HOGESCHOOL ROTTERDAM

C++

Objectgeoriënteerd Programmeren in C++

Dictaat

Versie 1.97

Work In Progress

J.Z.M. (Harry) Broeders

Objectgeoriënteerd Programmeren in C++ van J.Z.M. (Harry) Broeders is in licentie gegeven volgens een Creative Commons Naamsvermelding-NietCommercieel-GelijkDelen 3.0 Nederland-licentie.

Voorwoord

Dit dictaat is lang geleden begonnen als een korte introductie in C++ en inmiddels uitgegroeid tot dit met LaTeX opgemaakte document[1]. In de tussentijd hebben veel studenten en collega's mij feedback gegeven. De opbouwende kritiek van Harm Jongsma, Cor Diependaal, Henk van den Bosch, Dave Stikkelorum, John Visser, Gerard Tuk, Fidelis Theinert, Ben Kuiper, René Theil, René Boeije en vele anderen hebben dit dictaat ontegenzeggelijk beter gemaakt. Desondanks heb ik niet de illusie dat dit dictaat foutvrij is. Op- en aanmerkingen zijn dus nog altijd welkom, mail me op J.Z.M.Broeders@hr.nl, of maak een issue aan.

In dit dictaat is zoveel mogelijk gebruik gemaakt van de C++23 standaard. Zo wordt bijvoorbeeld gebruik gemaakt van `print` in plaats van `cout`, van modules in plaats van include files, van concepts en van ranges. Veel code die je op het internet vindt, is geschreven in oudere versies van C++. Taalconstructies die veelvuldig in oudere versies van C++ gebruikt werden, worden in dit dictaat kort besproken en er wordt uitgelegd waarom ze vervangen zijn door modernere alternatieven.

Alle programmacode in dit dictaat is getest met GCC[2] versie 15.2.0. Op https://github.com/HR-ELEKTRO/cppprog//wiki vind je de source code van alle in dit dictaat besproken programma's. Als je dit dictaat leest op een device met internettoegang, dan kun je op de bestandsnamen in dit dictaat klikken om de programma's te downloaden.

Veel literatuur over C++ en objectgeoriënteerd programmeren is geschreven in het Engels. Om er voor te zorgen dat je deze literatuur na (of tijdens) het lezen van dit dictaat eenvoudig kunt gebruiken heb ik veel van het jargon dat in deze literatuur wordt gebruikt niet vertaald naar het Nederlands. De keuze om een term wel of niet te vertalen is arbitraal. Zo heb ik

[1] De LaTeX code is beschikbaar op https://github.com/HR-ELEKTRO/cppprog/.

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

bijvoorbeeld het Engelse begrip ‘return type’ niet vertaal in het Nederlandse ‘terugkeertype’ of ‘retourtype’, maar heb ik het Engelse begrip ‘parameter type’ wel vertaald naar het Nederlandse ‘parametertype’. De Engelse begrippen ‘class’ en ‘class template’ zijn niet vertaald in ‘klasse’ en ‘klassetemplate’ maar de Engelse begrippen ‘function’ en ‘function template’ zijn wel vertaald in ‘functie’ en ‘functietemplate’.

Inhoudsopgave

Inleiding 13

1 Van C naar C++ 19
- 1.1 Modules in plaats van include files . . . 19
- 1.2 Output met `print` en `println` . . . 22
- 1.3 Input en output met >> en << . . . 25
- 1.4 Universele initialisatie . . . 27
- 1.5 `auto` type . . . 28
- 1.6 `auto` return type . . . 29
- 1.7 Range-based `for` . . . 30
- 1.8 Constante waarden met `constexpr` . . . 31
- 1.9 Read-only variabelen met `const` . . . 32
- 1.10 Het type `string` . . . 33
- 1.11 Het type `array` en `vector` . . . 34
- 1.12 Function name overloading . . . 34
- 1.13 Default argumenten . . . 35
- 1.14 Naam van een `struct` . . . 36
- 1.15 C++ als een betere C . . . 37

2 Objects and classes 43
- 2.1 Object Oriented Design (OOD) and Object Oriented Programming (OOP) . . . 43
- 2.2 UDT's (User-defined Data Types) . . . 49
- 2.3 Voorbeeld class `Breuk` (tweede versie) . . . 55
- 2.4 Constructor `Breuk` . . . 58
- 2.5 Initialization list van de constructor . . . 59
- 2.6 Constructors en type conversies . . . 60

2.7 Default copy constructor . . . 61
2.8 Default assignment operator . . . 62
2.9 const memberfuncties . . . 63
2.10 Class invariant . . . 65
2.11 Voorbeeld class Breuk (derde versie) . . . 65
2.12 Operator overloading . . . 67
2.13 this-pointer . . . 69
2.14 Reference variabelen . . . 70
2.15 Reference parameters . . . 70
2.16 const reference parameters . . . 72
2.17 Parameter FAQ . . . 73
2.18 Reference in range-based for . . . 74
2.19 Reference return type . . . 74
2.20 Reference return type (deel 2) . . . 76
2.21 Voorbeeld module breuk . . . 77

3 Templates 80
3.1 Functietemplates . . . 80
3.2 Class templates . . . 84
3.3 Voorbeeld class template Dozijn . . . 86
3.4 Voorbeeld class template Rij . . . 88
3.5 Standaard templates . . . 90
3.6 std::array . . . 91
3.7 std::vector . . . 92
3.8 Template Argument Deduction . . . 96
3.9 Alternatieve notatie met auto . . . 97
3.10 Concepts . . . 99

4 Inheritance 104
4.1 De syntax van inheritance . . . 105
4.2 Polymorfisme . . . 108
4.3 Memberfunctie overriding . . . 109
4.4 Abstract base class . . . 112
4.5 Constructors bij inheritance . . . 112
4.6 protected members . . . 113
4.7 Voorbeeld: ADC kaarten . . . 114
4.7.1 Probleemdefinitie . . . 114

4.7.2 Een gestructureerde oplossing . . . 114
4.7.3 Een oplossing door middel van een UDT . . . 118
4.7.4 Een objectgeoriënteerde oplossing . . . 121
4.7.5 Een kaart toevoegen . . . 125
4.8 Voorbeeld: impedantie calculator . . . 126
4.8.1 Weerstand, spoel en condensator . . . 127
4.8.2 Serie- en parallelschakeling . . . 131

5 Dynamic memory allocation en destructors 134
5.1 Dynamische geheugenallocatie met `new` en `delete` . . . 134
5.2 Smart pointers . . . 136
5.3 Destructor `~Breuk` . . . 138
5.4 Destructors bij inheritance . . . 140
5.5 Virtual destructor . . . 140
5.6 Toepassing van `std::unique_ptr` . . . 143

6 Exceptions 146
6.1 Het gebruik van `assert` . . . 147
6.2 Het gebruik van een `bool` returnwaarde . . . 148
6.3 Het gebruik van standaard exceptions . . . 149
6.4 Het gebruik van zelfgedefinieerde exceptions . . . 152
6.5 De volgorde van `catch` blokken . . . 155
6.6 Exception details . . . 156

7 Inleiding algoritmen en datastructuren 157
7.1 Analyse van algoritmen . . . 160
7.2 Datastructuren . . . 162

8 Gebruik van een stack 165
8.1 Haakjes balanceren . . . 166
8.2 Eenvoudige rekenmachine . . . 167
8.2.1 Postfix-notatie . . . 168
8.2.2 Een postfix calculator . . . 169
8.2.3 Een infix-calculator . . . 171

9 Implementaties van een stack 176
9.1 Stack met behulp van een array . . . 176
9.2 Stack met behulp van een gelinkte lijst . . . 178

9.3 Stack met array versus stack met gelinkte lijst . . . 181
9.4 Dynamisch kiezen voor een bepaald type stack . . . 181

10 De standaard C++ library 184
10.1 Voorbeeldprogramma met `std::string` . . . 185
10.2 Overzicht van de standaard library . . . 186

11 Containers 187
11.1 Sequentiële containers . . . 188
11.1.1 Voorbeeldprogramma met `std::vector` . . . 191
11.2 Generieke verdubbel functie voor containers . . . 194
11.3 Container adapters . . . 196
11.3.1 Voorbeeldprogramma met `std::stack` . . . 197
11.4 Associatieve containers . . . 198
11.4.1 Voorbeeldprogramma met `std::set` . . . 200
11.4.2 Voorbeeldprogramma met `std::multiset` (bag) . . . 202
11.4.3 Voorbeeldprogramma met `std::unordered_set` . . . 203
11.4.4 Voorbeeldprogramma met `std::map` . . . 204

12 Iteratoren 206
12.1 Voorbeeldprogramma's met verschillende iterator soorten . . . 210
12.2 Reverse-iteratoren . . . 214
12.3 Insert-iteratoren . . . 215
12.4 Stream-iteratoren . . . 216
12.5 Voorbeeldprogramma's met iteratoren . . . 217
12.6 Ranges . . . 218

13 Algoritmen 219
13.1 Zoeken, tellen, testen, bewerken en vergelijken . . . 220
13.1.1 Voorbeeldprogramma met `std::find` . . . 222
13.1.2 Voorbeeldprogramma met `std::find_if` . . . 224
13.1.3 Voorbeeldprogramma met `std::find_first_of` . . . 226
13.1.4 Voorbeeldprogramma met `std::for_each` . . . 227
13.2 Lambda-functies . . . 228
13.2.1 Closure . . . 229
13.2.2 Returntype van een lambda-functie . . . 230
13.2.3 Lambda-functieparametertype met `auto` . . . 231

13.3 Kopiëren, bewerken, vervangen, roteren, schudden, verwisselen, verwijderen en vullen . . . 231
13.3.1 Voorbeeldprogramma met `std::ranges::transform` . . . 233
13.3.2 Voorbeeldprogramma met `std::remove` . . . 235
13.3.3 Voorbeeldprogramma met `std::generate_n` . . . 236
13.4 Sorteren en bewerkingen op gesorteerde ranges . . . 236
13.4.1 Voorbeeldprogramma met `std::sort` . . . 237
13.4.2 Voorbeeldprogramma met `std::includes` . . . 239

14 Toepassingen van datastructuren 242
14.1 Boter, Kaas en Eieren . . . 242
14.1.1 Minimax algoritme . . . 242
14.1.2 Alpha-beta pruning . . . 256
14.1.3 Verschillende implementaties van Boter, Kaas en Eieren . . . 265
14.2 Het zoeken van het kortste pad in een graph . . . 266

15 Van C naar C++ (de details) 270
15.1 Binaire getallen . . . 270
15.2 Het type `bool` . . . 270
15.3 Het type `byte` . . . 271
15.4 Meer floating point types . . . 271
15.5 Digit separators . . . 272
15.6 Zero-initialisatie . . . 272
15.7 Aangewezen initialisatie . . . 272
15.8 Zelf gedefinieerde module . . . 273
15.9 Namespaces . . . 276
15.10 `using` in plaats van `typedef` . . . 278
15.11 `decltype` type . . . 278
15.12 Compile time functies . . . 280
15.13 Compile time `if` . . . 282
15.14 Read-only pointers met `const` . . . 282
15.14.1 `const *` . . . 282
15.14.2 `* const` . . . 283
15.14.3 `const * const` . . . 284
15.15 Casting . . . 284
15.16 `static_assert` . . . 287
15.17 Structured binding . . . 287

15.18 std::string_view . . . 288
15.19 Attributen . . . 292

16 Objects and classes (de details) 294
16.1 Operator overloading (deel 2) . . . 294
16.2 operator+ FAQ . . . 295
16.3 Operator overloading (deel 3) . . . 297
16.4 Overloaden operator++ en operator- . . . 299
16.5 Conversie operatoren . . . 300
16.6 Voorbeeld class Breuk (vierde versie) . . . 301
16.7 friend functions . . . 305
16.8 Default operator== . . . 306
16.9 Drieweg-vergelijkingsoperator <=> . . . 307
16.10 Default operator<=> . . . 310
16.11 Operator overloading (deel 4) . . . 311
16.12 static class members . . . 313
16.13 Initialiseren van datavelden . . . 315
16.14 Compile time constanten in een class . . . 317
16.15 Inline memberfuncties . . . 319
16.16 User-defined literals . . . 320
16.17 enum class . . . 321

17 Templates (de details) 323
17.1 Variable-template . . . 323
17.2 Template argument binding . . . 324
17.3 Template parameters automatisch bepalen . . . 324
17.4 Zelf concepts definiëren . . . 325
17.5 Class template specialisatie . . . 328
17.6 Zelf een formatter definiëren . . . 333
17.6.1 formatter<Dobbelsteen> . . . 333
17.6.2 formatter<Breuk> . . . 339
17.6.3 formatter<std::complex> . . . 344
17.7 Nog meer template details . . . 347

18 Inheritance (de details) 348
18.1 Overloading en overriding van memberfuncties . . . 348
18.2 Dynamic binding werkt niet in constructors en destructors . . . 353

18.3 Expliciet overridden van memberfuncties . . . 357
18.4 `final` overriding van memberfuncties . . . 359
18.5 `final` overerving . . . 361
18.6 Slicing problem . . . 361
18.7 Voorbeeld: opslaan van polymorfe objecten in een `vector` . . . 364
18.8 Casting en overerving . . . 365
18.9 Dynamic casting en RTTI . . . 369
18.10 Maak geen misbruik van RTTI en `dynamic_cast` . . . 370
18.11 Nog meer inheritance details . . . 371

19 Dynamic memory allocation en destructors (de details) 372
19.1 Voorbeeld class `Array` . . . 372
19.2 `explicit` constructor . . . 377
19.3 Copy constructor en default copy constructor . . . 377
19.4 Overloading `operator=` . . . 379
19.5 Wanneer moet je zelf een destructor, enz definiëren? . . . 381
19.6 Voorbeeld class template `Array` . . . 382
19.7 Ondersteuning range-based `for` voor class template `Array` . . . 385
19.8 Ondersteuning initialisatielijst voor class template `Array` . . . 386
19.9 Return value optimization . . . 387
19.10 Rvalue reference . . . 390
19.11 Move constructor en move assignment operator . . . 392

20 De standaard C++ library (de details) 400
20.1 Het meten van de executietijd van een stukje code . . . 400
20.2 Random getallen . . . 401
20.3 `vector::reserve` . . . 401
20.4 `emplace` . . . 401
20.5 `optional<T>` . . . 401
20.6 Van numeriek naar string en van string naar numeriek . . . 403
20.7 Unordered container met UDT's . . . 404
20.8 Voorbeeldprogramma dat generiek en objectgeoriënteerd programmeren combineert . . . 406
20.9 Standaard functie-objecten . . . 408
20.10 Views . . . 410
20.11 Dynamische format specifiers . . . 410
20.12 Lokalisatie . . . 413

Bibliografie 415

Inleiding

Halverwege de jaren ’70 werd steeds duidelijker dat de veelgebruikte softwareontwikkelmethode structured design (ook wel functionele decompositie genoemd) niet geschikt is om grote uitbreidbare en onderhoudbare softwaresystemen te ontwikkelen. Ook bleken de onderdelen van een applicatie die met deze methode waren ontwikkeld, meestal niet herbruikbaar in andere applicaties. Het heeft tot het begin van de jaren ’90 geduurd voordat een alternatief voor structured design het zogenoemde, object oriented design (OOD), echt doorbrak. Objectgeoriënteerde programmeertalen bestaan al sinds het begin van de jaren ’70. Deze manier van ontwerpen (OOD) en programmeren (OOP) is echter pas in het begin van de jaren ’90 populair geworden nadat in het midden van de jaren ’80 de programmeertaal C++ door Bjarne Stroustrup was ontwikkeld. Deze taal voegt taalconstructies toe aan de op dat moment in de praktijk meest gebruikte programmeertaal C. Deze objectgeoriënteerde versie van C heeft de naam C++ gekregen en heeft zich ontwikkeld tot één van de meest gebruikte programmeertalen van dit moment[3]. C++ is echter geen pure OO-taal (zoals bijvoorbeeld Smalltalk) en kan ook gebruikt worden als procedurele programmeertaal. Dit heeft als voordeel dat de overstap van C naar C++ eenvoudig te maken is maar heeft als nadeel dat C++ gebruikt kan worden als een soort geavanceerd C zonder gebruik te maken van OOP. De taal C++ is nog steeds in ontwikkeling. In dit dictaat gebruiken we de versie van de C++-standaad die C++23 wordt genoemd [10][4]. Daarbij volgen we zoveel mogelijk de *C++ Core Guidelines*[5].

[3] Bron: https://www.tiobe.com/tiobe-index/.

[4] Een zogenoemde draft version van de C++23-standaard is beschikbaar op https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2023/n4950.pdf. Ook van de C++26-standaard is al een draft version beschikbaar op https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2025/n5032.pdf.

[5] Zie https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines.

C++ is een zeer uitgebreide taal en dit dictaat moet dan ook zeker niet gezien worden als een cursus C++. In dit dictaat worden slechts de meest belangrijke delen van C++ behandeld. Als je meer achtergrondinformatie of diepgang zoekt, kun je gebruik maken van een van de boeken: *A Tour of C++* [20] of *Principles and Practice Using C++* [21] van Stroustrup of *Professional C++* [4] van Gregoire. Gedetailleerde gratis informatie kun je ook vinden in *C++ Annotations* [1] van Brokken en op http://en.cppreference.com/w/cpp.

Een terugblik op C

We starten dit dictaat met de overgang van gestructureerd programmeren in C naar gestructureerd programmeren in C++. *Ik ga ervan uit dat je de taal C*[6] *redelijk goed beheerst.* Misschien is het nodig om deze kennis op te frissen[7]. Vandaar dat dit dictaat begint met een terugblik op C. We doen dit aan de hand van het voorbeeldprogramma C.c dat een lijst met tijdsduren (in uren en minuten) inleest vanaf het toetsenbord en de totale tijdsduur (in uren en minuten) bepaalt en afdrukt. Dit programma kun je bijvoorbeeld gebruiken om, van een aantal gewerkte tijden, de totale gewerkte tijd te bepalen.

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

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

// Deze functie drukt een Tijdsduur af
void drukaf(Tijdsduur td) {
    if (td.uur == 0)
        printf("%13d minuten\n", td.minuten);
    else
        printf("%3d uur en %2d minuten\n", td.uur, td.minuten);
}

// Deze functie drukt een rij met een aantal Tijdsduren af
```

[6] We gaan in dit dictaat uit van de C23 standaard [11].

[6] De draft version is beschikbaar op https://www.open-std.org/jtc1/sc22/wg14/www/docs/n3096.pdf.

[7] Zie eventueel https://github.com/HR-ELEKTRO/cppprog//wiki.

```
void drukaf_rij(const Tijdsduur rij[], size_t aantal) {
    for (size_t teller = 0; teller < aantal; teller++)
        drukaf(rij[teller]);
}

// Deze functie berekent de totale Tijdsduur van een rij met een ↩
 ↪ aantal Tijdsduren
Tijdsduur som(const Tijdsduur rij[], size_t aantal) {
    Tijdsduur s = {0, 0};
    for (size_t teller = 0; teller < aantal; teller++) {
        s.uur += rij[teller].uur;
        s.minuten += rij[teller].minuten;
    }
    s.uur += s.minuten / 60;
    s.minuten %= 60;
    return s;
}

#define MAX_TIJDSDUREN 5

int main(void) {
    Tijdsduur tijdsduren[MAX_TIJDSDUREN];
    size_t aantal = 0;
    int aantal_gelezen_variabelen;
    do {
        printf("Type uren en minuten in: ");
        aantal_gelezen_variabelen = scanf("%d%d", ↩
 ↪ &tijdsduren[aantal].uur, &tijdsduren[aantal].minuten);
    }
    while (aantal_gelezen_variabelen == 2 && ++aantal < ↩
 ↪ MAX_TIJDSDUREN);
    printf("\n");
    drukaf_rij(tijdsduren, aantal);
    printf("De totale tijdsduur is:\n");
    drukaf(som(tijdsduren, aantal));
    return 0;
}
```

Verklaring:

- In de eerste regel wordt de file `stdio.h` geïnclude. Dit is nodig om gebruik te kunnen maken van functies en types die in de standaard C I/O library zijn opgenomen. In dit programma maak ik gebruik van `printf` (om te schrijven naar het scherm) en van `scanf` (om getallen te lezen vanaf het toetsenbord).
- Vervolgens is het samengestelde type `Tijdsduur` gedeclareerd. Variabelen van het **`struct`**-type `Tijdsduur` bevatten twee datavelden (Engels: data members) van het type **`int`**. Deze datavelden heten `uur` en `minuten` en zijn bedoeld voor de opslag van de uren en de minuten van de betreffende tijdsduur.
- Vervolgens worden er drie functies gedefinieerd:
  - `drukaf`.
    Deze functie drukt de parameter `td` van het type `Tijdsduur` af op het scherm door gebruik te maken van de standaard schrijffunctie `printf`. Het return type van de functie `drukaf` is **`void`**. Dit betekent dat de functie geen waarde teruggeeft.
  - `drukaf_rij`.
    Deze functie drukt een rij met tijdsduren af. Deze functie heeft twee parameters. De eerste parameter genaamd `rij` is een array met elementen van het type `Tijdsduur`. Een array wordt in C via call by reference doorgegeven. Dat wil zeggen dat in feite het beginadres van de array wordt doorgegeven aan de functie[8]. De functie kan dus de inhoud van de array wijzigen. Om aan te geven dat deze functie dit niet doet is de parameter als read only gedefinieerd met behulp van het keyword **`const`**[9]. De tweede parameter (een variabele van het type `size_t`[10] genaamd `aantal`) geeft aan hoeveel elementen uit de array afgedrukt moeten worden. Tijdens het uitvoeren van de **`for`**-lus krijgt de lokale variabele `teller`, van het type `size_t`, achtereenvolgens de waarden `0` t/m `aantal-1`. Deze teller wordt gebruikt om de elementen uit `rij` één voor één te selecteren. Elk element (een variabele van het type `Tijdsduur`) wordt met de functie `drukaf` afgedrukt.

[8] Zie eventueel [5, paragraaf 6.1.4].

[9] Zie eventueel [5, takeaway 1.5.6.3].

[10] In C gebruik je het type `size_t` om het aantal elementen in een array, of de index van een arrayelement, mee aan te geven. Het is een **`unsigned`** type en een variabele van dit type kan dus alleen positieve getallen bevatten. In de praktijk wordt ook vaak het type **`int`** gebruikt om het aantal elementen in een array mee aan te geven. Maar (in theorie) is het mogelijk dat een array meer dan `MAX_INT` elementen bevat, dus het is beter om `size_t` te gebruiken.

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

- Vervolgens wordt met de preprocessor directive **#define** de constante MAX_TIJDSDUREN gedefinieerd met als waarde 5.
- Tot slot wordt de hoofdfunctie main gedefinieerd. Deze functie wordt bij het starten van het programma aangeroepen. In de functie main wordt een array tijdsduur aangemaakt met MAX_TIJDSDUREN elementen van het type Tijdsduur. Tevens wordt de integer variabele aantal gebruikt om het aantal ingelezen tijdsduren te tellen. Elke tijdsduur bestaat uit twee integers: een aantal uren en een aantal minuten. In de **do-while**-lus worden tijdsduren vanaf het toetsenbord ingelezen in de array tijdsduren.

  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 aantal_gelezen_variabelen.

  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_TIJDS-

DUREN. De ++ vóór aantal betekent dat de waarde van aantal eerst met één wordt verhoogd en daarna wordt vergeleken met MAX_TIJDSDUREN. Over deze conditie is echt goed nagedacht. De && wordt namelijk altijd van links naar rechts uitgevoerd. Als de expressie aan de linkerkant van de && niet waar is, dan wordt de expressie aan de rechterkant niet uitgevoerd en wordt de variabele aantal dus niet verhoogd. Als het lezen van twee integers met scanf niet gelukt is, dan is aantal_gelezen_variabelen ongelijk aan 2 en wordt de variabele aantal niet opgehoogd. De **do-while**-lus wordt dus beëindigd als het inlezen van twee integers niet gelukt is, bijvoorbeeld omdat een letter in plaats van een cijfer is ingetypt, óf als de array vol is.

Na afloop van de **do-while**-lus wordt er voor gezorgd dat er een regel wordt overgeslagen door de string "\n" naar het beeldscherm te schrijven met de functie printf. Vervolgens wordt de lijst afgedrukt met de functie drukaf_rij en wordt de totale tijdsduur (die berekend wordt met de functie som) afgedrukt met de functie drukaf. Tot slot geeft de functie main de waarde 0 aan het operating system terug. De waarde 0 betekent dat het programma op succesvolle wijze is geëindigd. Het statement **return** 0 mag sinds C99 ook weggelaten worden en wordt in dat geval automatisch door de compiler toegevoegd.

Merk op dat dit programma niet meer dan MAX_TIJDSDUREN tijdsduren kan verwerken. Dit probleem kun je oplossen door dynamisch geheugen te gebruiken, maar dat is in C nog niet zo gemakkelijk. Later in dit dictaat (paragraaf 3.7) leer je dat dit in C++ veel gemakkelijker is.

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

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

1

Van C naar C++

De ontwerper van C++ Bjarne Stroustrup gaf op de vraag: “Wat is C++?” een aantal jaren geleden het volgende antwoord [22]:

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

In dit hoofdstuk bespreken we eerst enkele kleine verbeteringen van C++ ten opzichte van zijn voorganger C. Grotere verbeteringen zoals data abstractie, objectgeoriënteerd programmeren en generiek programmeren komen in de volgende hoofdstukken uitgebreid aan de orde.

De kleine verbeteringen van C++ ten opzichte van C, die je moet weten om de rest van dit dictaat te kunnen lezen, vind je in dit hoofdstuk. Verdere kleine verbeteringen, die ook best handig zijn bij het programmeren in C++, vind je in hoofdstuk 15.

1.1 Modules in plaats van include files

In C++ worden bij voorkeur modules gebruikt in plaats van include files. De standaard C++ library is zeer uitgebreid en in hoofdstuk 10 komen we hier uitgebreid op terug.

Je kunt de gehele standaard C++ library als module importeren met de volgende regel:

