Dictaat Object Georiënteerd Programmeren in C++

© 2008 Harry Broeders

Inhoudsopgave.

Inleiding. . . . 3

Een terugblik op C. . . . 3

1 Inleiding van C naar C++. . . . 6
- 1.1 Commentaar met //. . . . 6
- 1.2 Plaats van variabelen definities. . . . 6
- 1.3 Constante waarden met const. . . . 7
- 1.4 Het type bool. . . . 7
- 1.5 Standaard include files. . . . 7
- 1.6 Input en output met << en >>. . . . 8
- 1.7 Het type string. . . . 9
- 1.8 Het type vector. . . . 10
- 1.9 Function name overloading. . . . 10
- 1.10 Default parameters. . . . 11
- 1.11 Naam van struct. . . . 11
- 1.12 C++ als een betere C. . . . 12

2 Objects and classes. . . . 14
- 2.1 Object Oriented Design (OOD) en Object Oriented Programming (OOP). . . . 14
- 2.2 ADT’s (Abstract Data Types). . . . 17
- 2.3 Voorbeeld class Breuk (eerste versie). . . . 20
- 2.4 Constructor Breuk. . . . 22
- 2.5 Constructors en type conversies. . . . 23
- 2.6 Initialisation list van de constructor. . . . 24
- 2.7 Default copy constructor. . . . 24
- 2.8 Default assignment operator. . . . 24
- 2.9 const memberfuncties. . . . 25
- 2.10 Class invariant. . . . 26
- 2.11 Voorbeeld class Breuk (tweede versie). . . . 26
- 2.12 Operator overloading. . . . 27
- 2.13 this pointer. . . . 29
- 2.14 Reference variabelen. . . . 29
- 2.15 Reference parameters. . . . 30
- 2.16 const reference parameters. . . . 31
- 2.17 Parameter FAQ. . . . 31
- 2.18 Reference return type. . . . 32
- 2.19 Reference return type (deel 2). . . . 33
- 2.20 Operator overloading (deel2). . . . 33
- 2.21 operator+ FAQ. . . . 34
- 2.22 Operator overloading (deel 3). . . . 35
- 2.23 Overloaden operator++ en operator--. . . . 36
- 2.24 Conversie operatoren. . . . 37
- 2.25 Voorbeeld class Breuk (derde versie). . . . 37
- 2.26 friend functions. . . . 39
- 2.27 Operator overloading (deel 4). . . . 40
- 2.28 Voorbeeld separate compilation van class MemoryCell. . . . 41

3 Templates. . . . 42
- 3.1 Template functies. . . . 42
- 3.2 Template classes. . . . 45
- 3.3 Voorbeeld template class Dozijn. . . . 46
- 3.4 Template details. . . . 48

3.5 Standaard Templates. 48
3.6 std::vector. 48

4 Inheritance. 50
4.1 De syntax van inheritance. 51
4.2 Polymorfisme. 52
4.3 Memberfunctie overriding. 53
4.4 Abstract base class. 55
4.5 Constructors bij inheritance. 55
4.6 protected members. 55
4.7 Voorbeeld: ADC kaarten. 56
4.7.1 Probleemdefinitie. 56
4.7.2 Een gestructureerde oplossing. 56
4.7.3 Een oplossing door middel van een ADT. 58
4.7.4 Een object georiënteerde oplossing. 60
4.7.5 Een kaart toevoegen. 63
4.8 Overloading en overriding van memberfuncties. 64
4.9 Slicing problem. 67
4.10 Voorbeeld: Opslaan van polymorfe objecten in een vector. 68
4.11 Voorbeeld: Impedantie calculator. 70
4.11.1Weerstand, spoel en condensator. 70
4.11.2Serie- en parallelschakeling. 72
4.11.3Een grafische impedantie calculator. 74
4.12 Inheritance details. 74

5 Dynamic memory allocation en destructors. 74
5.1 Dynamische geheugen allocatie (new en delete). 74
5.2 Destructor ~Breuk. 75
5.3 Destructors bij inheritance. 77
5.4 Virtual destructor. 77
5.5 Voorbeeld class Array. 79
5.6 explicit constructor. 81
5.7 Copy constructor en default copy constructor. 81
5.8 Overloading operator=. 83
5.9 Wanneer moet je zelf een destructor, copy constructor en operator= definiëren. 84
5.10 Voorbeeld template class Array. 84

6 Losse flodders. 86
6.1 static class members. 86
6.2 Constante pointers met const. 87
6.2.1 const * 88
6.2.2 * const 88
6.2.3 const * const 88
6.3 Constanten in een class. 88
6.4 inline memberfuncties. 89
6.5 Namespaces. 90
6.6 Exceptions. 92
6.7 Casting en runtime type information. 99

Inleiding.

Dit is het dictaat: "Object Georiënteerd Programmeren in C++". Dit dictaat kan zonder boek gebruikt worden. Als je meer achtergrondinformatie of diepgang zoekt kun je gebruik maken van het boek: *"Thinking in C++ 2nd Edition, Volume 1"* van Bruce Eckel (2000 Pearson Education). 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 zogenaamde, object oriented design (OOD), echt doorbrak. Object georiënteerde 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 object georiënteerde versie van C heeft de naam C++ gekregen en heeft zich in korte tijd (de eerste release van Borland C++ was in 1990 en de eerste release van Microsoft C++ was in 1992) 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.

Een terugblik op C.

We starten deze onderwijseenheid met met de overgang van C naar C++. **Ik ga er van uit dat je de taal C zoals behandeld in de propedeuse 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 een voorbeeldprogramma 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>

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

/* Deze functie drukt een Tijdsduur af */
void drukaf(struct Tijdsduur td) {
     if (td.uur==0)
          printf("           %2d minuten\n", td.min);
     else
          printf("%3d uur en %2d minuten\n", td.uur, td.min);
}

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

/* Deze functie berekent de totaal gewerkte tijd
   uit een rij met n gewerkte tijden */
struct Tijdsduur som(struct Tijdsduur trij[], int n) {
```

```
    int teller;
    struct Tijdsduur s;
    s.uur=s.min=0;
    for (teller=0;teller<n;++teller) {
        s.uur+=trij[teller].uur;
        s.min+=trij[teller].min;
    }
    s.uur+=s.min/60;
    s.min%=60;
    return s;
}

#define MAX   5

int main () {
    struct Tijdsduur rij[MAX];
    int aantal=0, gelezen;
    do {
        printf("Type gewerkte uren en minuten in (of Ctrl-Z): ");
        gelezen=scanf("%d%d", &rij[aantal].uur, &rij[aantal].min);
    }
    while (gelezen==2 && ++aantal<MAX);
    printf("\n\n");
    drukafRij(rij, aantal);
    printf("De totaal gewerkte tijd is:\n");
    drukaf(som(rij, aantal));
    getchar(); getchar();
    return 0;
}
```

Verklaring:

- In de eerste regel wordt de file `stdio.h` “included”. Dit is nodig om gebruik te kunnen maken van functies en typen 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 `getchar` (om een karakter te lezen vanaf het toetsenbord) en van `scanf` (om egtallen te lezen vanaf het toetsenbord).

- Vervolgens is het samengestelde type `struct Tijdsduur` gedeclareerd. Variabelen van dit type bevatten twee datavelden (Engels: datamembers) van het type `int`. Deze datavelden heten `uur` en `min` 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 als parameter `td` meegegeven `struct Tijdsduur` af op het scherm door gebruik te maken van de standaard schrijffunctie `printf`. Het return type van deze functie 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 `trij` is een array met elementen van het type `struct Tijdsduur`. De tweede parameter (een integer genaamd `n`) 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 `n-1`. Deze `teller` wordt gebruikt om de elementen uit `trij` één voor één te selecteren. Elk element (een variabele van het type `struct 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 struct 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 trij 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.min groter dan 59 zijn geworden. Om deze reden wordt de waarde van s.min/60 opgeteld bij s.uur. De waarde van s.min moet dan gelijk worden aan de resterende minuten. Het aantal resterende minuten kunnen we bereken met s.min=s.min%60. Of in verkorte notatie s.min%=60. Het return type van de functie som is van het type struct 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 rij aangemaakt met MAX elementen van het type struct 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 rij. 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 2 integers met scanf niet gelukt is, dan is gelezen ongelijk aan 2 en wordt de variabele aantal niet opgehoogd. De while lus wordt dus beëindigd als het inlezen van 2 integers niet gelukt is, bijvoorbeeld omdat het einde invoer teken ingetyped 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 de stdio functie getchar() twee maal aangeroepen. Deze functie wacht totdat de gebruiker een return intoetst. Dit is nodig omdat de debugger van C++ Builder het output window meteen sluit als het programma eindigt. 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 onderwijsdeel 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 dit kwartaal C++ bovenop te bouwen.

Het volgen van deze onderwijseenheid zonder voldoende kennis en begrip van C is vergelijkbaar met het bouwen van een huis op drijfzand!

1 Inleiding van C naar C++.

De ontwerper van C++ Bjarne Stroustrup geeft op de vraag:"Wat is C++?" het volgende antwoord[1]:

"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, object georiënteerd programmeren en generiek programmeren komen in de volgende hoofdstukken uitgebreid aan de orde. C++ is een zeer uitgebreide taal en dit onderwijsdeel moet dan ook zeker niet gezien worden als een cursus C++. Wij behandelen slechts de meest belangrijke delen van C++. De verbeteringen die in dit hoofdstuk worden besproken zijn slechts kleine verbeteringen. De meeste worden in de les niet behandeld.

1.1 Commentaar met `//`.

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++ 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 Plaats van variabelen definities. (Zie eventueel TICPP[2] chapter03.html#Heading126.)

Je bent gewend om in C programma's, variabelen te definiëren aan het begin van een blok (meteen na `{`). In C++ 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 zelf mogelijk om de besturingsvariabele van een `for` lus in het `for` statement zelf te definiëren[3]. 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 hoeft 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` loop:

```
/* 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] Zie het boek: *The C++ programming language 3ed* van Stoustrup.

[2] TCPP = Thinking in C++. De “links” verwijzen naar de HTML versie van dit boek gratis te downloaden van: http://mindview.net/Books/TICPP/ThinkingInCPP2e.html.

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

1.3 Constante waarden met const. (Zie eventueel TICPP Chapter03.html#Heading134)

Je bent gewend om in C programma's symbolische constanten te definiëren met de preprocessor directive `#define`. Het is in ANSI C[4] en in C++ ook mogelijk om constanten te definiëren met behulp van een `const` qualifier gebruiken[5].

Voorbeeld met `#define`:

```
#define aantalRegels 80
```

Hetzelfde voorbeeld met `const`:

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

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 een volgend hoofdstuk (blz. 25) zul je zien dat het met een `const` qualifier ook mogelijk wordt om constanten te definiëren van zelfgemaakte types.

Een constante moet je initialiseren:

```
const int k; // Error: Constant variable 'k' must be initialized
```

Een constante mag je (vanzelfsprekend) niet veranderen:

```
aantalRegels=79; // Error: Cannot modify a const object
```

1.4 Het type bool. (Zie eventueel TICPP Chapter03.html#Heading119.)

C++ kent in tegenstelling tot C[6] het keyword `bool`. Met dit keyword kun je booleanse 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 booleanse 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 booleanse variabele gaat.

1.5 Standaard include files. (Zie eventueel TICPP Chapter02.html#Heading78 en Chapter02.html#Heading86)

De ISO/ANSI standaard bevat ook een uitgebreide library. Als we de functies, types enz. uit deze standaard library willen gebruiken dan moeten we de definities van de betreffende functies, types, enz.

---

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

[5] In ANSI C kun je de qualifier `const` ook gebruiken. Maar in het in de P fase bij elektrotechniek gebruikte boek “De taal C van PSD tot C programma” van Daryl McAllister maakt hier echter geen gebruik van.

[6] In ANSI C99 (de laatste versie van de C standaard) kun je 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 ons programma opnemen door de juiste header file te includen[7]. De C++ standaard header files eindigen niet zoals bij C op `.h` maar hebben helemaal geen extensie. Dus `#include <complex>` in plaatst van `#include <complex.h>`[8]. Dit heeft als voordeel dat (delen van) oude C programma's waarin C headerfiles gebruikt worden nog steeds met de nieuwe ANSI/ISO C++ compiler vertaald kunnen worden. Bij het schrijven van nieuwe programma's gebruiken we uiteraard de C++ include files.

De standaard C++ library bevat ook alle functies, types enz. uit de standaard C library. De headerfiles die afkomstig zijn uit de “oude” 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 ANSI/ISO C++ namespaces opgenomen. Als je een symbool uit de C++ standaard library 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);
```

In plaats van elk symbool uit de C++ standaard library voorafgaan 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);
```

1.6 Input en output met << en >>[9]. (Zie eventueel TICPP Chapter02.html#Heading90)

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" omdat de inhoud van de als eerste argument meegegeven format string pas tijdens het uitvoeren van het programma verwerkt wordt. De compiler merkt het dus niet als de "type aanduidingen" zoals `%d` die in de format string gebruikt zijn niet overeenkomen met de typen 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 ISO/ANSI C++ standaard 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 stan-

---

[7] Headerfiles kunnen we op twee verschillende manieren includen: `#include <naam>` nu worden alleen de standaard include directories doorzocht om de include file `naam` te vinden en `#include "naam"` nu wordt eerst de huidige directory en pas daarna de standaard include directories doorzocht om `naam` te vinden.

[8] Headerfiles die je zelf maakt krijgen nog wel steeds de extensie `.h`.

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

daard 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.

Voorbeeld met stdio:

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

Dit programmadeel bevat 2 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 het toetsenbord
std::cout<<"d="<<d<<std::endl;  // druk d af op het 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 voor verdere informatie hoofdstuk 10 en 14 van het TICPP boek of de bij de iostream library behorende help files.[10]

1.7 Het type string.[11] (Zie eventueel TICPP Chapter02.html#Heading94.)

In C worden character strings opgeslagen in variabelen van het type char[] (character array). De afspraak is dan dat het einde van de character string aangegeven wordt door een null character '\0'. Vaak werden deze variabelen dan aan functies doorgegeven door middel van character pointers (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 ANSI/ISO standaard library is een nieuw type string opgenomen waarin character strings op een veilige manier opgeslagen kunnen worden. Het bewerken van deze strings is ook veel eenvoudiger dan strings opgeslagen in chararacter array's. Het type string komt in de eerste practicumopdracht uitgebreid aan de orde.

