
   class string {
   //  ...
   };
}
```

Je ziet dat in de namespace `Bd` een functie `sin` is opgenomen die ook in de standaard library is opgenomen. De in de namespace `Bd` gedefinieerde class `string` is ook in de namespace `Vi` en ook in de standaard library gedefinieerd. Je hebt in paragraaf 1.10 gezien dat alle functies en classes uit de standaard library in de namespace `std` zijn opgenomen. Je kunt met de *scope-resolution* operator `::` aangeven uit welke namespace je een class of functie wilt gebruiken:

```
Bd::string s1 = "Harry";
Vi::string s2 = "John";
std::string s3 = "Standard";
```

In het bovenstaande codefragment worden drie objecten gedefinieerd van drie verschillende classes. Het object `s1` is van de class `string` die gedefinieerd is in de namespace `Bd`, het object `s2` is van de class `string` die gedefinieerd is in de namespace `Vi` en het object `s3` is van de class `string` die gedefinieerd is in de namespace `std` (de standaard library). Je ziet dat je met behulp van namespaces classes die toevallig dezelfde naam hebben toch in 1 programma kunt combineren. namespaces maken dus het hergebruik van code eenvoudiger.

Als je in een stuk code steeds de `string` uit de namespace `Bd` wilt gebruiken, kun je dat opgeven met behulp van een *using declaration*.

```
using Bd::string;
string s4 = "Hallo";
string s5 = "Dag";
```

De objecten `s4` en `s5` zijn nu beide van de class `string` die gedefinieerd is in de namespace `Bd`. De usingdeclaratie blijft net zolang geldig als een gewone variabeledeclaratie. Tot de bijbehorende accolade sluiten dus.

Als je in een stuk code steeds classes en functies uit de namespace `Bd` wilt gebruiken, kun je dat opgeven met behulp van een *using directive*.

```
using namespace Bd;
string s6 = "Hallo";
double d = sin(0,785398163397448);
```

Het object `s6` is nu van de class `string` die gedefinieerd is in de namespace `Bd`. De functie `sin` die wordt aangeroepen is nu de in de namespace `Bd` gedefinieerde functie. De using directive blijft net zolang geldig als een gewone variabeledeclaratie. Tot de bijbehorende accolade sluiten dus.

6.3 Read-only pointers met const

Je hebt in paragraaf 1.8 gezien hoe je in C++ de const qualifier kunt gebruiken om read-only variabelen (met een constante waarde) te definiëren.

Voorbeeld met const:

```
const int aantalRegels = 80;
```

De qualifier const kan op verschillende manieren bij pointers gebruikt worden.

6.3.1 const *

```
int i = 3;
const int j = 4;
const int* p = &i;[112]
```

Dit betekent: p wijst naar i en je kunt i via p alleen lezen. Je kunt i dus *niet* wijzigen via p. Dit is zinvol als je een pointer als argument aan een functie wilt meegeven en als je niet wilt dat de variabele waar deze pointer naar wijst in de functie gewijzigd wordt. Je gebruikt dan als parameter een const T* in plaats van een T*. Dat de functie de variabele waar de pointer naar wijst niet mag wijzigen zouden we natuurlijk ook gewoon kunnen afspreken (en bijvoorbeeld in het commentaar van de functie vermelden) maar door deze afspraak expliciet als een const-declaratie vast te leggen kan deze afspraak door de compiler gecontroleerd worden. Dit vermindert de kans op het maken van fouten bij het implementeren of wijzigen van de functie. Let op: je kunt i zelf wel rechtstreeks wijzigen!

```
const int* p = &i;
// p wijst naar i en je kunt i niet via p wijzigen.
// Let op: je kunt i zelf wel rechtstreeks wijzigen!
i = 4;
*p = 5;
```

[112] De notatie int const* p = &i; heeft dezelfde betekenis maar wordt in de praktijk zelden gebruikt.

```
//  Error: (Microsoft) 'p' : you cannot assign to a variable ↩
 ↪ that is const
//  Error: (GCC)       assignment of read-only location '* p'
    p = &j;
```

De pointer p is in dit geval zelf *geen* read-only variabele. Zoals je hierboven ziet, kun je p wijzigen en bijvoorbeeld naar de variabele j laten wijzen. Merk op dat de variabele j een read-only variabele is, dat is geen probleem omdat j via p niet gewijzigd kan worden. Een gewone `int*` kun je *niet* naar j laten wijzen omdat j niet gewijzigd mag worden. Een `const int*` kan dus wijzen naar een `int` en naar een `const int` maar een `int*` kan alleen maar naar een `int` wijzen.

6.3.2 * const

```
int* const q = &i;
```

Dit betekent: q wijst naar i en je mag q alleen lezen. Je kan q dus nergens anders meer naar laten wijzen. Dit is zinvol als je een pointer altijd naar dezelfde variabele wilt laten wijzen. Let op: je kunt i wel via q (of rechtstreeks) wijzigen.

```
    int* const q = &i;
    // q wijst naar i en je kunt q nergens anders meer ↩
     ↪ naar laten wijzen.
    // Let op: je kunt i wel via q (of rechtstreeks) ↩
     ↪ wijzigen.
    i = 4;
    *q = 5;
    q = &j;
//  Error: (Microsoft) 'q' : you cannot assign to a variable ↩
 ↪ that is const
//  Error: (GCC)       assignment of read-only variable 'q'
```

De pointer q is in dit geval een read-only variabele.

6.3.3 const * const

```
const int* const r = &i;
```

Dit betekent: r wijst naar i, je kunt i via r alleen lezen en je kunt r alleen lezen. Je kunt i niet via r wijzigen en je kunt r nergens anders meer naar laten wijzen. Let op: je kunt i zelf wel wijzigen!

```
    const int* const r = &i;
    // r wijst naar i en je kunt i niet via r wijzigen en ↩
     ↪ je kunt r nergens anders meer naar laten wijzen.
    // Let op: je kunt i zelf wel rechtstreeks wijzigen!
    i = 4;
    *r = 5;
//  Error: (Microsoft) 'r' : you cannot assign to a variable ↩
 ↪ that is const
//  Error: (GCC)       assignment of read-only location ↩
 ↪ '*(const int*)r'
    r = &j;
//  Error: (Microsoft) 'r' : you cannot assign to a variable ↩
 ↪ that is const
//  Error: (GCC)       assignment of read-only variable 'r'
```

6.4 Initialiseren van datavelden

Sinds C++11[113] mogen datavelden ook direct geïnitialiseerd worden. Als dezelfde datavelden ook in een member initialization list (zie paragraaf 2.4) worden geïnitialiseerd, wordt de directe initialisatie genegeerd.
Zie Breuk1DirectMemberInit.cpp:

```
class Breuk {
public:
    Breuk();
```

[113] Deze zogenoemde *non-static data member initializers* zijn beschikbaar in Microsoft Visual Studio vanaf versie 2013 en in GCC vanaf versie 4.7.