```
import std;
```

Als je gebruik maakt van de GCC compiler, dan moeten modules geprecompiled worden met het commando:

```
g++ -std=c++23 -fmodules -fmodule-only -Wall -Wextra ↩
↪ -Wpedantic -g3 -O0 -c -fsearch-include-path bits/std.cc
```

Hierna kun je de module importeren in je programma met `import std;`. Bijvoorbeeld in een "Hello World" programma (hello_world.cpp):

```
import std;

int main() {
    std::println("Hello, World!");
}
```

Dit programma kan dan gecompileerd worden met het commando:

```
g++ -std=c++23 -fmodules -Wall -Wextra -Wpedantic -g3 -O0 ↩
↪ hello_world.cpp -o hello_world
```

En vervolgens uitgevoerd worden met het commando[11]

```
./hello_world
```

De voordelen van modules ten opzichte van include files zijn onder andere:

- Header files worden door de preprocessor in de broncode geplakt, modules worden door de compiler gecompileerd en gelinkt. Een module wordt dus maar één keer gecompileerd, terwijl een header file meerdere keren gecompileerd wordt als deze in meerdere bronbestanden wordt geinclude.
- De volgorde van includen van header files is van belang omdat de ene include de betekenis van de andere kan beïnvloeden. Modules kunnen in willekeurige volgorde geimporteerd worden, omdat ze elkaar niet beïnvloeden.
- Als je een header file include, dan worden alle includes in die header file ook geinclude. Bij modules is dat niet het geval, omdat modules alleen de symbolen exporteren die expliciet worden geëxporteerd.

[11] Onder windows kan het programma uitgevoerd worden met het commando `Hello_world.exe`.

Modules zijn dus beter voor de onderhoudbaarheid van de code, omdat ze minder afhankelijkheden hebben ze verminderen bovendien de compilatie-tijd. Je kunt ook zelf modules definiëren, zie paragraaf 15.8.

Om het mogelijk te maken dat de standaard library met andere (oude en nieuwe) libraries in één programma kan worden gebruikt zijn in C++ namespaces opgenomen[12]. Alle symbolen uit de C++ standaard library bevinden zich in een namespace genaamd `std`. Als je een symbool uit de namespace `std` wilt gebruiken, moet je dit symbool laten voorafgaan door `std::`. Vandaar dat in het bovenstaande programma de `println` functie uit de standaard library is aangeroepen met behulp van `std::println`.

In plaats van elk symbool uit de C++ standaard library vooraf te laten gaan door `std::` kun je ook na de `import std;` de volgende regel in het programma opnemen[13]:

```
using namespace std;
```

Hierna kun je de `println` functie uit de standaard library aanroepen zonder deze vooraf te laten gaan door `std::`, dus gewoon met `println`. Zie `hello_world_using.cpp`.

Pas sinds C++23 kun je de gehele standaard library als module importeren. Ook de `println` functie is pas sinds C++23 beschikbaar in de standaard library, zie paragraaf 1.2. Als je gebruikt maakt van oudere versies van C++, dan moet je gebruik maken van include files (zoals in C) in plaats van modules. In plaats van `println` kun je dan gebruik maken van `cout` en de `<<` operator, zie paragraaf 1.3.

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

Een "Hello World" programma dat gebruik maakt van include files in plaats van modules en van `cout` in plaats van `println` ziet er als volgt uit (`hello_world_include.cpp`):

```
#include <iostream>

int main() {
    std::cout << "Hello, World!\n";
```

[12] Namespaces worden verder besproken in paragraaf 15.9.

[13] Omdat in dit dictaat uitsluitend gebruik maken van de standaard library en niet van andere libraries wordt deze regel in elk programma, dat in dit dictaat worden besproken, opgenomen. Dit wordt ook aangeraden in de C++ Core Guideline SF.6.

```
}
```

1.2 Output met `print` en `println`

Je bent gewend om in C programma's de functies uit de `stdio` bibliotheek te gebruiken voor input en output. De meest gebruikte functies zijn `printf` en `scanf`. Deze functies zijn echter niet type veilig (*type-safe*) omdat de inhoud van het, als eerste argument meegegeven, format (bijvoorbeeld `"%d"`) pas tijdens het uitvoeren van het programma verwerkt wordt. De compiler merkt het dus niet als de type aanduidingen (zoals bijvoorbeeld %d) die in het format gebruikt zijn niet overeenkomen met de types van de volgende argumenten. Tevens is het niet mogelijk om een eigen format type aanduiding aan de bestaande toe te voegen. Om deze redenen zijn in de C++23 standaard [10] naast de oude `stdio` bibliotheek (om compatibel te blijven) ook de functies `print` en `println` opgenomen. `println` voegt automatisch een newline-karakter toe aan het einde van de output.

Voorbeeld van een C-programma met `stdio` (`printf.c`):

```
#include <stdio.h>
//  ...
    double d = 3.14;
    printf("d = %d\n", d); // deze regel bevat een fout!
```

Dit programmadeel bevat een fout die niet door de compiler gesignaleerd wordt[14] en pas bij executie blijkt. De uitvoer van dit programma is namelijk:

```
d = 1374389535
```

De gebruikte conversion specifier %d is namlijk bedoeld voor het afdrukken van een integer, maar de variabele `d` is van het type **`double`**. In plaats van %d had `%lf` gebruikt moeten worden.

Hetzelfde voorbeeld met `println` (`println.cpp`):

```
import std;
using namespace std;
//  ...
    double d = 3.14;
    println("d = {}", d);
```

[14] Als het programma gecompileerd wordt met het commando `gcc printf.c`, dan wordt inderdaad geen melding gegeven. Als je echter het commando `gcc -Wall printf.c` gebruikt, dan geeft de compiler de waarschuwing: `warning: format '%d' expects argument of type 'int', but argument 2 has type 'double'`.

De uitvoer van dit programma is:

```
d = 3.14
```

De accolades `{}` in de `println` functie geven aan waar de waarde van het volgende argument `d` moet worden ingevuld. De compiler bepaald zelf het type van `d` en zorgt er automatisch voor dat de juiste conversie (in dit geval van **`double`** naar tekst) plaatsvindt.

De functies `print` en `println` bieden zeer uitgebreide formatting mogelijkheden. Als je bijvoorbeeld in het bovenstaande programma de variabele `d` met drie decimalen wilt weergegeven, dan kun je dat doen met behulp van een zogenoemde *format specifier* `{:.3f}`:

```
println("d = {:.3f}", d);
```

De uitvoer is dan:

```
d = 3.140
```

Als er meerdere argumenten zijn, dan worden deze in de volgorde van de accolades ingevuld. Bijvoorbeeld:

```
int i {42};
int j {666};
println("i = {}, j = {}", i, j);
```

De uitvoer van dit programma is:

```
i = 42, j = 666
```

Je kunt de volgorde van invullen ook aanpassen door een getal toe te voegen aan de accolades:

```
println("j = {1}, i = {0}", i, j);
```

De uitvoer van deze regel is:

```
j = 666, i = 42
```

Je kunt `i` en `j` ook onder elkaar netjes uitgelijnd weergeven:

```
println("i = {:>4}\nj = {:>4}", i, j);
```

De uitvoer van deze regel is:

```
i =   42
j =  666
```

Je kunt ook nullen toevoegen aan de linkerkant van de getallen:

```
println("i = {:0>4}\nj = {:0>4}", i, j);
```

De uitvoer van deze regel is:

```
i = 0042
j = 0666
```

Je kunt een argument ook meerdere keren gebruiken:

```
println("decimaal {0} is gelijk aan hexadecimaal {0:X}", i);
```

De uitvoer van deze regel is:

```
decimaal 42 is gelijk aan hexadecimaal 2A
```

Er zijn nog veel meer formatting mogelijkheden, zie https://en.cppreference.com/w/cpp/utility/format/spec.html.

Sinds C++23 moeten compilers UTF-8[15] als standaard encoding van source files ondersteunen. Je kunt dus een hello world programma schrijven met emoji's[16] (hello_world_emoji.cpp):

```
import std;
using namespace std;

int main() {
    println("👋🌍!");
}
```

Je kunt ook printen naar een bestand in plaats van naar het beeldscherm (hello_world_file.cpp):

```
import std;
using namespace std;

int main() {
    ofstream f_out {"output.txt"};
    println(f_out, "Hello, World!");
}
```

[15] Met UTF-8 encoding kunnen alle mogelijke tekens uit de Unicode-standaard worden weergegeven, zie https://nl.wikipedia.org/wiki/UTF-8.

[16] Zie https://unicode.org/emoji/charts/emoji-list.html voor een lijst van alle beschikbare emoji's.

Dit programma maakt een bestand genaamd `output.txt` aan en schrijft de string `Hello, World!` in dit bestand. Als het bestand al bestaat, dan wordt de inhoud van dit bestand overschreven.

1.3 Input en output met >> en <<

Vóór C++23 werd de iostream library gebruikt voor het inlezen en wegschrijven van data. Sinds C++23 heeft het de voorkeur om voor output de functies `print` en `println` te gebruiken, zie paragraaf 1.2. Voor input is er nog geen alternatief voor het gebruik van de iostream library.

Je bent gewend om in C programma's de functies uit de `stdio` bibliotheek te gebruiken voor input. De meest gebruikte functie voor input is `scanf`. Deze functie is, zoals al eerder vermeld, niet type veilig.

De belangrijkste output faciliteiten van de iostream library zijn de standaard output stream `cout` (vergelijkbaar met `stdout`) en de bijbehorende << operator. De belangrijkste input faciliteiten van deze library zijn de standaard input stream `cin` (vergelijkbaar met `stdin`) en de bijbehorende >> operator.

Voorbeeld van een C-programma met `stdio` (`scanf.c`):

```
#include <stdio.h>
//  ...
    double d;
    scanf("%d", d); // deze regel bevat twee fouten!
    printf("d = %.3lf\n", d);
```

Dit programmadeel bevat twee fouten[17] die niet door de compiler gesignaleerd worden[18] en pas bij executie blijken.

[17] De eerste fout is dat de conversion specifier `%d` bedoeld is voor het lezen van een **`int`**, maar de variabele `d` is van het type **`double`**. In plaats van `%d` had `%lf` gebruikt moeten worden. De tweede fout is dat de variabele `d` als argument aan de functie `scanf` wordt, maar omdat de variabele moet worden ingelezen moet het adres van deze variabele `&d` meegegeven moet worden.

[18] Als het programma gecompileerd wordt met het commando `gcc scanf.c`, dan wordt inderdaad geen melding gegeven. Als je echter het commando `gcc -Wall scanf.c` gebruikt, dan geeft de compiler de waarschuwing: `warning: format '%d' expects argument of type 'int*', but argument 2 has type 'double'`.

Hetzelfde voorbeeld met `iostream` (iostream.cpp):

```
import std;
using namespace std;
//  ...
    double d;
    cin >> d; // lees d in vanaf toetsenbord
    cout << "d = " << fixed << setprecision(3) << d << '\n'; // ↩
 ↪ druk d af op scherm met drie decimalen achter de punt ga naar ↩
 ↪ het begin van de volgende regel
```

De `iostream` library is zeer uitgebreid. In dit dictaat gaan we daar niet verder op in. Zie eventueel voor verdere informatie [1, hoofdstuk 6: IOStreams] of http://en.cppreference.com/w/cpp/io.

De meeste programmeurs vinden het gebruik van `cout` en `<<` om output te formateren minder handig dan het gebruik van de functies `print` en `println`. In dit dictaat gebruiken we daarom voor output de functies `print` en `println` en voor input de `>>` operator (lees_en_print_double.cpp):

```
import std;
using namespace std;

int main() {
    print("Type een foating point getal in: ");
    double d;
    cin >> d;
    print("Hoeveel decimalen wil je afdrukken?: ");
    int aantal_decimalen;
    cin >> aantal_decimalen;
    println("d = {:.{}f}", d, aantal_decimalen);
}
```

De in- en uitvoer van dit programma is bijvoorbeeld als volgt[19]:

```
Type een foating point getal in: 666.66666666666666666666666666
Hoeveel decimalen wil je afdrukken?: 5
d = 666.66667
```

[19] De invoer is groen en onderstreept weergegeven.

Je ziet dat het ook mogelijk is om binnen een format specifier een invulveld te gebruiken dat tijdens de uitvoering van het programma ingevuld wordt, in dit geval met de waarde van de variabele `aantal_decimalen`.

1.4 Universele initialisatie

In C ben je gewend om een variabele te initialiseren met de volgende syntax:

```
double d = 4.6;
```

In C++ kan deze syntax nog steeds gebruikt worden. Het is in C++, om redenen die verderop in dit dictaat duidelijk worden, belangrijk om onderscheid te maken tussen het aanmaken en initialiseren van een variabele en het toekennen van een waarde aan een bestaande variabele (met =)[20]. De bovenstaande syntax kan dan verwarrend zijn. In C++ kan een variabele ook geïnitialiseerd worden via een zogenoemde *universele initialisatie*. De waarde waarmee de variabele geïnitialiseerd moet worden, wordt dan tussen accolades achter de variabelenaam geplaatst. Bijvoorbeeld:

```
double d {4.6};
```

Eventueel kan tussen de variabelenaam en de accolade openen nog een is-teken gebruikt worden:

```
double d = {4.6};
```

Het gebruik van universele initialisatie ten opzichte van de andere mogelijke manieren van initialiseren[21] heeft de volgende voordelen:

- Deze vorm kan gebruikt worden op alle plaatsen waar een initialisatie gebruikt kan worden. Dus bijvoorbeeld ook om een **`struct`** te initialiseren:

```
struct Breuk {
    int teller;
    int noemer;
} half {1, 2};
```

- Bij deze vorm van initialisatie vindt geen type-conversie plaats. Bijvoorbeeld:

```
int i = 4.6; // i wordt 4. Vreemd!
```

[20] Dat het belangrijk is om het verschil te weten tussen een initialisatie en een toekenning (Engels: assignment) wordt duidelijk in paragrafen 2.4, 2.7 en 2.8.

[21] Een variabele kan ook geïnitialiseerd worden met een is-teken of door het gebruik van ronde haakjes.

```
    int j {4.6} // error: narrowing conversion of '4.6' from ↩
↪ 'double' to 'int'
```

Om deze redenen is er voor gekozen om in dit dictaat altijd gebruik te maken van universele initialisatie[22].

1.5 auto type

Als een variabele geïnitialiseerd wordt, is het mogelijk om de compiler zelf het type van een variabele te laten bepalen. Dit doe je door in de variabeledefinitie, in plaats van een typenaam, het keyword **auto** te gebruiken. Bijvoorbeeld:

```
auto antwoord {42};
auto ongeveer_pi {3.14};
```

De variabele `antwoord` is nu van het type **int** en de variabele `ongeveer_pi` is nu van het type **double**. Meestal is het beter om toch zelf het type te definiëren[23]. Dat maakt je programma duidelijker en voorkomt fouten.

Verderop in dit dictaat kom je een aantal meer zinvolle manieren van het gebruik van **auto** tegen:

- Bij het gebruik van templates, zie hoofdstuk 3, is het soms wel erg handig om **auto** te gebruiken.
- Bij het gebruik van een range-based **for**-statement, zie paragraaf 1.7 is het handig om **auto** te gebruiken.

Het gebruik van **auto** kan ook de aanpasbaarheid van je programma vergroten. Bijvoorbeeld, in het onderstaande programma wordt een array `kampioensjarenFeyenoord` met integers aangemaakt en verderop in het programma wordt de variabele `eerste` gelijk gemaakt aan het eerste getal uit deze array.

```
    int kampioensjaren_feyenoord[] {1924, 1928, 1936, 1938, 1940, ↩
↪ 1961, 1962, 1965, 1969, 1971, 1974, 1984, 1993, 1999, 2017, ↩
↪ 2023};

    int eerste {kampioensjaren_feyenoord[0]};
```

[22] Dit wordt ook aangeraden in de C++ Core Guideline ES.23.

[23] Onder C++ experts is dit een punt van discussie. Er zijn aanhangers van het AAA-idiom (Almost Always Auto) die juist voorstellen om zoveel mogelijk **auto** te gebruiken. Wij volgen het advies uit de C++ Core Guideline ES.11 en gebruiken **auto** alleen als dit de code duidelijker maakt.

We kunnen het type van de variabele `eerste` in dit geval ook door de compiler laten bepalen (Engels: type deduction), zie `auto.cpp`:

```
    int kampioensjaren_feyenoord[] {1924, 1928, 1936, 1938, 1940, ↩
↪ 1961, 1962, 1965, 1969, 1971, 1974, 1984, 1993, 1999, 2017, ↩
↪ 2023};

    auto eerste {kampioensjaren_feyenoord[0]};
```

Dit komt de onderhoudbaarheid van het programma ten goede want als het type van de array nu veranderd wordt van `int` naar `uint16_t` dan wordt het type van de variabele `eerste` automatisch aangepast.

1.6 `auto` return type

Het is ook mogelijk om de compiler zelf het return type van een functie te laten bepalen. Dit doe je door in de functiedefinitie in plaats van een typenaam het keyword `auto` te gebruiken. Stel dat in een programma de onderstaande functie `samengestelde_interest` is gedefinieerd die een eindkapitaal berekent als een startkapitaal voor een aantal jaren op een spaarrekening wordt gezet tegen een vaste rente.

```
float samengestelde_interest(float startkapitaal, float ↩
↪ rente_per_jaar, int aantal_jaar) {
    float eindkapitaal {startkapitaal};
    for (int jaar {1}; jaar <= aantal_jaar; ++jaar) {
        eindkapitaal = eindkapitaal * (1 + rente_per_jaar);
    }
    return eindkapitaal;
}
```

We kunnen het return type van de functie ook door de compiler laten bepalen (Engels: return type deduction), zie `auto_return_type.cpp`:

```
auto samengestelde_interest(float startkapitaal, float ↩
↪ rente_per_jaar, int aantal_jaar) {
    float eindkapitaal {startkapitaal};
    for (int jaar {1}; jaar <= aantal_jaar; ++jaar) {
        eindkapitaal = eindkapitaal * (1 + rente_per_jaar);
    }
    return eindkapitaal;
}
```

Dit komt de onderhoudbaarheid van het programma ten goede. Als later blijkt dat het type `float` niet nauwkeurig genoeg is om het eindkapitaal mee te berekenen[24], dan hoeft alleen het type van de variabele `eindkapitaal` verandert te worden in `double`. Het return type van de functie wordt dan ook automatisch aangepast van `float` naar `double`.

1.7 Range-based `for`

In C++ kan dezelfde `for`-lus als in C gebruikt worden. Maar er is in C++ ook nieuwe variant van de `for`-lus, de zogenoemde *range-based for*, beschikbaar. Deze `for`-lus is speciaal bedoeld om de elementen van een array[25] één voor één te benaderen.

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

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

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

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

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

We kunnen het type van de variabele `element` ook door de compiler laten bepalen[26], zie paragraaf 1.5 (`range-based_for.cpp`):

[24] De functie berekent dat het voor 10 jaar vastzetten van € 100 000 000 tegen een rente van 1,35 % € 114 350 328,00 oplevert. Dit is echter € 2,47 te weinig.

[25] De standaard C++ library bevat nog vele andere datastructuren die ook met een range-based `for` doorlopen kunnen worden zie paragraaf 3.6 en paragraaf 3.7.

[26] Als je het type van de variabele `som` ook automatisch van hetzelfde type wilt definiëren als het type van de array `rij`, dan kun je *geen* gebruik maken van `auto` omdat deze variabel met de waarde `0` wordt geïnitialiseerd en dus altijd van het type integer is als je `auto` gebruikt. Met behulp van `decltype` is dit wel mogelijk, zie paragraaf 15.11.

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

Als het type van de variabele `rij` gewijzigd wordt, dan wordt het type van de variabele `element` automatisch aangepast.

Als je de elementen in de `rij` wilt veranderen in de range-based **for** dan heb je een zogenoemde reference variabele nodig. Dit wordt verderop in dit dictaat uitgelegd in paragraaf 2.14 en paragraaf 2.18.

1.8 Constante waarden met constexpr

Je bent gewend om in C programma's symbolische constanten[27] te definiëren met de preprocessor directive **#define**. In C++ definieer je een zogenoemde compile time constante met behulp van het keyword **constexpr**. Een compile time constante is een constante waarvan de waarde al tijdens het compileren van het programma bepaald kan worden. De grootte van een array moet bijvoorbeeld een compile time constante zijn. Ook een **case**-label in een **switch**-statement moet een compile time constante zijn. Een compile time constante kan door de compiler voor een embedded systeem in read-only geheugen worden geplaatst.

Voorbeeld met **#define**:

```
#define aantal_regels 80
```

Hetzelfde voorbeeld met **constexpr**:

```
constexpr int aantal_regels {80};
```

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

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

Een compile time constante moet je initialiseren:

```
constexpr int k;
//  error: uninitialized const 'k'
```

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

```
int i;
//  ...
constexpr int m {i};
//  error: the value of 'i' is not usable in a constant expression
```

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

```
aantalRegels = 79;
//  error: assignment of read-only variable 'aantal_regels'
```

Je kunt in C++ ook functies definiëren die tijdens compile time uitgevoerd kunnen worden, door het returntype met **constexpr** te definiëren, zie paragraaf 15.12.

1.9 Read-only variabelen met const

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

Voorbeeld met **const**:

```
const int aantal_regels {80};
```

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

```
const int k;
//  error: uninitialized const 'k'
```

Je mag een constante wel met een variabele initialiseren.

```
int i;
//  ...
const int m {i};
```

Als een symbolische constante geïnitialiseerd wordt met een compile time constante, zoals de bovenstaande constante `aantal_regels`, dan is deze symbolische constante ook te gebruiken als compile time constante. Een compile time constante kan door de compiler voor een

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

Een constante mag je (vanzelfsprekend) niet veranderen:

```
aantal_regels = 79;
//  error: assignment of read-only variable 'aantal_regels'
```

In paragraaf 2.9 ga je zien dat het met een **const** qualifier ook mogelijk wordt om constanten (read-only variabelen) te definiëren van zelfgemaakte types.

Bij het definiëren van pointers kun je het keyword **const** op verschillende manieren gebruiken om de waarde waar de pointer naar wijst of de pointer zelf of beide read-only te maken, zie paragraaf 15.14.

1.10 Het type `string`

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

Bijvoorbeeld het vergelijken van character strings in C:

```
#include <stdio.h>
#include <string.h>
//  ...
    char str[100];
    scanf("%99s", str);
    if (strcmp(str, "Hallo") == 0) {
        // invoer is Hallo
    }
```

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

```
import std;
using namespace std;
//  ...
    string str;
    cin >> str;
    if (str == "Hallo") {
        // invoer is Hallo
    }
```

De `string` library is zeer uitgebreid. In dit dictaat gaan we daar niet verder op in. Zie eventueel voor verdere informatie [1, hoofdstuk 5: String] of https://en.cppreference.com/w/cpp/string.

1.11 Het type `array` en `vector`

In de C++ standaard library zijn ook de types `array` en `vector` opgenomen. Deze types zijn bedoeld om de oude C-array te vervangen. In paragraaf 3.6 en paragraaf 3.7 komen we hier uitgebreid op terug.

1.12 Function name overloading

In C mag elke functienaam maar één keer gedefinieerd worden. Dit betekent dat twee functies om de modulus (absolute waarde) te bepalen van variabelen van de types **`int`** en **`double`** verschillende namen moeten hebben. In C++ mag een functienaam meerdere keren gedefinieerd worden (*function name overloading*). De compiler selecteert aan de hand van de gebruikte argumenten de juiste functie. Dit maakt deze functies eenvoudiger te gebruiken omdat de gebruiker (de programmeur die deze functies aanroept) slechts één naam hoeft te onthouden.[28] Dit is vergelijkbaar met het gebruik van ingebouwde operator + die zowel gebruikt kan worden om integers als om floating-point getallen op te tellen.

[28] 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.

Voorbeeld zonder function name overloading:

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

Hetzelfde voorbeeld met function name overloading:

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

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

```
    double d;
    cin >> d; // lees d in
    println("{}", absolute_waarde(d)); // druk de absolute waarde ↩
 ↪ van d af
```

De compiler bepaalt nu zelf aan de hand van het type van het gebruikte argument (d) welk van de twee bovenstaande absolute_waarde functies aangeroepen wordt.[29]

1.13 Default argumenten

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

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

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

[29] In C kun je een soortgelijk effect creëren door het gebruik van een macro in combinatie met het keyword _Generic, zie eventueel: https://en.cppreference.com/w/c/language/generic.

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

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

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

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

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

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

```
drukaf(12, 2);  // uitvoer: 1100
drukaf(12);     // uitvoer: 12
drukaf(12, 10); // uitvoer: 12
drukaf(12, 7);  // uitvoer: 15
```

Het default argument maakt de `drukaf` functie eenvoudiger te gebruiken omdat de gebruiker (de programmeur die deze functies aanroept) niet steeds het tweede argument hoeft mee te geven als zij/hij getallen decimaal wil afdrukken.[30]

1.14 Naam van een `struct`

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

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

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

```
struct Tijdsduur td1;
```

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

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

```
typedef struct Tijdsduur Tijdsduur_t;
```

Variabelen van dit type kunnen dan als volgt gedefinieerd worden:

```
    Tijdsduur_t 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 hoofdstuk 2 ga je zien dat C++ het mogelijk maakt om op een veel betere manier een tijdsduur te definiëren (als user-defined data type).