Bijvoorbeeld het vergelijken van "oude" strings:

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

Gaat bij het gebruik van ANSI/ISO strings als volgt:

[10] Zie practicum opgave 1 voor meer uitleg.

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

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

1.8 Het type `vector`. (Zie eventueel TICPP Chapter02.html#Heading96.)

In de ANSI/ISO C++ standaard library is ook het type `vector` opgenomen. Dit type is bedoeld om het oude type array `[]` te vervangen. In een volgend hoofdstuk komen we hier uitgebreid op terug.

1.9 Function name overloading. (Zie eventueel TICPP Chapter07.html.)

In C mag elke functienaam maar 1 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 1 naam hoeft te onthouden.[12] 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;
}
```

Als je één van deze functies wilt gebruiken om bijvoorbeeld de absolute waarde van een variabele van het type `double` te berekenen 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 de gebruikte parameter (`in`) welk van de twee bovenstaande `abs` functies aangeroepen wordt.

[12] 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.

1.10 Default parameters. (Zie eventueel TICPP Chapter07.html#Heading242.)

Het is in C++ mogelijk om voor de laatste parameters van een functie default waarden 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 de laatste parameter een default waarde mee te geven (in het prototype) zodat we niet steeds het talstelsel 10 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 parameter wordt aangeroepen wordt als tweede parameter `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
```

De default parameter maakt de `print` functie eenvoudiger te gebruiken omdat de gebruiker (de programmeur die deze functies aanroept) niet steeds de tweede parameter hoeft mee te geven als zij/hij getallen decimaal wil afdrukken.[13]

1.11 Naam van struct.

In C is de naam van een struct géén typenaam. Als 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.     */
};
```

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

```
struct Tijdsduur td1;
```

Het is in C gebruikelijk om met de volgende type definitie:

```
typedef struct Tijdsduur TTijdsduur;
```

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

een typenaam (in dit geval `TTijdsduur`) te declareren voor het type `struct Tijdsduur`. Variabelen van dit type kunnen dan 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).

1.12 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 blz. 3 dan ontstaat het volgende C++ programma:

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

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

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

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

int main () {
     const int MAX(5);
     Tijdsduur rij[MAX];
```

```
    int aantal=0;
    do {
        cout<<"Type gewerkte uren en minuten in (of Ctrl-Z): ";
        cin>>rij[aantal].uur>>rij[aantal].min;
    }
    while (cin && ++aantal<MAX);
    cout<<endl<<endl;
    drukaf(rij, aantal);
    cout<<"De totaal gewerkte tijd is:"<<endl;
    drukaf(som(rij, 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` “included”. Dit is nodig om gebruik te kunnen maken van functies, objecten en typen 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 zogenaamde I/O manipulator wordt in de tweede regel de file `iomanip` "included". In dit programma maak ik gebruik van de manipulator `setw()` waarmee de breedte van een outputveld 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 `/*...*/`.
- Als typenaam voor de parameter `td` van de functie `drukaf` is hier `Tijdsduur` in plaats van `struct Tijdsuur` gebruikt. Dit zelfde geldt voor de parameter `trij` in de twee volgende functies.
- 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 constante `MAX` is hier als `const int` gedefinieerd in plaats van met `#define`.
- Het inlezen vanaf het toestenbord 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 lees bewerking gelukt is. De variabele `gelezen` is dus niet meer nodig.
- Als het inlezen op de een of andere manier niet gelukt is, dan 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 onderwijsdeel zul je leren hoe dit probleem is op te lossen door het gebruik van dynamisch geheugen.

Merk op dat de vernieuwingen die in C++ zijn ingevoerd ten opzichte van C, namelijk het gebruik van abstracte data typen en object georiënteerde 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 object georiënteerde technieken die C++ biedt.

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) en 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 gebruikelijk 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 bus). Bij het ontwerpen van software is het niet gebruikelijk om gebruik te maken van componenten die door anderen ontwikkeld zijn (en waarvoor je dus moet betalen). Vaak worden wel bestaande datastructuren en algoritmen gekopieerd maar die moeten vaak toch aangepast worden. Het gebruik van functie libraries is wel gebruikelijk maar dit zijn in feite eenvoudige componenten. Een voorbeeld van een complexe component zou 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) kunnen zijn. In alle applicaties waarbij je tekst moet kunnen invoeren kun je deze editor dan hergebruiken. Er zijn geen databoeken vol met software componenten die we kunnen kopen en waarmee we vervolgens zelf applicaties kunnen ontwikkelen. Dit heeft volgens mij verschillende redenen:

- Voor een hardware component moet je betalen. Bij een software component vindt men dit niet normaal. Zo hoor je vaak zeggen: "Maar dat algoritme staat toch gewoon in het boek van ... dus dat kunnen we toch zelf wel implementeren."
- Als je van een bepaalde hardware component 1000 stuks gebruikt dan moet je er ook 1000 maal voor betalen. Bij een software component vindt men dit niet normaal en er valt eenvoudiger mee te sjoemelen.
- Programmeurs denken vaak: maar dat kan ik zelf toch veel eenvoudiger en sneller programmeren.
- De omgeving (taal, operating system, hardware enz.) van software is divers. Dit maakt het produceren van software componenten niet eenvoudig. Heeft u deze component ook in C++ voor Linux in plaats van in Visual Basic voor Windows?[14]

[14] Er zijn de laatste jaren 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.

De object georiënteerde benadering is vooral bedoeld om het ontwikkelen van herbruikbare software-componenten mogelijk te maken. In deze onderwijseenheid zul je kennismaken met deze object georiënteerde benadering. Wij hebben daarbij gekozen voor de taal C++ omdat dit één van de meest gebruikte object georiënteerde programmeertalen is. Het is echter niet de bedoeling dat je na deze onderwijseenheid op de hoogte bent van alle aspecten en details van C++. Wel zul je na afloop van deze onderwijseenheid de algemene ideeën achter OOP begrijpen en die toe kunnen passen in C++. De object georiënteerde 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 object georiënteerde technieken lijkt op de manier van probleem oplossen die we in het dagelijks leven gebruiken. Een object georiënteerde 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 "*message*" aan dat object. Aan deze message kunnen indien nodig *argumenten* worden meegegeven. Het is de verantwoordelijkheid van het ontvangende object (de "*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 "*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 runtime worden bepaald. (*Late binding between the message (function name) and 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 "*class*" genoemd. Elk object behoort tot een bepaalde class, een object wordt ook wel een "*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 "*base class*" genoemd en de class die daarvan afgeleid wordt, wordt "*derived class*" genoemd. Als van een afgeleide class weer nieuwe classes worden afgeleid ontstaat een hiërarchisch netwerk van classes. Een derived class overerft alle eigenschappen van een base class (overerving = "*inheritance*"). Als een object een message ontvangt wordt de bijbehorende method als volgt bepaald ("*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 ("*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 ("*composition*"). Inheritance wordt ook wel generalisatie (*"generalization"*) genoemd en leidt tot een zogenaamde "*is een*" relatie. Je kunt een class Bakker bijvoorbeeld afleiden door middel van overerving van een class Winkelier. Dit betekent dan dat een Bakker minimaal dezelfde messages ondersteunt als Winkelier (maar hij 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 zogenaamde "*heeft een*" relatie. Je kunt in de definitie van een class Auto vijf objecten van de class Wiel opnemen. Dit betekent dan dat een Auto deze Wielen gebruikt 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 object georiënteerd 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 typen 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 data types.**
  In C++ kunnen door het gebruik van *templates* generieke functies en data types gedefinieerd worden. Een generieke functie is een functie die gebruikt kan worden voor meerdere parameter types. In C kan om een array met `int`'s te sorteren een functie geschreven worden. Als we vervolgens een array 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 definieren 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 data types 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 en de in deze base classes gedefinieerde messages in afgeleide (Engels: derived) classes 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 laatste wordt mogelijk gemaakt door de message pas tijdens run time aan een bepaalde method te verbinden en wordt "*polymorfisme*" genoemd. Op deze begrippen komen we later nog uitgebreid terug.

In het propedeuse project heb je geleerd hoe je vanuit een probleemstelling voor een programma de functies en procedures 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 object georiënteerde programmeertalen ontstaat al snel de vraag: "Hoe vind ik uitgaande van de probleemstelling de in dit programma benodigde classes en het verband tussen die classes?" Deze vraag is afhankelijk van de gebruikte OOA (Object Oriented Analyse) en OOD (Object Oriented Design) methode. Deze onderwerpen komen bij E later in dit onderwijsdeel aan de orde. Bij TI zijn deze onderwerpen al aan de orde geweest en komen ze nog uitgebreid bij verschillende onderwijsdelen in H2 aan de orde.

2.2 ADT's (Abstract Data Types).

De eerste stap op weg naar het object georiënteerde denken en programmeren is het leren programmeren met ADT's. Een ADT[15] is een user-defined type dat voor een gebruiker 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 typen 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 werken[16], dan kun je dit (niet in C) opgenomen "type" `Breuk` alleen maar op de volgende manier definiëren:

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

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

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

Het normaliseren[17] van de breuk zorgt ervoor dat 3/8 + 1/8 als resultaat 1/2 in plaats van 4/8 heeft. Dit zorgt ervoor dat een overflow minder snel optreedt als met het resultaat weer verder wordt gerekend.

---

[15] 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 "user-defined type".

[16] Een belangrijke reden om te werken met breuken in plaats van floating point getallen is het voorkomen van afrondingsproblemen. Een breuk zoals 1/3 moet als floating point getal afgerond worden tot bijvoorbeeld: 0.333333333. Deze afrondig kan ervoor zorgen dat een berekening zoals 3*(1/3) 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.

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

Deze manier van werken heeft de volgende nadelen:

- Iedere programmeur die gebruikt maakt van het type `struct 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.
- Iedere programmeur die gebruikt maakt van het type `struct 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 `struct 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 `struct Breuk` (en de bijbehorende bewerkingen) dit kan doen.

Deze nadelen komen voort uit het feit dat in C de definitie van het type `struct 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:

```
class Breuk {              // Op een object van de class Breuk
public:                    // kun je de volgende bewerkingen uitvoeren:
    void leesin();                 //  inlezen vanuit het toetsenbord.
    void drukaf() const[18];       //  afdrukken op het scherm.
    void plus(Breuk[19] b);        //  een Breuk erbij optellen.
private:                   // Een object van de class Breuk heeft privé:
    int boven;                     //  een teller,
    int onder;                     //  een noemer en
    void normaliseer();            //  een functie normaliseer.
};
```

De class `Breuk` koppelt een bepaalde datastructuur (2 interne integer variabelen genaamd `boven` en `onder`) met de bij deze datastructuur behorende bewerkingen[20] (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[21]. Deze private functies en variabelen zitten ingekapseld in de class `Breuk`.

---

[18] `const` memberfunctie wordt pas behandeld op blz. 25.

[19] Het is beter om hier een `const` reference parameter te gebruiken. Dit wordt pas behandeld op blz. 31.

[20] 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).

[21] 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.

Functies de opgenomen zijn in een class worden *memberfuncties* genoemd. De memberfunctie `plus` kan als volgt gedefinieerd worden[22]:

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

Je kunt objecten (variabelen) van de class (zelf gedefinieerde 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
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 class declaratie. Het object `a` is een instantie van de class `Breuk` en kan, zoals uit de class declaratie 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 class declaratie. 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 zogenaamde 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 datamembers `boven` en `onder` gebruikt worden dan zijn dit de datamembers 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 (blz. 25) 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.

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

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 `struct 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:

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

Je ziet dat het zelf gedefinieerde 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 zelf gedefinieerde type, als hij of zij de naam `Breuk` maar kent, als vanzelf (intuïtief) gebruiken. Er is daarbij geen handleiding of helpfile 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 blz. 18 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 source code presenteren van een programma 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;

// Class declaratie:
class Breuk {
public:
     // Class interface.
     // Vertelt wat je met een object van de class kunt doen.
     Breuk();                     // constructors zie blz. 22
     Breuk(int t);
     Breuk(int t, int n);
     int teller() const;          // const memberfunctie zie blz. 25
     int noemer() const;
     void plus(Breuk[23] b);
     void abs();
private:
     // Class implementatie.
     // 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: bepaalt 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;
}

// Class definitie:
// 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();
```

[23] Het is beter om hier een `const` reference parameter te gebruiken. Dit wordt pas behandeld op blz. 31.

```
}
void Breuk::abs() {
     if (boven<0) boven=-boven;
}
void Breuk::normaliseer() {
     assert(onder!=0);24
     if (onder<0) {
          onder=-onder;
          boven=-boven;
     }
     int d=ggd(boven, onder);
     boven/=d;
     onder/=d;
}

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 blz. 24
     cout<<"b3(b2) ==> "<<b3.teller()<<'/'<<b3.noemer()<<endl;
     b3.abs();
     cout<<"b3.abs() ==> "<<b3.teller()<<'/'<<b3.noemer()<<endl;
     b3=b2;              // kan dit zomaar? Zie blz. 24
     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
b3(b2) ==> -23/5
b3.abs() ==> 23/5
b3=b2 ==> -23/5
b3.plus(5) ==> 2/5
halve ==> 1/2
b3=halve ==> 1/2
```

2.4 Constructor `Breuk`. (Zie eventueel TICPP Chapter06.html#Heading225.)

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 datamembers `boven` met `0` en `onder`

[24] De standaard functie `assert` doet niets als de, als parameter opgegeven, expressie `true` oplevert maar breekt het programma met een passende foutmelding af als dit niet zo is. Je kunt zogenaamde "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.

met `1` geïnitialiseerd. Dit gebeurt in een zogenaamde *initialisation list*. Na het prototype van de constructor volgt een `:` waarna de datamembers één voor één geïnitialiseerd worden door middel van de `(…)` notatie.

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

In dit geval is er verder geen code in de constructor opgenomen. Door tussen de `{}` 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 `b` de waarde 0/1.

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

De overige constructors worden gebruikt als je een `Breuk` bij het aanmaken met 1 of met 2 integers ïnitialiseert. Na afloop van de onderstaande code zal het object `b2` de waarde 1/2 bevatten. (3/6 wordt namelijk in de constructor genormaliseerd tot 1/2.)

```
Breuk b2(3,6);         // roep 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();
}
```

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.[25]

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 (=*memberwise construction*).