```
    Breuk(int t);
    Breuk(int t, int n);
    int teller() const;
    int noemer() const;
private:
    int boven = 0; // directe initialisatie
    int onder = 1; // directe initialisatie
    void normaliseer();
};

Breuk::Breuk() { // geen initialization list
}

Breuk::Breuk(int t): boven(t) { // de datamember onder ↩
 ↪ wordt niet in de initialization list geïnitialiseerd
}

Breuk::Breuk(int t, int n): boven(t), onder(n) {
    normaliseer();
}

//  ...
int main() {
    Breuk b1;
    cout << "b1 = " << b1.teller() << '/' << b1.noemer() ↩
     ↪ << endl;
    Breuk b2(4);
    cout << "b2(4) = " << b2.teller() << '/' << ↩
     ↪ b2.noemer() << endl;
    Breuk b3(23, -5);
    cout << "b3(23, -5) = " << b3.teller() << '/' << ↩
     ↪ b3.noemer() << endl;
    cin.get();
    return 0;
}
```

Uitvoer:

```
b1 = 0/1
b2(4) = 4/1
b3(23, -5) = -23/5
```

6.5 Compile time constanten in een class

Met behulp van het keyword `constexpr`[114] kun je globale compile time constanten definiëren. Globale constanten komen de onderhoudbaarheid niet ten goede. Als je alle objecten van een class dezelfde compile time constante wilt laten delen, kun je deze constante `static`[115] definiëren.

Voorbeeld met `static` compile time constanten (Color.cpp):

```
class Color {
public:
    Color();
    Color(int c);
    int getValue() const;
    void setValue(int c);
//  constanten:
    static constexpr int BLACK = 0x00000000;
    static constexpr int RED = 0x00FF0000;
    static constexpr int YELLOW = 0x00FFFF00;
    static constexpr int GREEN = 0x0000FF00;
    static constexpr int LIGHTBLUE = 0x0000FFFF;
    static constexpr int BLUE = 0x000000FF;
    static constexpr int PURPER = 0x00FF00FF;
    static constexpr int WHITE = 0x00FFFFFF;
//  ...
private:
```

[114] Het C++11 keyword `constexpr` wordt helaas nog niet ondersteund door Microsoft Visual Studio 2013. Zie paragraaf 6.6 voor een alternatief.

[115] Zie paragraaf 6.1

```
    int value;
};

ostream& operator<<(ostream& o, Color c) {
    return o << setw(6) << setfill('0') << hex << ↩
     ↪ c.getValue();
}

Color::Color(): value(BLACK) {
}

Color::Color(int v): value(v) {
}

int Color::getValue() const {
    return value;
}

void Color::setValue(int v) {
    value = v;
}
```

Deze constanten kunnen als volgt gebruikt worden:

```
    Color c(Color::YELLOW);
    cout << "c = " << c << endl;
    c.setValue(Color::BLUE);
    cout << "c = " << c << endl;
```

Uitvoer:

```
c = ffff00
c = 0000ff
```

6.6 Read-only variabelen in een class

Als het keyword `constexpr` nog niet ondersteund wordt door je compiler, dan kun je in plaats van compile time constanten ook read-only variabelen gebruiken. Je moet dan in het voorbeeld uit paragraaf 6.5 het woord `constexpr` vervangen door `const`, zie Color.cpp.

Als alternatief kun je ook een `enum` gebruiken.

Voorbeeld met anonymous (naamloze) `enum`:

```
class Color {
public:
    Color();
    Color(int c);
    int getValue() const;
    void setValue(int c);
//  constanten:
    enum {
        BLACK = 0x00000000, RED = 0x00FF0000,
        YELLOW = 0x00FFFF00, GREEN = 0x0000FF00,
        LIGHTBLUE = 0x0000FFFF, BLUE = 0x000000FF,
        PURPER = 0x00FF00FF, WHITE = 0x00FFFFFF
//  ...
    };
private:
    int value;
};

ostream& operator<<(ostream& o, Color c) {
    return o << setw(6) << setfill('0') << hex << ↩
     ↪ c.getValue();
}

Color::Color(): value(BLACK) {
}
```

```
Color::Color(int v): value(v) {
}

int Color::getValue() const {
    return value;
}

void Color::setValue(int v) {
    value = v;
}
```

Deze constanten kunnen als volgt gebruikt worden:

```
    Color c(Color::YELLOW);
    cout << "c = " << c << endl;
    c.setValue(Color::BLUE);
    cout << "c = " << c << endl;
```

Uitvoer:

```
c = ffff00
c = 0000ff
```

6.7 Inline memberfuncties

Sommige programmeurs denken dat het gebruik van eenvoudige memberfuncties zoals `teller` en `noemer` te veel vertraging opleveren in hun applicatie. Dit is meestal ten onrechte! Maar voor die gevallen waar het echt nodig is biedt C++ de mogelijkheid om functies (ook memberfuncties) als *inline* te definiëren. Dit betekent dat de compiler een aanroep naar deze functie niet vertaalt naar een "jump to subroutine" machinecode maar probeert om de machinecode waaruit de functie bestaat rechtstreeks op de plaats van aanroep in te vullen. (Net zoals macro's in assembler.) Dit heeft (mogelijk) wel tot gevolg dat de code omvangrijker wordt. Memberfuncties mogen ook in de classdeclaratie gedefinieerd worden. Ze zijn dan "vanzelf" inline. Als de memberfunctie in de classdeclaratie alleen

maar gedeclareerd is, kan de definitie van die functie voorafgegaan worden door het keyword `inline` om de functie inline te maken. Veel compilers zullen als je aangeeft het programma voor snelheid te willen optimaliseren korte functies automatisch inline plaatsen.

Eerste manier om de memberfuncties `teller` en `noemer` inline te maken:

```
class Breuk {
//  ...
    int teller() const {
        return boven;
    }
    int noemer() const  {
        return onder;
    };
//  ...
};
```

Tweede manier om de memberfuncties `teller` en `noemer` inline te maken:

```
class Breuk {
//  ...
    int teller() const;
    int noemer() const;
//  ...
};

inline int Breuk::teller() const {
    return boven;
}
inline int Breuk::noemer() const {
    return onder;
}
```

6.8 Compile time functies

Stel dat je een functie hebt geschreven om n^m te bereken waarbij n en m gehele positieve getallen zijn, je bent er daarbij vanuit gegaan dat het resultaat past in een `unsigned int`:

```
unsigned int pow(unsigned int n, unsigned int m) {
    unsigned int result = 1;
    for (unsigned int i = 0; i < m; ++i) {
        result *= n;
    }
    return result;
}
```

Deze functie kun je bijvoorbeeld als volgt aanroepen:

```
    cout << "3^5 = " << pow(3, 5) << endl;
```

De uitvoer van deze regel is dan:

```
3^5 = 243
```

Als je een compile time constante wilt definiëren met de waarde 3^5 dan kun je die niet initialiseren met de zojuist besproken functie pow.