1.15 C++ als een betere C

Als we de verbeteringen die in C zijn doorgevoerd bij de definitie van C++ toepassen in het voorbeeld programma van pagina 14, dan ontstaat het C++ programma `C.cpp`:

```
import std;
using namespace std;

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

// Deze functie drukt een Tijdsduur af
void drukaf(Tijdsduur td) {
    if (td.uur == 0)
        println("{:13} minuten", td.minuten);
    else
        println("{:3} uur en {:2} minuten", td.uur, td.minuten);
}

// Deze functie drukt een rij met een aantal Tijdsduren af
void drukaf(const Tijdsduur rij[], size_t aantal) {
    for (size_t teller {0}; teller < aantal; ++teller)
        drukaf(rij[teller]);
```

```
}

// Deze functie berekent de totale Tijdsduur van een rij met een ↩
 ↪ aantal Tijdsduren
auto som(const Tijdsduur rij[], size_t aantal) {
    Tijdsduur s {0, 0};
    for (size_t teller {0}; teller < aantal; ++teller) {
        s.uur += rij[teller].uur;
        s.minuten += rij[teller].minuten;
    }
    s.uur += s.minuten / 60;
    s.minuten %= 60;
    return s;
}

int main() {
    constexpr size_t MAX_TIJDSDUREN {5};
    Tijdsduur tijdsduren[MAX_TIJDSDUREN];
    size_t aantal {0};
    do {
        print("Type uren en minuten in: ");
        cin >> tijdsduren[aantal].uur >> tijdsduren[aantal].minuten;
    }
    while (cin && ++aantal < MAX_TIJDSDUREN);
    println();
    drukaf(tijdsduren, aantal);
    println("De totale tijdsduur is:");
    drukaf(som(tijdsduren, aantal));
}
```

Verklaring van de verschillen:

- Hier is gebruik gemaakt van de standaard library van C++, zie paragrafen 1.2 en 1.3 in plaats van de standaard C library. In de eerste regel wordt de module `std` geïmporteerd, paragraaf 1.1. Dit is nodig om gebruik te kunnen maken van functies, objecten en types die in de standaard C++ library zijn opgenomen. In dit programma maak ik gebruik van `print` en `println` (om te schrijven naar het scherm) en `cin` (om te lezen vanaf het toetsenbord).

- In de derde regel wordt aangegeven dat de namespace `std` wordt gebuikt, zie pagina 21. Dit is nodig om de functies, types enz. die in de standard C++ library zijn gedefinieerd te kunnen gebruiken zonder er steeds `std::` voor te zetten.
- De naam van de **`struct`** `Tijdsuur` kan meteen als typenaam gebruikt worden. Een **`typedef`** is hier dus niet nodig, zie paragraaf 1.14.
- Om de datavelden van een tijdsduur af te drukken is hier gebruik gemaakt van `print` en `println` 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 functie om een rij af te drukken heeft hier de naam `drukaf` in plaats van `drukaf_rij`. 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, zie paragraaf 1.12.
- De functie `som` heeft nu het returntype **`auto`**. De compiler kan namelijk aan de hand van het **`return`**-statement zelf bepalen dat het returntype `Tijdsduur` is, zie paragraaf 1.6.
- De lokale variabele `teller` wordt geïnitialiseerd met een universele initialisatie in plaats van met een toekenning, zie paragraaf 1.4.
- De compile time constante `MAX_TIJDSDUREN` is hier als **`constexpr`** gedefinieerd in plaats van met **`#define`**, zie paragraaf 1.8.
- Het inlezen vanaf het toetsenbord is hier gedaan met behulp van `cin` in plaats van `scanf`. Het object `cin` kan gebruikt worden in een conditie om te testen of de vorige leesbewerking gelukt is. De variabele `gelezen` is dus niet meer nodig.
- De functie `main` heeft in C++ net zoals in C99 en hoger geen **`return`**-statement meer nodig.

Merk op dat dit programma niet meer dan `MAX_TIJDSDUREN` tijdsduren kan verwerken. Later in dit dictaat leer je hoe dit probleem is op te lossen door gebruik te maken van het type `vector`, zie paragraaf 3.7. Aan het einde van dit hoofdstuk geef ik je al een voorproefje.

Misschien vraag je jezelf af waarom in de functies `drukaf` en `som` geen gebruik is gemaakt van de range-based **`for`**, zie paragraaf 1.7. De reden is simpel: het werkt niet. De parameter `rij` is namelijk geen echte array maar een referentie naar het beginadres van de array `tijdsduren` die als argument wordt meegegeven bij aanroep. In de functie is het aantal elementen van `tijdsduren` dus niet bekend. Dat is ook de reden dat we dit aantal zelf als tweede parameter `aantal` mee moeten geven aan de functie. Het is dus niet zo handig om in C++ de ingebouwde arrays, die overgenomen zijn uit C, te gebruiken. Het is beter om

gebruik te maken van de nieuwe types uit de standaard C++ library zoals `array` en `vector`, zie paragraaf 1.11 en het onderstaande programma.

Als je toch probeert om een range-based **for** te gebruiken, dan krijg je de volgende cryptische foutmeldingen:

```
error: 'begin' was not declared in this scope; did you mean ↩
 ↪ 'std::begin'?
error: 'end' was not declared in this scope; did you mean 'std::end'?
```

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

Tot slot van dit hoofdstuk bekijken we een versie van bovenstaand programma waarbij gebruik gemaakt wordt van het type `vector` uit de standaard library, zie `C_vector.cpp`. Het is bedoeld om je een indruk te geven van mogelijkheden van C++. Het is niet nodig om alles al helemaal te begrijpen, we komen er in paragrafen 2.16 en 3.7 nog uitgebreid op terug.

```
import std;
using namespace std;

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

// Deze functie drukt een Tijdsduur af
void drukaf(Tijdsduur td) {
    if (td.uur == 0)
        println("{:13} minuten", td.minuten);
    else
        println("{:3} uur en {:2} minuten", td.uur, td.minuten);
}

// Deze functie drukt een rij met Tijdsduren af
void drukaf(const vector<Tijdsduur>& rij) {
    for (auto t: rij)
```

```
        drukaf(t);
}

// Deze functie berekent de totale Tijdsduur van een rij met ↩
 ↪ Tijdsduren
auto som(const vector<Tijdsduur>& rij) {
    Tijdsduur s {0, 0};
    for (auto t: rij) {
        s.uur += t.uur;
        s.minuten += t.minuten;
    }
    s.uur += s.minuten / 60;
    s.minuten %= 60;
    return s;
}

int main() {
    vector<Tijdsduur> tijdsduren;
    do {
        Tijdsduur t;
        print("Type uren en minuten in: ");
        cin >> t.uur >> t.minuten;
        if (cin)
            tijdsduren.push_back(t);
    }
    while (cin);
    println();
    drukaf(tijdsduren);
    println("De totale tijdsduur is:");
    drukaf(som(tijdsduren));
}
```

Verklaring van de verschillen:

- Hier is gebruik gemaakt van het type `vector` uit de standaard library. Een `vector<T>`[31] is een datastructuur waarin een onbepaald aantal elementen van het type `T` kan worden opgeslagen.

[31] De < en > worden hier gebruikt als een soort van haakjes, zie eventueel hoofdstuk 3

- Dit type wordt in `main` gebruikt om een `vector<Tijdsduur>` genaamd `tijdsduren` aan te maken. Hierin kunnen dus een onbepaald aantal elementen van het type `Tijdsduur` worden opgeslagen. Bij het aanmaken is de `vector` nog leeg.
- In `main` kunnen we nu telkens als een `Tijdsduur t` is ingelezen deze toevoegen aan de `vector` met het statement `tijdsduren.push_back(t)`. De tijdsduur `t` wordt hiermee aan het einde van `tijdsduren` toegevoegd.
- De functies `drukaf` en `som` hebben nu nog maar één parameter `rij`, van het type `const vector<Tijdsduur>&`. Met het teken `&` geven we aan dat de parameter call by reference moet gebruiken en met **`const`** geven we aan dat de parameter read-only moet zijn. Dit doen we omdat we niet willen dat de hele `vector` gekopieerd wordt bij het aanroepen van de functie en omdat we de `vector` alleen uit willen lezen. De `vector` 'weet' zelf hoeveel elementen er in zitten, dus dit hoeven we niet meer als tweede parameter aan deze functies mee te geven.
- Omdat de `vector` zelf weet hoeveel elementen er in zitten, kunnen we nu de range-based **`for`** gebruiken om elementen van de vector één voor één te behandelen. Zie paragraaf 1.7.

In dit hoofdstuk zijn de kleine verbeteringen van C++ ten opzichte van zijn voorganger C besproken die je moet weten om de rest van dit dictaat te kunnen lezen. Verdere kleine verbeteringen, die ook best handig zijn bij het programmeren in C++ vind je in hoofdstuk 15.

2

Objects and classes

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

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

In deze paragraaf gaan we eerst in 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 Express bus).

Bij het ontwerpen van software was het, voor het ontstaan van OOP, niet gebruikelijk om gebruik te maken van complexere componenten die door anderen ontwikkeld zijn (en waarvoor je dus moet betalen). Vaak werden wel bestaande datastructuren en algoritmen gekopieerd maar die moesten vaak toch aangepast worden. Het gebruik van functie libraries was wel gebruikelijk maar dit zijn in feite eenvoudige componenten. Een voorbeeld van een complexe component is bijvoorbeeld een editor (inclusief menu opties: openen, opslaan, opslaan als, print, printer set-up, bewerken (knippen, kopiëren, plakken), zoeken, zoek&vervang en inclusief een knoppenbalk). In alle applicaties waarbij je tekst moet kunnen invoeren, kun je deze editor dan hergebruiken.

Ondanks dat het gebruik van softwarecomponenten, die we kunnen kopen en waarmee we vervolgens zelf applicaties kunnen ontwikkelen, sinds de introductie van OOP is toegenomen blijft het gebruik van softwarecomponenten toch achter bij het gebruik van hardwarecomponenten. Dit heeft volgens mij verschillende redenen:

- Voor een softwarecomponent moet je net als voor een hardwarecomponent betalen. Bij veel hardwarecomponenten komt het niet in je op om ze zelf te maken (een videokaart bijvoorbeeld), maar een softwarecomponent kun je ook altijd zelf (proberen te) maken. Zo hoor je bijvoorbeeld soms zeggen: “Maar dat algoritme staat toch gewoon in het boek van . . . dus dat implementeren we zelf wel even”.
- Als je van een bepaalde hardwarecomponent 1000 stuks gebruikt, moet je er ook 1000 maal voor betalen. Bij een softwarecomponent vindt men dit minder normaal en er valt eenvoudiger mee te sjoemelen.
- Programmeurs denken vaak: maar dat kan ik zelf toch veel eenvoudiger en sneller programmeren. Daarbij overschatten ze vaak hun eigen kunnen en onderschatten ze de tijd die nodig is om een component grondig te testen.

- De omgeving (taal, operating system, hardware enz.) van software is divers. Dit maakt het produceren van softwarecomponenten niet eenvoudig. “Heeft u deze component ook in C++ voor Linux in plaats van in C# voor Windows?”[32]

De objectgeoriënteerde benadering is vooral bedoeld om het ontwikkelen van herbruikbare softwarecomponenten mogelijk te maken. In dit dictaat maak je kennis met deze objectgeoriënteerde benadering. Wij hebben daarbij gekozen voor de taal C++ omdat dit de meest gebruikte objectgeoriënteerde programmeertaal is voor embedded systems[33]. Het is echter niet zo dat je na het bestuderen van dit dictaat op de hoogte bent van alle aspecten en details van C++. Wel begrijp je na het bestuderen van dit dictaat de algemene ideeën achter OOP en kun je die toepassen in C++. De objectgeoriënteerde benadering is een (voor jou) nieuwe manier van denken over hoe je informatie in een computer kunt structureren.

De manier waarop we problemen oplossen met objectgeoriënteerde technieken lijkt op de manier van problemen oplossen die we in het dagelijks leven gebruiken. Een objectgeoriënteerde applicatie is opgebouwd uit deeltjes die we *objecten* noemen. Elk object is verantwoordelijk voor een bepaald deel van de functionaliteit van de applicatie. Een object kan aan een ander object vragen of dit object wat voor hem wil doen, dit gebeurt door het zenden van een boodschap (Engels: *message*) naar dat object. Aan deze message kunnen indien nodig *argumenten* worden meegegeven. Het is de verantwoordelijkheid van het ontvangende object (Engels: the *receiver*) hoe het de message afhandelt. Zo kan dit object de verantwoordelijkheid afschuiven door zelf ook weer een message te versturen naar een ander object.

Het algoritme dat de ontvanger gebruikt om de message af te handelen wordt de methode (Engels: *method*) genoemd. Het object dat de message verstuurt is dus niet op de hoogte van de door de ontvanger gebruikte method. Dit wordt *information hiding* genoemd. Het versturen van een message lijkt in eerste instantie op het aanroepen van een functie. Toch zijn er enkele belangrijke verschillen:

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

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

[33] Bron: AspenCore 2025 Embedded Market Study.

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

Een derived class overerft alle eigenschappen van een base class (overerving = Engels: *inheritance*). Als een object een message ontvangt, wordt de bijbehorende method als volgt bepaald (Engels: *method binding*):

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

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

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

De volgende abstractietechnieken zijn achtereenvolgens ontstaan:

- **Functies en procedures.**
  In de eerste hogere programmeertalen zoals Algol, 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.
- **Modules.**
  In latere 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.
- **User-defined data types (UDT's).**
  In latere programmeertalen zoals Ada en JavaScript kan een bepaalde datastructuur met de bij deze datastructuur behorende functies en procedures ingekapseld worden in een UDT. Het verschil tussen een module en een UDT is dat een module alleen gebruikt kan worden en dat een UDT geïnstantieerd kan worden. Dit wil zeggen dat er meerdere variabelen van dit UDT (=type) aangemaakt kunnen worden. Als de specificatie van het UDT bekend is, kun je dit type op dezelfde wijze gebruiken als de ingebouwde types van de programmeertaal (zoals `char`, `int` en `double`) zonder dat je de implementatie details van dit type hoeft te kennen of te begrijpen. Dit is dus weer een vorm van information hiding. Op het begrip UDT komen we nog uitgebreid terug.

- **Generieke functies en datatypes.**
  In moderne programmeertalen zoals Java, C♯, Python, Ada en ook in C++ kunnen door het gebruik van *templates* generieke functies en datatypes gedefinieerd worden. Een generieke functie is een functie die gebruikt kan worden voor meerdere parametertypes. In C kan om een array met `int`'s te sorteren een functie geschreven worden. Als we vervolgens een 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 arrays van elk willekeurig type gesorteerd kunnen worden. Een generiek type is een UDT dat met verschillende andere types gecombineerd kan worden. In C++ kunnen we bijvoorbeeld een UDT array definiëren waarin we variabelen van het type `int` kunnen opslaan. Door het gebruik van een template kunnen we een generiek UDT array definiëren waarmee we dan arrays van elk willekeurig type kunnen gebruiken. Op generieke functies en datatypes komen we nog uitgebreid terug.
- **Classes.**
  In moderne programmeertalen zoals Java, C♯, Python, PHP, Ruby, Perl, Smalltalk en ook in C++ kunnen een aantal logisch bij elkaar behorende UDT's van elkaar worden afgeleid door middel van inheritance. Door nu programma's te schrijven in termen van base classes (basisklassen) en de in deze base classes gedefinieerde messages in derived classes (afgeleide klassen) te implementeren kan je deze base classes gebruiken zonder dat je de implementatie van deze classes hoeft te kennen of te begrijpen en kun je eenvoudig van implementatie wisselen. Zo'n base class wordt dan ook wel een Abstract Data Type (ADT) genoemd omdat zo'n class geen implementatie bevat en dus niet geïnstantieerd kan worden. De derived classes die dit ADT implementeren worden ook wel Concrete Data Types (CDT's) genoemd.
  Tevens kun je code die alleen messages aanroept van de base class zonder opnieuw te compileren hergebruiken voor alle direct of indirect van deze base class afgeleide classes. Dit wordt veelvormigheid of met een moeilijk woord polymorfisme (Engels: *polymorphism*) genoemd. Dit laatste wordt mogelijk gemaakt door de message pas tijdens run time aan een bepaalde method te verbinden en wordt late binding (Engels: *late binding*) genoemd. Op deze begrippen komen we later nog uitgebreid terug.

Toen je leerde programmeren in C heb je geleerd hoe je vanuit een probleemstelling voor een programma de benodigde functies kan vinden. Deze methode bestond uit het opdelen van het probleem in deelproblemen, die op hun beurt weer werden opgedeeld in deelproblemen enz. net zo lang tot de oplossing eenvoudig werd. Deze methode heet functionele decompositie. De softwareontwikkelmethoden die hiervan zijn afgeleid worden *structured analyse and design* genoemd. De bekende Yourdon methode [27] is daar een voorbeeld van. Bij het

gebruik van objectgeoriënteerde 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? Het antwoord op deze vraag wordt bepaald door de gebruikte OOA (Object Oriented Analyse) en OOD (Object Oriented Design) methode.

2.2 UDT's (User-defined Data Types)

De eerste stap op weg naar het objectgeoriënteerde denken en programmeren is het leren programmeren met UDT's. Een UDT is een user-defined type dat voor een gebruiker[34] niet te onderscheiden is van ingebouwde types (zoals **int**). Een UDT koppelt een bepaalde datastructuur (interne variabelen) met de bij deze datastructuur behorende bewerkingen (functies). Deze functies en variabelen kunnen voor de gebruikers van het UDT zichtbaar (*public*) of onzichtbaar (*private*) gemaakt worden. Functies en variabelen die private zijn kunnen alleen door functies van het UDT zelf gebruikt worden. De private functies en variabelen zitten ingekapseld in het UDT. Als de specificatie van het UDT bekend is, kun je dit type op dezelfde wijze gebruiken als de ingebouwde types van de programmeertaal (zoals **char**, **int** en **double**) zonder dat je de implementatie details van dit type hoeft te kennen of te begrijpen. Dit is een vorm van information hiding.

De programmeertaal C biedt geen directe ondersteuning om UDT's te definiëren[35]. Als je in C een programma wilt schrijven waarbij je met breuken in plaats van met floating-point getallen wilt werken[36], dan kun je dit (niet in C opgenomen) type `Breuk` op de volgende manier zelf definiëren (`Breukc.c`):

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

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

[35] Het is erg omslachtig om UDT's te definiëren in C, in C++ is dit veel eenvoudiger.

[36] 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 afronding kan ervoor zorgen dat een berekening zoals $3 \times {}^1/_3$ 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.

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

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

Het normaliseren[37] 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.

Deze manier van werken heeft de volgende nadelen:

- Iedere programmeur die gebruikt maakt van het type `Breuk` kan een waarde toekennen aan de datavelden `boven` en `onder`. Het is in C niet te voorkomen dat een programmeur het dataveld `onder` van een variabele van het type `Breuk` per ongeluk op nul zet. Als een programmeur het dataveld `onder` van een variabele van het type `Breuk` op nul zet, kan dit een fout veroorzaken in code van een andere programmeur die bijvoorbeeld deze variabele van het type `Breuk` naar het type **`double`** converteert. Als deze fout geconstateerd wordt, kunnen alle functies waarin breuken gebruikt worden de boosdoeners zijn.
- Iedere programmeur die gebruik maakt van het type `Breuk` kan er voor kiezen om zelf de code voor het optellen van breuken 'uit te vinden' in plaats van gebruik te maken van de functie `som`. Er valt dus niet te garanderen dat alle optellingen van breuken correct en genormaliseerd zijn. Ook niet als we wel kunnen garanderen dat de functies `som` en `normaliseer` correct zijn. Iedere programmeur die breuken gebruikt kan ze namelijk op zijn eigen manier optellen.
- Iedere programmeur die gebruikt maakt van het type `Breuk` kan zelf nieuwe bewerkingen (zoals bijvoorbeeld het vermenigvuldigen van breuken) definiëren. Het zou beter zijn als alleen de programmeur die verantwoordelijk is voor het onderhouden van het type `Breuk` (en de bijbehorende bewerkingen) dit kan doen.

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

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

In C++ kunnen we door gebruik te maken van een *class*, een eigen type `Breuk` als volgt declareren (`Breuk0.cpp`):

```
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[38];//-  afdrukken op het scherm;
    void plus(Breuk[39] b);// -  een Breuk erbij optellen.
private:                // Een object van de class Breuk heeft privé:
    int boven;          // -  een teller;
    int onder;          // -  een noemer;
    void normaliseer(); // -  een functie normaliseer.
};
```

De class `Breuk` koppelt een bepaalde datastructuur (twee interne integer variabelen genaamd `boven` en `onder`) met de bij deze datastructuur behorende bewerkingen[40] (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.[41] Deze private functies en variabelen zitten ingekapseld in de class `Breuk`. De implementatie van de class is zichtbaar in de definitie van de class (als **`private`** datamembers). Dit zorgt ervoor dat compiler een object van een UDT, ook wel een *concrete* class genoemd, efficiënt (zowel in ruimte als in tijd) kan implementeren en bewerken. Het nadeel is echter dat een concrete class opnieuw gecompileerd moet worden als de implementatie wijzigt. Verderop in dit dictaat kun je leren hoe je door een zogenoemde *abstracte* class te gebruiken ervoor kunt zorgen dat niet alles opnieuw gecompileerd hoeft te worden bij een wijziging of uitbreiding, zie paragraaf 4.7.4.

[38] **`const`** memberfunctie wordt behandeld in paragraaf 2.9.

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

[40] 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).

[41] 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 die opgenomen zijn in een class worden *memberfuncties* genoemd. De memberfunctie `plus` kan als volgt gedefinieerd worden[42]:

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

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

```
    Breuk a, b; // definieer de objecten a en b van de class Breuk
    a.leesin(); // lees a in
    b.leesin(); // lees b in
    a.drukaf();
    print(" + ");
    b.drukaf();
    print(" = ");
    a.plus(b);  // tel b bij a op
    a.drukaf(); // druk a af
    println();
```

Een object heeft drie kenmerken:

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

[42] De definitie van de memberfuncties `leesin`, `drukaf` en `normaliseer` zijn te vinden in `Breuk0.cpp`.

een eigen identiteit. De twee objecten van de class `Breuk` in het voorgaande voorbeeld hebben bijvoorbeeld elk een eigen naam (`a` en `b`) waarmee ze geïdentificeerd kunnen worden.

Bij de memberfunctie `plus` wordt bij de definitie met de zogenoemde qualifier `Breuk::` aangegeven dat deze functie een member is van de class `Breuk`. De memberfunctie `plus` kan alleen op een object van de class `Breuk` uitgevoerd worden. Dit wordt genoteerd als: `a.plus(b);` Het object `a` wordt de *receiver* (ontvanger) van de memberfunctie genoemd. Als in de definitie van de memberfunctie `plus` de datavelden `boven` en `onder` gebruikt worden, dan zijn dit de datavelden van de receiver (in dit geval object `a`). Als in de definitie van de memberfunctie `plus` de memberfunctie `normaliseer` gebruikt wordt, dan wordt deze memberfunctie op de receiver uitgevoerd (in dit geval object `a`). De betekenis van **`const`** achter de declaratie van de memberfunctie `drukaf` komt later (paragraaf 2.9) aan de orde.

Deze manier van werken heeft de volgende voordelen:

- Een programmeur die gebruik maakt van de class (het type) `Breuk` kan *géén* waarde toekennen aan de private datavelden `boven` en `onder`. Alleen de memberfuncties `leesin`, `drukaf`, `plus` en `normaliseer` kunnen een waarde toekennen aan deze datavelden. Als ergens een fout ontstaat omdat het dataveld `onder` van een `Breuk` op nul is gezet, kunnen alleen de memberfuncties van `Breuk` de boosdoeners zijn.
- Een programmeur die gebruik maakt van de class `Breuk` kan er *niet* voor kiezen om zelf de code voor het optellen van breuken 'uit te vinden' in plaats van gebruik te maken van de memberfunctie `plus`. Als we kunnen garanderen dat de functies `plus` en `normaliseer` correct zijn, dan kunnen we dus garanderen dat alle optellingen van breuken correct en genormaliseerd zijn. Iedere programmeur die breuken gebruikt kan ze namelijk alleen optellen door gebruik te maken van de memberfunctie `plus`.
- Iedere programmeur die gebruikt maakt van de class `Breuk` kan *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 C++ in plaats van de **`typedef struct`** `Breuk` met de bijbehorende functies in C 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 UDT is een user-defined type dat voor een gebruiker niet te onderscheiden is van ingebouwde types (zoals **int**)”. Een UDT Breuk moet dus uiteindelijk als volgt gebruikt kunnen worden:

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

Je ziet dat het zelfgedefinieerde type Breuk uiteindelijk 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 bespreek ik hoe de class Breuk stap voor stap uitgebreid en aangepast kan worden zodat uiteindelijk een UDT Breuk ontstaat. Dit blijkt nog behoorlijk lastig te zijn en je zou jezelf af kunnen vragen: “Is het al die inspanningen wel waard om een Breuk zo te maken dat je hem net zoals een **int** kan gebruiken?” Bedenk dan het volgende: het maken van de class Breuk is dan wel een hele inspanning maar iedere programmeur kan vervolgens dit zelfgedefinieerde type, als hij of zij de naam Breuk maar kent, als vanzelf (intuïtief) gebruiken. Er is daarbij geen handleiding of helpfunctie nodig, omdat het type Breuk op dezelfde wijze als het ingebouwde type **int** te gebruiken is. De class Breuk hoeft maar één keer gemaakt te worden, maar kan talloze malen gebruikt worden.