2.5 Constructors en type conversies. (Zie eventueel TICPP Chapter12.html#Heading372.)

Aan het einde van de functie `main` wordt 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 argument en niet met een `int` als argument. 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 dan wordt deze memberfunctie aangeroepen. Als dit niet het geval is dan "kijkt" de compiler of er in de class `Breuk` een memberfunctie is met een argument van een ander type waarnaar het type `int` omgezet kan worden[26]. 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

[25] Door het slim gebruik van default parameters 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 parameters vervangen kunnen worden: `Breuk::Breuk(int t=0, int n=1);`

[26] Als dit er meerdere zijn, dan zijn er allerlei conversie 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 zelfgefinieerd type.

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[27].

2.6 Initialisation list van de constructor. (Zie eventueel TICPP Chapter08.html#Heading267.)

Het initialiseren van data fields vanuit de constructor kan op twee manieren geprogrammeerd worden:
- door gebruik te maken van een initialisation 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 datamember kan bijvoorbeeld niet met een assignment geïnitialiseerd worden.

Dus gebruik:

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

in plaats van:

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

2.7 Default copy constructor. (Zie eventueel TICPP Chapter11.html#Heading339.)

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 value parameter wordt doorgegeven aan 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*). Het is ook mogelijk om zelf een copy constructor te definiëren (wordt later behandeld) maar voor de class Breuk voldoet de door de compiler gedefinieerde copy constructor prima.

2.8 Default assignment operator.

Als geen assignment operator (=) gedefinieerd is dan wordt door de compiler een *default assignment operator* aangemaakt. Deze default assignment operator roept voor elk dataveld de assignment operator van dit veld aan (=*memberwise assignment*). Het is ook mogelijk om zelf een assignment operator te

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

definiëren (wordt later behandeld) maar voor de class `Breuk` voldoet de door de compiler gedefinieerde assignment operator prima.

2.9 `const` memberfuncties. (Zie eventueel TICPP Chapter08.html#Heading271.)

Een object (variabele) van de class (het zelf gedefinieerde type) `Breuk` kan ook als constante 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: Cannot modify a const object
```

Stel jezelf nu eens de vraag welke memberfuncties je aan mag roepen op het object `halve`[28]. De memberfunties `teller` en `noemer` kunnen zonder problemen worden aangeroepen op een constante breuk omdat ze de waarde van de breuk niet veranderen. De memberfunties `plus` en `abs` mogen echter niet op het object `halve` worden aangeroepen omdat deze memberfuncties dit object zouden wijzigen en een constant object mag je vanzelfsprekend niet veranderen. De compiler kan niet (altijd[29]) 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 constante breuken:

```
const Breuk halve(1,2);          // constante halve met waarde 1/2
cout<<halve.teller()<<endl;
```

Het aanroepen van de memberfunctie `plus` voor de `const Breuk halve` geeft echter een foutmelding:

```
halve.plus(b);
//   Error30: Non-const function called for const object
```

[28] Zie blz. 21 voor de declaratie van de class `Breuk`.

[29] De compiler kan dit zeker niet als de class definitie seperaat van de applictie gecompileerd wordt. Zie blz. 41.

[30] De Borland C++ Builder 6 compiler geeft slechts een warning in plaats van een error. Dit betekent dat de constante `halve` door het aanroepen van de functie `plus` gewijzigd kan worden. Dit gaat lijnrecht in tegen de C++ standaard! Als we Borland om opheldering vragen (door op F1 te drukken) geven zij de volgende toelichting: *Non-const function called for const object is an error, but was reduced to a warning to give existing programs a chance to work.* wxDev-C++ (gcc) geeft wel een

(wordt vervolgd...)

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: Cannot modify a const object
}
```

Door het toepassen van function 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[31].

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.
- *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 datamembers `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 datamembers private zijn, dan weet je zeker dat voor elk object van de class `Breuk` deze invariant altijd geldig is[32]. Dit vermindert de kans op het maken van fouten.

2.11 Voorbeeld class `Breuk` (tweede versie).

Dit voorbeeld is een uitbreiding van de op blz. 20 gegeven class `Breuk` (versie 1). In deze versie 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 source code presenteren van een programma waarin het type `Breuk` gedeclareerd, geïmplementeerd en gebruikt wordt. Meteen daarna zal ik op het bovengenoemde punt ingaan.

```
#include <iostream>
#include <cassert>
```

---

[30] (...vervolg)
error: *passing `const Breuk' as `this' argument of `void Breuk::drukaf()' discards qualifiers.* Cryptisch maar wel correct.

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

[32] Door het definieren 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.

```
using namespace std;

class Breuk {
public:
     Breuk(int t, int n);
     int teller() const;
     int noemer() const;
     void operator+=(Breuk[33] right);
private:
     int boven;
     int onder;
     void normaliseer();
};

// ...

void Breuk::operator+=(Breuk right) {
     boven=boven*right.onder + onder*right.boven;
     onder=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 TICPP Chapter12.html.)

In de taal C++ kun je de betekenis van operatoren (zoals bijvoorbeeld `+=`) definiëren voor zelf gedefinieerde typen. 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 (niet erg duidelijke) foutmelding op:

```
//   Error: Illegal structure operation.
```

[33] Het is beter om hier een `const` reference parameter te gebruiken. Dit wordt pas behandeld op blz. 31.

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

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

De memberfunctie operator+= in deze tweede versie van Breuk heeft dezelfde implementatie als de memberfunctie plus in de eerste versie van Breuk (zie blz. 19).

Je kunt voor een zelf gedefinieerd type alle operatoren zelf definiëren behalve de operator . waarmee een member geselecteerd wordt en de operator ?:[34]. 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 typen 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 zelf gedefinieerde typen 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 zelf gedefinieerde 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 links naar rechts 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:

```
//   Error: Illegal structure operation.
```

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 operator+=(Breuk right);
```
[35]

De definitie is dan als volgt:

```
Breuk Breuk::operator+=(Breuk right) {
     boven=boven*right.onder + onder*right.boven;
     onder=onder*right.onder;
     normaliseer();
     return DIT_OBJECT;      // 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.

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

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

2.13 `this` pointer.

Elke memberfunctie kan beschikken over een impliciet argument 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)[36] {
     boven=boven*right.onder + onder*right.boven;
     onder=onder*right.onder;
     normaliseer();
     return *this;
}
```

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` veranderd 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 TICPP Chapter11.html#Heading326.)

Een *reference* is niets anders dan een alias (andere naam) voor de variabele waarnaar hij refereert. Alle operaties die op een variabele zijn toegestaan zijn ook toegestaan op een reference die naar die variabele verwijst[37].

```
int i(3)[38];
int& j(i);      // een reference moet geïnitialiseerd worden
                // er bestaat nu 1 variabele met 2 namen (i en j)
```

---

[36] Deze `operator+=` is niet juist! Zie blz. 33 voor een correcte implementatie van de `operator+=`.

[37] 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 refereert 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` refereert kun je later niet naar een andere variabele laten refereren.
- Een pointer kan ook naar geen enkele variabele wijzen (de waarde is dan `0`) maar een reference verwijst altijd naar een variabele.

[38] 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);` gebruiken, alsof het ingebouwde type `int` een constructor heeft die je aanroept. Deze syntax heeft de voorkeur omdat deze syntax ook werkt bij zelf gedefinieerde types (classes) zoals `Breuk`. Een object `b` van de class `Breuk` kun je alleen initialiseren met de syntax `Breuk b(3, 7);`

```
j=4;           // i is nu gelijk aan 4
               // een reference is gewoon een "pseudoniem"
```

2.15 Reference parameters. (Zie eventueel TICPP Chapter11.html#Heading327.)

In C worden bij het aanroepen van een functie de parameters gekopieerd. Dit betekent dat de parameters 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 formele parameter genoemd. Een parameter die bij een aanroep van een functie gebruikt wordt, wordt actuele parameter genoemd. Bij een functieaanroep wordt dus de **waarde** van de actuele parameter naar de formele parameter gekopieerd[39]. Het is dus geen probleem als de actuele parameter een constante is.

```
void skipLines(int l) {
     while (l-- > 0)
          cout<<endl;
}
// ...
int n(7);
skipLines(n);             // waarde van n wordt gekopieerd naar l
cout<<"n = "<<n<<endl;    // n = 7
skipLines(3);             // waarde 3 wordt gekopieerd naar l
```

Als je de parameter die 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);
```

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

```
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 te kopiëren maar door het adres te kopiëren. Als er dan in de functie aan de formele parameter een nieuwe waarde wordt toegekend, dan wordt deze waarde dus direct op het adres van de actuele parameter opgeslagen.

---

39 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 de meegegeven parameter zelf (in dit geval het adres van de variabele) kan in de functie niet veranderd worden.

Dit betekent dat er een probleem ontstaat als de actuele parameter een constante is omdat aan een constante geen nieuwe waarde toegekend mag worden. Bijvoorbeeld:

```
swapInts(i, 5);
```

De C++ Builder compiler geeft de volgende foutmeldingen:

```
Reference initialized with 'int', needs lvalue of type 'int'
Type mismatch in parameter 'q' (wanted 'int &', got 'int')
```

2.16 `const` reference parameters. (Zie eventueel TICPP Chapter11.html#Heading328.)

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 (op de meeste systemen 4 bytes) gekopieerd terwijl bij een value parameter de waarde (aantal bytes afhankelijk van het type) gekopieerd wordt. Als de waarde veel geheugenruimte in beslag neemt dan is het om performance redenen beter om een reference parameter te gebruiken.[40] Om er voor te zorgen dat deze functie toch aangeroepen kan worden met een constante (zonder dat er een tijdelijke kopie wordt gemaakt) 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
}
```

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

```
void drukaf(const Tijdsduur& td) {
     td.uur=23;                         // drukaf probeert vals te spelen
//   Error: Cannot modify a const object
}
```

In alle voorafgaande code waarin een `Breuk` als parameter is gebruikt kan dus beter een `const Breuk&` als parameter worden gebruikt. (Bijvoorbeeld op blz. 18, 21 en 27.)

2.17 Parameter FAQ.[41]

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 (de parameter die 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.

---

[40] Later zullen we zien dat er nog een veel belangrijke reden is om een reference parameter in plaats van een gewone parameter te gebruiken. (Zie paragraaf over polymorfisme.)

[41] FAQ=Frequently Asked Questions (veel gestelde vragen).

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`).

2.18 Reference return type.

Als een functie een waarde teruggeeft door middel van een `return` statement dan 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 gebruiken kun je een adres in plaats van een waarde teruggeven. Dit adres kun je dan gebruiken om een waarde aan toe te kennen.

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

int main() {
     int x(3), y(4);
     max(x,y)=0;              // y is nu 0
     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 zogenaamde “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[42]. De fout in de definitie van de functie `som` komt pas aan het licht als we de functie bijvoorbeeld als volgt aanroepen:

```
d=som(c, som(a, b));
```

Hetzelfde gevaar is trouwens ook aanwezig als je een pointer (bijvoorbeeld `Tijdsduur*`) als return type kiest (het zogenaamde “dangling pointer problem”). Je moet er dan voor oppassen dat de pointer niet naar een variabele wijst die na afloop van de functie niet meer bestaat[43].

---

[42] De C++ Builder 6 compiler herkent het dangling reference probleem en geeft de volgende foutmelding: “Attempting to return a reference to local variable `td`”. De gcc compiler geeft alleen maar een warning: “reference to local variable `s` returned”

[43] Vreemd genoeg herkent de C++ Builder 6 compiler het dangling pointer probleem niet. Wel wordt de volgende warning gegeven: “Suspicious pointer conversion”. De gcc compiler geeft de warning: “address of local variable `s` returned”.

2.19 Reference return type (deel 2).

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. 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& operator+=(const Breuk& right);
```

De definitie is dan als volgt:

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

2.20 Operator overloading (deel2).

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 b(*this);     // maak een kopie van dit object
     b+=right;           // tel daar het object right bij op
     return b;           // geef deze waarde terug
}

int main() {
     Breuk a(1,2);
     Breuk b(3,4);
     Breuk c;
     c=a+b;
// ...
```

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

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

2.21 `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 (zie de boeken: *Effective C++* en *More Effective C++*). Als `a`, `b` en `c` van het type `int` zijn dan levert de expressie `(a+b)+=c` de volgende (onduidelijke) foutmelding op "Error: Lvalue required". Dit is goed omdat het optellen van `c` bij een tijdelijke variabele (de som `a+b` wordt namelijk 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.[44] 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.[45]

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 zogenaamde dangling reference (zie blz. 32) omdat de lokale variabele na afloop van de memberfunctie opgeruimd wordt.

Q: Kun je in de `operator+` de benodigde lokale variabele niet met aanmaken zodat we toch een reference kunnen teruggeven? De met `new`[46] 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!

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.

---

[44] Zoals al eerder vermeld is geeft de C++ Builder 6 compiler bij deze fout geen error (zoals volgens de standaard zou moeten) maar alleen een warning.

[45] Dat dit nogal subtiel is blijkt wel uit het feit dat Bjarne Stroustrup (de ontwerper van C++) in een vergelijkbaar voorbeeld in zijn boek *The C++ programming language 3ed* geen `const` bij het return type gebruikt.

[46] De operator `new` wordt pas besproken op pagina 74 van dit dictaat.

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[47]. De Borland Builder C++ compiler genereert de volgende foutmelding:

```
//   Error: illegal structure operation.
```

De gcc compiler genereert een veel duidelijkere foutmelding:

```
//   Error: 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.

2.22 Operator overloading (deel 3).

Naast het definiëren van operatoren als memberfuncties van zelf gedefinieerde typen kun je ook de globale operatoren overloaden[48]. 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 (=function) 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;
}
```

Deze implementatie roept simpel de `Breuk::operator+` memberfunctie aan!

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);
    // ...
};
```

[47] Op blz. 37 zal ik bespreken hoe je deze type conversie indien gewenst zelf kunt definiëren.

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

```
const Breuk operator+(const Breuk& left, const Breuk& right) {
     Breuk copyLeft(left);
     copyLeft+=right;
     return copyLeft;49
}

int main() {
     Breuk b(1,2);
     Breuk b1(b+3); // wordt: Breuk b1(operator+(b, Breuk(3)));
     Breuk b2(3+b); // wordt: Breuk b2(operator+(Breuk(3), b));
// ...
```

De unary operatoren (dat zijn operatoren met 1 operand, zoals `!`) en de assignment operatoren (zoals `+=`) kunnen het beste als memberfunctie overloaded worden. De overige binary operatoren 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.

2.23 Overloaden `operator++` en `operator--`. (Zie eventueel TICPP Chapter12.html#Heading353.)

Bij het overloaden van de `operator++` en `operator--` ontstaat een probleem omdat beide zowel een *prefix* als een *postfix* operator variant kennen[50]. Dit probleem is opgelost door de postfix versie te voorzien van een (dummy) `int` argument.