```
    constexpr unsigned int c = pow(3, 5);
//  Error: (GCC) call to non-constexpr function 'unsigned ↩
 ↪ int pow(unsigned int, unsigned int)'
```

Nu kun je natuurlijk je rekenmachine pakken en berekenen dat 3^5 gelijk is aan 243. De compile time constante kan dan als volgt gedefinieerd worden:

```
    constexpr unsigned int c = 243; // 3^5
```

Nu is in commentaar weergegeven hoe deze compile time constante berekend is. Deze code is echter niet goed onderhoudbaar. Als de compile time constante bijvoorbeeld gewijzigd moet worden in 4^6, dan moet je eerst uitrekenen dat dit

gelijk is aan 4096 en vervolgens moet je 243 vervangen door 4096 en tot slot moet je niet vergeten om ook het commentaar aan te passen. Het zou natuurlijk veel beter zijn voor de onderhoudbaarheid als de compiler deze waarde zelf zou kunnen berekenen. Sinds C++11 is dat inderdaad mogelijk door de functie vooraf te laten gaan door het keyword `constexpr`. Zo'n functie wordt een compile time functie genoemd omdat een aanroep naar deze functie niet tijdens het uitvoeren van het programma maar tijdens het vertalen van het programma wordt uitgevoerd. In C++11 mag een compile time functie alleen bestaan uit een enkel `return`-statement. We kunnen de bovenstaande functie `pow` dus in C++11 niet simpel als compile time functie definiëren door er `constexpr` voor te zetten:

```
constexpr unsigned int cpow(unsigned int n, unsigned int ↩
 ↪ m) {
//  Error: (GCC) body of constexpr function 'constexpr ↩
 ↪ unsigned int powc(unsigned int, unsigned int)' not a ↩
 ↪ return-statement
    unsigned int result = 1;
    for (unsigned int i = 0; i < m; ++i) {
        result *= n;
    }
    return result;
}
```

In C++14 werkt dit wel omdat veel beperkingen voor compile time functies daar zijn opgeheven[116].

Door gebruik te maken van de operator `?:` en recursie is het echter wel mogelijk om in C++11 een compile time functie te definiëren die n^m kan bereken:

```
constexpr unsigned int cpowr(unsigned int n, unsigned int ↩
 ↪ m) {
    return m == 0 ? 1 : n * cpowr(n, m - 1);
}
```

[116] Dit wordt helaas nog niet ondersteund door GCC 4.9.2. Ook niet als de optie `-std=c++14` wordt gebruikt.

De compile time constante kan nu als volgt gedefinieerd worden (ConstexprFunctie.cpp):

```
    constexpr unsigned int c = cpowr(3, 5);
```

6.9 Dynamic binding werkt niet in constructors en destructors

Als in een memberfunctie een andere memberfunctie van dezelfde class wordt aangeroepen en als die laatste memberfunctie virtual gedefinieerd is, dan wordt een message verstuurd naar de receiver van de eerste memberfunctie. Bij het zoeken van de method voor deze message wordt eerst gekeken in de class van de receiver en daarna in de base class van de receiver enz. Dit wordt *dynamic binding* genoemd. Hieronder is een voorbeeld van dynamic binding gegeven (DynamicBinding.cpp):

```
class Fruit {
public:
    virtual void print(ostream& o) {
        o << soort() << endl;
    }
private:
    virtual string soort() const {
        return "Fruit";
    }
//  ...
};

class Appel: public Fruit {
private:
    virtual string soort() const {
        return "Appel";
    }
//  ...
};
```

```
class Peer: public Fruit {
private:
    virtual string soort() const {
        return "Peer";
    }
//  ...
};

int main() {
    Fruit f;
    f.print(cout);
    Appel a;
    a.print(cout);
    Peer p;
    p.print(cout);
//  ...
```

Uitvoer:

```
Fruit
Appel
Peer
```

Laten we eens kijken hoe de regel `a.print(cout);` wordt uitgevoerd. De message `print(cout)` wordt verstuurd naar het object `a`. Er wordt in de class van de receiver (de class `Appel`) gekeken of er een method `print` is gedefinieerd. Dit is niet het geval. Vervolgens wordt in de base class van de class `Appel` (de class `Fruit`) gekeken of er een method `print` is gedefinieerd. Dit is het geval en de betreffende method heeft de juiste parameter. Deze method `Fruit::print` wordt dus uitgevoerd. In deze method wordt de message `soort()` verstuurd naar de receiver. De receiver van de method `print` is het object `a`. Er wordt dus in de class van de receiver (de class `Appel`) gekeken of er een method `soort` is gedefinieerd. Dit is het geval. De method `Appel::soort` wordt dus uitgevoerd en de string `"Appel"` wordt teruggegeven. Deze string wordt vervolgens in de method `Fruit::print` afgedrukt. De method die wordt uitgevoerd bij de ontvangst van

een message is dus afhankelijk van de receiver en wordt tijdens run time "opgezocht"[117], dit wordt *dynamic binding* (ook wel *late binding*) genoemd. Maar dit wist je natuurlijk allemaal al lang (lees paragraaf 2.1 en paragraaf 4.2 nog maar eens door).

In een constructor en in een destructor werkt dynamic binding echter niet! Op het eerste gezicht is dit vreemd maar als je er goed over nadenkt dan blijkt het toch logisch te zijn. Dit wordt uitgelegd aan de hand van het volgende voorbeeld. Als we in het bovenstaande programma de class `Fruit` als volgt wijzigen (DynamicBindingInConstructorEnDestructor.cpp):

```
class Fruit {
public:
    Fruit() {
        cout << "Er is een " << soort() << " aangemaakt." ↩
         ↪ << endl;
    }
    virtual ~Fruit() {
        cout << "Er is een " << soort() << " verwijderd." ↩
         ↪ << endl;
    }
    virtual void print(ostream& o) {
        o << soort() << endl;
    }
private:
    virtual string soort() const {
        return "Fruit";
    }
};
```

[117] De method wordt niet echt opgezocht tijdens run time. Late binding wordt in C++ geïmplementeerd door middel van een zogenoemde virtuele functietabel die uit pointers naar memberfuncties bestaat. Elke class (die virtuele functies bevat) heeft zo'n virtuele functietabel en elk object van zo'n class bevat een (verborgen) pointer naar deze tabel. Via deze tabel met memberfunctiepointers kan late binding op een zeer snelle manier (zonder te zoeken) worden geïmplementeerd.

dan wordt de uitvoer:

```
Er is een Fruit aangemaakt.
Fruit
Er is een Fruit aangemaakt.
Appel
Er is een Fruit aangemaakt.
Peer
Er is een Fruit verwijderd.
Er is een Fruit verwijderd.
Er is een Fruit verwijderd.
```

Deze uitvoer is anders dan je (uitgaande van dynamic binding) zou verwachten. Toch is deze uitvoer goed te verklaren.

Bij het aanmaken van een object a van de class `Appel` door middel van `Appel a;` wordt namelijk eerst de constructor van de base class (de class `Fruit`) aangeroepen. De constructor van de derived class (de class `Appel`) is nog niet uitgevoerd. Je zou dus kunnen zeggen dat de appel nog niet af is, het is al wel een stuk fruit maar het heeft nog niet de specifieke eigenschappen van een appel. Het is dus logisch dat vanuit de constructor `Fruit::soort` wordt aangeroepen en niet `Appel::soort` want het betreffende object is (nog) geen appel. Stel dat de memberfunctie `Appel::soort` zou worden aangeroepen vanuit de constructor `Fruit::Fruit` dan zou die memberfunctie gebruik kunnen maken van nog niet geïnitialiseerde datavelden van de class `Appel`, de constructor van `Appel` is namelijk nog niet uitgevoerd. Het zal duidelijk zijn dat dit ongewenst gedrag kan veroorzaken en daarom vermeden moet worden. Als vanuit de constructor van een class een message wordt verstuurd, wordt geen dynamic binding maar *static binding* toegepast. Dat wil zeggen dat de memberfunctie die in dezelfde class zelf is gedefinieerd wordt aangeroepen.

Bij het verwijderen van een object a van de class `Appel` wordt eerst de destructor van de derived class (de class `Appel`) uitgevoerd. Daarna wordt pas de destructor van de base class (de class `Fruit`) aangeroepen. Je zou dus kunnen zeggen dat de appel dan al (gedeeltelijk) verwijderd is, het is nog wel een stuk fruit maar

het heeft niet meer de specifieke eigenschappen van een appel. Het is dus logisch dat vanuit de destructor `Fruit::soort` wordt aangeroepen en niet `Appel::soort` want het betreffende object is geen appel (meer). Stel dat de memberfunctie `Appel::soort` zou worden aangeroepen vanuit de destructor `Fruit::~Fruit` dan zou die memberfunctie gebruik kunnen maken van al verwijderde datavelden van de class `Appel`, de destructor van `Appel` is namelijk al uitgevoerd. Het zal duidelijk zijn dat dit ongewenst gedrag kan veroorzaken en daarom vermeden moet worden. Als vanuit de destructor van een class een message wordt verstuurd, wordt geen dynamic binding maar *static binding* toegepast. Dat wil zeggen dat de memberfunctie die in dezelfde class zelf is gedefinieerd wordt aangeroepen.

6.10 Exceptions

Vaak zal in een functie of memberfunctie gecontroleerd worden op uitzonderlijke situaties (fouten). De volgende functie berekent de impedantie van een condensator van `c` Farad bij een frequentie van `f` Hz. Zie ImpedanceC1.cpp:

```
complex<double> impedanceC(double c, double f) {
    return complex<double>(0, -1 / (2 * PI * f * c));
}
```

Deze functie kan als volgt aangeroepen worden om de impedantie van een condensator van 1 mF bij 1 kHz op het scherm af te drukken:

```
    cout << impedanceC(1e-6, 1e3) << endl;
```

Als deze functie aangeroepen wordt om de impedantie van een condensator van 0 F uit te rekenen, verschijnt de volgende vreemde uitvoer[118]:

```
(0,-1.#INF)
```

De waarde `-1.#INF` staat voor negatief oneindig. Ook het berekenen van de impedantie van een condensator bij 0 Hz veroorzaakt dezelfde vreemde uitvoer. Het

[118] De uitvoer `(0,-1.#INF)` verschijnt als je het programma onder Windows draait. Als je het onder Linux draait, is de uitvoer `(0,-inf)`.

zal voor iedereen duidelijk zijn dat zulke vreemde uitvoer tijdens het uitvoeren van het programma voorkomen moet worden.

6.10.1 Het gebruik van assert

Op voetnote 34 heb je al kennis gemaakt met de standaard functie `assert`. Deze functie doet niets als de, als argument meegegeven, expressie `true` oplevert maar breekt het programma met een passende foutmelding af als dit niet zo is. Je kunt zogenoemde *assertions* gebruiken om tijdens de ontwikkeling van het programma te controleren of aan een bepaalde voorwaarden (waarvan je "zeker" weet dat ze geldig zijn) wordt voldaan. Je kunt de functie `impedanceC` voorzien van een assertion (ImpedanceC2.cpp):

```
complex<double> impedanceC(double c, double f) {
    assert(c != 0.0 && f != 0.0);
    return complex<double>(0, -1 / (2 * PI * f * c));
}
```

Als je nu tijdens het debuggen deze functie aanroept om de impedantie van een condensator van 0 F uit te rekenen (of om de impedantie van een condensator bij 0 Hz uit te rekenen), zal de `assert` functie het programma beëindigden. De volgende foutmelding verschijnt:

```
(0,-159.155)
Assertion failed: c != 0.0 && f != 0.0, file impedancec2.cpp, line 11
```

Het programma stopt na deze melding. Als het programma echter gecompileerd wordt zonder zogenoemde debug informatie[119], worden alle `assert` functies verwijderd en verschijnt weer de vreemde uitvoer `(0,-1.\#INF)`.

De standaard functie `assert` is bedoeld om tijdens het ontwikkelen van programma's te controleren of iedereen zich aan bepaalde afspraken houdt. Als

[119] In Microsoft Visual Studio kun je in een dropdownbox in de knoppenbalk kiezen voor Debug of Release. In GCC moet je het programma met de optie `-DNDEBUG` compileren om de assertions te verwijderen.

bijvoorbeeld is afgesproken dat de functie `impedanceC` alleen mag worden aangeroepen met argumenten die ongelijk aan 0 zijn, is dat prima met `assert` te controleren. Elk programmadeel waarin `impedanceC` wordt aangeroepen moet nu voor de aanroep zelf controleren of de argumenten geldige waarden hebben. Bij het testen van het programma (gecompileerd met debug informatie) zorgt de `assert` ervoor dat het snel duidelijk wordt als de afspraak geschonden wordt. Als na het testen duidelijk is dat iedereen zich aan de afspraak houdt, is het niet meer nodig om de `assert` uit te voeren. Het programma wordt gecompileerd zonder debug informatie en alle `assert` aanroepen worden verwijderd.

Het gebruik van `assert` heeft de volgende nadelen:

- Het programma wordt abrupt afgebroken als niet aan de assertion wordt voldaan.
- Op elke plek waar de functie `impedanceC` aangeroepen wordt moeten, voor aanroep, eerst de argumenten gecontroleerd worden. Als de functie op veel plaatsen aangeroepen wordt, is het logischer om de controle in de functie zelf uit te voeren. Dit maakt het programma ook beter onderhoudbaar (als de controle aangepast moet worden dan hoeft de code maar op 1 plaats gewijzigd te worden) en betrouwbaarder (je kunt de functie niet meer zonder controle aanroepen, de controle zit nu immers in de functie zelf).

We kunnen concluderen dat `assert` in dit geval niet geschikt is.

6.10.2 Het gebruik van een `bool` returnwaarde

In C (en ook in C++) werd dit traditioneel opgelost door de functie een returnwaarde te geven die aangeeft of het uitvoeren van de functie gelukt is (ImpedanceC3.cpp):

```
bool impedanceC(complex<double>& res, double c, double f) {
    if (c != 0.0 && f != 0.0) {
        res = complex<double>(0, -1 / (2 * PI * f * c));
        return true;
    }
```