Met een beetje geluk ben je als ontwerper van de class Breuk later zelf ook gebruiker van de class Breuk, zodat je zelf de vruchten van je inspanningen kunt plukken.

2.3 Voorbeeld class Breuk (tweede versie)

Dit voorbeeld is een eerste uitbreiding van de op pagina 51 gegeven class Breuk (eerste versie). In deze versie leer je:

- 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 presenteer nu eerst de complete broncode van het programma Breuk1.cpp waarin het type Breuk gedeclareerd, geïmplementeerd en gebruikt wordt. Meteen daarna ga ik, één voor één, in op de bovengenoemde punten.

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

// Classdeclaratie:

class Breuk {
public:
    // Classinterface. Vertelt:
    // - hoe je een object van deze class kunt maken;
    // - wat je aan een object van deze class kunt vragen;
    // - wat je met een object van de class kunt doen.
    // Constructors zie paragraaf 2.4.
    Breuk();
    Breuk(int t);
    Breuk(int t, int n);
    // Vraag memberfuncties zie pagina 65.
    int teller() const; // const memberfunctie zie paragraaf 2.9.
    int noemer() const;
    // Doe memberfuncties zie pagina 65.
```

```
    void plus(Breuk[43] b);
    void abs();
    // ...
    // Er zijn nog veel uitbreidingen mogelijk.
    // ...
private:
    // Classimplementatie. Vertelt:
    // - wat je nodig hebt om een object van de class te maken.
    // Dit is voor gebruikers van de class niet van belang.
    int boven;
    int onder;
    void normaliseer();
};

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

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

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

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

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

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

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

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

void Breuk::normaliseer() {
    assert(onder != 0)[44];
    if (onder < 0) {
        onder = -onder;
        boven = -boven;
    }
    int d {gcd(boven < 0 ? -boven : boven, onder)};
    boven /= d;
    onder /= d;
}

// Hoofdprogramma:

int main() {
    Breuk b1 {4};
    println("b1 {{4}}[45] = {}/{}", b1.teller(), b1.noemer());
    Breuk b2 {23, -5};
    println("b2 {{23, -5}} = {}/{}", b2.teller(), b2.noemer());
    Breuk b3 {b2}; // kan dit zomaar? Zie paragraaf 2.7.
    println("b3 {{b2}} = {}/{}", b3.teller(), b3.noemer());
    b3.abs();
    println("b3.abs() = {}/{}", b3.teller(), b3.noemer());
    b3 = b2; // kan dit zomaar? Zie paragraaf 2.8.
    println("b3 = b2 = {}/{}", b3.teller(), b3.noemer());
    b3.plus(5);
    println("b3.plus(5) = {}/{}", b3.teller(), b3.noemer());
}
```

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

[44] De standaard macro `assert` doet niets als de, als argument meegegeven, expressie **true** oplevert maar breekt het programma met een passende foutmelding af als dit niet zo is. Je kunt zogenoemde *assertions* gebruiken om tijdens de ontwikkeling van het programma te controleren of aan bepaalde voorwaarden (waarvan je 'zeker' weet dat ze geldig zijn) wordt voldaan. De header `cassert` moet geïnclude worden om deze macro te kunnen gebruiken. Zie paragraaf 6.1.

[45] In de format string van een `print` of `println` functie kun je `"{{"` of `"}}"` gebruiken om respectievelijk een `"{"` of `"}"` af te drukken.

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
```

2.4 Constructor Breuk

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

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

Als een Breuk bij het aanmaken niet geïnitialiseerd wordt, dan wordt de constructor zonder parameters aangeroepen. In de definitie van deze constructor worden dan de datavelden boven met 0 en onder met 1 geïnitialiseerd. Dit gebeurt in een zogenoemde *initialization list*. Na het prototype van de constructor volgt een : waarna de datavelden één voor één geïnitialiseerd worden door middel van een universele initialisatie, zie paragraaf 1.4.

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

In dit geval wordt het dataveld boven geïnitialiseerd met de waarde 0 en wordt het dataveld onder geïnitialiseerd met de waarde 1, verder is er geen code in de constructor opgenomen. Door in de constructor een print opdracht op te nemen zou je een melding op het scherm kunnen geven telkens als een Breuk aangemaakt wordt. Deze constructor wordt dus aangeroepen als je een Breuk aanmaakt zonder deze te initialiseren. Na het uitvoeren van de onderstaande code heeft de breuk b1 de waarde $^0/_1$.

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

De overige constructors worden gebruikt als je een Breuk bij het aanmaken met één of met twee integers initialiseert. Na afloop van de onderstaande code bevat het object b2 de waarde $^1/_2$ ($^3/_6$ 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);` is als volgt gedefinieerd:

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

Het is ook mogelijk om vanuit een constructor een andere constructor van dezelfde class aan te roepen in de initialization list. De constructor zonder parameters kan dus ook als volgt worden geïmplementeerd (Breuk1_constructor_calls_constructor.cpp):

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

Door het definiëren van constructors kun je er dus voor zorgen dat elk object bij het aanmaken geïnitialiseerd wordt. Fouten die voortkomen uit het gebruik van een niet geïnitialiseerde variabele worden hiermee voorkomen. Het is dus verstandig om voor elke class één of meer constructors te definiëren[46].

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

2.5 Initialization list van de constructor

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

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

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

[46] Door het slim gebruik van default argumenten, zie paragraaf 1.13, kun je het aantal constructors vaak beperken. Alle drie de constructors die ik voor de class `Breuk` gedefinieerd heb zouden door één constructor met default argumenten vervangen kunnen worden: `Breuk::Breuk(int t = 0, int n = 1);`.

Dus gebruik:

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

in plaats van:

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

2.6 Constructors en type conversies

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

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

Uit de uitvoer blijkt dat dit 'gewoon' werkt: `b3` wordt verhoogd met $^5/_1$. Op zich is dit vreemd want de memberfunctie `plus` is gedefinieerd met een `Breuk` als parameter en niet met een **`int`** als parameter. Als je de memberfunctie `plus` aanroept op een `Breuk` met een **`int`** als argument, gebeurt het volgende: eerst 'kijkt' de compiler of in de class `Breuk` de memberfunctie `plus(int)` gedefinieerd is. Als dit het geval is, wordt deze memberfunctie aangeroepen. Als dit niet het geval is, 'kijkt' de compiler of er in de class `Breuk` een memberfunctie is met een parameter van een ander type waarnaar het type **`int`** omgezet kan worden[47]. In dit geval 'ziet' de compiler de memberfunctie `Breuk::plus(Breuk)`. De compiler 'vraagt zich nu dus af' hoe een variabele van het type **`int`** omgezet kan worden naar het type `Breuk`. Of met andere woorden hoe een `Breuk` gemaakt kan worden en geïnitialiseerd kan worden met een **`int`**. Of met andere woorden of de constructor `Breuk(int)` bestaat. Deze constructor bestaat in dit geval. De compiler maakt nu een tijdelijke naamloze variabele aan van het type `Breuk` (door gebruik te maken van de constructor `Breuk(int)`) en geeft een kopietje van deze variabele door aan de memberfunctie `plus`. De memberfunctie `plus` telt de waarde van deze variabele op bij de receiver. Na afloop van de memberfunctie `plus` wordt de tijdelijke naamloze variabele weer vrijgegeven.

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

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`[48].

2.7 Default copy constructor

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

- een object geïnitialiseerd wordt met een object van dezelfde class;
- een object als argument wordt doorgegeven aan een functie;[49]
- een object als waarde wordt teruggegeven vanuit een functie.[50]

De compiler genereert, als de programmeur geen copy constructor definieert, zelf een *default copy constructor*. Deze default copy constructor kopieert elk deel waaruit de class bestaat vanuit het originele object naar het gekopieerde object (Engels: *memberwise copy*). Het is ook mogelijk om zelf een copy constructor te definiëren (wordt later behandeld, zie eventueel paragraaf 19.3) maar voor de class `Breuk` voldoet de door de compiler gedefinieerde copy constructor prima.

In het programma op pagina 57 wordt de copy constructor van de class `Breuk` als volgt gebruikt:

```
Breuk b3 {b2};
```

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

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

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

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

De default copy constructor maakt een zogenoemde diepe kopie (Engels: deep copy).[51] Dit wil zeggen dat, als een private variabele zelf ook weer uit onderdelen bestaat, dat deze onderdelen dan ook gekopieerd worden. Stel dat we de class Rechthoek definiëren met twee private variabelen van het type Breuk genaamd lengte en breedte.

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

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

Als er in de class Rechthoek geen enkele constructor gedefinieerd is dan wordt door de compiler een default constructor gedefinieerd, zie pagina 59. Deze default constructor roept de default constructor van de class Breuk aan om de private variabelen lengte en breedte te initialiseren. Als we dus een object van de class Rechthoek aanmaken dan zorgt de default constructor ervoor dat beide private variabelen de waarde $^0/_1$ krijgen.

2.8 Default assignment operator

Als geen assignment operator (=) gedefinieerd is, wordt door de compiler een *default assignment operator* aangemaakt. Deze default assignment operator roept voor elk dataveld de assignment operator van dit veld aan (Engels: *memberwise assignment*). Het is ook mogelijk om zelf een assignment operator te definiëren (wordt later behandeld, zie eventueel paragraaf 19.4) maar voor de class Breuk voldoet de door de compiler gedefinieerde assignment operator prima.

In het programma op pagina 57 wordt de assignment operator van de class Breuk als volgt gebruikt:

```
    b3 = b2;
```

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

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

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

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

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

2.9 const memberfuncties

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

```
Breuk b {1, 3};            // variabele b met waarde 1/3
const Breuk halve {1, 2}; // constante halve met waarde 1/2
```

Een **const** Breuk mag je (vanzelfsprekend) niet veranderen.

```
    halve = b;
//  error: passing 'const Breuk' as 'this' argument discards ↩
 ↪ qualifiers[52]
```

Stel jezelf nu eens de vraag welke memberfuncties je aan mag roepen op het object halve[53]. De memberfuncties teller en noemer kunnen zonder problemen worden aangeroepen op een read-only breuk omdat ze de waarde van de breuk niet veranderen. De memberfuncties plus en abs mogen echter niet op het object halve worden aangeroepen omdat deze memberfuncties dit object zouden (kunnen) wijzigen en een read-only object mag je vanzelfsprekend niet veranderen. De compiler kan niet (altijd[54]) controleren of het aanroepen van een memberfunctie een verandering in het object tot gevolg heeft. Daarom mag je

[52] Deze foutmelding is niet erg helder. De **this**-pointer wordt behandeld in paragraaf 2.13.

[53] Zie pagina 55 voor de declaratie van de class Breuk.

[54] De compiler kan dit zeker niet als de classdefinitie separaat van de applicatie gecompileerd wordt. Zie paragraaf 2.21.

een memberfunctie alleen aanroepen voor een **const** object als je expliciet aanduidt dat deze memberfunctie het object niet verandert. Dit doe je door het keyword **const** achter de memberfunctie te plaatsten. Bijvoorbeeld:

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

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

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

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

```
    const Breuk halve {1, 2}; // constante halve met waarde 1/2
    println({}, halve.teller();
```

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

```
    halve.plus(b);
//  error: passing 'const Breuk' as 'this' argument discards ↩
 ↪ qualifiers[55]
```

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: assignment of member 'Breuk::boven' in read-only object
```

Door het toepassen van function name overloading is het mogelijk om 2 functies te definiëren met dezelfde naam en met dezelfde parameters waarbij de ene **const** is en de andere niet, maar dit is erg gedetailleerd en behandel ik hier niet[56].

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

[55] Deze foutmelding is niet erg helder. De **this**-pointer wordt behandeld in paragraaf 2.13.

[56] Voor degene die echt alles wil weten: als een memberfunctie zowel const als non-const gedefinieerd is dan wordt bij aanroep op een read-only object de const memberfunctie aangeroepen en wordt bij aanroep op een variabel object de non-const memberfunctie aangeroepen. (Dat is ook wel logisch, nietwaar?)

- **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 grootste gemene deler van de teller en noemer 1 is[57]. Door de noemer altijd > 0 te maken kan het teken van de `Breuk` eenvoudig bepaald worden (= teken van teller). Door de `Breuk` altijd te normaliseren wordt onnodige overflow van de datavelden `boven` en `onder` voorkomen. Een voorwaarde waaraan elk object van een class voldoet wordt een *class invariant* genoemd. Als je ervoor zorgt dat de class invariant aan het einde van elke public memberfunctie geldig is en als alle datavelden private zijn, dan weet je zeker dat voor elk object van de class `Breuk` deze invariant altijd geldig is.[58] Dit vermindert de kans op het maken van fouten.

2.11 Voorbeeld class `Breuk` (derde versie)

Dit voorbeeld is een uitbreiding van de in paragraaf 2.3 gegeven class `Breuk` (tweede versie). Met behulp van dit voorbeeld leer je, 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 presenteer nu eerst de complete broncode

[57] Dit wordt gedaan door de teller en noemer te delen door de grootste gemene deler van teller en noemer. Deze waarde wordt berekend met behulp van de standaardfunctie gcd. Deze functie is beschikbaar sinds C++17. Als je nog gebruik maakt van een oudere versie van C++, dan moet je de functie gcd zelf definiëren, zie `Breuk1_met_eigen_gcd.cpp`.

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

van het programma Breuk2.cpp waarin het type Breuk gedeclareerd, geïmplementeerd en gebruikt wordt. Meteen daarna ga ik op het bovengenoemde punt in.

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

// Classdeclaratie.

class Breuk {
public:
    Breuk(int t, int n);
    int teller() const;
    int noemer() const;
    void operator+=(Breuk[59] right);
private:
    int boven;
    int onder;
    void normaliseer();
};

//  ...

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

int main() {
    Breuk b1 {14, 4};
    println("b1 {{14, 4}} = {}/{}", b1.teller(), b1.noemer());
    Breuk b2 {23, -5};
    println("b2 {{23, -5}} = {}/{}", b2.teller(), b2.noemer());
    b1 += b2;
    println("b1 += b2 = {}/{}", b1.teller(), b1.noemer());
}
```

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

Uitvoer:

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

2.12 Operator overloading

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

```
b1 += b2;
```

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

```
b1 += b2;
//  error: no match for 'operator+=' (operand types are 'Breuk' and ↩
 ↪ 'Breuk')
```

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

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

De memberfunctie **operator**+= in deze derde versie van Breuk heeft dezelfde implementatie als de memberfunctie plus in de tweede versie van Breuk (zie pagina 52).

Je kunt voor een zelfgedefinieerd type alle operatoren zelf definiëren behalve de operator . waarmee een member geselecteerd wordt en de operator ?:[60]. Dus zelfs operatoren zoals [] (array index), -> (pointer dereferentie naar member) en () (functie aanroep) kun je zelf definiëren! Je kunt de prioriteit van operatoren niet zelf definiëren. Dit betekent dat a + b * c altijd wordt geïnterpreteerd als a + (b * c). Ook de associativiteit van de operatoren met gelijke prioriteit kun je niet zelf definiëren. Dit betekent dat a + b + c altijd wordt geïnterpreteerd als (a + b) + c. Ook is het niet mogelijk om de operatoren die al gedefinieerd zijn voor de ingebouwde types zelf te (her)definiëren. Het zelf definiëren van operatoren die in de taal C++ nog niet bestaan bijvoorbeeld @ of ** is niet mogelijk. Bij het definiëren van operatoren voor zelfgedefinieerde types moet je er natuurlijk wel voor

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

zorgen dat het gebruik voor de hand liggend is. De compiler heeft er geen enkele moeite mee als je bijvoorbeeld de memberfunctie `Breuk::operator+=` definieert die de als argument meegegeven `Breuk` van de receiver aftrekt. De programmeurs die de class `Breuk` gebruiken zullen dit minder kunnen waarderen.

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

```
    a += b += c;
//  error: no match for 'operator+=' (operand types are 'Breuk' and ↩
 ↪ 'void')
```

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

De definitie is dan als volgt:

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

Ik bespreek nu eerst hoe je de receiver (`DIT_OBJECT`) kunt benaderen en daarna kom ik weer op de definitie van de **`operator`**+= terug.

[61] Even verderop bespreek ik 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 impliciete parameter genaamd **this** die het adres bevat van het object waarop de memberfunctie wordt uitgevoerd. Met deze pointer kun je bijvoorbeeld het object waarop een memberfunctie wordt uitgevoerd als return waarde van deze memberfunctie teruggeven. Bijvoorbeeld:

```
Breuk Breuk::operator+=(Breuk right) { ↩
 ↪ // Let op: deze code is niet correct!;
// Zie paragraaf 2.20 voor een correcte implementatie van de ↩
 ↪ operator+=.
    boven = boven * right.onder + onder * right.boven;
    onder = onder * right.onder;
    return *this;
}
```

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

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

Toch is het (nog steeds) niet correct. In de bovenstaande `Breuk::`**operator**+= memberfunctie wordt bij het uitvoeren van het **return**-statement een kopie van het huidige object 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

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

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

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

- Als je de variabele waar een pointer `p` naar wijst wilt benaderen, moet je de notatie `*p` gebruiken maar als je de variabele waar een reference `r` naar verwijst wilt benaderen kun je gewoon `r` gebruiken.
- Een pointer die naar de variabele `v` wijst kun je later naar een andere variabele laten wijzen maar een reference die naar de variabele `v` verwijst kun je later niet naar een andere variabele laten verwijzen.
- Een pointer kan ook naar geen enkele variabele wijzen (de waarde is dan gelijk aan **`nullptr`**[62]) maar een reference verwijst altijd naar een variabele.

2.15 Reference parameters

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

[62] In de programmeertaal C gebruikte je `NULL` als waarde voor een pointer die nergens naar wees. Deze constante is in een aantal include file gedefinieerd, onder andere in `<stdio.h>`. In C++ wordt de waarde **`nullptr`** gebruikt, die in de taal zelf is gedefinieerd In de C23 standaard is de waarde **`nullptr`** ook gedefinieerd.

een functieaanroep wordt dus de *waarde* van de actuele parameter naar de formele parameter gekopieerd[63]. Het is dus geen probleem als de actuele parameter een constante is.

```
void sla_regels_over(int aantal) {
    while (aantal > 0) {
        println();
        --aantal;
    }
}
//  ...
    int n {7};
    sla_regels_over(n); // waarde van n wordt gekopieerd naar aantal
    println("n = {}", n); // uitvoer: n = 7
    sla_regels_over(3); // waarde 3 wordt gekopieerd naar aantal
```

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

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

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

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

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

```
//  ...
    int i {3};
    int j {4};
    swap_ints(i, j);
```

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

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

```
    swap_ints(i, 5);
//  error: cannot bind non-const lvalue reference of type 'int&' to ↩
 ↪ an rvalue of type 'int'
```

Zie param.cpp voor een compleet voorbeeldprogramma.

2.16 const reference parameters

Er is nog een andere reden om een reference parameter in plaats van een normale value parameter te gebruiken. Bij een reference parameter wordt zoals we hebben gezien een adres (een vast aantal bytes[64]) gekopieerd terwijl bij een value parameter de waarde (een variabel aantal bytes, afhankelijk van het type) gekopieerd wordt. Als de waarde veel geheugenruimte in beslag neemt, is het om performance redenen beter om een reference parameter te gebruiken.[65] Om er voor te zorgen dat deze functie toch aangeroepen kan worden met een constante kan deze parameter dan het beste als **const** reference parameter gedefinieerd worden. Het wordt dan ook onmogelijk om in de functie een waarde aan de formele parameter toe te kennen.

```
void drukaf(Tijdsduur td) {         // kopieer een Tijdsduur
    // ...
}
void drukaf(const Tijdsduur& td) { // kopieer een adres maar
```

[64] Op een MSP430-microcontroller bestaat een adres uit 2 bytes, op een CC3220-microcontroller uit 4 bytes en op een 64-bit pc uit 8 bytes.

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

```
    // ...                          // voorkom toekenningen aan
}                                   // de inhoud van dit adres
```

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

```
void drukaf(const Tijdsduur& td) {
    td.uur = 23; // drukaf probeert vals te spelen
//  error: assignment of member 'Tijdsduur::uur' in read-only object
```

In alle voorafgaande code waarin een Breuk als parameter is gebruikt kan dus beter[66] een **const** Breuk& als parameter worden gebruikt (Bijvoorbeeld op pagina 51, 56 en 66.)

2.17 Parameter FAQ[67]

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

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

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

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

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

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

[66] Een Breuk bestaat uit 2 **int**s. Op een AVR-microcontroller bestaat elke **int** uit 2 bytes en een Breuk dus uit $2 \times 2 = 4$ bytes. Een adres bestaat op een MSP430 uit 2 bytes en kan dus sneller gekopieerd worden dan een Breuk. Op een CC3220-microcontroller bestaat een **int** uit 4 bytes en een Breuk dus uit $2 \times 4 = 8$ bytes. Een adres bestaat op een CC3220-microcontroller uit 4 bytes en kan dus sneller gekopieerd worden dan een Breuk. Op een 64-bit pc bestaat een **int** uit 4 bytes en een Breuk dus uit $2 \times 4 = 8$ bytes. Een adres bestaat op een 64-bit pc uit 8 bytes en kan dus net zo snel gekopieerd worden als een Breuk.

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

2.18 Reference in range-based for

In paragraaf 1.7 heb je kennis gemaakt met de zogenoemde *range-based for*. Deze **for**-lus is speciaal bedoeld om de elementen van een array[68] één voor één te benaderen.

De in paragraaf 1.7 gepresenteerde code laat zien hoe je een array element voor element kunt uitlezen:

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

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

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

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

De reference `element` is nu één voor één een alias voor een element uit de array genaamd `rij`. Zie `ref_range-based_for.cpp` voor een compleet voorbeeldprogramma.

2.19 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, als het ware, ingevuld op de plaats waar de functie aangeroepen is. Door nu een reference als return type te gebruiken kun je een alias in plaats van een waarde teruggeven. Deze alias kun je dan gebruiken om een waarde aan toe te kennen. Zie voorbeeldprogramma `ref_ret.cpp`:

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

[68] De standaard C++ library bevat nog vele andere datastructuren die ook met een range-based **for** doorlopen kunnen worden zie paragraaf 3.6 en paragraaf 3.7.

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

Als je probeert om een reference naar een lokale variabele terug te geven, geeft de compiler de volgende foutmelding: `error: cannot bind non-const lvalue reference of type 'int&' to an rvalue of type 'int'`. De compiler geeft deze foutmelding omdat een lokale variabele alleen binnen de functie bestaat. Je kunt dus geen reference teruggeven naar een lokale variabele omdat deze variabele na afloop van de functie niet meer bestaat. De referentie zou dan verwijzen naar een niet meer bestaande variabele. Dit wordt een *dangling reference* genoemd.

```
int& som(int i1, int i2) {
    int s {i1 + i2};
    return s; // Error!
}
```

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

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

De compiler geeft in dit geval geen foutmelding, maar allen de waarschuwing `warning: address of local variable 's' returned`. Deze fout (het zogenoemde *dangling pointer 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[69]. De fout in de definitie van de functie `som` komt pas aan het licht als we de functie bijvoorbeeld als volgt aanroepen:

```
    int* p1 {som2(1, 2)};
    int* p2 {som2(3, 4)};
    println("1 + 2 = {}", *p1);
    println("3 + 4 = {}", *p2);
```

[69] De GCC C++-compiler geeft de warning: `reference to local variable 's' returned`.

De uitvoer van dit programma is:

```
Segmentation fault
```

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

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

Zie ref_ret.cpp voor een compleet voorbeeldprogramma.

2.20 Reference return type (deel 2)

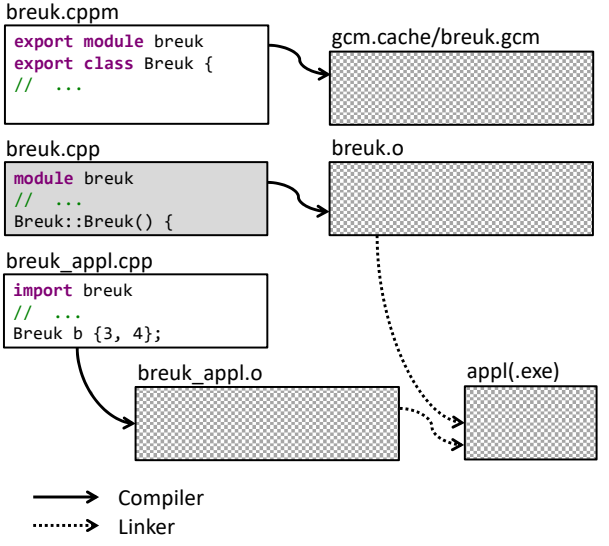
In de Breuk::**operator**+= memberfunctie die gegeven is op pagina 68 wordt bij het uitvoeren van het **return**-statement een kopie van het huidige object teruggegeven. Dit is echter niet correct. Want als we deze operator als volgt gebruiken: `(a += b) += c;` dan wordt eerst `a += b` uitgerekend en een kopie van a teruggegeven, c wordt dan bij deze kopie van a opgeteld waarna de kopie wordt verwijderd. Dit is natuurlijk niet de bedoeling, het is de bedoeling dat c bij a wordt opgeteld. We kunnen dit probleem oplossen door een reference naar het object zelf terug te geven. Hiermee zorgen we er meteen voor dat de expressie `(a += b) += c` gewoon goed (zoals bij integers) werkt. De **operator**+= memberfunctie kan dan als volgt gedeclareerd worden:

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

Behalve de operator += kun je ook andere operatoren overloaden. Deze kennis heb je echter niet nodig om de rest van dit dictaat te begrijpen. Hoe je andere operatoren kunt overloaden bij het type Breuk kun je vinden in hoofdstuk 16.