Voorbeeld van het overloaden van `operator++` voor objecten van de class `Breuk`. De implementie van deze memberfuncties wordt op blz. 38 gegeven.

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

Het gebruik van het return type `Breuk&` in plaats van `const Breuk&` bij de prefix `operator++` zorgt ervoor dat de expressie `++++b` als `b` van het type `Breuk` is gewoon werkt (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 (warning in C++ Builder 6) geeft (net zoals bij het type `int`).

---

[49] 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;
}
```

[50] 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;`.

2.24 Conversie operatoren. (Zie eventueel TICPP Chapter12.html#Heading374.)

Een constructor van class `Breuk` met 1 argument van het type `T` wordt door de compiler gebruikt als een variabele van het type `T` moet worden geconverteerd naar het type `Breuk` (zie blz. 23). 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`?

2.25 Voorbeeld class `Breuk` (derde versie).

Dit voorbeeld is een uitbreiding van de op blz. 26 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 operator `<<` 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 source code presenteren van een programma 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& left, const Breuk& right);
friend bool operator==(const Breuk& left, const Breuk& right);
};

istream& operator>>(istream& left, Breuk& right);
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) {
     // ...
}

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() {
     // ...
}

const Breuk operator+(const Breuk& left, const Breuk& right) {
     return Breuk(left)+=right;
}
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;
     cout<<"Geef Breuk: ";
     cin>>b1;
     cout<<"Geef nog een Breuk: ";
     cin>>b2;
     cout<<b1<<" + "<<b2<<" = "<<(b1+b2)<<endl;
     Breuk b3(18, -9);
     if (-2!=b3)
          cout<<"Error."<<endl;
     else
          cout<<"OK."<<endl;
     cout<<b3++<<endl;
     cout<<b3<<endl;
     b3+=5;
     cout<<b3<<endl;
     cin.get();
     cin.get();
     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:

```
//   Error: 'Breuk::boven' is not accessible
//   Error: 'Breuk::onder' is not accessible
```

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*.

2.26 `friend` functions. (Zie eventueel TICPP Chapter05.html#Heading212.)

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 dan 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:
     // ...
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 function binnen de class gedeclareerd wordt is een friend function 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 function in tegenspraak is met het principe van information hiding. Dit is echter niet het geval; een class beslist namelijk zelf wie zijn vrienden zijn. Voor alle overige functies (geen member en geen friend) geldt nog steeds dat de private datamembers 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 worden 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 function van de class Breuk te definiëren.

2.27 Operator overloading (deel 4). (Zie eventueel TICPP Chapter12.html#Heading364.)

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[51].

```
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 parameter 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;
```

[51] 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.

Omdat alle ingebouwde overloaded `<<` operatoren met als eerste argument een `ostream&` deze referen-ce ook weer als return waarde teruggeven kun je de bovenstaande `operator<<` vereenvoudigen tot:

```
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.

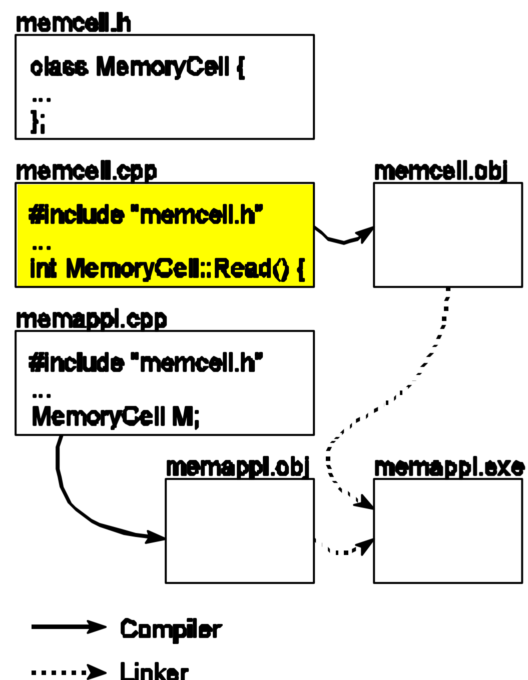
```
cin>>a>>b;
```

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 blz. 18 dan is eigenlijk het enige belangrijke verschil dat het zelf gedefinieerde type `Breuk` nu op precies dezelfde wijze als de ingebouwde typen te gebruiken is. Je hebt er ongeveer 20 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 20 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 zelf gedefinieerde type, als hij of zij de naam `Breuk` maar kent, als vanzelf (intuïtief) gebruiken. Er is daarbij geen handleiding of helpfile 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.

2.28 Voorbeeld separate compilation van class `MemoryCell`.

Je kunt de *interface* (=declaratie) en *implementatie* (=definitie) van een class splitsen in een `.h` en een `.cpp` file. Dit heeft als voordeel dat je de imple-mentatie afzonderlijk kunt compileren tot een `.obj` file. De gebruiker van de class kan dan de `.h` file gebruiken in zijn eigen programma en ver-volgens de `.obj` file met zijn eigen programma mee "linken". De gebruiker hoeft dan dus niet te beschikken over de implementatie.



Voor de class `MemoryCell` zien de `.h` en `.cpp` file er als volgt uit:

```
// Dit is file memcell.h
// prevent multiple inclusion.
```

```
#ifndef _memcell_
#define _memcell_
class MemoryCell {
public:
     int Read() const;
     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>
using namespace std;
#include "memcell.h"
int main() {
     MemoryCell M;
     M.Write(5);
     cout<<"Cell contents are "<<M.Read()<<endl;
// ...
```

[52]

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 blz. 30 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;
}
```

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

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 int array een double array 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 template functie 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 zogenaamde *generieke* functie, een functie die je als het ware voor verschillende datatypes kunt gebruiken.

De template functie definitie voor swap ziet er als volgt uit:

```
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[53]. Dit wordt aangegeven door het keyword typename[54] 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[55] in de parameterlijst van de functie gebruikt worden. Deze template functie definitie 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 parameters de benodigde "versie" van swap genereren[56] door voor het template argument (in dit geval T) het betreffende type in te vullen.

---

53 Een template kan ook "normale" parameters hebben. Dit komt verderop in dit dictaat aan de orde.

54 In plaats van typename mag ook class gebruikt worden. Omdat het keyword typename pas laat in de ISO/ANSI C++ standaard is opgenomen 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 zelfgedefineerde types (classes) als ingebouwde typen (zoals int) gebruikt kunnen worden.

55 Dit is niet helemaal waar. Zie de volgende voetnoot.

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

Dus de aanroep:

```
int x(3);
int y(4);
swap(x, y);
```

heeft tot gevolg dat de volgende functie gegenereerd wordt[57]:

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

Als de functie `swap` daarna opnieuw met twee `int`'s als parameters 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[58]:

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

Het gebruik van een template functie 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 template functie aan te brengen en het programma opnieuw te compileren.

Het gebruik van een template functie heeft echter ook het volgende nadeel:

- Doordat de compiler de volledige template functie definitie nodig heeft om een functie aanroep te kunnen vertalen moet de template functie definitie in een headerfile (`.h` file) opgenomen worden en "included" worden in elke `.cpp` file waarin de template functie gebruikt wordt. Het is niet mogelijk om de template functie afzonderlijk te compileren tot een `.obj` 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 template functie in plaats van "met de hand" verschillende versies van een functie aan te maken wordt een programma **beter onderhoudbaar** en **eenvoudiger uitbreidbaar**.

---

[57] Als je zelf deze functie al gedefinieerd hebt dan zal de compiler geen functie genereren maar de al gedefineerde functie gebruiken. Dit geeft ons de mogelijkheid om een template functie te definiëren met een aantal uitzonderingen voor bepaalde (van tevoren gedefinieerde) typen.

[58] Hier blijkt duidelijk het belang van function name overloading (zie blz. 10).

3.2 Template classes. (Zie eventueel TICPP Chapter16.html.)

In de vorige paragraaf hebben we gezien dat je een template van een functie kunt maken waardoor de functie generiek wordt. Een genierieke functie kun je voor verschillende datatypes gebruiken. Als je de generieke functie aanroept zal de compiler zelf afhankelijk van het type van de gebruikte parameters de benodigde "versie" van de generieke functie genereren door voor het template argument het betreffende type in te vullen. Natuurlijk is het ook mogelijk om een template memberfunctie 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 12 integers kunt opslaan. Daarna maken we een template van deze class zodat we deze class niet alleen kunnen gebruiken voor het opslaan van 12 integers maar voor het opslaan van 12 elementen van *elk* type.

De class `Dozijn` waarin 12 integers kunnen worden opgeslagen kan als volgt gedeclareerd worden:

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

In een `Dozijn` kunnen dus 12 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()<<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[59] */
}
```

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:

[59] Het is in C++ mogelijk om een functie te verlaten zonder dat een returnwaarde wordt teruggegeven door een zogenaamde exception te gooien. Exceptions worden in dit dictaat niet behandeld.

```
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 d1;
     for (int j=0; j<12; ++j)
          d1.zetIn(j, j*j);  // vul d1 met kwadraten
     cout<<"d1 = "<<d1<<endl;
     cin.get();
     return 0;
}
```

Dit programma geeft de volgende uitvoer:

```
d1 = 0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100, 121
```

In objecten van de class `Dozijn` kunnen 12 elementen van het type `int` worden opgeslagen. Als je een `Dozijn` met elementen van het type `double` nodig hebt, dan 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 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 template class `Dozijn` wordt bepaald. Bij het gebruik van de template class `Dozijn` kan de compiler niet zelf bepalen wat het type `T` moet zijn. Vandaar dat je dit bij het gebruik van de template class `Dozijn` zelf moet specificeren. Bijvoorbeeld:

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

3.3 Voorbeeld template class `Dozijn`.

De *template class* `Dozijn` kan als volgt worden gedeclareerd:

```
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 pameter `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 returntype 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 *template* memberfuncties. 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 blz. 46) 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 template class `Dozijn` kan nu als volgt gebruikt worden:

```
int main() {
     Dozijn<int> d1;
     for (int j=0; j<12; ++j)
          d1.zetIn(j, j*j);  // vul d1 met kwadraten
     cout<<"d1 = "<<d1<<endl;
     Dozijn<string> d2;
     d2.zetIn(0, "Drenthe");
     d2.zetIn(1, "Flevoland");
// ...
     d2.zetIn(10, "Zeeland");
     d2.zetIn(11, "Zuid-Holland");
     cout<<"d2 = "<<d2<<endl;
     cin.get();
     return 0;
}
```

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

De uitvoer van het bovenstaande programma is:

```
d1 = 0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100, 121
d2 = Drenthe, Flevoland, Friesland, Gelderland, Groningen, Limburg,
Noord-Brabant, Noord-Holland, Overijssel, Utrecht, Zeeland,
Zuid-Holland
```

Je ziet dat je de template class `Dozijn` kunt gebruiken om 12 elementen van het type `int` op te slaan maar ook om 12 elementen van het type `string` op te slaan.

3.4 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 typen (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 typen) specificeren.

Voor al deze details verwijs ik je naar Volume 2 van TICPP.

3.5 Standaard Templates.

In september 1998 is de ISO/ANSI C++ standaard officieel vastgesteld. In deze standaard zijn een groot aantal standaard templates voor datastructuren en algoritmen opgenomen. Deze in de standaard opgenomen verzameling tempates is grotendeels afkomstig uit de STL (Standard Template Library) een verzameling templates die in het begin van de jaren '90 ontwikkeld werd door Alex Stepanov en Meng Lee van Hewlett Packard Laboratories en sinds 1994 via internet gratis is verspreid. In deze standaard library is ook een generiek type `vector` opgenomen.

3.6 `std::vector`.

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

- Een `vector` is in tegenstelling tot een built-in array een “echt” object.
- Je kunt een `vector` “gewoon” vergelijken en toekennen.
- 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[60] 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`.
  - ...

Voorbeeld van een programma met een `vector`.

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

int main() {
     // definieer vector van integers
     vector<int> v;
     // vul met kwadraten
     for (int i=0; i<15; ++i) {
          v.push_back(i*i);
```

[60] Een exception is een manier om een fout vanuit een component door te geven aan de code die deze component gebruikt. Exceptions worden in dit dictaat niet behandeld.

```
    }
    // druk af
    for (vector<int>::size_type i=0; i<v.size(); ++i) {
        cout<<v[i]<<" ";
    }
    cout<<endl;

    // toekennen van de een vector
    vector<int> w;
    w=v;
    for (vector<int>::size_type i=0; i<w.size(); ++i) {
        cout<<w[i]<<" ";
    }
    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.

```
#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 (vector<int>::size_type i(0); i<v.size(); ++i) {
        cout<<v[i]<<" ";
```

```
    }
    cout<<endl;
    cin.get();
    cin.get();
    return 0;
}
```

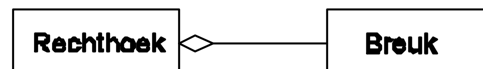
4 Inheritance.

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) software-component 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 template class `Breuk` lijkt op het eerste gezicht al een prima herbruik-bare software component. 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 bij-voorbeeld `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 object georiënteerde techniek (inheritance) een onderhoudbare variant van een herbruikbare software component kan maken. Door middel van deze techniek kun je dus software componenten niet alleen hergebruiken op de manier zoals de ontwerper van de component dat bedoeld heeft, maar kun je de software component ook naar je eigen wensen omvor-men.