```
    else
        return false;
}
```

Het programma wordt nu niet meer abrupt afgebroken. Het gebruik van een return waarde om aan te geven of een functie gelukt is heeft echter de volgende nadelen:

- Bij *elke* aanroep moet de returnwaarde getest worden.
- Op de plaats waar de fout ontdekt wordt kan hij meestal niet opgelost worden.
- Het "echte" resultaat van de functie moet nu via een call by reference parameter worden teruggegeven. Dit betekent dat je om de functie aan te roepen altijd een variabele aan moet maken (om het resultaat in op te slaan) ook als je het resultaat alleen maar wilt doorgeven aan een andere functie of operator.

De C library stdio werkt bijvoorbeeld met returnwaarden van functies die aangeven of de functie gelukt is. Zo geeft de functie `printf` bijvoorbeeld een `int` returnwaarde. Als de functie gelukt is, geeft `printf` het aantal geschreven karakters terug maar als er een error is opgetreden, geeft `printf` de waarde `EOF` terug. Een goed geschreven programma moet dus bij elke aanroep naar `printf` de returnwaarde testen![120]

6.10.3 Het gebruik van standaard exceptions

C++ heeft *exceptions* ingevoerd voor het afhandelen van uitzonderlijke fouten. Een exception is een *object* dat in de functie waar de fout ontstaat "gegooid" kan worden en dat door de aanroepende functie (of door zijn aanroepende functie enz...) "opgevangen" kan worden. In de standaard library zijn ook een aantal standaard exceptions opgenomen. De class van de exception die bedoeld is om te

[120] Kijk nog eens terug naar oude C programma's die je hebt geschreven. Hoe werd daar met de returnwaarde van `printf` omgegaan?

gooien als een parameter van een functie een ongeldige waarde heeft is de naam `domain_error`[121]. Zie ImpedanceC4.cpp

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

complex<double> impedanceC(double c, double f) {
    if (c == 0.0)
        throw domain_error("Capaciteit == 0");
    if (f == 0.0)
        throw domain_error("Frequentie == 0");
    return complex<double>(0, -1 / (2 * PI * f * c));
}
```

Je kunt een exception object gooien door het C++ keyword `throw` te gebruiken. Bij het aanroepen van de constructor van de class `domain_error` die gedefinieerd is in de include file `<exception>` kun je een string meegeven die de oorzaak van de fout aangeeft. Als de `throw` wordt uitgevoerd, wordt de functie meteen afgebroken. Lokale variabelen worden wel netjes opgeruimd (de destructors van deze lokale objecten wordt netjes aangeroepen). Ook de functie waarin de functie `impedanceC` is aangeroepen wordt meteen afgebroken. Dit proces van afbreken wordt gestopt zodra de exception wordt opgevangen. Als de exception nergens wordt opgevangen, wordt het programma gestopt.

```
    try {
        cout << impedanceC(1e-6, 1e3) << endl;
        cout << impedanceC(1e-6, 0) << endl;
        cout << "Dit was het!" << endl;
    } catch (domain_error& e) {
        cout << e.what() << endl;
    }
    cout << "The END." << endl;
```

[121] Als je opgelet hebt bij de wiskunde lessen, komt deze naam je bekend voor.

Exceptions kunnen worden opgevangen in een zogenaamd *try-blok*. Dit blok begint met het keyword `try` en wordt afgesloten met het keyword `catch`. Na het `catch` keyword moet je tussen haakjes aangeven welke exceptions je wilt opvangen gevolgd door een codeblok dat uitgevoerd wordt als de gespecificeerde exception wordt opgevangen. Na het eerste catch blok kunnen er nog een willekeurig aantal catch blokken volgen om andere exceptions ook te kunnen vangen. Als je alle mogelijke exceptions wilt opvangen, kun je dat doen met: `catch(...){ /*code*/}`.

Een exception kun je het beste als reference opvangen. Dit heeft 2 voordelen:

- Het voorkomt een extra kopie.
- We kunnen op deze manier ook van `domain_error` afgeleide classes opvangen zonder dat het slicing probleem (zie paragraaf 4.12) optreedt.

De class `domain_error` heeft een memberfunctie `what()` die de bij constructie van het object meegegeven string weer teruggeeft. De uitvoer van het bovenstaande programma is:

```
(0,-159.155)
Frequentie == 0
The END.
```

Merk op dat het laatste statement in het try-blok niet meer uitgevoerd wordt omdat bij het uitvoeren van het tweede statement een exception optrad. Het gebruik van exceptions in plaats van een `bool` returnwaarde heeft de volgende voordelen:

- Het programma wordt niet meer abrupt afgebroken. Door de exception op te vangen op een plaats waar je er wat mee kunt heb je de mogelijkheid om het programma na een exception gewoon door te laten gaan of op z'n minst netjes af te sluiten.
- Je hoeft niet bij elke aanroep te testen. Je kunt meerder aanroepen opnemen in hetzelfde try-blok. Als in het bovenstaande programma een exception zou optreden in het eerste statement, wordt het tweede (en derde) statement niet meer uitgevoerd.

- De returnwaarde van de functie kan nu weer gewoon gebruikt worden om de berekende impedantie terug te geven.

Vraag:

Pas de impedance calculator (zie paragraaf 4.14) aan zodat dit programma geen vreemde uitvoer meer produceert door een divide by zero error.

Antwoord:

De classes `C` en `P` moeten worden aangepast (Netwerk2.cpp):

```
class C: public Component { // C = Condensator
public:
    C(double c): c(c) {
    }
    virtual complex<double> Z(double f) const {
        if (c == 0.0)
            throw domain_error("Capacity == 0");
        if (f == 0.0)
            throw domain_error("Frequency == 0");
        return complex<double>(0, -1 / (2 * PI * f * c));
    }
    virtual void print(ostream& o) const {
        o << "C(" << c << ")";
    }
private:
    double c;
};
class P: public Component { // P = Parallel schakeling van ↩
 ↪ twee componenten
public:
    P(const Component& c1, const Component& c2): c1(c1), ↩
     ↪ c2(c2) {
    }
    virtual complex<double> Z(double f) const {
```

```
        if (c1.Z(f) + c2.Z(f) == complex<double>(0, 0))
            throw domain_error("Impedance of parallel ↩
             ↪ circuit can not be calculated (due to ↩
             ↪ divide by zero)");
        return (c1.Z(f) * c2.Z(f)) / (c1.Z(f) + c2.Z(f));
    }
    virtual void print(ostream& o) const {
        o << "(" << c1 << "//" << c2 << ")";
    }
private:
    const Component& c1;
    const Component& c2;
};
```

In het hoofdprogramma kunnen exceptions als volgt worden opgevangen:

```
    try {
        R r1(1E2);
        C c1(0); // om te testen!
        L l1(3E-2);
        S s1(r1, c1);
        S s2(r1, l1);
        P p(s1, s2);
        printImpedanceTable(p);
    } catch (domain_error& e) {
        cout << "Exception: " << e.what() << endl;
    }
```