2.21 Voorbeeld module breuk

Je kunt de *declaratie* en *definitie* (implementatie) van een class splitsen in een `.cppm` (module interface) en een `.cpp` (module implementatie) file, paragraaf 15.8. Dit heeft als voordeel dat je de implementatie afzonderlijk kunt compileren tot een zogenoemde object file[70]. De gebruiker van de class kan dan de module importeren in zijn eigen programma en vervolgens de object file met zijn eigen programma mee 'linken'. Zie figuur 2.1. De gebruiker hoeft dan dus niet te beschikken over de implementatie.



Figuur 2.1: Afzonderlijk compileren van de module breuk.

Om de voorbeeldcode compact te houden wordt hieronder uitgelegd hoe de zeer eenvoudige en volledig nutteloze class `Memory_cell` opgesplitst kan worden in een module interface en een module implementatie. In een object van de class `Memory_cell` kun je een waarde van het type **`int`** opslaan en deze waarde later weer opvragen[71].

Voor de class `Memory_cell` zien de files `memory_cell.cppm`, `memory_cell.cpp` en `memory_appl.cpp` er als volgt uit:

```
// Dit is file Memcell.cppm
export module memcell;
```

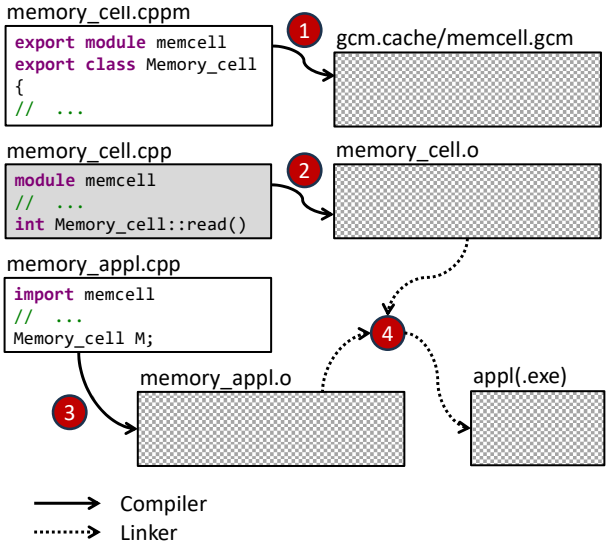
[70] Een object file bevat machine code en heeft afhankelijk van de gebruikte compiler de extensie `.o` of `.obj`

[71] Dat kan natuurlijk ook gewoon in een variabele van het type **`int`** maar het gaat hier alleen om een voorbeeld van het gebruik van een zelfgedefinieerde module.

```
export class Memory_cell {
public:
    int read() const;
    void write(int x);
private:
    int stored_value;
};

// Dit is file memory_cell.cpp
module memcell;

int Memory_cell::read() const {
    return stored_value;
}

void Memory_cell::write(int x) {
    stored_value = x;
}

// Dit is file memory_appl.cpp
import std;
import memcell;

int main() {
    Memory_cell m;
    m.write(5);
    std::println("m = {}", m.read());
//  ...
}
```

De verschillende commando's voor de compiler en linker zoals aangegeven in figuur 2.2 zijn:

1. `g++ -std=c++23 -fmodules -fmodule-only -c memory_cell.cppm`
2. `g++ -std=c++23 -fmodules -c memory_cell.cpp`
3. `g++ -std=c++23 -fmodules -c memory_appl.cpp`
4. `g++ -std=c++23 -fmodules memory_cell.o memory_appl.o -o appl`



Figuur 2.2: Afzonderlijk compileren van de module `memory_cell`.

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 Functietemplates

Op pagina 71 heb ik de functie `swap_ints` besproken waarmee twee **`int`** variabelen verwisseld kunnen worden:

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

Als je twee **`double`** variabelen wilt verwisselen, kun je deze functie niet rechtstreeks gebruiken. Waarschijnlijk ben je op dit moment gewend om de functie `swap_ints` 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, moet je opnieuw knippen, plakken, zoeken en vervangen.
- Bij een wat ingewikkelder algoritme, bijvoorbeeld sorteren, is het niet altijd duidelijk welke **int** je wel en welke **int** je niet moet vervangen in **double** als je in plaats van een array met elementen van het type **int** een array met elementen van het type **double** wilt sorteren. Hierdoor kunnen in een goed getest algoritme toch weer fouten opduiken.
- Als er zich in het algoritme een logische fout bevindt of als je het algoritme wilt vervangen door een efficiëntere versie, dan moet je de benodigde wijzigingen in elke gekopieerde versie aanbrengen.

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

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

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

Na het keyword **template** volgt een lijst van template-parameters tussen < en >. Een template-parameter is meestal een type[72]. Dit wordt aangegeven door het keyword **typename**[73] gevolgd door een naam voor de parameter. Ik heb hier de naam `T` gebruikt maar ik had net zo goed de naam `Vul_maar_in` kunnen gebruiken. De template-parameter moet[74] in de parameterlijst

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

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

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

van de functie gebruikt worden. De definitie van deze functietemplate genereert nog geen enkele machinecode instructie. Het is alleen een 'mal' waarmee (automatisch) functies aangemaakt kunnen worden.

Als je nu de functie swap aanroept, genereert de compiler zelf afhankelijk van het type van de gebruikte argumenten de benodigde 'versie' van swap[75] door voor het template-parameter (in dit geval T) het betreffende type in te vullen.

Dus de aanroep:

```
int n {2};
int m {3};
swap(n, m);
```
[76]

heeft tot gevolg dat de volgende functie gegenereerd wordt[77]:

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

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

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

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

[76] In het bestand swap_template.cpp zie je dat hier ::swap(n, m) staat. In dit bestand is namelijk de regel **using namespace** std; opgenomen. De extra :: is dan nodig, omdat de compiler anders niet kan kiezen tussen de door ons gedefinieerde functietemplate swap en de in de standaard gedefinieerde functietemplate std::swap. Als alternatief kun je ook de regel **using namespace** std; niet gebruiken, zie swap_template_zonder_using_std.cpp.

[77] Als je zelf deze functie al gedefinieerd hebt, genereert de compiler geen nieuwe functie, maar wordt de al gedefinieerde functie gebruikt.

dan heeft dit tot gevolg dat een tweede functie swap gegenereerd wordt[78]:

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

Het gebruik van een functietemplate heeft de volgende voordelen:

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

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

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

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

De functietemplate swap is hier als voorbeeld zelf gedefinieerd, maar in de praktijk is dit niet nodig omdat deze functietemplate al in de standaard C++-library is opgenomen.

[78] Hier blijkt duidelijk het belang van function name overloading (zie paragraaf 1.12).

3.2 Class templates

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

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

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

In een `Dozijn` kunnen dus twaalf integers worden opgeslagen in de private array genaamd `data`. De programmeur die een object van de class `Dozijn` gebruikt kan een integer in dit object 'zetten' door de boodschap `zet_in` 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.zet_in(3,13);
```

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

```
    println("De plaats nummer 3 in d bevat de waarde: {}", ↩
↪ d.lees_uit(3));
```

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

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

int Dozijn::lees_uit(int index) const {
    if (index >= 0 && index < 12)
        return data[index];
    return 0; /* ik weet niets beters[79] */
}
```

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

Om de inhoud van een object van de class `Dozijn` eenvoudig om te kunnen zetten naar een `string` is de functie `naar_string` gedefinieerd:

```
string naar_string(const Dozijn& d) {
    string s {format("{}", d.lees_uit(0))};
    for (int i {1}; i < 12; ++i)
        s += format(", {}", d.lees_uit(i));
    return s;
}
```

In deze functie wordt gebruik gemaakt van de standaard functie `format` die op dezelfde manier werkt als de standaard functie `print` alleen wordt het resultaat niet weergegeven op het scherm, maar opgeslagen in een string.

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

```
int main() {
    Dozijn kwadraten;
    for (int j {0}; j < 12; ++j)
        kwadraten.zet_in(j, j * j);
    println("kwadraten = {}", naar_string(kwadraten));
}
```

Dit programma geeft de volgende uitvoer:

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

[79] Het is in C++ mogelijk om een functie te verlaten zonder dat een returnwaarde wordt teruggegeven door een zogenoemde exception te gooien. Exceptions worden in hoofdstuk 6 behandeld.

In objecten van de class Dozijn kunnen twaalf elementen van het type **int** worden opgeslagen. Als je een Dozijn met elementen van het type **double** nodig hebt, kun je natuurlijk gaan kopiëren, plakken, zoeken en vervangen maar daar zitten weer de in de vorige paragraaf besproken nadelen aan. Als je verschillende versies van Dozijn 'met de hand' genereert, moet je bovendien elke versie een andere naam geven omdat een class naam uniek moet zijn. In plaats daarvan kun je ook het template mechanisme gebruiken om een Dozijn met elementen van het type T te definiëren, waarbij het type T pas bij het gebruik van de class template Dozijn wordt bepaald. Bij het gebruik van de class template Dozijn kan de compiler niet zelf bepalen wat het type T moet zijn. Vandaar dat je dit bij het gebruik van de class template Dozijn zelf moet specificeren. Bijvoorbeeld:

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

3.3 Voorbeeld class template Dozijn

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

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

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

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

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

De memberfunctie `lees_uit` geeft nu de waarde van plaats `0` terug als de `index <= 0` is en geeft de waarde van plaats 11 terug als de `index >= 11` is. Je kunt in dit geval namelijk niet de waarde `0` teruggeven zoals we bij het `Dozijn` met integers (op pagina 85) 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 om te kunnen zetten naar een `string` is de template functie `naar_string` gedefinieerd:

```
template<typename T> string naar_string(const Dozijn<T>& d) {
    string s {format("{}", d.lees_uit(0))};
    for (int i {1}; i < 12; ++i)
        s += format(", {}", d.lees_uit(i));
    return s;
}
```

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

```
int main() {
    Dozijn<int> kwadraten;
    for (int j {0}; j < 12; ++j)
        kwadraten.zet_in(j, j * j);
    println("kwadraten = {}", naar_string(kwadraten));
    Dozijn<string> provincies;
    provincies.zet_in(0, "Drenthe");
    provincies.zet_in(1, "Flevoland");
    provincies.zet_in(2, "Friesland");
    provincies.zet_in(3, "Gelderland");
    provincies.zet_in(4, "Groningen");
    provincies.zet_in(5, "Limburg");
    provincies.zet_in(6, "Noord-Brabant");
    provincies.zet_in(7, "Noord-Holland");
    provincies.zet_in(8, "Overijssel");
    provincies.zet_in(9, "Utrecht");
    provincies.zet_in(10, "Zeeland");
```

```
    provincies.zet_in(11, "Zuid-Holland");
    println("provincies = {}", naar_string(provincies));
}
```

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

De uitvoer van het bovenstaande programma is:

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

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

3.4 Voorbeeld class template `Rij`

In de, in de vorige paragraaf beschreven, class template `Dozijn` kunnen we twaalf elementen van een type naar keuze opslaan. Als je meer dan twaalf elementen wilt opslaan, kun je deze template niet gebruiken. Je zou nu bijvoorbeeld een class template `Gros` kunnen definiëren waar honderdvierenveertig elementen in passen door de class template `Dozijn` te kopiëren en overal de waarde 12 te vervangen door de waarde 144. Het kopiëren van code is echter, zoals je weet, slecht voor de onderhoudbaarheid. Je kunt beter een class template `Rij` definiëren waarbij je het gewenste aantal elementen als (tweede) template-parameter kunnen opgeven.

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

```
template<typename T, size_t N> class Rij {
public:
    void zet_in(size_t index, const T& waarde);
    const T& lees_uit(size_t index) const;
    constexpr size_t aantal_plaatsen();
private:
    T data[N];
};
```

Je ziet dat het standaard type `size_t` gebruikt is als type van de tweede template-parameter `N`. Het type `size_t` wordt in C en C++ gebruikt om de grootte van een object mee aan te

geven en een variabele van dit type kan alleen positief zijn. Dit is een logische keuze omdat het aantal elementen in een `Rij` ook alleen maar positief kan zijn. Het type `size_t` is ook gebruikt als type van de parameter `index` van de memberfuncties `zet_in` en `lees_uit`. Deze `index` kan ook alleen maar positief zijn.

De functie `aantal_plaatsen` is een zogenoemde compile time functie, die zoals de naam al zegt ook tijdens het compileren van het programma kan worden uitgevoerd, zie paragraaf 15.12. De returnwaarde van deze functie is dus te gebruiken op plaatsen waar een compile time constante vereist is, zie paragraaf 1.8. Je kunt bijvoorbeeld een rij definiëren die evenveel plaatsen bevat als een al eerder gedefinieerde rij[80]:

```
    Rij<int, 10> kwad;
    Rij<double, kwad.aantal_plaatsen()> wortels;
```

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

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

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

template<typename T, size_t N> constexpr size_t Rij<T, ↩
 ↪ N>::aantal_plaatsen() {
    return N;
}
```

Om de inhoud van een `Rij` eenvoudig om te kunnen zetten naar een `string` is de template functie `naar_string` gedefinieerd:

```
template<typename T, size_t N> string naar_string(const Rij<T, N>& ↩
 ↪ r) {
```

[80] Het argument dat wordt meegegeven aan een non-type template-parameter moet een compile time constante zijn, zie https://en.cppreference.com/w/cpp/language/constant_expression.

```
    string s {format("{}", r.lees_uit(0))};
    for (size_t i {1}; i < N; ++i)
        s += format(", {}", r.lees_uit(i));
    return s;
}
```

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

```
int main() {
    Rij<int, 10> kwad;
    for (size_t i {0}; i < kwad.aantal_plaatsen(); ++i)
        kwad.zet_in(i, i * i);
    println("kwad = {}", naar_string(kwad));

    Rij<char, 26> alfabet;
    for (size_t i {0}; i < alfabet.aantal_plaatsen(); ++i)
        alfabet.zet_in(i, 'A' + i);
    println("alfabet = {}", naar_string(alfabet));
}
```

De uitvoer van het bovenstaande programma is:

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

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

3.5 Standaard templates

In de C++-standaard [8] is een groot aantal standaard templates voor datastructuren en algoritmen opgenomen. In hoofdstuk 10 komen we hier nog uitgebreid op terug. De basis voor deze verzameling templates is in de jaren ’90 gelegd door Alex Stepanov en Meng Lee in de zogenoemde STL (Standard Template Library). In de standaard library zijn ook een generiek type `array` en een generiek type `vector` opgenomen.

3.6 std::array

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

- Een `array` is in tegenstelling tot een C-array een echt object.
- Je kunt een `array` gewoon vergelijken, toekennen en kopiëren.
- Het aantal elementen van een `array` moet bij het compileren bekend zijn (net zoals bij een C-array) en kan (anders dan bij een C-array) ook opgevraagd worden[81]. Met andere woorden: een `array` weet zelf hoe groot hij is.
- Een `array` heeft memberfuncties:
  - `size()`: geef het aantal elementen in de `array`[82].
  - `at(...)`: geef element op de opgegeven positie. Bijna gelijk aan `operator[]` maar `at` controleert of de index geldig is en gooit een exception[83] als dit niet zo is.
  - `...`[84]

De elementen van een `array` kunnen, net zoals de elementen van een C-array, één voor één gelezen of bewerkt worden met behulp van een range-based **`for`** (zie paragraaf 1.7 en paragraaf 2.18).

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

```
import std;
using namespace std;

int main() {
    // definieer array van 15 integers
    array<int, 15> a;
    // vul met kwadraten
    int i {0};
```

[81] Bij een C-array genaamd a kan het aantal elementen bepaald worden met de enigszins cryptische code: **`sizeof`** `a / ` **`sizeof`** `a[0]`.

[82] Niet te verwarren met de **`sizeof`** operator die het aantal bytes bepaalt dat een variabele of type in beslag neemt.

[83] Een exception is een manier om een fout vanuit een component door te geven aan de code die deze component gebruikt. Exceptions worden in hoofdstuk 6 behandeld.

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

```
    for (auto& e: a) {
        e = i * i;
        ++i;
    }
    // druk af
    println("{}", a);

    // kopiëren van de een array
    auto b {a};
    println("{}", b);
    // vergelijken van arrays
    if (a != b) {
        println("Dit kan niet waar zijn!");
    }
    try {
        a.at(100) = 12;
        // ongeldige index ==> foutmelding (exception)
    } catch(const out_of_range& e) {
        println(cerr, "Error: {}", e.what());
    }
    a[i] = 12;
    // ongeldige index ==> Abort in debug mode, ongedefinieerd ↩
 ↪ gedrag in release mode
}
```

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]
Error: array::at: __n (which is 100) >= _Nm (which is 15)
Assertion '__n < this->size()' failed.
Aborted
```

3.7 std::vector

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

- Een `vector` is in tegenstelling tot een C-array een echt object.
- Een `vector` kan op twee manieren geïnitialiseerd worden:
  - `vector<int> v1 {10}` is een vector met één integer element met de waarde 10.
  - `vector<int> v1(10)` is een vector met tien integer elementen met de waarde 0.
- Je kunt een `vector` gewoon vergelijken, toekennen en kopiëren.
- Een `vector` kan groeien en krimpen.
- Een `vector` heeft memberfuncties:
  - `size()`: geef het aantal elementen in de vector.
  - `at(...)`: geef element op de opgegeven positie. Bijna gelijk aan `operator[]` maar `at` controleert of de index geldig is en gooit een exception[85] 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 zodra het aantal elementen groter wordt dan de capaciteit.
  - `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`.
  - ...[86]

De elementen van een `vector` kunnen, net zoals de elementen van een C-array, één voor één gelezen of bewerkt worden met behulp van een range-based `for` (zie paragraaf 1.7 en paragraaf 2.18).

Voorbeeld van een programma met een `vector` (`std_vector1.cpp`):

```
import std;
using namespace std;

int main() {
    // definieer vector van integers
    vector<int> v;
    // vul met kwadraten
```

[85] Een exception is een manier om een fout vanuit een component door te geven aan de code die deze component gebruikt. Exceptions worden in hoofdstuk 6 behandeld.

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

```
    for (int i {0}; i < 15; ++i) {
        v.push_back(i * i);
    }
    // druk af
    println("{}", v);

    // kopiëren van een vector
    auto w {v};
    println("{}", w);
    // vergelijken van vectoren
    if (v != w) {
        println("Dit kan niet waar zijn!");
    }
    try {
        v.at(100) = 12;
        // ongeldige index ==> foutmelding (exception)
    } catch(const out_of_range& e) {
        println(cerr, "Error: {}", e.what());
    }
    v[100] = 12;
    // ongeldige index  ==> Abort in debug mode, ongedefinieerd ↩
 ↪ gedrag in release mode
}
```

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]
Error: vector::_M_range_check: __n (which is 100) >= this->size() ↩
 ↪ (which is 15)
Assertion '__n < this->size()' failed.
Aborted
```

Voorbeeldprogramma om een onbekend aantal elementen in een `vector` in te lezen (std_vector2.cpp):

```
import std;
using namespace std;

void lees_ints(vector<int>& vec) {
    // gooi huidige inhoud vec weg
    vec.resize(0);
    println("Voer een willekeurig aantal integers in,");
    println("sluit af met een tekst:");
    int i;
    while (cin >> i) {
        vec.push_back(i);
    }
    // zorg dat cin na de "onjuiste" invoer weer gebruikt kan worden
    cin.clear();
    cin.ignore(numeric_limits<streamsize>::max(), '\n');
}

int main() {
    // definieer vector
    vector<int> v;
    // vul deze vector
    lees_ints(v);
    // druk af
    println("{}", v);
    // vul deze vector opnieuw
    lees_ints(v);
    // druk af
    println("{}", v);
}
```

Mogelijke invoer[87] en uitvoer van dit programma:

```
Voer een willekeurig aantal integers in, sluit af met een tekst:
1 2 3 4 5 6 7 8 9 10 stop
[1 2 3 4 5 6 7 8 9 10]
Voer een willekeurig aantal integers in, sluit af met een tekst:
1 2 3 4 Hoedje van papier
```

[87] De invoer is groen en onderstreept weergegeven.

```
[1 2 3 4]
```

De naam `vector` is misschien niet zo handig gekozen omdat het niet overeenkomt met het wiskundige begrip *vector*. De standaard library bevat ook de template `valarray<T>`. Dit is een speciale container waarbij `T` een numeriek type moet zijn. Er kunnen rekenkundige bewerkingen op dit type uitgevoerd worden. Een `valarray` lijkt dus meer op een vector zoals we die kennen uit de wiskunde dan een `vector`. Zie `valarray.cpp`.

Deze rekenkundige bewerkingen kunnen efficiënt geïmplementeerd worden. Zo kan de bewerking: `v3 = 2 * v1 + v2` door de compiler vertaald worden in één loop waarin voor elke element van `v1`, `v2` en `v3` de volgende code wordt uitgevoerd: `v3[i] = 2 * v1[i] + v2[i];`.

3.8 Template Argument Deduction

Zoals je hebt gezien moet bij het definiëren van een `array` of `vector` het type van de elementen tussen < en > worden opgegeven. Bij het definiëren van een `array` moet bovendien het aantal elementen worden opgegeven. Als de `array` of de `vector` geïnitialiseerd wordt, dan kan de compiler zelf dit type (en aantal) bepalen. Dit wordt *Class Template Argument Deduction* (CTAD) genoemd. Let wel op dat een string literal van het type `const char*` is. Je kunt het suffix `s` gebruiken om een string literal van het type `string` te maken, bijvoorbeeld `"Hello"s` is van het type `string`.

Dus de definitie:

```
array a {1, 2, 3, 4};
vector v1 {3.14, 6.28};
vector v2 {"Hallo", "Wereld"};
vector v3 {"Hallo"s, "Wereld"s};
```

Is gelijk aan:

```
array<int, 4> a {1, 2, 3, 4};
vector<double> v1 {3.14, 6.28};
vector<const char*> v2 {"Hallo", "Wereld"};
vector<string> v3 {"Hallo"s, "Wereld"s};
```

Zie CTAD.cpp.

3.9 Alternatieve notatie met auto

In deze en volgende paragraaf worden verschillende varianten besproken van de functietemplate number_of_zero_bits die het aantal nullen in de binaire representatie van een gegeven geheel getal berekent. Het gaat daarbij niet om de implementatie van deze functie, maar om de verschillende manieren waarop een functietemplate gedefinieerd kan worden.

De eerste variant van deze functietemplate number_of_zero_bits1.cpp is als volgt gedefinieerd:

```
template<typename T>
int number_of_zero_bits(T n) {
    using unsigned_T = make_unsigned_t<T>;
    unsigned_T unsigned_n {static_cast<unsigned_T>(n)};
    int count {0};
    unsigned_T mask {1};
    for (size_t bit {0}; bit < sizeof(T) * 8; ++bit) {
        if ((unsigned_n & mask) == 0) {
            ++count;
        }
        mask <<= 1;
    }
    return count;
}
```

Je kunt deze functietemplate bijvoorbeeld als volgt gebruiken:

```
    auto n {numeric_limits<int8_t>::max()};
    println("Het int8_t getal {} heeft {} bit(s) met de waarde ↩
 ↪ 0.", n, number_of_zero_bits(n));
    auto m {numeric_limits<uint64_t>::min()};
    println("Het uint64_t getal {} heeft {} bit(s) met de waarde ↩
 ↪ 0.", m, number_of_zero_bits(m));
```

De uitvoer van dit programmafragment is:

```
Het int8_t getal 127 heeft 1 bit(s) met de waarde 0.
Het uint64_t getal 0 heeft 64 bit(s) met de waarde 0.
```

Met behulp van het keyword **auto** kun je deze functietemplate ook als volgt definiëren:

```
int number_of_zero_bits(auto n) {
    using T = decltype(n);
    using unsigned_T = make_unsigned_t<T>;
    unsigned_T unsigned_n {static_cast<unsigned_T>(n)};
    int count {0};
    unsigned_T mask {1};
    for (size_t bit {0}; bit < sizeof(T) * 8; ++bit) {
        if ((unsigned_n & mask) == 0) {
            ++count;
        }
        mask <<= 1;
    }
    return count;
}
```

Hoewel dat nu niet meteen duidelijk is, is deze functie `number_of_zero_bits` nog steeds een functietemplate, omdat de parameter `n` een **auto**-parameter is. De compiler zal namelijk nog steeds voor elke unieke type van `n` een aparte versie van de functie genereren. De implementatie blijft (bijna[88]) hetzelfde, maar de functietemplate is nu iets eenvoudiger gedefinieerd. Elke functie met één of meer **auto**-parameters is dus in feite een functietemplate in een verkorte notatie[89].

Je moet wel oppassen als je deze verkorte notatie gebruikt, omdat elke **auto**-parameter een aparte template-parameter is, dus als je meerdere **auto**-parameters hebt, dan kunnen deze allemaal verschillende types hebben. Op pagina 81 heb je de definitie van de functietemplate `swap` gezien:

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

[88] Het enige verschil is dat het parametertype van `n` nu moet worden achterhaald met behulp van **decltype**`(n)`, zie paragraaf 15.11.

[89] In de C++ Core Guideline CG: T.13. wordt geadviseerd om voor eenvoudige functietemplates met één parameter de verkorte notatie met **auto** te gebruiken.,

```
    q = t;
}
```

Het is nu *geen* goed idee om deze functietemplate in een verkorte notatie te definiëren (swap_auto.cpp):

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

Als deze functietemplate op de volgende wijze wordt gebruikt:

```
    float f {0.4}
    int i {4};
    swap(f, i);
    println("f = {}, i = {}", f, i);
```

Dan zal dit programma zonder fouten compileren en uitvoeren, maar de uitvoer zal waarschijnlijk niet zijn wat je verwacht:

```
f = 4, i = 0
```

Als je de oorspronkelijke functietemplate swap had gebruikt, dan zou dit programma bij het compileren de volgende foutmelding hebben gegeven:

```
error: no matching function for call to 'swap(int&, double&)'
```

Het is beter om een foutmelding te krijgen bij het compileren van het programma dan onverwacht gedrag bij het uitvoeren van het programma. Dus in dit geval is het beter om de oorspronkelijke functietemplate te gebruiken, waarbij beide parameters dezelfde template parameter hebben, zodat je alleen waarden van hetzelfde type kunt verwisselen, zie swap_template.cpp.

3.10 Concepts

In C++20 zijn zogenoemde *concepts* (concepten) geïntroduceerd. Een concept is een predicate template voor een type. Een predicate is een bewering die waar of onwaar is.

Er zijn een aantal standaard concepten gedefinieerd in de standaard library, zie https://en.cppreference.com/w/cpp/concepts.html.

Zo kan bijvoorbeeld het concept `integral<T>` worden gebruikt om te bepalen of het type `T` een integer getal kan bevatten.

De volgende code:

```
    println("integral<int> = {}", integral<int>);
    println("integral<int8_t> = {}", integral<int8_t>);
    println("integral<uint64_t> = {}", integral<uint64_t>);
    println("integral<double> = {}", integral<double>);
```

Geeft de volgende output:

```
integral<int> = true
integral<int8_t> = true
integral<uint64_t> = true
integral<double> = false
```

Een concept kan gebruikt worden om het type waarmee een template gebruikt kan worden te beperken (Engels: to constrain).

Als je de functietemplate `number_of_zero_bits`, zie pagina 97 bijvoorbeeld als volgt gebruikt:

```
    double d {3.14};
    println("number_of_zero_bits({}) = {}", d, ↩
 ↪ number_of_zero_bits(d));
```

Dan zal de compiler de volgende foutmelding geven:

```
error: invalid use of incomplete type 'class ↩
 ↪ std::__make_unsigned_selector<double, false, false>'
```

Dit is een vrij cryptische foutmelding, die aangeeft dat de compiler probeert een unsigned versie van het type **`double`** te maken, maar dat dit niet mogelijk is omdat het type **`double`** geen unsigned variant heeft. De compiler genereert namelijk met behulp van de functietemplate een functie door voor het type `T` **`double`** in te vullen, waarna de compiler deze functie probeert te compileren, met de bovenstaande foutmelding als gevolg.

Het zou veel beter zijn als je de compiler zou kunnen vertellen dat de functietemplate `number_of_zero_bits` alleen gebruikt mag worden met types die een integer getal kunnen bevatten.

Met behulp van een concept is dat sinds C++20 mogelijk. De eerste twee regels van de functietemplate number_of_zero_bits kunnen dan als volgt worden gedefinieerd (number_of_zero_bits3

```
template<typename T> requires integral<T>
int number_of_zero_bits(T n) {
```

Als deze functietemplate nu als volgt gebruikt:

```
    double d {3.14};
    println("number_of_zero_bits({}) = {}", d, ↩
↪ number_of_zero_bits(d));
```

Dan zal de compiler de volgende duidelijkere foutmelding geven:

```
error: no matching function for call to 'number_of_zero_bits(double&)'
note: constraints not satisfied
```

De compiler zal nu niet meer proberen om in de functietemplate number_of_zero_bits **double** in te vullen voor het type T, omdat het concept integral<**double**> niet waar (**false**) is.

Het gebruik van een concept zorgt ervoor dat de compiler betere foutmeldingen kan geven als een template wordt gebruikt met een type dat niet aan de eisen van het concept voldoet. Een template waarvan het gebruik beperkt is, wordt een "constrained template" genoemd[90].

Het beperken van een template met behulp van een concept kan op verschillende manieren worden gedaan. De eerste twee regels van de functietemplate number_of_zero_bits kunnen ook als volgt worden gedefinieerd (number_of_zero_bits4.cpp):

```
template<typename T>
int number_of_zero_bits(T n) requires integral<T> {
```

Je kunt de **requires**-voorwaarde dus ook achter de parameterlijst plaatsen.

Er is ook een derde manier om een template te beperken met behulp van een concept, namelijk door de naam van het concept in plaats van het keyword **typename** in de template parameterlijst te plaatsen. Er is dan helemaal geen **requires**-voorwaarde meer nodig. De eerste twee regel van de functietemplate number_of_zero_bits kunnen dan als volgt worden gedefinieerd (number_of_zero_bits5.cpp):

```
template<integral T>
```

[90] In de C++ Core Guideline CG: T.10. wordt geadviseerd om voor alle template parameters een concept te specificeren.

```
int number_of_zero_bits(T n) {
```

Ook als de functietemplate in de verkorte notatie is gedefinieerd, zie pagina 98, kun je een concept gebruiken om de template te beperken. Je plaatst dan de naam van het concept vóór het keyword **auto**.

De eerste regel van de verkorte notatie van de functietemplate `number_of_zero_bits` kan dan als volgt gedefinieerd worden (number_of_zero_bits6.cpp):

```
int number_of_zero_bits(integral auto n) {
```

De vier verschillende manieren om een template te beperken met behulp van een concept zijn functioneel allemaal gelijk.

Er zit nog een klein addertje onder het gras, het concept `integral<T>` is niet alleen **true** voor de standaard integer types, maar ook voor de standaard character types en voor het type **bool**. Het aantal bits met de waarde `0` in een character is wel te bepalen, maar als je probeert het aantal bits met de waarde `0` in een **bool** te bepalen, dan krijg je de volgende foutmelding van de compiler, omdat het type **bool** geen unsigned variant heeft:

```
error: invalid use of incomplete type 'struct std::make_unsigned<bool>'
```

Als je wilt voorkomen dat de functietemplate `number_of_zero_bits` gebruikt kan worden met het type **bool**, dan kun je dit als volgt doen (number_of_zero_bits7.cpp):

```
template<typename T> requires (integral<T> && !same_as<T, bool>)
int number_of_zero_bits(T n) {
```

Hierbij is gebruik gemaakt van het standaard concept `same_as<T, U>`, dat waar is als de types `T` en `U` hetzelfde zijn. Je ziet dat je achter het keyword **requires** meerdere voorwaarden kunt plaatsen, die met logische operatoren kunnen worden gecombineerd. De logische expressie moet wel tussen haakjes worden geplaatst.

Als je de verkorte notatie van de functietemplate `number_of_zero_bits` gebruikt, zie pagina 102 en je wilt voorkomen dat de template met het type **bool** wordt gebruikt, dan moet je zelf een **concept** definiëren (number_of_zero_bits8.cpp):

```
template <class T>
concept integer = integral<T> && !same_as<T, bool>;

int number_of_zero_bits(integer auto n) {
```

Je kunt concepten dus gebruiken om duidelijkere foutmeldingen te krijgen als een template wordt gebruikt met een type dat niet aan de eisen van het concept voldoet. Daarnaast kun je door concepten te gebruiken templates ook overloaden (concept-based overloading). De compiler kiest dan de template waarbij de meegegeven argumenten het beste aan de gedefinieerde voorwaarden (concepten) voldoen. In paragraaf 12.1 vind je hiervan een voorbeeld.

Het is ook mogelijk om zelf concepten te definiëren, die *niet* bestaan uit logische combinaties van standaard concepten, zie paragraaf 17.4[91].

[91] In de C++ Core Guideline CG: T.11. wordt geadviseerd om waar mogelijk standaard concepten te gebruiken.

4

Inheritance

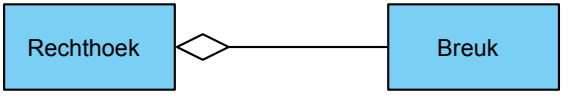
Een van de belangrijkste doelen van C++ is het ondersteunen van het hergebruik van code. In het vorige hoofdstuk werd één van de taalconstructies die C++ biedt om hergebruik van code mogelijk te maken, de *template*, besproken. In dit hoofdstuk worden de twee belangrijkste manieren besproken waarop een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

De in hoofdstuk 2 gedefinieerde class `Breuk` lijkt op het eerste gezicht al een prima herbruikbare softwarecomponent. Als je echter een variant van deze class `Breuk` wilt definiëren, dan heb je op dit moment geen andere keuze dan de class `Breuk` te kopiëren, te voorzien van een andere naam, bijvoorbeeld `My_breuk`, 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 leert in dit hoofdstuk hoe je door het toepassen van een objectgeoriënteerde techniek (inheritance) een onderhoudbare variant van een herbruikbare softwarecomponent kan maken. Door middel van deze techniek kun je dus softwarecomponenten niet alleen hergebruiken op de manier zoals de ontwerper van de component dat bedoeld heeft, maar kun je de softwarecomponent ook naar je eigen wensen omvormen.

Hergebruik door middel van *composition* is niet specifiek voor OOP. Ook bij de gestructureerde programmeermethode paste je deze vorm van hergebruik al toe. Het hergebruik van een softwarecomponent door middel van composition is niets anders als het gebruiken van deze component als onderdeel van een andere (nieuwe) softwarecomponent. Als je bijvoorbeeld

rechthoeken wilt gaan gebruiken waarbij de lengte en de breedte als breuk moeten worden weergegeven, dan kun je het UDT `Breuk` als volgt (her)gebruiken:

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



Figuur 4.1: Schematische weergave van composition.

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

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

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

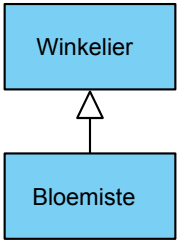
In dit hoofdstuk kun je de informatie over inheritance vinden die je moet weten om de rest van dit dictaat te kunnen lezen. Verdere details vind je in hoofdstuk 18.

4.1 De syntax van inheritance

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

In C++ code wordt dit als volgt gedeclareerd:

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

class Bloemiste: public[92] Winkelier {
//  Bloemiste is afgeleid van Winkelier
//  ...
};
```



Figuur 4.2: Schematische weergave van inheritance.

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

Een derived class heeft (minimaal) dezelfde datavelden en (minimaal) dezelfde public memberfuncties als de base class waarvan hij is afgeleid. Om deze reden mag je een object van de derived class ook gebruiken als de compiler een object van de base class verwacht[93]. De relatie tussen de derived class en de base class wordt een IS-A (is een) relatie genoemd. De derived class is een (speciaal geval van de) base class. Volgens figuur 4.2 is een `Bloemiste` een (speciaal geval van een) `Winkelier`. Omdat een derived class datavelden 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. Volgens figuur 4.2 is een `Winkelier` dus geen (speciaal geval van een) `Bloemiste`.

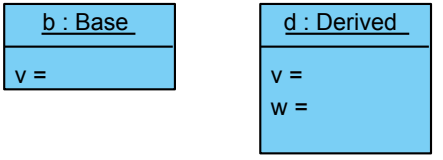
De derived class erft alle datavelden van de base class over. Dit wil zeggen dat een object van een derived class (minimaal) dezelfde datavelden heeft als een object van de base class. Private datavelden uit de base class zijn in objecten van de derived class wel aanwezig maar kunnen vanuit memberfuncties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

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

[93] Maar pas op voor het slicing probleem dat ik later (paragraaf 18.6) bespreek.

dataveld v is alleen toegankelijk vanuit de memberfuncties van de class Base en het dataveld w is alleen toegankelijk vanuit de memberfuncties van de class Derived.

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

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

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



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

Ook erft de derived class alle memberfuncties van de base class over. Dit wil zeggen dat op een object van een derived class (minimaal) dezelfde memberfuncties uitgevoerd kunnen worden als op een object van de base class. Private memberfuncties uit de base class zijn in de derived class wel aanwezig maar kunnen vanuit de derived class *niet* rechtstreeks aangeroepen worden. In de derived class kun je bovendien extra memberfuncties toevoegen.

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

```
class Base {
public:
    // ...
    int get_v() const { return v; }[94]
private:
    void set_v(int i) { v = i; }
    int v;
};

class Derived: public Base {
```

```
public:
    // ...
    int get_w() const { return w; }
private:
    void set_w(int i) { w = i; }
    int w;
};

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

4.2 Polymorfisme

Als de classes `Base` en `Derived` zoals hierboven gedefinieerd zijn, kan een pointer van het type `Base*` niet alleen wijzen naar een object van de class `Base` maar ook naar een object van de class `Derived`. Want een `Derived` IS-A (is een) `Base`. Omdat een pointer van het type `Base*` naar objecten van verschillende classes kan wijzen wordt zo'n pointer een *polymorfe* (veelvormige) pointer genoemd. Evenzo kan een reference van het type `Base&` niet alleen verwijzen naar een object van de class `Base` maar ook naar een object van de class `Derived`. Want een `Derived` IS-A (is een) `Base`. Omdat een reference van het type `Base&` naar objecten van verschillende classes kan verwijzen wordt zo'n reference een *polymorfe* (veelvormige) reference genoemd. Later in dit hoofdstuk blijkt dat *polymorfisme* de kern is waar het bij OOP om draait[95]. Door het toepassen van polymorfisme kun je software maken die eenvoudig aangepast, uitgebreid en hergebruikt kan worden. (Zie het uitgebreide voorbeeld in paragraaf 4.7.1.)

Als ik nu de volgende functie definieer:

```
void druk_v_af(const Base& p) {
    print("{}", p.get_v());
}
```

[94] Alle memberfuncties zijn in dit voorbeeld in de class zelf gedefinieerd. Dit zogenaamd *inline* definiëren (zie ook paragraaf 16.15) heeft als voordeel dat ik iets minder hoef te typen, maar het nadeel is dat de class niet meer afzonderlijk gecompileerd kan worden (zie paragraaf 2.21).

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

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, datavelden en memberfuncties toevoegen aan de base class. Een derived class kan bovendien memberfuncties die in de base class geïmplementeerd zijn in de derived class een andere implementatie geven (*overridden*). Dit kan alleen als de base class de memberfunctie *virtual* heeft gedeclareerd:[96]

```
class Base {
public:
    virtual void print_name() const {
        println("Ik ben een Base.");
    }
};
class Derived: public Base {
public:
    [97] void print_name() const override[98] {
        println("Ik ben een Derived.");
    }
};
```

Als via een polymorfe pointer een memberfunctie wordt aangeroepen, wordt de memberfunctie van de class van het object waar de pointer naar wijst aangeroepen. Omdat een polymorfe pointer tijdens het uitvoeren van het programma naar objecten van verschillende classes kan wijzen, kan de keuze van de memberfunctie pas tijdens het uitvoeren van het programma worden bepaald. Dit wordt *late binding* of ook wel *dynamic binding* genoemd. Op soortgelijke wijze kan een polymorfe reference verwijzen naar een object van verschillende

[96] Dit is niet waar. Maar als je een non-virtual memberfunctie uit de base class in de derived class toch opnieuw implementeert dan wordt de functie in de base class niet overridden maar overloaded. Overloading geeft onverwachte (en meestal ongewenste) effecten zoals je later (paragraaf 18.1) gaat zien.

[97] Het keyword **`virtual`** kan ook hier gebruikt worden maar kan ook weggelaten worden. Als een memberfunctie eenmaal virtual gedeclareerd is, blijft hij namelijk virtual.

[98] Het is mogelijk om expliciet met behulp van het keyword **`override`** aan te geven dat deze memberfunctie een, reeds in de base class gedeclareerde memberfunctie, override. Zie paragraaf 18.3. Omdat dit keyword pas in C++11 is toegevoegd, is het niet verplicht om dit te gebruiken. In de C++ Core Guideline C.128 wordt aangeraden om bij het overridden van een memberfunctie het keyword **`override`** te gebruiken en het keyword **`virtual`** weg te laten.

classes. Als via deze polymorfe reference een memberfunctie wordt aangeroepen, wordt de memberfunctie van de class van het object waar de reference naar verwijst aangeroepen. Ook in dit geval is er sprake van *late binding*.

Voorbeeld van het gebruik van polymorfisme:

```
Base b;
Derived d;
Base* bp1 {&b};
Base* bp2 {&d};
bp1->print_name();
bp2->print_name();
Base& br1 {b};
Base& br2 {d};
br1.print_name();
br2.print_name();
```

Uitvoer:

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

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

```
class Base {
public:
    virtual void print_name() const {
        println("Ik ben een Base.");
    }
};

class Derived: public Base {
public:
    void print_name() const override {
        println("Ik ben een Derived en ");
        Base::print_name();
    }
};
```

Voorbeeld van het gebruik van polymorfisme:

```
Base b;
Derived d;
Base* bp1 {&b};
Base* bp2 {&d};
bp1->print_name();
bp2->print_name();
Base& br1 {b};
Base& br2 {d};
br1.print_name();
br2.print_name();
```

Uitvoer:

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

4.4 Abstract base class

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

Een base class wordt vaak als ABC gedefinieerd omdat de implementatie van een virtual functie pas in de derived classes ingevuld kan worden. Verderop in dit hoofdstuk definiëren we een base class voor een passieve elektrische component. We weten dat zo'n passieve elektrische component een impedantie heeft. De base class heeft dan ook een memberfunctie om deze impedantie te berekenen. We weten echter pas hoe we deze functie moeten implementeren als we weten om welke passieve elektrische component het gaat. De verschillende passieve elektrische componenten: weerstand, spoel en condensator worden als derived classes afgeleid van de base class. In deze derived classes kunnen we de betreffende formule voor het berekenen van de impedantie van deze specifieke passieve elektrische component invullen. De functie om de impedantie te berekenen wordt in de base class dus pure virtual gedefinieerd en overridden in de derived classes. Een class met één of meer pure virtual functies wordt een ABC (Abstact Base Class) of ook wel een ADT (Abstract Data Type) genoemd. De derived classes waarin alle pure virtual functies overridden wordt een Concrete Class of ook wel een CDT (Concrete Data Type) genoemd.

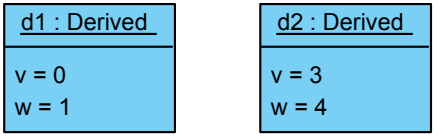
Bovendien voorkomt het gebruik van een ABC het slicing probleem, zie paragraaf 18.6.

4.5 Constructors bij inheritance

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

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

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



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

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

//  ...
    Derived d1;
    Derived d2 {3, 4};
```

4.6 protected members

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

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

Het definiëren van protected datavelden wordt afgeraden[99] omdat dit slecht is voor de onderhoudbaarheid van de code. Als een protected dataveld een illegale waarde krijgt, moet alle broncode worden doorzocht om de plaatsen te vinden waar dit dataveld veranderd wordt. In elke derived class is het protected dataveld namelijk te veranderen. Het is soms wel zinvol om protected memberfuncties te definiëren. Deze protected functies kunnen dan niet door gebruikers van de objecten van deze class worden aangeroepen maar wel in de memberfuncties van derived classes.

4.7 Voorbeeld: ADC kaarten

Ik beschrijf nu een praktisch programmeerprobleem. Vervolgens bespreek ik een gestructureerde oplossing, een oplossing door middel van een UDT en een objectgeoriënteerde oplossing. Daarna vergelijk ik deze oplossingen met elkaarn (nu mag je één keer raden welke oplossing de beste blijkt 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 twee verschillende types ADC kaarten kunnen werken. Deze kaarten hebben de typenamen AD178 en NI323. Deze kaarten zijn functioneel gelijk en hebben beide een 8 kanaals 16 bits ADC met instelbare voorversterker. Het initialiseren van de kaarten, het selecteren van een kanaal, het uitlezen van de 'sampled' waarde en het instellen van de versterkingsfactor moet echter bij elke kaart op een andere wijze gebeuren (andere adressen, andere bits en/of andere procedures). In de applicatie moeten meerdere ADC kaarten van verschillende types gebruikt kunnen worden. In de applicatie is alleen de spanning in volts van 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 bespreek ik 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:

[99] Zie de C++ Core Guideline NR.7.

- `init_card` voor het initialiseren van de kaart;
- `select_channel` voor het selecteren van een kanaal;
- `get_channel` voor het opvragen van het op dit moment geselecteerde kanaal;
- `set_amplifier` voor het instellen van de versterkingsfactor;
- `sample_card` voor het uitlezen van een sample;
- `read_card` voor het uitlezen van de spanning in volts.

Voor elke kaart die in het programma gebruikt wordt moeten een aantal gegevens bijgehouden worden zoals: het kaarttype, de ingestelde versterkingsfactor en het geselecteerde kanaal. Om deze gegevens per kaart netjes bij elkaar te houden heb ik de struct `ADCCard` gedeclareerd. Voor elke kaart die in het programma gebruikt wordt, wordt dan een variabele van dit struct type aangemaakt. Het type van de kaart wordt in de struct met behulp van het enumeratie-type `Card_type` opgeslagen[100]. Aan elk van de eerder genoemde functies wordt de te gebruiken kaart dan als parameter van het type `ADCCard` doorgegeven. Zie `kaart0.cpp`:

```
enum class Card_type {AD178, NI323};

struct ADCCard {
    Card_type type;
    double amplifying_factor;
    int selected_channel;
};

void init_card(ADCCard& card, Card_type type) {
    card.type = type;
    card.amplifying_factor = 1.0;
    card.selected_channel = 1;
    // ... eventueel voor alle kaarten benodigde code
    switch (card.type) {
        case Card_type::AD178:
            // ... de specifieke voor de AD178 benodigde code
            break;
        case Card_type::NI323:
            // ... de specifieke voor de NI323 benodigde code
            break;
    }
```

[100] In C++ kun je naast het enumeratie-type `enum` ook het enumeratie-type `enum class` gebruiken. Deze `enum class` heeft als voordeel dat de enumerators niet in de globale namespace terecht komen en dat er geen impliciete conversie naar een geheel getal plaatsvindt, zie paragraaf 16.17.

```
}

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

int get_channel(const ADCCard& card) {
    return card.selected_channel;
}

void set_amplifier(ADCCard& card, double factor) {
    card.amplifying_factor = factor;
    // ... eventueel voor alle kaarten benodigde code
    switch (card.type) {
        case Card_type::AD178:
            // ... de specifieke voor de AD178 benodigde code
            break;
        case Card_type::NI323:
            // ... de specifieke voor de NI323 benodigde code
            break;
        }
}

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

```
            break;
    }
    return sample;
}

double read_card(const ADCCard& card) {
    return sample_card(card) * card.amplifying_factor / 6553.5;
}

int main() {
    ADCCard c1;
    init_card(c1, Card_type::AD178);
    set_amplifier(c1, 10);
    select_channel(c1, 3);
    println("Kanaal {} van kaart c1 = {:.2f} V.", get_channel(c1), ↩
 ↪ read_card(c1));

    ADCCard c2;
    init_card(c2, Card_type::NI323);
    set_amplifier(c2, 5);
    select_channel(c2, 4);
    println("Kanaal {} van kaart c2 = {:.2f} V.", get_channel(c2), ↩
 ↪ read_card(c2));
}
```

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

- Iedere programmeur die gebruikt maakt van het type **struct** ADCCard kan een waarde toekennen aan de datavelden. Zo zou een programmeur die de struct ADCCard gebruikt in plaats van de functie select_channel het statement ++c1.selected_channel; 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 read_card. Er valt dus niet te garanderen dat altijd de juiste formule wordt gebruikt. Ook niet als we wel kunnen garanderen dat de functies read_card en sample_card correct zijn.
- Iedere programmeur die gebruikt maakt van het type **struct** ADCCard kan 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 Card_type uitgebreid worden. Bovendien moeten alle functies, waarin door middel van een **switch** afhankelijk van het kaarttype verschillende code wordt uitgevoerd, gewijzigd en opnieuw gecompileerd worden.

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

4.7.3 Een oplossing door middel van een UDT

Deze problemen kunnen voorkomen worden als een UDT gebruikt wordt, waarin zowel de data van een kaart als de functies die op een kaart uitgevoerd kunnen worden, ingekapseld zijn[101]. Zie kaart1.cpp:

```
enum class Card_type {AD178, NI323};

class ADCCard {
public:
    ADCCard(Card_type name);
    void select_channel(int channel);
    int get_channel() const;
    void set_amplifier(double factor);
    double read() const;
private:
    Card_type type;
    double amplifying_factor;
    int selected_channel;
    int sample() const;
};

ADCCard::ADCCard(Card_type name): type{name}, ↩
 ↪ amplifying_factor{1.0}, selected_channel{1} {
    // ... eventueel voor alle kaarten benodigde code
    switch (type) {
        case Card_type::AD178:
```

[101] 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 UDT ADCCard te gebruiken rechtstreeks de kaart aanspreekt.

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

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

int ADCCard::get_channel() const {
    return selected_channel;
}

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

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

```
    switch (type) {
        case Card_type::AD178:
            // ... de specifieke voor de AD178 benodigde code
            break;
        case Card_type::NI323:
            // ... de specifieke voor de NI323 benodigde code
            break;
    }
    return sample;
}

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

int main() {
    ADCCard k1 {Card_type::AD178};
    k1.set_amplifier(10);
    k1.select_channel(3);
    println("Kanaal {} van kaart k1 = {:.2f} V.", ↩
 ↪ k1.get_channel(), k1.read());