Hergebruik door middel van *composition* is niet specifiek voor OOP. Ook bij de gestructureerde pro-grammeer methode paste je deze vorm van hergebruik al toe. Het hergebruik van een software compo-nent door middel van composition is niets anders als het gebruiken van deze component als onderdeel van een andere (nieuwe) software component. 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;
};
```

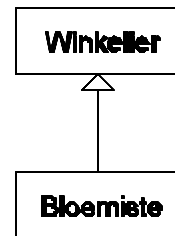


We zeggen dan dat de class `Rechthoek` een HAS-A (heeft een of meer) relatie heeft met de class `Breuk`. Schematisch kan dit zoals hierboven getekend worden weergegeven.[61] Bij gestructureerd programmeren is dit de enige relatie die software componenten met elkaar kunnen hebben. Bij object georiënteerd programmeren bestaat ook de z.g 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 blz. 56. Ook zal ik dan bespreken wat de voordelen van een object georiënteerde benadering zijn ten opzichte van een gestructureerde of ADT benadering.

[61] De hier gebruikte tekennotatie heet UML (Unified Modelling Language) en is een standaard notatie die veel bij object georiënteerd ontwerpen wordt gebruikt.

4.1 De syntax van inheritance. (Zie eventueel TICPP Chapter14.html#Heading406.)



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 super class). De class die hiervan afgeleid (Engels: derived) wordt, wordt de *derived* class genoemd (of ook wel child class of sub class). Schematisch kan dit zoals hiernaast getekend worden aangegeven. In C++ code wordt dit als volgt gedeclareerd:

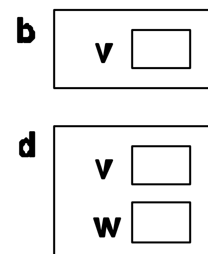
```
class Winkelier {
     // ...
};
class Bloemiste: public[62] Winkelier {
     // Bloemiste is afgeleid van Winkelier
     // ...
};
```

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 datamembers en (minmaal) 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.[63] 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) base class. Omdat een derived class datamembers en memberfuncties kan toevoegen aan de base class is het omgekeerde niet waar. Je kunt een object 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 datamembers van de base class over. Dit wil zeggen dat een object van een derived class (minimaal) dezelfde datamembers heeft als een object van de base class. Private datamembers 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” datamembers toevoegen. Als de objecten `b` en `d` op onderstaande wijze gedefinieerd zijn dan kan de structuur van deze objecten weergegeven worden zoals daarnaast getekend is. Het object `b` bevat alleen een datamember `v` en een object `d` bevat zowel een datamember `v` als een datamember `w`. Het datamember `v` is alleen toegankelijk vanuit de memberfuncties van de class `Base` en het datamember `w` is alleen toegankelijk vanuit de memberfuncties van de class `Derived`.

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

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



---

62 Er bestaat ook private inheritance maar dit wordt in de praktijk niet veel gebruikt en zal ik 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).

63 Maar pas op voor het slicing probleem dat ik later (blz. 67) zal bespreken.

```
// ...
Base b;
Derived 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 dan 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; }[64]
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. (Zie eventueel TICPP Chapter15.html.)

Als de classes `Base` en `Derived` zoals hierboven gedefinieerd zijn, dan 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[65]. Door het toepassen van polymorfisme kan je

[64] Alle memberfuncties zijn in dit voorbeeld in de class zelf gedefinieerd. Dit zogenaamd *inline* definieren heeft als voordeel dat ik iets minder hoef te typen, maar het nadeel is dat de class niet meer afzonderlijk gecompileerd kan worden (zie blz. 41).

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

software maken die eenvoudig aangepast, uitgebreid en hergebruikt kan worden. (Zie het uitgebreide voorbeeld op blz. 56.)

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).

4.3 Memberfunctie overriding.

Een derived class kan, zoals we al hebben gezien, datamembers 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.[66]

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

class Derived: public Base {
public:
     virtual67 void printName() {
          cout<<"Ik ben een Derived."<<endl;
     }
};
```

Als via een polymorfe pointer een memberfunctie wordt aangeroepen dan 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 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 dan 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);
```

---

[66] 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. Overlaoding geeft onverwachte (en meestal ongewenste) effecten zoals je later (blz. 64) zult zien.

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

```
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 orginele 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() {
          cout<<"Ik ben een Base."<<endl;
     }
};

class Derived: public Base {
public:
     virtual void printName() {
          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) om er voor te zorgen dat er geen problemen ontstaan bij polymorf gebruik van de class. Simpel gesteld luidt de regel van Liskov: *"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` dan 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 datamembers 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.

4.4 Abstract base class. (Zie eventueel TICPP Chapter15.html#Heading447.)

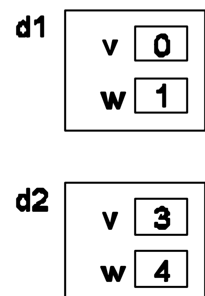
Het is mogelijk om een virtual memberfunctie in een Base class alleen maar te declareren en nog niet te implementeren. Dit wordt dan een "*pure virtual*" memberfunctie genoemd en de betreffende base class wordt een zogenaamde "*Abstract Base Class (ABC)*". Een virtual memberfunctie kan pure virtual gemaakt 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, dan 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 wil "aanroepen" vanuit de constructor van de derived class dan kan dit door deze aanroep in de initialisation 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 initialisation list).

Als de objecten `d1` en `d2` op onderstaande wijze gedefinieerd zijn dan zijn deze objecten geïnitialiseerd zoals daarnaast getekend is.

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

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

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



4.6 `protected` members. (Zie eventueel TICPP Chapter14.html#Heading420.)

Het is in C++ ook mogelijk om in een base class datamembers 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 class declaratie 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 afge-
     // leid zijn van de class Toegang en vanuit de class Toegang zelf.
private:
     // alleen toegankelijk vanuit de class Toegang zelf.
};
```

Het definiëren van protected datamembers wordt afgeraden omdat dit slecht is voor de onderhoudbaarheid van de code. Als een protected datamember een illegale waarde krijgt moet alle source code worden doorzocht om de plaatsen te vinden waar deze datamember veranderd wordt. In elke afgeleide class is het protected datamember 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 afgeleide 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 object georiënteerde oplossing bespreken.[68] 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 typen 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 typen gebruikt kunnen worden. In de applicatie is alleen de spanning in volts van 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 en
- `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: de kaartsoort, 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 struct `ADCCard` doorgegeven.

```
enum CardName {AD178, NI323};

struct ADCCard {
    CardName t;         // card type
    double f;           // amplifying factor
```

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

```
     int c;              // selected channel
};

void initCard(ADCCard& card, CardName name) {
     card.t=name;
     card.f=1.0;
     card.c=1;
     // eventueel voor alle kaarten benodigde code
     switch (card.t) {
          case AD178:
               // de specifieke voor de AD178 benodigde code
               cout<<"AD178 is geinitialiseeerd."<<endl;
               break;
          case NI323:
               // de specifieke voor de NI323 benodigde code
               cout<<"NI323 is geinitialiseeerd."<<endl;
               break;
     }
     // eventueel voor alle kaarten benodigde code.
}

void selectChannel(ADCCard& card, int channel) {
     card.c=channel;
//   ... zelfde switch instructie als bij initCard
}

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

void setAmplifier(ADCCard& card, double factor) {
     card.f=factor;
//   ... zelfde switch instructie als bij initCard
}

int sampleCard(const ADCCard& card) {
     int sample; // Niet portable! Gaat alleen goed als int 16 bits is.
//   ... zelfde switch instructie als bij initCard
     return sample;
}

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

int main() {
     ADCCard c2;
     initCard(c2, NI323);
     setAmplifier(c2, 5);
     selectChannel(c2, 4);
     cout<<"Kanaal "<<getChannel(c2)<<" van kaart c2 = ";
     cout<<readCard(c2)<<" V."<<endl;
// ...
```

Het `switch` statement uit `initCard` wordt in de functies `selectChannel`, `setAmplifier` en `sampleCard` op soortgelijke wijze gebruikt.

Het initialiseren, het instellen van een versterkingsfactor van 10 en het afdrukken van de waarde van kanaal 3 gaat dan bij het gebruik van een AD178 als volgt:

```
ADCCard adc;
initCard(adc, AD178);
setAmplifier(adc, 10);
selectChannel(adc, 3);
cout<<"Kanaal "<<getChannel(adc)<<" = "<<readCard(adc)<<" V."<<endl;
```

Hopelijk heb je al lang 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 kaart type verschillende code worden 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[69].

```
enum CardName {AD178, NI323};

class ADCCard {
public:
     ADCCard(CardName name);
     void selectChannel(int channel);
     int getChannel() const;
     void setAmplifier(double factor);
     double read() const;
private:
     CardName t;          // card type
     double f;            // amplifying factor
     int c;               // selected channel
     int sample() const;
};

ostream& operator<<(ostream& out, const ADCCard& card) {
     return out<<card.read()<<" V.";
}
```

[69] 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.

```
ADCCard::ADCCard(CardName name): t(name), f(1.0), c(1) {
     // eventueel voor alle kaarten benodigde code
     switch (t) {
          case AD178:
               // de specifieke voor de AD178 benodigde code
               cout<<"AD178 is geinitialiseeerd."<<endl;
               break;
          case NI323:
               // de specifieke voor de NI323 benodigde code
               cout<<"NI323 is geinitialiseeerd."<<endl;
               break;
     }
     // eventueel voor alle kaarten benodigde code.
}

void ADCCard::selectChannel(int channel) {
     c=channel;
//   ... zelfde switch instructie als bij initCard
}

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

void ADCCard::setAmplifier(double factor) {
     f=factor;
//   ... zelfde switch instructie als bij initCard
}

int ADCCard::sample() const {
     int sample;
//   ... zelfde switch instructie als bij initCard
     return sample;
}

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

Het `switch` statement uit `initCard` wordt in de functies `selectChannel`, `setAmplifier` en `sampleCard` op soortgelijke wijze gebruikt. Het overloaden van de `operator<<` voor een `ostream` en een `ADCCard` maakt het afdrukken van de waarde in volts erg eenvoudig.

Het initialiseren, het instellen van een versterkingsfactor van 10 en het afdrukken van de waarde van kanaal 3 gaat dan bij het gebruik van een AD178 als volgt:

```
ADCCard adc(AD178);
adc.setAmplifier(10);
adc.selectChannel(3);
cout<<"Kanaal "<<adc.getChannel()<<" = "<<adc<<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 kaart type verschillende code worden 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.

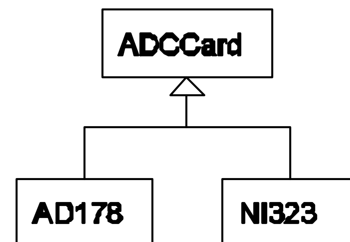
Stel dat ik niet zelf het ADT `ADCCard` heb ontwikkeld, maar dat ik deze herbruikbare ADT heb ingekocht. Als een geschikte ADT te koop is dan heeft kopen de voorkeur boven zelf maken om een aantal redenen:

- 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), dan 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 zelf ontwikkelde 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 source code `adccard.cpp` file niet aan ons beschikbaar stelt[70]. Het toevoegen van de nieuwe ADC kaart (BB647) is dan niet mogelijk.

De (gekochte) ADT `ADCCard` is een herbruikbare software component die echter niet door de gebruiker uit te breiden is. Je zult zien dat je door het toepassen van de object georiënteerde technieken inheritance en polymorfisme wel een software component kan maken die door de gebruiker uitgebreid kan worden zonder dat de gebruiker de source code van de originele component hoeft te wijzigen.

4.7.4 Een object georiënteerde oplossing.



Bij het toepassen van de object georiënteerde benadering constateer ik dat in het programma twee typen ADC kaarten gebruikt worden. Beide kaarten hebben dezelfde functionaliteit en kunnen dus worden afgeleid van dezelfde base class[71]. Dit is hier schematisch weergeven. Ik heb de base class als volgt gedeclareerd:

```
class ADCCard {
public:
     ADCCard();
     virtual ~ADCCard()72;
     virtual void selectChannel(int channel) =0;
     int getChannel() const;
```

[70] Ook als de source code 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.

[71] 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 E studenten later in deze onderwijseenheid behandeld. Bij TI is dit al behandeld en komt het later in andere onderwijseenheden nog uitgebreid aan de orde.

[72] Dit is een zoganaamde virtual destructor, waarom het nodig is om een virtual destructor te definiëren wordt verderop in dit dictaat besproken.

```
     virtual void setAmplifier(double factor) =0;
     double read() const;
protected:
     void rememberChannel(int channel);
     void rememberAmplifier(double factor);
private:
     double f;          // amplifying factor
     int c;             // selected channel
     virtual int sample() const =0;
};
```

De memberfuncties `selectChannel`, `setAmplifier` en `sample` heb ik pure virtual (zie blz. 55) gedeclareerd omdat de implementatie per kaart type 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 op blz. 64 bespreken. De datamembers `f` en `c` 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.

```
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;
};
```

Ik heb operator overloading toegepast om een object van een van `ADCCard` afgeleide class eenvoudig te kunnen afdrukken:

```
ostream& operator<<(ostream& out, const ADCCard& card) {
     return out<<card.read()<<" V.";
}
```

De diverse classes heb ik als volgt geïmplementeerd:

```
ADCCard::ADCCard(): f(1.0), c(1) {
     // voor alle kaarten benodigde code
}
int ADCCard::getChannel() const {
     return c;
```

```
}
double ADCCard::read() const {
     return sample()*f/6553.5;
}
void ADCCard::rememberChannel(int channel) {
     c=channel;
}
void ADCCard::rememberAmplifier(double factor) {
     f=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 {
     int sample;
     // de specifieke voor de AD178 benodigde code
     return sample;
}

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 {
     int sample;
     // de specifieke voor de NI323 benodigde code
     return sample;
}
```

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

```
void doIt(ADCCard& c) {
     c.setAmplifier(10);
     c.selectChannel(3);
     cout<<"Kanaal "<<c.getChannel()<<" = "<<c<<endl;
}
```

Deze functie kan ik dan als volgt gebruiken:

```
int main() {
     AD178 card1;
     doIt(card1);
     NI323 card2;
```

```
    doIt(card2);
// ...
```

Merk op dat door de overloaded `operator<<` in de functie `doIt` de in de base class `ADCCard` gedefinieerde memberfunctie `read` aangeroepen wordt 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:

```
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::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 {
    int sample;
    // de specifieke voor de NI323 benodigde code
    return sample;
}
```

Het programma `main` kan nu als volgt aangepast worden:

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

Als alle functie en class declaraties in aparte `.h` en alle functie en class definities in aparte `.cpp` file 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) mee gelinkt 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 object georiënteerde 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 blz. 10 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 functies aanroept) slechts 1 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[73] van het gebruik van overloading van memberfuncties:

```
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).

Voorbeeld van het verkeerd gebruik van overloading en de hiding-rule:

```
// 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:
```

[73] In dit voorbeeld (en ook in enkele volgende voorbeelden) heb ik de memberfunctie in de class zelf gedefinieerd. Dit zogenaamd *inline* definiëren heeft als voordeel dat ik iets minder hoef te typen, maar het nadeel is dat de class niet meer afzonderlijk gecompileerd kan worden (zie blz. 41).

```
    void f(int i) const[74] { // Verberg f() !! Geen goed idee !!!
        cout<<"Ik ben f(int)"<<endl;
    }
};

int main() {
    Base b;
    Derived d;
    b.f();
//  d.f();
//  [C++ Error]: Too few parameters in call to 'Derived::f(int)'[75]
    d.f(3);
    d.Base::f();[76]
// ...
```

Uitvoer:

```
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`.