6.10.4 Het gebruik van zelfgedefinieerde exceptions

In plaats van het gebruik van de standaard gedefinieerde exceptions kun je ook zelf exception classes definiëren. Zie ImpedanceC5.cpp:

```
class FrequencyError {};
class CapacityError {};
```

```
complex<double> impedanceC(double c, double f) {
    if (c == 0.0)
        throw CapacityError();
    if (f == 0.0)
        throw FrequencyError();
    return complex<double>(0, -1 / (2 * PI * f * c));
}
```

Voorbeeld van gebruik:

```
int main() {
    try {
        cout << impedanceC(1e-6, 1e3) << endl;
        cout << impedanceC(1e-6, 0) << endl;
        cout << "Dit was het!" << endl;
    } catch (CapacityError&) {
        cout << "Capaciteit == 0" << endl;
    } catch (FrequencyError&) {
        cout << "Frequentie == 0" << endl;
    }
    cout << "The END." << endl;
```

Uitvoer:

```
(0,-159.155)
Frequentie == 0
The END.
```

In het bovenstaande voorbeeld gebruiken we lege classes. Bij verschillende fouten gooien we objecten van verschillende (lege) classes. Bij het opvangen van de fout kunnen we de fout identificeren aan de hand van zijn class.

We kunnen bij het definiëren van exception classes ook overerving toepassen. Zelfgedefinieerde exception classes kunnen dan met behulp van overerving volgens een generalisatie- of specialisatiestructuur ingedeeld worden. Bij een `catch` kunnen we nu kiezen of we een specifieke of een generieke exception willen afvangen. De specifieke zijn polymorf met de generieke.

Doordat exceptions gewoon objecten zijn kun je ze ook data en gedrag geven.

Je kunt bijvoorbeeld een `virtual` memberfunctie definiëren die een bij de exception passende foutmelding geeft. Als je deze memberfunctie in specifiekere exceptions override, kun je een generieke exception vangen en toch door middel van dynamic binding de juiste foutmelding krijgen! Zie ImpedanceC6.cpp:

```
class ImpedanceError {
public:
    virtual ~ImpedanceError() = default;
    virtual string getErrorMessage() const = 0;
};

class FrequencyError: public ImpedanceError {
public:
    virtual string getErrorMessage() const;
};

string FrequencyError::getErrorMessage() const {
    return "Frequentie == 0";
}

class CapacityError: public ImpedanceError {
public:
    virtual string getErrorMessage() const;
};

string CapacityError::getErrorMessage() const {
    return "Capaciteit == 0";
}

complex<double> impedanceC(double c, double f) {
    if (c == 0.0)
        throw CapacityError();
    if (f == 0.0)
        throw FrequencyError();
    return complex<double>(0, -1 / (2 * PI * f * c));
```

```
}
```

Voorbeeld van gebruik:

```
try {
    cout << impedanceC(1e-6, 1e3) << endl;
    cout << impedanceC(1e-6, 0) << endl;
    cout << "Dit was het!" << endl;
} catch (ImpedanceError& e) {
    cout << e.getErrorMessage() << endl;
}
cout << "The END." << endl;
```

Uitvoer:

```
(0,-159.155)
Frequentie == 0
The END.
```

Als je nu bijvoorbeeld alleen de `CapacityError` exception wilt opvangen maar de `FrequencyError` exception niet, dan kan dat eenvoudig door in de bovenstaande catch het type `ImpedanceError` te vervangen door `CapacityError`.

6.10.5 De volgorde van `catch` blokken

Als je meerdere catch blokken gebruikt om exceptions af te vangen, moet je er op letten dat de meest specifieke exceptions vóór de generieke exceptions komen. Omdat de catch blokken van boven naar beneden worden “geprobeerd” als er een exception gevangen wordt. Dus:

```
try {
//  ...
} catch (FrequencyError&) {
    cout << "FrequencyError exception" << endl;
} catch (ImpedanceError&) {
    cout << "Other exception derived from ↩
     ↪ ImpedanceError" << endl;
```

```
} catch (...) {
    cout << "Other exception" << endl;
}
```

Vraag:

Waarom verschijnt bij het uitvoeren van de onderstaande code de foutmelding `FrequencyError exception` nooit op het scherm?

```
try {
//  ...
} catch (ImpedanceError&) {
    cout << "Other exception derived from ↩
     ↪ ImpedanceError" << endl;
} catch (FrequencyError&) {
    cout << "FrequencyError exception" << endl;
}
```

Antwoord:

Als er een `FrequencyError` exception wordt gegooid, wordt deze al opgevangen in het eerste `catch` blok. Een `FrequencyError` is namelijk afgeleid van een `ImpedanceError` dus een `FrequencyError` *is een* `ImpedanceError`.

6.10.6 Exception details

Over exceptions valt nog veel meer te vertellen:

- **Exceptions in constructors en destructors.**
  Exceptions zijn heel handig om te melden dat het aanmaken van een object in de constructor mislukt is. Bijvoorbeeld omdat ongeldige argumenten aan de constructor zijn meegegeven, je kunt in zo'n geval vanuit de constructor een exception gooien. Het gebruik van exceptions in een destructor is echter niet aan te raden. Als een exception optreedt, worden objecten opgeruimd (waarbij destructors worden aangeroepen). Voordat je het weet wordt dan

een exception gegooid tijdens het afhandelen van een exception en dat is niet toegestaan in C++. Als dit gebeurd wordt het programma afgesloten.

- **Function try-blok.**
  Speciale syntax om exceptions die optreden bij het initialiseren van datavelden in een constructor op te vangen.
- **Re-throw.**
  Gebruik van `throw` zonder argument in een catch blok om de zojuist opgevangen exception weer door te gooien.
- **Exception specification.**
  Speciale syntax om in het prototype van een functie aan te geven welke exceptions deze functie kan veroorzaken. Zie Impedance7.cpp.
- **Exceptions in de std library.**
  Een overzicht van alle exceptions die in de standaard library gedefinieerd zijn en gebruikt worden, is te vinden op: `http://en.cppreference.com/w/cpp/error/exception`.

Voor al deze details verwijs ik je naar [3, hoofdstuk 7] en [9, hoofdstuk 10].

6.11 Casting en run time type information

Door middel van *casting*[122] kun je variabelen van het ene type omzetten naar een ander type en kun je objecten van de ene class omzetten naar een andere class. Dit zijn "gevaarlijke" bewerkingen die alleen moeten worden gebruikt als het echt niet anders kan.

[122] Het Engelse woord casting betekent onder andere in een mal gieten. Als je een variabele cast naar een ander type, giet je die variabele als het ware in een andere mal.

6.11.1 Casting

In C kun je met een eenvoudige vorm van casting typeconversies doen. Stel dat we het adres van een C string in een integer willen opslaan[123] dan kun je dit als volgt proberen (Casting1.cpp):

```
    int i;
    i = "Hallo";
```

Als je dit probeert te compileren, krijg je de volgende foutmelding:

```
    i = "Hallo";
//  Error: (Microsoft) '=' : cannot convert from 'const char ↩
 ↪ [6]' to 'int'
//  Error: (GCC)       invalid conversion from 'const char*' ↩
 ↪ to 'int'
```

Het is namelijk helemaal niet zeker dat een `char*` in een `int` variabele past. Stel dat je zeker weet dat het past (omdat je het programma alleen maar onder Win32 wilt gebruiken) dan kun je de compiler "dwingen" met behulp van een zogenoemde *cast*:

```
    i = (int)"Hallo";
```

In C++ mag je deze cast ook als volgt coderen:

```
    i = int("Hallo");
```

Het vervelende van beide vormen van casting is dat een cast erg moeilijk te vinden is. Omdat een cast in bijna alle gevallen code oplevert die niet portable is, is het echter wel belangrijk om alle casts in een programma op te kunnen sporen.