    ADCCard k2 {Card_type::NI323};
    k2.set_amplifier(5);
    k2.select_channel(4);
    println("Kanaal {} van kaart k2 = {:.2f} V.", ↩
 ↪ k2.get_channel(), k2.read());
}
```

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 `Card_type` uitgebreid worden. Bovendien moeten alle memberfuncties, waarin door middel van een **`switch`** afhankelijk van het kaarttype verschillende code wordt uitgevoerd, gewijzigd en opnieuw gecompileerd worden.

Op zich is dit nadeel bij deze oplossing minder groot dan bij de gestructureerde oplossing omdat in dit geval alleen de UDT `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 UDT tot een probleem voor wat betreft de uitbreidbaarheid leiden.

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

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

Het is waarschijnlijk dat de leverancier van de UDT alleen de module interface `ADCCard.cppm` en de library file `libADCCard.a` aan ons levert maar de broncode `ADCCard.cpp` file niet aan ons beschikbaar stelt[102]. Het toevoegen van de nieuwe ADC kaart (BB647) is dan niet mogelijk.

De (gekochte) UDT `ADCCard` is een herbruikbare softwarecomponent die echter niet door de gebruiker uit te breiden is. Je gaat zien dat je door het toepassen van de objectgeoriënteerde technieken inheritance en polymorfisme wel een softwarecomponent kan maken die door de gebruiker uitgebreid kan worden zonder dat de gebruiker de broncode van de originele component hoeft te wijzigen.

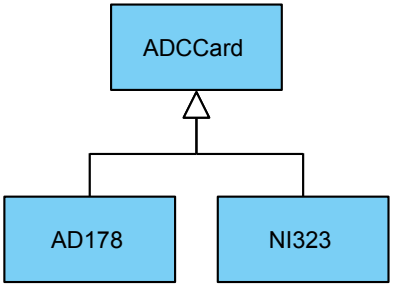
4.7.4 Een objectgeoriënteerde oplossing

Bij het toepassen van de objectgeoriënteerde benadering constateer ik dat in het programma twee types ADC kaarten gebruikt worden. Beide kaarten hebben dezelfde functionaliteit en

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

kunnen dus worden afgeleid van dezelfde base class[103]. Dit is schematisch weergeven in het UML class diagram dat gegeven is in figuur 4.5. Ik heb de base class als volgt gedeclareerd:

```
class ADCCard {
public:
    ADCCard();
    virtual ~ADCCard() = default;[104]
    virtual void select_channel(int channel) = 0;
    int get_channel() const;
    virtual void set_amplifier(double factor) = 0;
    double read() const;
protected:
    void remember_channel(int channel);
    void remember_amplifier(double factor);
private:
    double amplifying_factor;
    int selected_channel;
    virtual int sample() const = 0;
};
```



Figuur 4.5: Overerving bij ADC kaarten.

De memberfuncties `select_channel`, `set_amplifier` en `sample` heb ik *pure virtual* (zie paragraaf 4.4) gedeclareerd omdat de implementatie per kaarttype verschillend is. Dit maakt de class `ADCCard` abstract. Je kunt dus geen objecten van dit type definiëren, alleen references en pointers. Elke afgeleide concrete class (elk kaarttype) moet deze functies 'overridden'. De memberfuncties `get_channel` en `read` heb ik non-virtual gedeclareerd omdat het niet

[103] 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.

[104] Dit is een zogenoemde *virtual destructor*, waarom het nodig is om een virtual destructor te definiëren wordt verderop in dit dictaat besproken. Zie paragraaf 5.5.

mijn bedoeling is dat een derived class deze functies 'override'. Wat er gebeurt als je dit toch probeert, bespreek ik in paragraaf 18.1. De memberfuncties `remember_channel` en `remember_amplifier` heb ik protected gedeclareerd om er voor te zorgen dat ze vanuit de derived classes bereikbaar zijn. De memberfunctie `sample` heb ik private gedefinieerd omdat deze functie alleen vanuit de memberfunctie `read` gebruikt hoeft te kunnen worden. Derived classes moeten deze memberfunctie dus wel definiëren maar ze mogen hem zelf niet aanroepen! Ook gebruikers van deze derived classes hebben mijn inziens de memberfunctie `sample` niet nodig. Ze worden dus door mij verplicht de memberfunctie `read` te gebruiken zodat de returnwaarde altijd in volts is (ik hoop hiermee fouten bij het gebruik te voorkomen).

Van de abstracte base class `ADCCard` heb ik vervolgens de concrete classes `AD178` en `NI323` afgeleid. Zie `kaart2.cpp`:

```
class AD178: public ADCCard {
public:
    AD178();
    void select_channel(int channel) override;
    void set_amplifier(double factor) override;
private:
    int sample() const override;
};

class NI323: public ADCCard {
public:
    NI323();
    void select_channel(int channel) override;
    void set_amplifier(double factor) override;
private:
    int sample() const override;
};
```

De diverse classes heb ik als volgt geïmplementeerd:

```
ADCCard::ADCCard(): amplifying_factor{1.0}, selected_channel{1} {
    // ... voor alle kaarten benodigde code
}
int ADCCard::get_channel() const {
    return selected_channel;
}
double ADCCard::read() const {
    return sample() * amplifying_factor / 6553.5;
```

```
}
void ADCCard::remember_channel(int channel) {
    selected_channel = channel;
}
void ADCCard::remember_amplifier(double factor) {
    amplifying_factor = factor;
}

AD178::AD178() {
    // ... de specifieke voor de AD178 benodigde code
}
void AD178::select_channel(int channel) {
    remember_channel(channel);
    // ... de specifieke voor de AD178 benodigde code
}
void AD178::set_amplifier(double factor) {
    remember_amplifier(factor);
    // ... de specifieke voor de AD178 benodigde code
}
int AD178::sample() const {
    // ... de specifieke voor de AD178 benodigde code
}

NI323::NI323() {
    // ... de specifieke voor de NI323 benodigde code
}
void NI323::select_channel(int channel) {
    remember_channel(channel);
    // ... de specifieke voor de NI323 benodigde code
}
void NI323::set_amplifier(double factor) {
    remember_amplifier(factor);
    // ... de specifieke voor de NI323 benodigde code
}
int NI323::sample() const {
    // ... de specifieke voor de NI323 benodigde code
}
```

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

```
void do_measurement(ADCCard& card, double factor, int channel) {
```

```
    card.set_amplifier(factor);
    card.select_channel(channel);
    println("Kanaal {} = {:.2f} V.", card.get_channel(), ↩
 ↪ card.read());
}
```

Deze functie kan ik dan als volgt gebruiken:

```
int main() {
    AD178 card1;
    do_measurement(card1, 10, 3);
    NI323 card2;
    do_measurement(card2, 5, 4);
}
```

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

4.7.5 Een kaart toevoegen

Als het programma nu aangepast moet worden zodat een nieuwe kaart (typenaam BB647) ook gebruikt kan worden, dan kan dit heel eenvoudig door de volgende class te declareren (kaart4/BB647.cppm):

```
export class BB647: public ADCCard {
public:
    BB647();
    void select_channel(int channel) override;
    void set_amplifier(double factor) override;
private:
    int sample() const override;
};
```

Met de volgende implementatie (kaart4/BB647.cpp):

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

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

Het programma `main` kan nu als volgt aangepast worden (kaart4/kaart4.cpp):

```
int main() {
    AD178 card1;
    do_measurement(card1, 10, 3);
    NI323 card2;
    do_measurement(card2, 5, 4);
    BB647 card3;   // new!
    do_measurement(card3, 2, 7);
}
```

Als alle functie- en classdeclaraties in aparte module interfaces (.cppm) en alle functie- en classdefinities in aparte `.cpp` files opgenomen zijn, dan hoeft alleen de nieuwe class en de nieuwe `main` functie opnieuw vertaald te worden. De rest van het programma kan dan eenvoudig (zonder hercompilatie) meegelinkt worden. Dit voordeel komt voort uit het feit dat de functie `do_measurement` polymorf is. Aan de parameter die gedefinieerd is als een `ADCCard&` kun je objecten van elke van deze class afgeleide classes (`AD178`, `NI323` of `BB647`) gebruiken. Je ziet dat de objectgeoriënteerde oplossing een zeer goed *onderhoudbaar* en *uitbreidbaar* programma oplevert. In het directory `kaart3` en `kaart4` vind je een volledige implementatie van deze objectgeoriënteerde oplossing, respectievelijk zonder en met de nieuw toegevoegde class BB647. In deze directories zijn ook `CMakelists.txt` bestanden opgenomen zodat de applicaties eenvoudig gebouwd kunnen worden met CMake[105]

4.8 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.

[105] Zie https://cmake.org/.

4.8.1 Weerstand, spoel en condensator

Passieve elektrische componenten (hierna componenten genoemd) hebben een complexe[106] impedantie Z. Deze impedantie is een functie van de frequentie f. Er bestaan 3 soorten (basis)componenten:

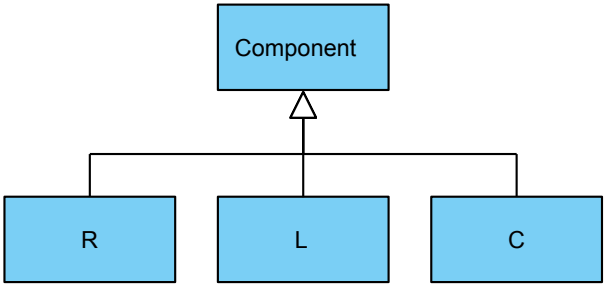
- R (weerstand): heeft een weerstandswaarde r uitgedrukt in Ohm. $Z(f) = r$.
- L (spoel): heeft een zelfinductie l uitgedrukt in Henry. $Z(f) = \mathrm{j} \cdot 2 \cdot \pi \cdot f \cdot l$.
- C (condensator): heeft een capaciteit c uitgedrukt in Farad. $Z(f) = -\mathrm{j}/(2 \cdot \pi \cdot f \cdot c)$.

De classes `Component`, `R`, `L` en `C` hebben de volgende relaties (zie ook figuur 4.6):

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

We willen een programma maken waarin gebruik gemaakt kan worden van passieve elektrische componenten. De ABC (Abstract Base Class) `Component` kan dan als volgt gedefinieerd worden:

```
class Component {
public:
    virtual ~Component() = default;[107]
    virtual complex<double> Z(double f) const = 0;
    virtual string naar_string() const = 0;
};
```



Figuur 4.6: `R`, `L` en `C` zijn afgeleid van `Component`

[106] Voor wie niet weet wat complexe getallen zijn: http://nl.wikipedia.org/wiki/Complex_getal.

[107] Dit is een zogenoemde *virtual destructor*, waarom het nodig is om een virtual destructor te definiëren wordt verderop in dit dictaat besproken. Zie paragraaf 5.5.

Het type `complex` is opgenomen in de standaard C++ library[108]. De functie `Z` moet in een van `Component` afgeleide class de impedantie berekenen (een complex getal) bij de als parameter gegeven frequentie `f`. De functie `naar_string` moet in een van `Component` afgeleide class het type en de waarde opslaan in een string en deze retourneren. Bijvoorbeeld: `"L(1E-3)"` voor een spoel van 1 mH.

Om componenten met behulp van de `print` en `println` af te kunnen drukken, is de template `std::formatter` gespecialiseerd voor `Component` (`netwerk0.cpp`), zie paragraaf 17.6:

```
template<>
struct std::formatter<Component>: public formatter<string> {
    auto format(const Component& c, auto& context) const {
        return formatter<string>::format(c.naar_string(), context);
    }
};
```

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

```
void print_impedance_table(const Component& c) {
    println("Impedantie tabel voor: {}\n", c);
    println("{:>7} | {}", "freq", "Z");
    println("{}", string(25, '-'));
    for (double freq(10); freq < 10E6; freq *= 10)
        println("{:>7.0f} | {:.3f}", freq, c.Z(freq));
    println();
}
```

Merk op dat de functie `print_impedance_table` niet 'weet' welke `Component` gebruikt wordt. Dit betekent dat deze polymorfe functie voor alle huidige soorten componenten te gebruiken is. De functie is zelfs ook voor toekomstige soorten componenten bruikbaar. Dit maakt het programma eenvoudig uitbreidbaar.

Om complexe getallen te kunnen printen, is in `netwerk0.cpp` de template `std::formatter` gespecialiseerd voor `complex<T>`, zie paragraaf 17.6.

De uitvoer van het bovenstaande programma is:

```
Impedantie tabel voor: R(100)

   freq | Z
```

[108] Zie eventueel https://en.cppreference.com/w/cpp/numeric/complex.

```
-------------------------
     10 | 100.000+0.000j
    100 | 100.000+0.000j
   1000 | 100.000+0.000j
  10000 | 100.000+0.000j
 100000 | 100.000+0.000j
1000000 | 100.000+0.000j

Impedantie tabel voor: C(1e-05)

   freq | Z
-------------------------
     10 | 0.000-1591.549j
    100 | 0.000-159.155j
   1000 | 0.000-15.915j
  10000 | 0.000-1.592j
 100000 | 0.000-0.159j
1000000 | 0.000-0.016j

Impedantie tabel voor: L(0.001)

   freq | Z
-------------------------
     10 | 0.000+0.063j
    100 | 0.000+0.628j
   1000 | 0.000+6.283j
  10000 | 0.000+62.832j
 100000 | 0.000+628.319j
1000000 | 0.000+6283.185j
```

Vraag:

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

Antwoord:

```
class R: public Component { // R = Weerstand
public:
```

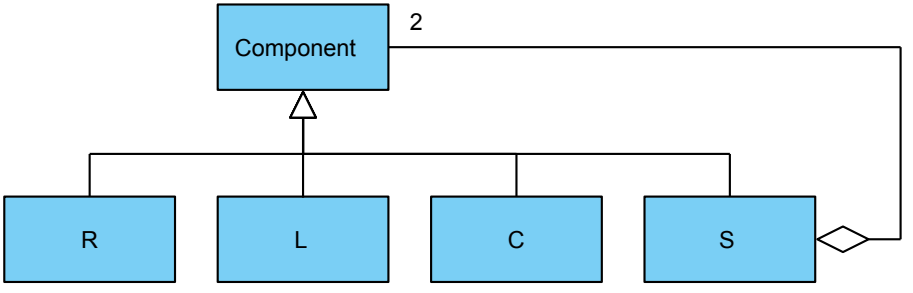
```
    R(double r): r{r} {
    }
    complex<double> Z(double) const override {
        return r;
    }
    string naar_string() const override {
        return format("R({})", r);
    }
private:
    double r;
};

class L: public Component { // L = Spoel
public:
    L(double l): l{l} {
    }
    complex<double> Z(double f) const override {
        return complex<double> {0, 2 * numbers::pi * f * l};
    }
    string naar_string() const override {
        return format("L({})", l);
    }
private:
    double l;
};

class C: public Component { // C = Condensator
public:
    C(double c): c{c} {
    }
    complex<double> Z(double f) const override {
        return complex<double> {0, -1 / (2 * numbers::pi * f * c)};
    }
    string naar_string() const override {
        return format("C({})", c);
    }
private:
    double c;
};
```

4.8.2 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, is duidelijk dat een serieschakeling een soort component *is*. De class `S` (serieschakeling) moet dus van de class `Component` afgeleid worden maar bestaat ook uit andere componenten. Zie figuur 4.7. Dit heeft als bijkomend voordeel dat je het aantal componenten waaruit een serieschakeling bestaat tot twee kunt beperken. Als je bijvoorbeeld een serieschakeling wilt doorrekenen van een `R`, `L` en `C`, dan maak je eerst van de `R` en `L` een serieschakeling, die je vervolgens met de `C` combineert tot een tweede serieschakeling. Op soortgelijke wijze kun je het programma uitbreiden met parallelschakelingen. Met dit programma kun je dan van elk passief elektrisch netwerk de impedantie berekenen.



Figuur 4.7: `S` is afgeleid van `Component` en `S` heeft 2 `Components`.

De classes `S` en `P` kunnen dan als volgt gebruikt worden (`netwerk1.cpp`):

```
int main() {
    R r1 {1E2};
    C c1 {1E-6};
    L l1 {3E-2};
    S s1 {r1, c1};
    S s2 {r1, l1};
    P p {s1, s2};
    print_impedance_table(p);
```

```
}
```

Je ziet dat je de al bestaande polymorfe functie `print_impedance_table` 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+1.257j
    100 | 101.591+12.515j
   1000 | 197.893-14.361j
  10000 | 101.132-10.580j
 100000 | 100.011-1.061j
1000000 | 100.000-0.106j
```

Vraag:

Implementeer nu zelf de classes `S` en `P`.

Antwoord:

```
class S: public Component { // S = Serie schakeling van twee componenten
public:
    S(const Component& c1, const Component& c2): c1{c1}, c2{c2} {
    }
    complex<double> Z(double f) const override {
        return c1.Z(f) + c2.Z(f);
    }
    string naar_string() const override {
        return format("({}+{})", c1.naar_string(), c2.naar_string());
    }
private:
    const Component& c1;
    const Component& c2;
};
```

```
class P: public Component { // P = Parallel schakeling van twee componenten
public:
    P(const Component& c1, const Component& c2): c1{c1}, c2{c2} {
    }
    complex<double> Z(double f) const override {
        return (c1.Z(f) * c2.Z(f)) / (c1.Z(f) + c2.Z(f));
    }
    string naar_string() const override {
        return format("({}//{})", c1.naar_string(), c2.naar_string());
    }
private:
    const Component& c1;
    const Component& c2;
};
```

In dit hoofdstuk is de informatie over inheritance besproken die je moet weten om de rest van dit dictaat te kunnen lezen. Verdere details vind je in hoofdstuk 18.

5

Dynamic memory allocation en destructors

In dit hoofdstuk worden de volgende onderwerpen besproken:

- Dynamic memory allocation;
- Destructor.

Als C++-programmeur maak je meestal niet rechtstreeks gebruik van dynamic memory allocation maar gebruik je classes uit de standaard library die gebruik maken van dynamic memory allocation. In hoofdstuk 19 wordt dynamic memory allocation toegepast bij het maken van een UDT `Array`. Aan het einde van dat hoofdstuk laat ik zien hoe je er voor zorgt dat je deze UDT ook met een range-based `for` kunt doorlopen en met een initialisatielijst kunt initialiseren.

5.1 Dynamische geheugenallocatie met `new` en `delete`

Je bent gewend om variabelen globaal of lokaal te definiëren. De geheugenruimte voor globale variabelen wordt gereserveerd zodra het programma start en pas weer vrijgegeven bij het beëindigen van het programma. De geheugenruimte voor een lokale variabele wordt, zodra die variabele gedefinieerd wordt, gereserveerd op de stack. Als het blok (compound statement), waarin de variabele 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 arrays (waarvan de grootte dus tijdens het uitvoeren van het programma bepaald kan worden) beschikt C++ over de operatoren **new**[] en **delete**[]. De operatoren **new** en **new**[] geven een pointer naar het nieuw gereserveerde geheugen terug. Deze pointer kan dan gebruikt worden om dit geheugen te gebruiken. Als dit geheugen niet meer nodig is, kan dit worden vrijgegeven door de operator **delete** of **delete**[] uit te voeren op de pointer die bij **new** of respectievelijk **new**[] is teruggegeven. De met **new** aangemaakte variabelen bevinden zich in een speciaal geheugengebied *heap* genaamd. Tegenover het voordeel van dynamische geheugenallocatie, een grotere flexibiliteit, staat het gevaar van een geheugenlek (memory leak). Een geheugenlek ontstaat als een programmeur vergeet een met **new** aangemaakte variabele weer met **delete** te verwijderen.

In C werden de functies `malloc` en `free` gebruikt om geheugenruimte op de heap te reserveren en weer vrij te geven. Deze functies zijn echter niet type-safe (type veilig) omdat het return type van `malloc` een **void*** is die vervolgens door de gebruiker naar het gewenste type moet worden omgezet. De compiler merkt het dus niet als de gebruikte type aanduidingen niet overeenkomen. Om deze reden zijn in C++ nieuwe memory allocatie operatoren (**new** en **delete**) toegevoegd.

Voorbeeld met **new** en **delete**:

```
    double* dp {new double}; // reserveer een double
    int i; cin >> i;
    double* drij {new double[i]}; // reserveer een array met i ↩
↪   doubles
    // ...
    delete dp; // geef de door dp aangewezen geheugenruimte vrij
    delete[] drij; // idem voor de door drij aangewezen array
```

In paragraaf 3.7 heb je gezien hoe, door het gebruik van een `std::vector` in plaats van het gebruik van een statische array, geen grens gesteld hoeft te worden aan het aantal elementen in een array. De enige grens is dan de grootte van het beschikbare (virtuele) werkgeheugen. De `std::vector` is geïmplementeerd met behulp van dynamische geheugenallocatie.

5.2 Smart pointers

Het resultaat van **new** toekennen aan een 'naakte' pointer is vragen om problemen. Voordat je het weet, vergeet je om **delete** aan te roepen en heb je een geheugenlek veroorzaakt.

Een eenvoudige oplossing is het gebruik van zogenoemde *smart pointers* die in de standaard library zijn opgenomen. Er zijn twee soorten smart pointers die veel gebruikt worden. De `unique_pointer` en de `shared_pointer`.

Een `unique_ptr` gebruik je als er één eigenaar is van de data waar de pointer naar wijst. Als deze pointer ophoudt te bestaan, dan wordt automatisch de operator **delete** aangeroepen op de pointer. Zie `unique_ptr.cpp`.

In de onderstaande functie wordt dynamisch een object aangemaakt van de class `Test`. De functie kan via twee paden verlaten worden. De programmeur is echter vergeten om in een van deze paden **delete** aan te roepen. Ook als de functie `is_waarde` een exception gooit, zie hoofdstuk 6, dan wordt **delete** niet aangeroepen. Deze functie kan dus een geheugenlek veroorzaken.

```
void f1(int i) {
    Test* p {new Test{i}};
    if (p->is_waarde(0)) {
        println("Test object heeft waarde 0");
        return;
    }
    println("Test object heeft waarde ongelijk aan 0");
    delete p;
}
```

In de onderstaande functie wordt dynamisch een object aangemaakt van de class `Test`. De functie kan via twee paden verlaten worden. In dit geval wordt de pointer die door **new** wordt teruggegeven opgeslagen in een `unique_ptr`. De programmeur hoeft er nu niet meer aan te denken om **delete** aan te roepen. De `unique_ptr` neemt deze verantwoordelijkheid over. Ook als de functie `is_waarde` een exception, zie hoofdstuk 6, gooit, dan wordt **delete** door `unique_ptr` aangeroepen. Als een functie verlaten wordt doordat er een excption wordt gegooid, dan worden de lokale variabelen namelijk netjes verwijderd. Deze functie kan dus *geen* geheugenlek veroorzaken.

```
void f2(int i) {
    unique_ptr<Test> p {new Test {i}};
    if (p->is_waarde(0)) {
```

```
        println("Test object heeft waarde 0");
        return;
    }
    println("Test object heeft waarde ongelijk aan 0");
}
```

Je kunt gebruik maken van de standaard functie `make_unique` om een dynamisch object aan te maken en het beheer over te dragen aan een `unique_ptr`[109]. Het is dan ook niet meer nodig om zelf **new** aan te roepen:

```
auto p {make_unique<Test>(i)};
```

Een `unique_ptr` kan *niet* gekopieerd worden. Er kan namelijk maar één eigenaar zijn van de data waar de pointer naar wijst. Als er twee pointers naar dezelfde data wijzen is niet meer duidelijk wie de eigenaar van die data is (en wie er dus verantwoordelijk is om de operator **delete** op deze data aan te roepen). Een `unique_ptr` kan *wel* verplaatst worden, bijvoorbeeld bij zo'n pointer wordt teruggegeven vanuit een functie De verantwoordelijkheid om de data vrij te geven wordt dan overgedragen van de ene naar de andere `unique_ptr`. Dit gebeurt door het aanroepen van de move constructor, zie eventueel paragraaf 19.11.

Een `shared_ptr` gebruik je als het eigenaarschap van de data waar de pointer naar wijst moet worden gedeeld door meerdere pointers. Een `shared_ptr` kan in tegenstelling tot een `unique_ptr` *wel* gekopieerd worden. Als een `shared_ptr` wordt gekopieerd, dan wordt de data waar de pointer naar wijst pas vrijgegeven als alle `shared_ptrs` naar deze data zijn verwijderd.

Een `shared_ptr` kan het beste geïnitialiseerd worden met de standaard library functie `make_shared` zodat je **new** niet zelf expliciet hoeft aan te roepen[110].

Smart pointers zijn te verkiezen boven gewone pointers omdat ze de kans op memory leaks sterk verkleinen. Maar smart pointers blijven een soort pointers, dus gebruik ze alleen als er geen eenvoudigere oplossing mogelijk is. Vaak kan het gebruik van een standaard container, zie hoofdstuk 11, het gebruik van pointers voorkomen.

[109] Dit wordt ook aangeraden in de C++ Core Guideline R.23.

[110] Dit wordt ook aangeraden in de C++ Core Guideline R.22.

5.3 Destructor ~Breuk

Een class kan naast een aantal constructors (zie paragraaf 2.4) ook één *destructor* hebben. De destructor heeft als naam, de naam van de class voorafgegaan door het teken ~. Als programmeur hoef je niet zelf de destructor aan te roepen. De compiler zorgt ervoor dat de destructor aangeroepen wordt net voordat het object opgeruimd wordt. Dit is:

- aan het einde van het blok waarin de variabele gedefinieerd is voor een lokale variabele;
- aan het einde van het programma voor globale variabele;
- bij het aanroepen van **delete** voor dynamische variabelen.