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

Uitvoer:

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

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

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

[76] Voor degene die echt alles wil weten: De hidden memberfunctie kan nog wel aangeroepen worden door gebruik te maken van zijn zogenaamde qualified name (`baseclassname::memberfunctionname`).

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 dan 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.

Op blz. 53 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[77]. Alleen memberfuncties van een "ouder of voorouder class" kunnen overridden worden in de "kind class". Als een overridden memberfunctie via een polymorfe pointer of reference (zie blz. 52) aangeroepen wordt, dan wordt tijdens het uitvoeren van het programma bepaald naar welk type object de pointer wijst (of de reference refereert). 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 de 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) of
- 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).[78] [79]

Voorbeeld van het gebruik van overriding en verkeerd gebruik van overloading.
De functie `f` wordt overloaded en de functie `g` wordt overridden:

```
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;
     }
     // ...
```

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

[78] Voor degene die echt alles wil weten: de overridden memberfunctie kan wel door gebruik te maken van zijn zogenaamde qualified name (`Baseclassname::memberfunctionname`) aangeroepen worden.

[79] Voor degene die echt alles wil begrijpen: een functie met dezelfde parameters in twee classes zonder overervings relatie zijn overloaded omdat de impliciete parameter "`this`" verschillend is.

```
};

class Derived: public Base {
public:
     void f(int i) const {
          cout<<"Derived::f(int) called."<<endl;
     }
     virtual void g(int i) const {
          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 Slicing problem. (Zie eventueel TICPP Chapter15.html#Heading450.)

Een object van een class `Derived`, die public overerft van class `Base`, mag worden toegekend aan een object van de class `Base` (`b=d`). Er geldt immers: een `Derived` is een `Base`. Dit levert problemen op als het object van de class `Derived` meer geheugenruimte inneemt dan een object van de class `Base`. Dit is het geval als in class `Derived` (extra) datamembers zijn opgenomen. Deze extra datamembers 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:

```
class Mens {
public:
     virtual void printSoort() {
          cout<<"Mens.";
     }
     virtual void printSalaris() {
          cout<<"Salaris = 0";
     }
     // ...
};
```

```
class Docent: public Mens {
public:
     Docent(): salaris(30000) {
     }
     virtual void printSoort() {
          cout<<"Docent.";
     }
     virtual void printSalaris() {
          cout<<"Salaris = "<<salaris;
     }
     virtual void verhoogSalarisMet(unsigned short v) {
          salaris+=v;
     }
     // ...
private:
     unsigned short salaris80;
};

int main() {
     Docent Bd;
     Bd.printSoort(); cout<<" ";
     Bd.printSalaris(); cout<<endl;
     Bd.verhoogSalarisMet(10000);
     Bd.printSalaris(); cout<<endl;

     Mens m(Bd);    // Waar blijft het salaris?
     m.printSoort(); cout<<" ";
     m.printSalaris(); cout<<endl;

     Mens& mr(Bd);
     mr.printSoort(); cout<<" ";
     mr.printSalaris(); cout<<endl;

     Mens* mp(&Bd);
     mp->printSoort(); cout<<" ";
     mp->printSalaris(); cout<<endl;
// ...
```

Uitvoer:

```
Docent. Salaris = 30000
Salaris = 40000
Mens. Salaris = 0
Docent. Salaris = 40000
Docent. Salaris = 40000
```

4.10 Voorbeeld: Opslaan van polymorfe objecten in een `vector`. (Zie eventueel TICPP Chapter15.html#Heading456.)

In het voorgaande programma heb je gezien dat alleen pointers en references polymorf kunnen zijn. Als we dus polymorfe objecten willen opslaan dan 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:

[80] De waarde van een `unsigned short` variable ligt tussen 0 en 65535 :-)

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

class Fruit {
public:
     virtual void printSoort() const =0;
// ...
};

class Appel: public Fruit {
public:
     virtual void printSoort() const {
          cout<<"Appel."<<endl;
     }
// ...
};

class Peer: public Fruit {
public:
     virtual void printSoort() const {
          cout<<"Peer."<<endl;
     }
// ...
};

class FruitMand {
public:
     void voegToe(Fruit& p) {
         fp.push_back(&p);
     }
     void printInhoud() const {
          cout<<"De fruitmand bevat:"<<endl;
          for (vector<Fruit*>::size_type i(0); i<fp.size(); ++i)
               fp[i]->printSoort();
     }
private:
     vector<Fruit*> fp;
};

int main() {
     FruitMand m;
     Appel a1, a2;
     Peer p1;
     m.voegToe(a1);
     m.voegToe(p1);
     m.voegToe(a2);
     m.printInhoud();

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

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.

4.11 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.

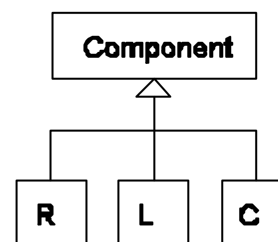
4.11.1 Weerstand, spoel en condensator.

Passieve elektrische componenten (hierna componenten genoemd) hebben een complexe impedantie Z. Deze impedantie is een functie van de frequentie f. Componenten hebben een impedantie en een waarde. Er bestaan 3 soorten (basis)componenten:

- R (weerstand): Z=waarde.
- L (spoel): Z=j*2*π*frequentie*waarde.
- C (condensator): Z=-j/(2*π*frequentie*waarde).

De classes `Component`, `R`, `L` en `C` hebben de volgende relaties (zie ook nevenstaande figuur):

- een `R` **is een** `Component`.
- een `L` **is een** `Component`.
- een `C` **is een** `Component`.



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()[81] {
     }
     virtual complex<double> Z(double f) const=0;
     virtual void print(ostream& o) const=0;
};
```

Het type `complex` is opgenomen in de ISO/ANSI standaard C++ library. Zie eventueel Volume 2 van TICPP.

De functie `Z` moet in een van `Component` afgeleide class de impedantie berekenen (een complex getal) bij de als parameter meegegeven frequentie `f`. De functie `print` moet in een van `Component` afgeleide class het type en de waarde afdrukken op de als parameter meegegeven output stream `o`. Bijvoorbeeld: `L(1E-3)` voor een spoel van 1mH.

Als we componenten ook met behulp van de operator<< willen afdrukken dan moeten we deze operator als volgt overloaden:

```
ostream& operator<<(ostream& o, const Component& c) {
     c.print(o);
     return o;
}
```

De classes `R`, `L` en `C` kunnen dan als volgt gebruikt worden:

```
void printImpedanceTable(const Component& c) {
```

[81] Dit is een zoganaamde virtual destructor, waarom het nodig is om een virtual destructor te definiëren wordt verderop in dit dictaat besproken.

```
    cout<<"Impedantie tabel voor: "<<c<<endl<<endl;
    cout<<"freq\tZ"<<endl;
    for (double freq(10);freq<10E6;freq*=10)
        cout<<setw(5)<<freq<<'\t'<<c.Z(freq)<<endl;
    cout<<endl<<endl;
}

int main() {
    R r(1E2);
    printImpedanceTable(r);
    cin.get();
    C c(1E-5);
    printImpedanceTable(c);
    cin.get();
    L l(1E-3);
    printImpedanceTable(l);
    cin.get();
    return 0;
}
```

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 zelf 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+06   (100,0)

Impedantie tabel voor: C(1e-05)

freq    Z
   10   (0,-1591.55)
  100   (0,-159.155)
 1000   (0,-15.9155)
10000   (0,-1.59155)
100000  (0,-0.159155)
1e+06   (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+06   (0,6283.19)
```

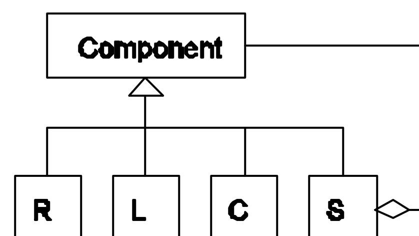
Vraag:
Implementeer nu zelf de classes `R`, `C` en `L`.

Antwoord:
```
class R: public Component { // R=Weerstand
public:
     R(double r): value(r) {
     }
     virtual complex<double> Z(double) const {
          return value;
     }
     virtual void print(ostream& o) const {
          o<<"R("<<value<<")";
     }
private:
     double value;
};

class L: public Component { // L=Spoel
public:
     L(double l): value(l) {
     }
     virtual complex<double> Z(double f) const {
          return complex<double>(0, 2*M_PI*f*value);
     }
     virtual void print(ostream& o) const {
          o<<"L("<<value<<")";
     }
private:
     double value;
};

class C: public Component { // C=Condensator
public:
     C(double c): value(c) {
     }
     virtual complex<double> Z(double f) const {
          return complex<double>(0, -1/(2*M_PI*f*value));
     }
     virtual void print(ostream& o) const {
          o<<"C("<<value<<")";
     }
private:
     double value;
};
```

4.11.2 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) kan dus van de class `Component` afgeleid worden. Dit heeft als bijkomend voordeel dat je het aantal componenten waaruit een serieschakeling bestaat tot 2 kunt beperken. Als je dan een

serieschakeling wilt doorrekenen van een R, L en C maak je bijvoorbeeld 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.

De classes S en P kunnen dan als volgt gebruikt worden:

```
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-06))//(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+06   (100,-0.106103)
```

Vraag:
Implementeer nu zelf de classes S en P.

Antwoord:

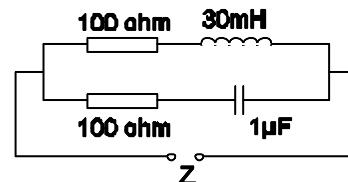
```
class S: public Component { // S = Serieschakeling van 2 componenten
public:
     S(const Component& c1, const Component& c2): comp1(c1), comp2(c2) {
     }
     virtual complex<double> Z(double f) const {
          return comp1.Z(f)+comp2.Z(f);
     }
     virtual void print(ostream& o) const {
          o<<"("<<comp1<<"+"<<comp2<<")";
     }
private:
     const Component& comp1;
     const Component& comp2;
     S(const S&);              // voorkom gebruik
     void operator=(const S&); // voorkom gebruik
};

class P: public Component { // P = Parallelschakeling van 2 componenten
public:
     P(const Component& c1, const Component& c2): comp1(c1), comp2(c2) {
     }
     virtual complex<double> Z(double f) const {
          return (comp1.Z(f)*comp2.Z(f)) / (comp1.Z(f)+comp2.Z(f));
```

```
    }
    virtual void print(ostream& o) const {
        o<<"("<<comp1<<"//"<<comp2<<")";
    }
private:
    const Component& comp1;
    const Component& comp2;
    P(const P&);                // voorkom gebruik
    void operator=(const P&); // voorkom gebruik
};
```

4.11.3 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/sopx2/winapps/pso2windows.html kun je, als je dat leuk vind, zelf leren hoe je met Borland C++ Builder grafische applications kunt maken.

Het nevenstaande netwerk moet als volgt worden ingevoerd:
(R(100)+L(3E-2))//(R(100)+C(1E-6))



4.12 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 TICPP Chapter14.html en Chapter15.html.

5 Dynamic memory allocation en destructors.

In dit hoofdstuk worden de volgende onderwerpen besproken:

- Dynamic memory allocation. (Zie eventueel TICPP Chapter04.html#Heading190 en Chapter13.html.)
- Destructor. (Zie eventueel TICPP Chapter06.html#Heading226.)

Deze technieken worden vervolgens toegepast bij het maken van een ADT `Array`.

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 (zogenaamde automatic) variabele wordt, zodra die variabele gedefinieerd wordt, gereserveerd op de stack. Als het blok (compound statement), waarin de variabele gedefinieerd 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[] (zie eventueel TICPP Chapter13.html#Heading393). 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 dan 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 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. Bij het ontwerpen van nieuwe software kan je het best van deze nieuwe operatoren gebruik maken.

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 5.5 zul je zien hoe, door het gebruik van dynamische geheugenallocatie 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.

5.2 Destructor ~Breuk. (Zie eventueel TICPP Chapter06.html#Heading226.)

Een class kan naast een aantal constructors (zie blz. 22) 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 blz. 20) 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;
     cout<<"Druk op enter om verder te gaan..."<<endl;
```

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

Het hoofdprogramma om de eerste versie van `Breuk` te testen is (zie blz. 22):

```
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 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
halve ==> 1/2
b3=halve ==> 1/2

Een breuk met de waarde 1/2 is verwijderd uit het geheugen.
Druk op enter om verder te gaan...

Een breuk met de waarde 1/2 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`, `b3` en `halve` 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[82]. 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 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 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 echt 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 2.5 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 dan 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?[83]

5.4 Virtual destructor. (Zie eventueel TICPP Chapter15.html#Heading456.)

Als een class nu of in de toekomst als base class gebruikt wordt dan moet de destructor virtual zijn zodat van deze class afgeleide classes via een polymorfe pointer “deleted” kan worden.

Hier volgt een voorbeeld van het gebruik van een ABC en polymorfisme:

```
class Fruit {
public:
    virtual ~Fruit() {
        cout<<"Er is een stuk Fruit verwijderd."<<endl;
    }
    virtual void printSoort()=0;
    // ...
};

class Appel: public Fruit {
public:
```

[82] 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.

[83] Omdat geheugenruimte in de omgekeerde volgorde van aanmaken moet worden vrijgegeven.

```
    virtual ~Appel() {
        cout<<"Er is een Appel verwijderd."<<endl;
    }
    virtual void printSoort() {
        cout<<"Appel."<<endl;
    }
    // ...
};

class Peer: public Fruit {
public:
    virtual ~Peer() {
        cout<<"Er is een Peer verwijderd."<<endl;
    }
    virtual void printSoort() {
        cout<<"Peer."<<endl;
    }
    // ...
};

class FruitMand {
public:
    ~FruitMand() {
        for (vector<Fruit*>::size_type i(0); i<fp.size(); ++i)
            delete fp[i];
    }
    void voegToe(Fruit* p) {
        fp.push_back(p);
    }
    void printInhoud() const {
        cout<<"De fruitmand bevat:"<<endl;
        for (vector<Fruit*>::size_type i(0); i<fp.size(); ++i)
            fp[i]->printSoort();
    }
private:
    vector<Fruit*> fp;
};

int main() {
    FruitMand m;
    m.voegToe(new Appel);
    m.voegToe(new Peer);
    m.voegToe(new Appel);
    m.printInhoud();
// ...
```

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 dan 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 blz. 52) aangeroepen wordt. Als de destructor virtual gedefinieerd is dan 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[84]. Omdat de destructor van een derived class ook altijd de destructor van zijn base class aanroept (zie blz. 55) wordt de destructor van `Fruit` ook aangeroepen. Als de destructor niet virtual gedefinieerd is dan wordt tijdens het compileren van het programma bepaald van welk type de pointer is. Vervolgens wordt de destructor van deze class (`Fruit`) aangeroepen[85]. In dit geval wordt dus alleen de destructor van de base class aangeroepen.

Let op! Als we een class aanmaken zonder destructor dan zal de compiler zelf een zogenaamde *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 definiëren!

5.5 Voorbeeld class `Array`.

Het in C (en dus ook 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, dan 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[86] waarbij:

- 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 voorbeeld worden de belangrijkste aspecten toegelicht en worden verwijzingen naar het TICPP boek gegeven.

---

[84] De destructor is dan dus *overridden*.

[85] De destructor is dan dus *overloaded*.

[86] In de ISO/ANSI standaard C++ library is ook een type `std::vector` opgenomen. Dit type is in paragraaf 3.6 besproken. Het verschil tussen onze zelfgemaakte `Array` en `std::vector` is dat een `std::vector` object kan groeien en krimpen nadat het is aangemaakt.