In de C++ standaard is om deze reden een nieuwe syntax voor casting gedefinieerd die eenvoudiger te vinden is:

[123] Er is geen enkele reden te bedenken waarom je dit zou willen. Het adres van een C string moet je natuurlijk in een `char*` opslaan!

```
i = reinterpret_cast<int>("Hallo");
```

In dit geval moeten we een `reinterpret_cast` gebruiken omdat de cast niet portable is.

Stel dat je een C++ programma schrijft dat moet draaien op een AVR ATmega32 microcontroller. Als je de output poort B van deze controller wilt aansturen, kan dat via adres 0x38. Dit kun je in C++ als volgt doen:

```
volatile[124] uint8_t* portb = ↩
 ↪ reinterpret_cast<uint8_t*>(0x38);
*portb = 0xFE; // schrijf 0xFE (hex) naar adres 0x38 ↩
 ↪ (hex)
```

Als we een cast willen doen die wel portable is, kan dat met `static_cast`.

Vraag:

Wat is de uitvoer van het volgende programmadeel (Casting2.cpp)?

```
int i1 = 1;
int i2 = 2;
double d = i1 / i2;
cout << "d = " << d << endl;
```

Antwoord:

```
d = 0
```

Dit is niet wat de meeste mensen verwachten. De computer gebruikt bij het berekenen van `i1 / i2` echter een integer deling omdat `i1` en `i2` beiden van het type `int` zijn. Het antwoord van deze integer deling is de integer waarde 0. Vervolgens wordt deze waarde omgezet naar het type `double`.

[124] Zie voor uitleg over het gebruik van `volatile` http://bd.eduweb.hhs.nl/micprg/volatile.htm.

Als we willen dat het bovenstaande programma `0.5` als antwoord geeft, moeten we ervoor zorgen dat in plaats van een integer deling een floating point deling wordt gebruikt. De computer gebruikt een floating point deling als één van de twee argumenten een floating point getal is. De juiste deling is dus:

```
    double e = static_cast<double>(i1) / i2;
```

Er bestaat ook een speciale cast om een `const` weg te casten de zogenoemde `const_cast`. Voorbeeld (Casting3.cpp):

```
void stiekem(const string& a) {
    const_cast<string&>(a) = "Hallo";
}

int main() {
    string s("Dag");
    cout << "s = " << s << endl;
    stiekem(s);
    cout << "s = " << s << endl;
//  ...
```

Uitvoer:

```
s = Dag
s = Hallo
```

Het zal duidelijk zijn dat je het gebruik van `const_cast` zoveel mogelijk moet beperken. Als de reference a in het bovenstaande programma naar een `const string` verwijst, is het resultaat onbepaald omdat een compiler `const` objecten in het ROM geheugen kan plaatsen (dit gebeurt veel bij embedded systems).

6.11.2 Casting en overerving

Als we een pointer naar een `Base` class hebben, mogen we een pointer naar een `Derived` (van `Base` afgeleide) class toekennen aan deze `Base` class pointer, maar niet vice versa. Bijvoorbeeld:

```
class Hond { /* ... */ };
class SintBernard: public Hond { /* ... */ };
//  ...
    Hond* hp = new SintBernard; // OK: een SintBernard is ↩
     ↪ een Hond
    SintBernard* sbp = new Hond; // ERROR: een Hond is geen ↩
     ↪ SintBernard
//  Error: (Microsoft) 'initializing' : cannot convert from ↩
 ↪ 'Hond *' to 'SintBernard *'
//  Error: (GCC)       invalid conversion from 'Hond*' to ↩
 ↪ 'SintBernard*'
```

Het omzetten van een `Hond*` naar een `SintBernard*` kan soms toch nodig zijn. We noemen dit een *downcast* omdat we afdalen in de class hiërarchie.

Voorbeeld (Casting4.cpp):

```
class Hond {
public:
    virtual ~Hond() {
    }
    virtual void blaf() const {
        cout << "Blaf." << endl;
    }
//  ...
};

class SintBernard: public Hond {
public:
    SintBernard(int w = 10): whisky(w) {
    }
    virtual void blaf() const {
        cout << "Woef!" << endl;
    }
    int geefDrank() {
        cout << "Geeft drank." << endl;
        int w(whisky);
```

```
        whisky = 0;
        return w;
    };
//  ...
private:
    int whisky;
};

void geefHulp(Hond* hp) {
    hp->blaf();
//  cout << hp->geefDrank() << " liter." << endl;
//  Error: (Microsoft) 'geefDrank' : is not a member of 'Hond'
//  Error: (GCC)       'class Hond' has no member named ↩
 ↪ 'geefDrank'
//  We kunnen een cast gebruiken maar dat geeft foutieve ↩
 ↪ uitvoer als hp niet naar een SintBernard wijst.
    cout << static_cast<SintBernard*>(hp)->geefDrank() << ↩
     ↪ " liter." << endl;
}
```

In dit geval is een `static_cast` gebruikt om een downcast te maken. Als je de functie `geefHulp` aanroept met een `Hond*` als argument die wijst naar een `SintBernard`, gaat alles goed.[125]

```
    Hond* borisPtr = new SintBernard;
    geefHulp(borisPtr);
    delete borisPtr;
```

Uitvoer:

```
Woef!
Geeft drank.
10 liter.
```

[125] Als de functie `geefHulp` alleen maar aangeroepen wordt met een `SintBernard*` als argument, is het natuurlijk veel slimmer om deze functie te definiëren als: `void geefHulp(SintBernard* sbp)`. De downcast is dan niet meer nodig!

Als je de functie `geefHulp` echter aanroept met een `Hond*` als argument die niet wijst naar een `SintBernard`, dan geeft het programma onvoorspelbare resultaten (en/of loopt het vast):

```
    Hond* fikkiePtr = new Hond;
    geefHulp(fikkiePtr);
    delete fikkiePtr;
```

Uitvoer:

```
Blaf.
Geeft drank.
-33686019 liter.
```

Een `static_cast` is dus alleen maar geschikt als downcast als je zeker weet dat de cast geldig is. In het bovenstaande programma zou je de mogelijkheid willen hebben om te kijken of een downcast mogelijk is. Dit kan met een zogenoemde `dynamic_cast`. Zie Casting5.cpp:

```
void geefHulp(Hond* hp) {
    hp->blaf();
    SintBernard* sbp(dynamic_cast<SintBernard*>(hp));
    if (sbp)
        cout << sbp->geefDrank() << " liter." << endl;
}
```

Je kunt de functie `geefHulp` nu veilig aanroepen zowel met een `SintBernard*` als met een `Hond*` als argument.

```
    Hond* borisPtr = new SintBernard;
    geefHulp(borisPtr);
    delete borisPtr;
    Hond* fikkiePtr = new Hond;
    geefHulp(fikkiePtr);
    delete fikkiePtr;
```

Uitvoer:

```
Woef!
Geeft drank.
10 liter.
Blaf.
```

Een `dynamic_cast` is alleen mogelijk met polymorfe pointers en polymorfe references. Als een `dynamic_cast` van een pointer mislukt, geeft de cast een nul pointer terug. Bij een `dynamic_cast` van een reference is dit niet mogelijk (omdat een nul reference niet bestaat). Als een `dynamic_cast` van een reference mislukt, wordt de standaard exception `bad_cast` gegooid.