```
#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;
     int length() const;
     bool operator==(const Array& r) const;
     bool operator!=(const Array& r) const;
     // ...
     // Er zijn vele uitbreidingen mogelijk.
private:
     int size;
     int* data;
};

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 { //87
     assert(index>=0 && index<size);
     return data[index];
```

[87] 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 m.b.v `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 :-).

```
}
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<<"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. (Zie eventueel TICPP Chapter12.html#Heading373.)

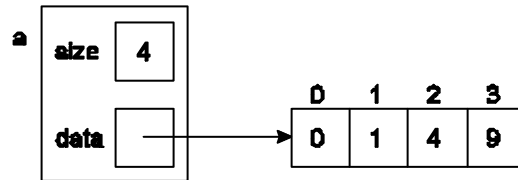
De constructor `Array(int)` is `explicit` gedeclareerd om te voorkomen dat de compiler deze constructor gebruikt om een `int` "automatisch" om te zetten naar een `Array`. Zie blz. 23.

5.7 Copy constructor en default copy constructor. (Zie eventueel TICPP Chapter11.html#Heading331.)

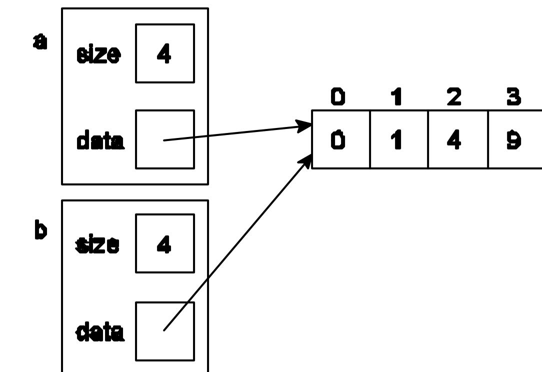
Een copy constructor wordt gebruikt als een object gekopieerd moet worden. Dit is het geval als:
- een object geïnitialiseerd wordt met een object van dezelfde class.
- een object als value parameter wordt doorgegeven aan een functie.

- 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.



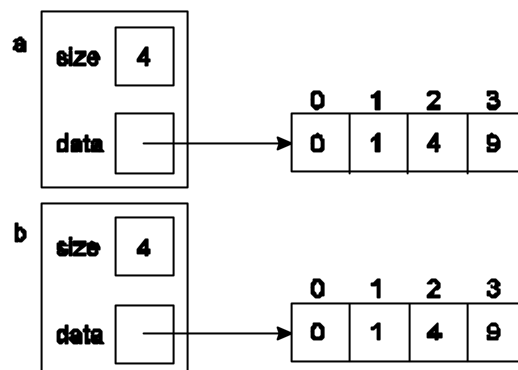
Dat je voor de class `Array` zelf een destructor moet definiëren waarin je het in de constructor met `new` gereserveerde geheugen met `delete` weer vrij moet geven zal niemand verbazen. Dat je voor de class `Array` zelf een copy constructor en `operator=` moet definiëren ligt misschien minder voor de 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 4 elementen gevuld met kwadraten is hierboven schematisch weergegeven.



De door de compiler gegenereerde copy constructor zal een memberwise copy uitvoeren. De datamembers `size` en `data` worden dus gekopieerd. Als je de `Array a` naar de `Array b` kopieert door middel van het statement `Array b(a);`[88] dan ontstaat de hiernaast weergegeven situatie.

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 hiernaast weergegeven. Om deze situatie te bereiken moeten je zelf de copy constructor van de class `Array` declareren:
`Array::Array(const Array&);`

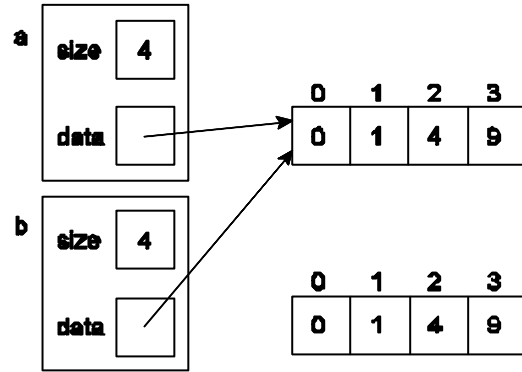


Ik zal nu eerst de problematiek van de assignment operator bespreken omdat die vergelijkbaar is met de problematiek van de copy constructor.

[88] 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 gebruik van het `=` teken bij een initialisatie is verwarrend omdat het lijkt alsof er een assigment 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);` te gebruiken. Ook bij de ingebouwde types, bijvoorbeeld `int i(0);`

5.8 Overloading `operator=`. (Zie eventueel TICPP Chapter12.html#Heading366.)

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 hiernaast weergegeven ontstaan. Je ziet dat de door de compiler gegenereerde assignment operator niet alleen onjuist werkt maar bovendien een memory leak veroorzaakt.



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];
}
```

Het argument van de copy constructor moet een `Array&` zijn en kan geen `Array` zijn. Als je namelijk een `Array` als (call by value) argument gebruikt dan 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.

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 blz. 32. 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 `<algorithm>`:

```
template <class T>
inline 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 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 "destructen" van een object van de class niet de pointer, maar de data waar de pointer naar wijst moet worden "destructed".

Dit betekent dat de class `Breuk` geen zelf gedefinieerde assignment operator, geen zelf gedefinieerde copy constructor en ook geen zelf gedefinieerde destructor nodig heeft. De class `Array` heeft wel een zelf gedefinieerde assignment operator, een zelf gedefinieerde copy constructor en ook een zelf gedefinieerde destructor nodig.

In de vorige paragraaf heb ik het zelf gedefinieerde 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 zelf gedefinieerde 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, dan 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 template class `Array` wordt bepaald. Bij het gebruik van de template class `Array` kan de compiler niet (snel) zelf bepalen wat het type `T` moet zijn. Vandaar dat je dit bij het gebruik van de template class `Array` zelf moet specificeren. Bijvoorbeeld:

```
Array<Breuk> vb(300);   // een Array met 300 breuken.
```

5.10 Voorbeeld template class `Array`.

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

```
template <typename T> class Array {
public:
     explicit Array(int s);
     Array(const Array<T>& r);
     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;
     Array<double> a(i);
     for (int j(0); j<a.length(); ++j)
          a[j]=sqrt(j);                    // Vul v met wortels
     cout<<"a[12] = "<<a[12]<<endl;
     Array<int> b(i);
     for (int t(0); t<b.length(); ++t)
          b[t]=t*t;                        // Vul w met kwadraten
     cout<<"b[12] = "<<b[12]<<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 template class kunnen definiëren:

```
template <typename T, int size> class FixedArray {        // geen goed idee!
     // ...                                                // zie hieronder
private:
     T data[size];
}
```

Deze template class kan dan als volgt gebruikt worden:

```
FixedArray<Breuk, 300> vb;        // Een Array met 300 breuken.
```

Er zijn grote verschillen tussen deze template class `FixedArray` en de eerder gedefinieerde template class `Array`:

- De `size` moet bij de laatste template tijdens het compileren bekend zijn. De compiler genereert (instantieert) namelijk de benodigde “versie” van de template class en vult daarbij de opgegeven template parameters in.
- De compiler genereert een nieuwe “versie” van de class telkens als deze class gebruikt wordt met andere template parameters. Dit betekent dat `FixedArray<int, 3> a` en `FixedArray<int, 4> b` verschillende type’s 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 gegenereerd[89]. Bij de template class `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.

We kunnen dus concluderen dat de template class `FixedArray` niet zo’n goed idee was.

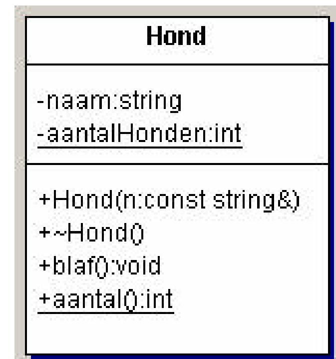
6 Losse flodders.

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 datamembers 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` datamember is een onderdeel van de class en wordt door alle objecten van de class gedeeld. Z’n `static` datamember kan bijvoorbeeld gebruikt worden om het aantal “levende” objecten van een class te tellen. In UML worden `static` members onderstreept weergegeven.




```
#include <iostream>
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;
}
```

[89] Dit is niet de waarheid. Een memberfunctie van een template class 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.

```
int main() {
     cout<<"Er zijn nu "<<Hond::aantal()<<" honden."<<endl;
     {
          Hond h1("Boris");
          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.member_functie_naam(parameters)`
  Voorbeeld: `cout<<h1.aantal()<<endl;`
- direct via de classnaam: `class_naam::member_functie_naam(parameters)`
  Voorbeeld: `cout<<Honden::aantal()<<endl;`

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" datamembers gebruiken.

6.2 Constante pointers met const. (Zie eventueel Chapter08.html#Heading248 en Chapter08.html#Heading253.)

Je hebt in paragraaf 1.3 gezien hoe je in C++ de `const` qualifier kunt gebruiken om constante waarden te definiëren.

Voorbeeld met `const`:

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

De qualifier `const` kan op verschillende manieren bij pointers gebruikt worden.

6.2.1 const *

```
int i(3);
int j(4);
const int* p(&i);
```
[90]

Dit betekent: `p` wijst naar `i` en je kan `i` niet via `p` wijzigen[91]. Let op: je kan `i` zelf wel wijzigen!

```
i=4;      // Goed
*p=5;     // Error: Cannot modify a const object
p=&j;     // Goed
```

6.2.2 * const

```
int* const q(&i);
```

Dit betekent: `q` wijst naar `i` en je kan `q` nergens anders meer naar laten wijzen[92]. Let op: je kan `i` wel via `q` (of rechtstreeks) wijzigen.

```
i=4;      // Goed
*q=5;     // Goed
q=&j;     // Error: Cannot modify a const object
```

6.2.3 const * const

```
const int* const r(&i);
```

Dit betekent: `r` wijst naar `i` en je kan `i` niet via `r` wijzigen en je kan `r` nergens anders meer naar laten wijzen. Let op: je kan `i` zelf wel wijzigen!

```
i=4;      // Goed
*r=5;     // Error: Cannot modify a const object
r=&j;     // Error: Cannot modify a const object
```

6.3 Constanten in een class.

Met behulp van het keyword `const` kun je globale constanten definiëren. Globale constanten komen de onderhoudbaarheid niet ten goede. Als een datamember `const` wordt gedefinieerd dan krijgt elk object zijn eigen constante. Als je alle objecten van een class dezelfde constante wilt laten delen dan kun je deze constante `static` definiëren. Als alternatief kun je ook een `enum` gebruiken.

---

[90] De notatie `int const* p(&i);` heeft dezelfde betekenis maar wordt in de praktijk zelden gebruikt.

[91] Dit is zinvol als je een pointer als parameter 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.

[92] Dit is zinvol als je een pointer altijd naar dezelfde variabele wilt laten wijzen.

Voorbeeld met `static` constanten:

```
class Color {
public:
     Color();
     Color(int c);
     int getValue() const;
     void setValue(int c);
// constanten:
     static const int BLACK = 0x00000000;
     static const int RED = 0x00FF0000;
     static const int YELLOW = 0x00FFFF00;
     static const int GREEN = 0x0000FF00;
     static const int LIGHTBLUE = 0x0000FFFF;
     static const int BLUE = 0x000000FF;
     static const int PURPER = 0x00FF00FF;
     static const int WHITE = 0x00FFFFFF;
// ...
private:
     int value;
};
```

Deze constanten kunnen als volgt gebruikt worden:

```
Color c(Color::YELLOW);
//...
c.setValue(Color::BLUE);
```

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;
};
```

Deze constanten kunnen als volgt gebruikt worden:

```
Color c(Color::YELLOW);
//...
c.setValue(Color::BLUE);
```

6.4 `inline` memberfuncties. (Zie eventueel TICPP Chapter09.html#Heading280.)

Veel 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 wel tot gevolg dat de code omvangrijker wordt. Memberfuncties mogen ook in de class declaratie gedefinieerd worden. Ze zijn dan "vanzelf" inline. Als de memberfunctie in de class declaratie alleen maar gedeclareerd is dan kan de definitie van die functie voorafgegaan worden door het keyword `inline` om de functie inline te maken.

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;
}
```

6.5 Namespaces.

Bij grote programma’s kunnen verschillende classes “per ongeluk” dezelfde naam krijgen. In C++ kun je classes (en functies etc.) groeperen in zogenaamde *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 op blz. 8 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 3 objecten gedefinieerd van 3 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 dan 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 using declaratie blijft net zolang geldig als een gewone variabele declaratie. Tot de bijbehorende accolade sluiten dus.

Als je in een stuk code steeds classes en functies uit de namespace Bd wilt gebruiken dan 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 variabele declaratie. Tot de bijbehorende accolade sluiten dus.

6.6 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.

```
complex<double> impedanceC(double c, double f) {
     return complex<double>(0, -1/(2*M_PI*f*c));
}
```

Deze functie kan als volgt aangeroepen worden om de impedantie van een condensator van 1 µF 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 loopt het programma volledig vast. Er verschijnt een window waarin (als op de knop details wordt gedrukt) de volgende foutmelding verschijnt:

```
IMPC heeft een uitzondering 10H veroorzaakt in module IMPC.EXE op
0177:0040116b.
Registers:
EAX=0063fdf8 CS=0177 EIP=0040116b EFLGS=00010207
...
```

Ook het berekenen van de impedantie van een condensator bij 0 Hz veroorzaakt dezelfde fout. Het programma wordt door deze fout abrupt afgebroken en krijgt niet de kans om tijdelijk opgeslagen data op te slaan. Deze fout wordt een general protection error genoemd en wordt in dit geval veroorzaakt door delen door 0. Uit de documentatie van de in het bovenstaande voorbeeld gebruikte processor (Pentium) blijkt dat “uitzondering 10H” (H staat voor Hexadecimaal) door de FDIV (Floating point Divide) machinecode instructie wordt veroorzaakt als geprobeerd wordt om door nul te delen. Het zal voor iedereen duidelijk zijn dat zulke “errors” tijdens het uitvoeren van het programma voorkomen moeten worden.

6.6.1 Het gebruik van `assert`.

Op pagina 22 heb je al kennis gemaakt met de standaard functie `assert`. Deze functie doet niets als de, als parameter opgegeven, expressie `true` oplevert maar breekt het programma met een passende foutmelding af als dit niet zo is. Je kunt zogenaamde "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:

```
complex<double> impedanceC(double c, double f) {
     assert(c!=0.0 && f!=0.0);
     return complex<double>(0, -1/(2*M_PI*f*c));
}
```

Als je nu 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) dan zal de `assert` functie het programma beëindigden. De volgende foutmelding verschijnt:

```
Assertion failed: c!=0.0 && f!=0.0, file impC.cpp, line 9
Abnormal program termination
```

Het programma stopt nog steeds abrupt. De foutmelding is nu wel veel duidelijker. Als het programma echter gecompileerd wordt zonder zogenaamde debug informatie dan worden alle `assert` functies verwijderd en verschijnt weer de onduidelijke foutmelding (uitzondering 10H).

De standaard functie `assert` is bedoeld om tijdens het ontwikkelen van programma's te controleren of iedereen zich aan bepaalde afspraken houdt. Als bijvoorbeeld is afgesproken dat de functie `impedanceC` alleen mag worden aangeroepen met parameters die niet gelijk aan 0 zijn dan is dat prima met `assert` te controleren. Elk programmadeel waarin `impedanceC` wordt aangeroepen moet nu voor de aanroep zelf controleren of de parameters 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 dan 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 nog steeds abrupt afgebroken en krijgt niet de kans om tijdelijk opgeslagen data op te slaan.
- Op elke plek waar deze functie aangeroepen wordt moeten, voor aanroep, eerst de parameters gecontroleerd worden. Als de functie op veel plaatsen aangeroepen wordt dan 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.

6.6.2 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:

```
bool impedanceC(complex<double>& res, double c, double f) {
     if (c!=0.0 && f!=0.0) {
          res=complex<double>(0, -1/(2*M_PI*f*c));
          return true;
     }
     else
          return false;
}
```

Deze functie kan nu als volgt aangeroepen worden:

```
complex<double> z;
if (impedanceC(z, 1e-6, 1e3)) {
     cout<<z<<endl;
}
else {
     cout<<"Kan impedantie niet berekenen."<<endl;
}
if (impedanceC(z, 1e-6, 0)) {
     cout<<z<<endl;
}
else {
     cout<<"Kan impedantie niet berekenen."<<endl;
}
```

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.
- De "echte" returnwaarde 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 bytes 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![93]

6.6.3 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 gooien als een parameter van een functie een ongeldige waarde heeft is de naam `domain_error`[94].

```
#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*M_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 dan 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 dan 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;
```

---

[93] Kijk nog eens terug naar oude C prgramma's die je hebt geschreven. Hoe werd daar met de returnwaarde van `printf` omgegaan? Kijk ook eens in de sourcecode van een linux systeem.

[94] Als je opgelet hebt bij de wiskunde lessen dan komt deze naam je bekend voor :-)

Exceptions kunnen worden opgevangen als ze optreden 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 dan 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 blz. 67) 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 dan 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 blz. 70) aan zodat dit programma niet meer abrupt door een divide by zero error afgebroken kan worden.

Antwoord:
De classes `C` en `P` moeten worden aangepast:

```
class C: public Component { // C=Condensator
public:
     C(double c): value(c) {
     }
     virtual complex<double> Z(double f) const {
          if (value==0.0)
               throw domain_error("Capacity == 0");
          if (f==0.0)
               throw domain_error("Frequency == 0");
          return complex<double>(0, -1/(2*M_PI*f*value));
     }
     virtual void print(ostream& o) const {
          o<<"C("<<value<<")";
     }
private:
     double value;
};

class P: public Component { // P=Parallel schakeling van 2 componenten
```

```
public:
     P(const Component& c1, const Component& c2): comp1(c1), comp2(c2) {
     }
     virtual complex<double> Z(double f) const {
          if (comp1.Z(f)+comp2.Z(f)==0)
               throw domain_error("Illegal parallel circuit");
          return (comp1.Z(f)*comp2.Z(f)) / (comp1.Z(f)+comp2.Z(f));
     }
     virtual void print(ostream& o) const {
          o<<"("<<comp1<<"//"<<comp2<<")";
     }
private:
     const Component& comp1;
     const Component& comp2;
     P(const P&);               // voorkom gebruik
     void operator=(const P&); // voorkom gebruik
};
```

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<<e.what()<<endl;
}
```

6.6.4 Het gebruik van zelf gedefinieerde exceptions.

In plaats van het gebruik van de standaard gedefinieerde exceptions kun je ook zelf exception classes definiëren.

```
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*M_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 (CapacityError& e) {
   cout<<"Capaciteit == 0"<<endl;
} catch (FrequencyError& e) {
   cout<<"Frequentie == 0"<<endl;
}
```

```
cout<<"The END."<<endl;
```

Uitvoer:

```
(0,-159.155)
Frequentie == 0
The END.
```

Zelf gedefinieerde classes kunnen we m.b.v. overerving volgens een generalisatie/specialisatie structuur indelen. 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 b.v. een `virtual` memberfunctie definiëren die een bij de exception passende foutmelding geeft. Als je deze memberfunctie in specifiekere exceptions override dan kun je een generieke exception vangen en toch d.m.v. dynamic binding de juiste foutmelding krijgen!

```
class ImpedanceError {
public:
	virtual ~ImpedanceError();
	virtual string getErrorMessage() const =0;
};

ImpedanceError::~ImpedanceError() {
}

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*M_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`.

6.6.5 De volgorde van `catch` blokken

Als je meerdere catch blokken gebruikt om exceptions af te vangen dan 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 gegooid wordt. Dus:

```
try {
//   ...
} catch (FrequencyError& e) {
    cout<<"FrequencyError exception"<<endl;
} catch (ImpedanceError& e) {
    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& e) {
    cout<<"Other exception derived from ImpedanceError"<<endl;
} catch (FrequencyError& e) {
    cout<<"FrequencyError exception"<<endl;
}
```

Antwoord:
Als er een `FrequencyError` exception wordt gegooid dan wordt deze al opgevangen in het eerste `catch` blok. Een `FrequencyError` is namelijk afgeleid van een `ImpedanceError` dus een `FrequencyError` *is een* `ImpedanceError`.

6.6.6 Exception details.

Over exceptions valt nog veel meer te vertellen:

- Exceptions in constructors en destructors. Gebruik nooit throw in een destructor!
- Function try-blok. Speciale syntax om exceptions die optreden bij het initialiseren van datamembers 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 waarmee in het prototype van een functie aangegeven kan worden welke exceptions deze functie kan veroorzaken.

- Exceptions in de std. Een overzicht van alle exceptions die in de standaard library gedefinieerd zijn en een overzicht van alle exceptions die bij het gebruik van standaard C++ kunnen optreden.

Voor al deze details verwijs ik je naar hoofdstuk 1 van "Thinking in C++, Volume 2".

6.7 Casting en runtime type information.

6.7.1 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[95] dan kun je dit als volgt proberen:

```
int i;
i="Hallo";
```

Als je dit probeert te compileren dan krijg je de volgende foutmelding:

```
[C++ Error] Cannot convert '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 zogenaamde *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. Een cast wordt door C programmeurs ook vaak gebruikt terwijl dat helemaal niet nodig is (zoals in het bovenstaande geval).

In de C++ standaard is om deze reden een nieuwe syntax voor casting gedefinieerd die eenvoudiger te vinden is:

```
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 68HC11 microcontroller. Als je de output poort B van deze controller wilt aansturen dan kan dat via adres $1004. Dit kun je in C++ als volgt doen:

```
char* ptrPortB(reinterpret_cast<char*>(0x1004));
*ptrPortB=189; // schrijf 189 (decimaal) naar adres 0x1004 (hex)
```

Als we een cast willen doen die wel portable is dan kan dat met `static_cast`.

[95] 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!

Vraag:
Wat is de uitvoer van het volgende programma:

```
#include <iostream>
int main() {
     int i1(1);
     int i2(2);
     double d(i1/i2);
     std::cout<<"d = "<<d<<std::endl;
     return 0;
}
```

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`.

Als we willen dat het bovenstaande programma `0.5` als antwoord geeft dan 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 1 van de 2 argumenten een floating point getal is. De juiste deling is dus:

```
     double d(static_cast<double>(i1)/i2);
```

Er bestaat ook een speciale cast om een `const` weg te casten de zogenaamde `const_cast`. Voorbeeld:

```
#include <iostream>
#include <string>

void stiekem(const std::string& a) {
     const_cast<std::string&>(a)="Hallo";
}

int main() {
     std::string s("Dag");
     std::cout<<"s = "<<s<<std::endl;
     stiekem(s);
     std::cout<<"s = "<<s<<std::endl;
     std::cin.get();
     return 0;
}
```

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` refereert dan is het resultaat onbepaald omdat een compiler `const` objecten in het ROM geheugen kan plaatsen (dit gebeurt veel bij embedded systems).

6.7.2 Casting en overerving.

Als we een pointer naar een `Base` class hebben dan mogen we een pointer naar een `Derived` (van `Base` afgeleide) class toekennen aan deze `Base` class pointer. Voorbeeld:

```
class Hond { /* ... */ };
class SintBernard: public Hond { /* ... */ };

Hond* hp(new SintBernard); // OK: een SintBernard is een Hond
SintBernard* sp(new Hond); // ERROR: een Hond is geen SintBernard
```

Het omzetten van een `Hond*` naar een `SintBernard*` kan soms toch nodig zijn. We noemen dit een *down-cast* omdat we afdalen in de class hiërarchie. Voorbeeld:

```
class Hond {
public:
    virtual void blaf() const {
        std::cout<<"Blaf."<<std::endl;
    }
    virtual ~Hond() {
    }
// ...
};

class SintBernard: public Hond {
public:
    SintBernard(int w=10): whisky(w) {
    }
    virtual void blaf() const {
        std::cout<<"Woef!"<<std::endl;
    }
    int geefDrank() {
        std::cout<<"Geeft drank."<<std::endl;
        int i(whisky);
        whisky=0;
        return i;
    };
// ...
private:
    int whisky;
};

void geefHulp(Hond* hp) {
    hp->blaf();
//  std::cout<<hp->geefDrank()<<" liter."<<std::endl;
//  [C++ Error] 'geefDrank' is not a member of 'Hond'
    std::cout
        <<static_cast<SintBernard*>(hp)->geefDrank()
        <<" liter."<<std::endl;
}
```

In dit geval is een `static_cast` gebruikt om een down-cast te maken. Zolang je de functie `geefHulp` alleen maar aanroept met een `SintBernard*` als argument gaat alles goed.[96]

[96] 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 down-cast is dan niet meer nodig!

```
Hond* borisPtr(new SintBernard);
geefHulp(borisPtr);
delete borisPtr;
```

Uitvoer:

```
Woef!
Geeft drank.
10 liter.
```

Als we de functie `geefHulp` echter aanroepen met een `Hond*` als argument geeft het programma onvoorspelbare resultaten (of loopt het vast):

```
Hond* borisPtr(new Hond);
geefHulp(borisPtr);
delete borisPtr;
```

Uitvoer:

```
Blaf.
Geeft drank.
6684840 liter.
```

Een `static_cast` is dus alleen maar geschikt als down-cast als je zeker weet dat de cast geldig is. In het bovenstaande programma zou je de mogelijkheid willen hebben om te kijken of een down-cast mogelijk is. Dit kan met een zogenaamde `dynamic_cast`.

```
void geefHulp(Hond* hp) {
     hp->blaf();
     SintBernard* psb(dynamic_cast<SintBernard*>(hp));
     if (psb != 0) {
          std::cout<<psb->geefDrank()<<" liter."<<std::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;
borisPtr=new Hond;
geefHulp(borisPtr);
delete borisPtr;
```

Uitvoer:

```
Woef!
Geeft drank.
10 liter.
Blaf.
```

Een `dynamic_cast` is alleen mogelijk met polymorphic pointers en polymorphic references. Als een `dynamic_cast` van een pointer mislukt dan 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 dan 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:

```
#include <typeinfo>

void geefHulp(Hond& hr) {
     hr.blaf();
     try {
          SintBernard& sbr(dynamic_cast<SintBernard&>(hr));
          std::cout<<sbr.geefDrank()<<" liter."<<std::endl;
     } catch (std::bad_cast) {
          /* doe niets */
     }
}
```

Deze functie kun je als volgt aanroepen:

```
SintBernard boris;
geefHulp(boris);
Hond h;
geefHulp(h);
```

Uitvoer:

```
Woef!
Geeft drank.
10 liter.
Blaf.
```

6.7.3 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 virtual functions RTTI. Dat is logisch omdat polymorphism 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.

```
#include <typeinfo>

void printRas(Hond& hr) {
     std::cout<<typeid(hr).name()<<std::endl;
}
```

Deze functie kun je als volgt aanroepen:

```
SintBernard boris;
printRas(boris);
Hond h;
printRas(h);
```

Uitvoer:

```
SintBernard
Hond
```

6.7.4 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:

```
class Hond { public: virtual ~Hond(); /* ... */ };
class SintBernard: public Hond { /* ... */ };
class Tekkel: public Hond { /* ... */ };

// Deze code is NIET uitbreidbaar!
// ******* DON'T DO THIS AT HOME *******
// blaf moet als virtual memberfunctie geïmplementeerd worden!

void blaf(const Hond* hp) {
     if (dynamic_cast<const SintBernard*>(hp)!=0)
          std::cout<<"Woef!"<<std::endl;
     else if (dynamic_cast<const Tekkel*>(hp)!=0)
          std::cout<<"Kef kef!"<<std::endl;
     else
          std::cout<<"Blaf."<<std::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 1 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.