Een versie van `geefHulp` die werkt met een `Hond&` in plaats van met een `Hond*` kun je dus als volgt implementeren (Casting6.cpp):

```
#include <typeinfo>
//  ...
void geefHulp(Hond& hr) {
    hr.blaf();
    try {
        SintBernard& sbr(dynamic_cast<SintBernard&>(hr));
        cout << sbr.geefDrank() << " liter." << endl;
    } catch (bad_cast) {
        /* doe niets */
    }
}
```

Deze functie kun je als volgt aanroepen:

```
    SintBernard boris;
    geefHulp(boris);
    Hond fikkie;
    geefHulp(fikkie);
```

Uitvoer:

```
Woef!
Geeft drank.
10 liter.
Blaf.
```

6.11.3 Dynamic casting en RTTI

Om tijdens run time te kunnen controleren of een `dynamic_cast` mogelijk is moet informatie over het type tijdens run time beschikbaar zijn. Dit wordt *RTTI* = *R*un *T*ime *T*ype *I*nformation genoemd. In C++ hebben alleen classes met één of meer virtuele functies RTTI. Dat is logisch omdat polymorfisme ontstaat door het gebruik van virtual memberfuncties. RTTI maakt dus het gebruik van `dynamic_cast` mogelijk. Je kunt ook de RTTI gegevens behorende bij een object rechtstreeks opvragen. Deze gegevens zijn opgeslagen in een object van de class `type_info`. Deze class heeft een vraagfunctie `name()` waarmee de naam van de class opgevraagd kan worden. Om het `type_info` object van een (ander) object op te vragen moet je het C++ keyword `typeid` gebruiken. Zie Rtti.cpp:

```
using namespace std;

void printRas(Hond& hr) {
    cout << typeid(hr).name() << endl;
}
```

Deze functie kun je als volgt aanroepen:

```
    SintBernard boris;
    printRas(boris);
    Hond h;
    printRas(h);
```

Uitvoer:

```
class SintBernard
class Hond
```

6.11.4 Maak geen misbruik van RTTI en dynamic_cast

Het verkeerd gebruik van dynamic_cast en RTTI kan leiden tot code die *niet* uitbreidbaar en aanpasbaar is! Je zou bijvoorbeeld op het idee kunnen komen om een functie `blaf()` als volgt te implementeren (Casting7.cpp):

```
class Hond {
public:
    virtual ~Hond();
//  ...
};

Hond::~Hond() {
}

class SintBernard: public Hond {
//  ...
};

class Tekkel: public Hond {
//  ...
};

//  Deze code is NIET uitbreidbaar!
//  ******* DON'T DO THIS IN YOUR CODE *******
//  blaf moet als virtual memberfunctie geimplementeerd ↩
 ↪ worden!

void blaf(const Hond* hp) {
    if (dynamic_cast<const SintBernard*>(hp) != 0)
        cout << "Woef!" << endl;
    else if (dynamic_cast<const Tekkel*>(hp) != 0)
```

```
        cout << "Kef kef!" << endl;
    else
        cout << "Blaf." << endl;
}
```

Bedenk zelf wat er moet gebeuren als je de class `DuitseHerder` wilt toevoegen. In plaats van een losse functie die expliciet gebruikt maakt van dynamic binding (`dynamic_cast`) met behulp van RTTI moet je een `virtual` memberfunctie gebruiken. Deze `virtual` memberfunctie maakt impliciet gebruik van dynamic binding. Alleen in uitzonderingsgevallen (er is maar één soort hond die whisky bij zich heeft) moet je `dynamic_cast` gebruiken. Alle honden kunnen blaffen (alleen niet allemaal op dezelfde manier) dus is het gebruik van `dynamic_cast` om blaf te implementeren niet juist. In dit geval moet je een `virtual` memberfunctie gebruiken.

Bibliografie

[1] Harry Broeders. *Algoritmen en datastructuren*. 2015. URL: http://bd.eduweb.hhs.nl/algods/pdf/dictaat_ALGODS.pdf (zie pagina 187).

[2] Bruce Eckel. *Thinking in C++, Volume 1: Introduction to Standard C++*. 2nd Edition. Prentice Hall, 2000. ISBN: 978-0-13-979809-2. URL: http://www.tigernt.com/onlineDoc/tij/TIC2Vone.pdf (zie pagina's 10, 18, 23, 25, 26, 28, 29, 30, 56, 57, 60, 65, 68, 164).

[3] Bruce Eckel. *Thinking in C++, Volume 2: Standard Libraries & Advanced Topics*. 2nd Edition. Prentice Hall, 1999. ISBN: 978-0-13-035313-9. URL: http://www.tigernt.com/onlineDoc/tij/TIC2Vtwo.pdf (zie pagina's 27, 108, 223).

[4] ISO. *ISO/IEC 14882:2014 Information technology — Programming languages — C++*. Geneva, Switzerland: International Organization for Standardization, 2014. URL: http://www.iso.org/iso/home/store/catalogue_ics/catalogue_detail_ics.htm?csnumber=64029 (zie pagina's 11, 18, 26, 108).

[5] Barbara H. Liskov en Jeannette M. Wing. "A Behavioral Notion of Subtyping". In: *ACM Trans. Program. Lang. Syst.* 16.6 (nov 1994), p. 1811–1841. ISSN: 0164-0925 (zie pagina 123).

[6] Scott Meyers. *Effective C++: 55 Specific Ways to Improve Your Programs and Designs*. 3rd Edition. Addison-Wesley Professional, 2005. ISBN: 978-0-321-33487-9 (zie pagina 78).

[7] Scott Meyers. *More Effective C++: 35 New Ways to Improve Your Programs and Designs*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1995. ISBN: 978-0-201-63371-9 (zie pagina 78).

[8] Bjarne Stroustrup. *The C++ Programming Language*. Special Edition. Addison-Wesley Professional, 2000. ISBN: 978-0-201-70073-2. URL: http://my.safaribooksonline.com/book/programming/cplusplus/0201700735/a-tour-of-cplusplus/ch02lev1sec1 (zie pagina 17).

[9] Bjarne Stroustrup. *The C++ Programming Language*. 4th Edition. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2013. ISBN: 978-0-321-56384-2 (zie pagina's 79, 108, 164, 223).