Als voorbeeld voegen we aan de eerste versie van de class Breuk (zie paragraaf 2.3) een destructor toe:

```
class Breuk {
public:
    Breuk();
    Breuk(int t);
    Breuk(int t, int n);
    ~Breuk();
    // rest van de class Breuk is niet gewijzigd.
};

Breuk::~Breuk() {
    println("Een breuk met de waarde {}/{} is verwijderd uit het ↩
 ↪ geheugen.", boven, onder);
}
```

Het hoofdprogramma om deze versie van Breuk te testen is (zie pagina 57):

```
int main() {
    Breuk b1 {4};
    println("b1 {{4}} = {}/{}", b1.teller(), b1.noemer());
    Breuk b2 {23, -5};
    println("b2 {{23, -5}} = {}/{}", b2.teller(), b2.noemer());
    Breuk b3 {b2};
    println("b3 {{b2}} = {}/{}", b3.teller(), b3.noemer());
    b3.abs();
    println("b3.abs() = {}/{}", b3.teller(), b3.noemer());
    b3 = b2;
    println("b3 = b2 = {}/{}", b3.teller(), b3.noemer());
    b3.plus(5);
```

```
    println("b3.plus(5) = {}/{}", b3.teller(), b3.noemer());
}
```

De uitvoer van dit programma (Breuk1_destructor.cpp) is nu:

```
b1 4 = 4/1
b2 23, -5 = -23/5
b3 b2 = -23/5
b3.abs() = 23/5
b3 = b2 = -23/5
Een breuk met de waarde 5/1 is verwijderd uit het geheugen.
b3.plus(5) = 2/5
Een breuk met de waarde 2/5 is verwijderd uit het geheugen.
Een breuk met de waarde -23/5 is verwijderd uit het geheugen.
Een breuk met de waarde 4/1 is verwijderd uit het geheugen.
```

Zoals je ziet worden de in de functie `main` aangemaakte lokale objecten `b1`, `b2` en `b3` aan het einde van de functie `main` uit het geheugen verwijderd. De volgorde van verwijderen is *omgekeerd* ten opzichte van de volgorde van aanmaken. Dit is niet toevallig maar hier is door de ontwerper van C++ goed over nagedacht. Stel dat we aan het begin van een functie vier objecten van de class `Wiel` aanmaken en vervolgens een object van de class `Auto` aanmaken waarbij we aan de constructor van `Auto` de vier `Wiel` objecten meegeven. Bij het verlaten van deze functie worden de objecten dan in omgekeerde volgorde uit het geheugen verwijderd[111]. 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 het aanmaken gaat.

We zien dat halverwege de functie `main` ook nog een `Breuk` object wordt verwijderd. Dit lijkt op het eerste gezicht wat vreemd. Als we goed kijken, zien we dat deze destructor wordt aangeroepen bij het uitvoeren van de volgende regel uit `main`:

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

[111] 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.

Snap je welk `Breuk` object aan het einde van deze regel wordt verwijderd? Lees indien nodig paragraaf 2.6 nog eens door. De memberfunctie `plus` verwacht een `Breuk` object als argument. De integer `5` wordt met de constructor `Breuk(5)` omgezet naar een `Breuk` object. Dit impliciet door de compiler aangemaakte object wordt dus aan het einde van de regel (nadat de memberfunctie `plus` is aangeroepen en teruggekeerd) weer uit het geheugen verwijderd.

Als geen destructor gedefinieerd is, wordt door de compiler een *default destructor* aangemaakt. Deze default destructor roept voor elk dataveld de destructor van dit veld aan (*memberwise destruction*). In dit geval heb ik de destructor `~Breuk` een melding op het scherm laten afdrukken. Dit is in feite nutteloos en ik had net zo goed de door de compiler gegenereerde default destructor kunnen gebruiken.

5.4 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?[112]

5.5 Virtual destructor

Als een class nu of in de toekomst als base class gebruikt wordt, moet de destructor virtual zijn zodat objecten van deze class afgeleide classes via een polymorfe pointer 'deleted' kunnen worden.

Hier volgt een voorbeeld van het gebruik van een ABC en polymorfisme (`Fruitmand_destructor.cpp`):

```
class Fruit {
public:
    virtual ~Fruit() {
        println("Er is een stuk Fruit verwijderd.");
    }
    virtual string soort() const = 0;
// ...
```

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

```
};

class Appel: public Fruit {
public:
    ~Appel() override {
        println("Er is een Appel verwijderd.");
    }
    string soort() const override {
        return "Appel";
    }
// ...
};

class Peer: public Fruit {
public:
    ~Peer() override {
        println("Er is een Peer verwijderd.");
    }
    string soort() const override {
        return "Peer";
    }
// ...
};

class Fruitmand {
public:
    ~Fruitmand() {
        for (const Fruit* e: fp)
            delete e;
            // Als we de constructor in de class Fruit *NIET* ↩
 ↪ virtual maken en *NIET* overridden voor Appel en Peer, dan ↩
 ↪ geeft de GCC C++ compiler een warning:: deleting object of ↩
 ↪ abstract class type 'Fruit' which has non-virtual destructor ↩
 ↪ will cause undefined behavior
    }
    void voeg_toe(Fruit* p) {
        fp.push_back(p);
    }
    void print_inhoud() const {
        println("De fruitmand bevat:");
        for (const Fruit* e: fp)
```

```
            println("  {}", e->soort());
    }
private:
    vector<Fruit*> fp;
};

int main() {
    Fruitmand m;
    m.voeg_toe(new Appel);
    m.voeg_toe(new Appel);
    m.voeg_toe(new Peer);
    m.print_inhoud();
    // hier wordt de Fruitmand m verwijderd!
}
```

De uitvoer van dit programma is als volgt:

```
De fruitmand bevat:
  Appel
  Appel
  Peer
Er is een Appel verwijderd.
Er is een stuk Fruit verwijderd.
Er is een Appel verwijderd.
Er is een stuk Fruit verwijderd.
Er is een Peer verwijderd.
Er is een stuk Fruit verwijderd.
```

Als in de base class `Fruit` *geen* virtual destructor gedefinieerd wordt maar een gewone (non-virtual) destructor die *niet* overridden wordt voor `Appel` en `Peer`, dan wordt de uitvoer als volgt:

```
De fruitmand bevat:
  Appel
  Appel
  Peer
Er is een stuk Fruit verwijderd.
Er is een stuk Fruit verwijderd.
Er is een stuk Fruit verwijderd.
```

Dit komt doordat de destructor via een polymorfe pointer (zie paragraaf 4.2) aangeroepen wordt. Als de destructor virtual gedefinieerd is, wordt tijdens het uitvoeren van het programma bepaald naar welk type object (een appel of een peer) deze pointer wijst. Vervolgens wordt de destructor van deze class (`Appel` of `Peer`) aangeroepen[113]. Omdat de destructor van een derived class ook altijd de destructor van zijn base class aanroept (zie paragraaf 5.4) wordt de destructor van `Fruit` ook aangeroepen. Als de destructor niet virtual gedefinieerd is, wordt tijdens het compileren van het programma bepaald van welk type de pointer is. Vervolgens wordt de destructor van deze class (`Fruit`) aangeroepen[114]. In dit geval wordt dus alleen de destructor van de base class aangeroepen.

Let op! Als we een class aanmaken zonder destructor, maakt de compiler zelf een zogenoemde *default destructor* aan. Deze automatisch aangemaakte destructor is echter *niet* virtual. In elke class die nu of in de toekomst als base class gebruikt wordt moeten we dus zelf een virtual destructor declareren[115]. Achter de declaratie kan de code = `default` gebruikt worden om aan te geven dat de default implementatie van de destructor, zie pagina 140 gebruikt moet worden.

5.6 Toepassing van `std::unique_ptr`

In de voorgaande paragraaf is de class `Fruitmand` als volgt gedefinieerd:

```
class Fruitmand {
public:
    ~Fruitmand() {
        for (const Fruit* e: fp)
            delete e;
    }
    void voeg_toe(Fruit* p) {
        fp.push_back(p);
    }
    void print_inhoud() const {
        println("De fruitmand bevat:");
        for (const Fruit* e: fp)
            println("  {}", e->soort());
    }
```

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

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

[115] Zie ook de C++ Core Guideline C.127.

```
private:
    vector<Fruit*> fp;
};
```

Deze class werd als volgt gebruikt:

```
    Fruitmand m;
    m.voeg_toe(new Appel);
    m.voeg_toe(new Appel);
    m.voeg_toe(new Peer);
    m.print_inhoud();
```

De class `Fruitmand` heeft een destructor waarin al het aan de fruitmand toegevoegde fruit wordt ge**delete** als de fruitmand wordt vrijgegeven. We kunnen deze verantwoordelijkheid ook overdragen aan een smart pointer.De `Fruitmand` bevat dan een `vector` van `unique_ptrs` en heeft geen destructor meer nodig:

```
class Fruitmand {
public:
    void voeg_toe(unique_ptr<Fruit> p) {
        fp.push_back(move(p));
    }
    void print_inhoud() const {
        println("De fruitmand bevat:");
        for (const auto& e: fp)
            println("  {}", e->soort());
    }
private:
    vector<unique_ptr<Fruit>> fp;
};
```

De standaardfunctie `move`, zie pagina 397, wordt gebruikt om er voor te zorgen dat het eigenaarschap overgedragen wordt van de smart pointer `p` naar een in de `vector` opgeslagen `unique_ptr`.

Deze class kan nu als volgt gebruikt worden:

```
    Fruitmand m;
    m.voeg_toe(make_unique<Appel>());
    m.voeg_toe(make_unique<Appel>());
    m.voeg_toe(make_unique<Peer>());
    m.print_inhoud();
```

Als de variabele `m` wordt verwijderd, wordt de (default) destructor van `Fruitmand` aangeroepen. Deze (default) destructor roept de destructor aan van al zijn datamembers in dit geval is dat `m.fp`. De destructor van deze `vector` roept de destructor aan van alle elementen van de `vector`. Elk element is een `unique_pointer` en elke destructor van die `unique_pointer` roept **`delete`** aan om het door de `unique_pointer` beheerde dynamische geheugen weer vrij te geven.

De uitvoer van dit programma is als volgt:

```
De fruitmand bevat:
  Appel
  Appel
  Peer
Er is een Appel verwijderd.
Er is een stuk Fruit verwijderd.
Er is een Appel verwijderd.
Er is een stuk Fruit verwijderd.
Er is een Peer verwijderd.
Er is een stuk Fruit verwijderd.
```

6

Exceptions

Vaak wordt in een functie of memberfunctie gecontroleerd op uitzonderlijke situaties (fouten). De volgende functie berekent de impedantie van een condensator van c Farad bij een frequentie van f Hz. Zie impedance_C1.cpp:

```
complex<double> impedance_C(double c, double f) {
    return complex<double> {0, -1 / (2 * numbers::pi * f * c)};
}
```

Deze functie kan als volgt aangeroepen worden om de impedantie van een condensator van 1 mF bij 1 kHz op het scherm af te drukken:

```
    println("{:.3f}", impedance_C(1e-6, 1e3));
```

Als deze functie aangeroepen wordt om de impedantie van een condensator van 0 F uit te rekenen, verschijnt de volgende vreemde uitvoer:

```
0.000-infj
```

De waarde `-inf` staat voor negatief oneindig. Ook het berekenen van de impedantie van een condensator bij 0 Hz veroorzaakt dezelfde vreemde uitvoer. Het zal voor iedereen duidelijk zijn dat zulke vreemde uitvoer tijdens het uitvoeren van het programma voorkomen moet worden.

6.1 Het gebruik van assert

In voetnoot 44 heb je al kennis gemaakt met de standaard macro `assert`. Deze macro doet niets als de, als argument meegegeven, expressie **`true`** oplevert maar breekt het programma met een passende foutmelding af als dit niet zo is. Omdat macro's niet vanuit modules geëxporteerd kunnen worden, moet de file `cassert` worden geïnclude om gebruik te kunnen maken van de `assert` macro[116]. Je kunt zogenoemde *assertions* gebruiken om tijdens de ontwikkeling van het programma te controleren of aan een bepaalde voorwaarden (waarvan je 'zeker' weet dat ze geldig zijn) wordt voldaan. Je kunt de functie `impedance_C` voorzien van een assertion (`impedance_C2.cpp`):

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

Als je nu tijdens het debuggen deze functie aanroept om de impedantie van een condensator van 0 F uit te rekenen (of om de impedantie van een condensator bij 0 Hz uit te rekenen), beëindigt de `assert` functie het programma. De volgende foutmelding verschijnt:

```
assertion "c != 0.0 && f != 0.0" failed: file "impedance_C2.cpp", ↩
 ↪ line 18, function: std::complex<double> impedance_C(double, ↩
 ↪ double)
```

Het programma stopt na deze melding. Als het programma echter gecompileerd wordt zonder zogenoemde debug informatie[117], worden alle `assert` functies verwijderd en verschijnt weer de vreemde uitvoer `0.000-infj`.

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 `impedance_C` alleen mag worden aangeroepen met argumenten die ongelijk aan 0 zijn, is dat prima met `assert` te controleren. Elk programmadeel waarin `impedance_C` wordt aangeroepen moet nu voor de aanroep zelf controleren of de argumenten geldige waarden hebben. Bij het testen van het programma (gecompileerd met debug informatie) zorgt de `assert` ervoor dat het snel duidelijk wordt als de afspraak geschonden wordt. Als na het testen duidelijk is dat iedereen zich aan de afspraak houdt, is het niet meer nodig om

[116] Waarschijnlijk zijn "Contracts" de vervanging voor de assert macro in C++26. Maar die worden helaas nog niet ondersteund in GCC 15.

[117] In GCC moet je het programma met de optie `-DNDEBUG` compileren om de assertions te verwijderen.

de `assert` uit te voeren. Het programma wordt gecompileerd zonder debug informatie en alle `assert` aanroepen worden verwijderd.

Het gebruik van `assert` heeft de volgende nadelen:

- Het programma wordt abrupt afgebroken als niet aan de assertion wordt voldaan.
- Op elke plek waar de functie `impedance_C` aangeroepen wordt moeten, voor aanroep, eerst de argumenten gecontroleerd worden. Als de functie op veel plaatsen aangeroepen wordt, is het logischer om de controle in de functie zelf uit te voeren. Dit maakt het programma ook beter onderhoudbaar (als de controle aangepast moet worden dan hoeft de code maar op 1 plaats gewijzigd te worden) en betrouwbaarder (je kunt de functie niet meer zonder controle aanroepen, de controle zit nu immers in de functie zelf).

We kunnen concluderen dat `assert` in dit geval niet geschikt is.

6.2 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 (`impedance_C3.cpp`):

```
[[nodiscard]] bool impedance_C(complex<double>& res, double c, ↩
 ↪ double f) {
    if (c != 0.0 && f != 0.0) {
        res = complex<double> {0, -1 / (2 * numbers::pi * f * c)};
        return true;
    }
    else
        return false;
}
```

Het programma wordt nu niet meer abrupt afgebroken. Het gebruik van een return waarde om aan te geven of een functie gelukt is heeft echter de volgende nadelen:

- Bij *elke* aanroep moet de returnwaarde getest worden. Het attribuut `[[nodiscard]]`, zie paragraaf 15.19 zorgt ervoor dat de compiler een waarschuwing geeft als de returnwaarde niet wordt gebruikt.
- Op de plaats waar de fout ontdekt wordt kan hij meestal niet opgelost worden.
- Het echte resultaat van de functie moet nu via een call by reference parameter worden teruggegeven. Dit betekent dat je om de functie aan te roepen altijd een variabele aan

moet maken (om het resultaat in op te slaan) ook als je het resultaat alleen maar wilt doorgeven aan een andere functie of operator.

De C library stdio werkt bijvoorbeeld met returnwaarden van functies die aangeven of de functie gelukt is. Zo geeft de functie `printf` bijvoorbeeld een **`int`** returnwaarde. Als de functie gelukt is, geeft `printf` het aantal geschreven karakters terug maar als er een error is opgetreden, geeft `printf` de waarde `EOF` terug. Een goed geschreven programma moet dus bij elke aanroep naar `printf` de returnwaarde testen![118]

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`[119]. Zie `impedance_C4.cpp`

```
complex<double> impedance_C(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 * numbers::pi * f * c)};
}
```

Je kunt een exception object gooien door het C++ keyword **`throw`** te gebruiken. Bij het aanroepen van de constructor van de class `domain_error` die gedefinieerd is in de include file `<exception>` kun je een string meegeven die de oorzaak van de fout aangeeft. Als de **`throw`** wordt uitgevoerd, wordt de functie meteen afgebroken. Lokale variabelen worden wel netjes opgeruimd (de destructors van deze lokale objecten wordt netjes aangeroepen). Ook de functie waarin de functie `impedance_C` is aangeroepen wordt meteen afgebroken. Dit proces van afbreken wordt gestopt zodra de exception wordt opgevangen. Als de exception nergens wordt opgevangen, wordt het programma gestopt.

```
int main() {
```

[118] Kijk nog eens terug naar oude C programma's die je hebt geschreven. Hoe werd daar met de returnwaarde van `printf` omgegaan?

[119] Als je opgelet hebt bij de wiskunde lessen, komt deze naam je bekend voor.

```
    try {
        println("Z = {:.3f}", impedance_C(1e-6, 1e3));
        println("Z = {:.3f}", impedance_C(1e-6, 0));
        println("Dit was het!");
    } catch (const domain_error& e) {
        println("Fout: {}", e.what());
    }
    println("The END.");
}
```

Exceptions kunnen worden opgevangen in een zogenaamd *try-blok*. Dit blok begint met het keyword **try** en wordt afgesloten met het keyword **catch**. Na het **catch** keyword moet je tussen haakjes aangeven welke exceptions je wilt opvangen gevolgd door een codeblok dat uitgevoerd wordt als de gespecificeerde exception wordt opgevangen. Na het eerste catch blok kunnen er nog een willekeurig aantal catch blokken volgen om andere exceptions ook te kunnen vangen. Als je alle mogelijke exceptions wilt opvangen, kun je dat doen met: `catch(...) { /*code*/ }`.

Een exception kun je het beste als reference opvangen. Dit heeft 2 voordelen:

- Het voorkomt een extra kopie.
- We kunnen op deze manier ook van `domain_error` afgeleide classes opvangen zonder dat het slicing probleem (zie paragraaf 18.6) 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:

```
Z = 0.000-159.155j
Fout: Frequentie == 0
The END.
```

Merk op dat het laatste statement in het try-blok niet meer uitgevoerd wordt omdat bij het uitvoeren van het tweede statement een exception optrad. Het gebruik van exceptions in plaats van een **bool** returnwaarde heeft de volgende voordelen:

- Het programma wordt niet meer abrupt afgebroken. Door de exception op te vangen op een plaats waar je er wat mee kunt heb je de mogelijkheid om het programma na een exception gewoon door te laten gaan of op z'n minst netjes af te sluiten.
- Je hoeft niet bij elke aanroep te testen. Je kunt meerder aanroepen opnemen in hetzelfde try-blok. Als in het bovenstaande programma een exception zou optreden in het eerste statement, wordt het tweede (en derde) statement niet meer uitgevoerd.

- De returnwaarde van de functie kan nu weer gewoon gebruikt worden om de berekende impedantie terug te geven.

Vraag:

Pas de impedance calculator (zie paragraaf 4.8) aan zodat dit programma geen vreemde uitvoer meer produceert door een divide by zero error.

Antwoord:

De classes C en P moeten worden aangepast (netwerk2.cpp):

```
class C: public Component { // C = Condensator
public:
    C(double c): c{c} {
    }
    complex<double> Z(double f) const override {
        if (c == 0.0)
            throw domain_error {"Capacity == 0"};
        if (f == 0.0)
            throw domain_error {"Frequency == 0"};
        return complex<double> {0, -1 / (2 * numbers::pi * f * c)};
    }
    string naar_string() const override {
        return format("C({})", c);
    }
private:
    double c;
};
class P: public Component { // P = Parallel schakeling van twee componenten
public:
    P(const Component& c1, const Component& c2): c1{c1}, c2{c2} {
    }
    complex<double> Z(double f) const override {
        if (c1.Z(f) + c2.Z(f) == complex<double>(0, 0))
            throw domain_error {"Impedance of parallel circuit can not be calculated (due to divide by zero)"};
        return (c1.Z(f) * c2.Z(f)) / (c1.Z(f) + c2.Z(f));
    }
    string naar_string() const override {
```

```
        return format("({}//{})", c1.naar_string(), ↩
 ↪ c2.naar_string());
    }
private:
    const Component& c1;
    const Component& c2;
};
```

In het hoofdprogramma kunnen exceptions als volgt worden opgevangen:

```
    try {
        R r1 {1E2};
        C c1 {0}; // om te testen!
        L l1 {3E-2};
        S s1 {r1, c1};
        S s2 {r1, l1};
        P p {s1, s2};
        print_impedance_table(p);
    } catch (domain_error& e) {
        println("Exception: {}", e.what());
    }
```

6.4 Het gebruik van zelfgedefinieerde exceptions

In plaats van het gebruik van de standaard gedefinieerde exceptions kun je ook zelf exception classes definiëren. Zie impedance_C5.cpp:

```
class Frequency_error {};
class Capacity_error {};

complex<double> impedance_C(double c, double f) {
    if (c == 0.0)
        throw Capacity_error {};
    if (f == 0.0)
        throw Frequency_error {};
    return complex<double> {0, -1 / (2 * numbers::pi * f * c)};
}
```

Voorbeeld van gebruik:

```
int main() {
    try {
```

```
        println("Z = {:.3f}", impedance_C(1e-6, 1e3));
        println("Z = {:.3f}", impedance_C(1e-6, 0));
        println("Dit was het!");
    } catch (const Capacity_error&) {
        println("Capaciteit == 0");
    } catch (const Frequency_error&) {
        println("Frequentie == 0");
    }
    println("The END.");
}
```

Uitvoer:

```
Z = 0.000-159.155j
Frequentie == 0
The END.
```

In het bovenstaande voorbeeld gebruiken we lege classes. Bij verschillende fouten gooien we objecten van verschillende (lege) classes. Bij het opvangen van de fout kunnen we de fout identificeren aan de hand van zijn class.

We kunnen bij het definiëren van exception classes ook overerving toepassen. Zelfgedefinieerde exception classes kunnen dan met behulp van overerving volgens een generalisatie- of specialisatiestructuur ingedeeld worden. Bij een **catch** kunnen we nu kiezen of we een specifieke of een generieke exception willen afvangen. De specifieke zijn polymorf met de generieke.

Doordat exceptions gewoon objecten zijn kun je ze ook data en gedrag geven.

Je kunt bijvoorbeeld een **virtual** memberfunctie definiëren die een bij de exception passende foutmelding geeft. Als je deze memberfunctie in specifiekere exceptions override, kun je een generieke exception vangen en toch door middel van dynamic binding de juiste foutmelding krijgen! Zie impedance_C6.cpp:

```
class Impedance_error {
public:
    virtual ~Impedance_error() = default;
    virtual string get_error_message() const = 0;
};

class Frequency_error: public Impedance_error {
public:
```

```
    string get_error_message() const override;
};

string Frequency_error::get_error_message() const {
    return "Frequentie == 0";
}

class Capacity_error: public Impedance_error {
public:
    string get_error_message() const override;
};

string Capacity_error::get_error_message() const {
    return "Capaciteit == 0";
}

complex<double> impedance_C(double c, double f) {
    if (c == 0.0)
        throw Capacity_error {};
    if (f == 0.0)
        throw Frequency_error {};
    return complex<double> {0, -1 / (2 * numbers::pi * f * c)};
}
```

Voorbeeld van gebruik:

```
    try {
        println("Z = {:.3f}", impedance_C(1e-6, 1e3));
        println("Z = {:.3f}", impedance_C(1e-6, 0));
        println("Dit was het!");
    } catch (const Impedance_error& e) {
        println("Error: {}", e.get_error_message());
    }
    println("The END.");
}
```

Uitvoer:

```
Z = 0.000-159.155j
Frequentie == 0
The END.
```

Als je nu bijvoorbeeld alleen de `Capacity_error` exception wilt opvangen maar de `Frequency_error` exception niet, dan kan dat eenvoudig door in de bovenstaande catch het type `Impedance_error` te vervangen door `Capacity_error`.

6.5 De volgorde van `catch` blokken

Als je meerdere catch blokken gebruikt om exceptions af te vangen, moet je er op letten dat de meest specifieke exceptions vóór de generieke exceptions komen. Omdat de catch blokken van boven naar beneden worden geprobeerd als er een exception gevangen wordt. Dus:

```
try {
//  ...
} catch (Frequency_error&) {
    println("Frequency_error exception");
} catch (Impedance_error&) {
    println("Other exception derived from Impedance_error");
} catch (...) {
    println("Other exception");
}
```

Vraag:

Waarom verschijnt bij het uitvoeren van de onderstaande code de foutmelding `Frequency_error exception` nooit op het scherm?

```
try {
//  ...
} catch (Impedance_error&) {
    println("Other exception derived from Impedance_error");
} catch (Frequency_error&) {
    println("Frequency_error exception");
}
```

Antwoord:

Als er een `Frequency_error` exception wordt gegooid, wordt deze al opgevangen in het eerste `catch` blok. Een `Frequency_error` is namelijk afgeleid van een `Impedance_error` dus geldt: een `Frequency_error` *is een* `Impedance_error`.

6.6 Exception details

Over exceptions valt nog veel meer te vertellen:

- **Exceptions in constructors en destructors.**
  Exceptions zijn heel handig om te melden dat het aanmaken van een object in de constructor mislukt is. Bijvoorbeeld omdat ongeldige argumenten aan de constructor zijn meegegeven, je kunt in zo'n geval vanuit de constructor een exception gooien. Het gebruik van exceptions in een destructor is echter niet aan te raden. Als een exception optreedt, worden objecten opgeruimd (waarbij destructors worden aangeroepen). Voordat je het weet wordt dan een exception gegooid tijdens het afhandelen van een exception en dat is niet toegestaan in C++. Als dit gebeurd wordt het programma afgesloten.
- **Function try-blok.**
  Speciale syntax om exceptions die optreden bij het initialiseren van datavelden in een constructor op te vangen.
- **Re-throw.**
  Gebruik van `throw` zonder argument in een catch blok om de zojuist opgevangen exception weer door te gooien.
- **No exception specification.**
  Speciaal keyword `noexcept` om in het prototype van een functie aan te geven dat deze functie geen exceptions kan veroorzaken. Als zo'n functie toch een exception gooit, dan wordt `std::terminate` aangeroepen om het programma meteen te beëindigen.
- **Exceptions in de std library.**
  Een overzicht van alle exceptions die in de standaard library gedefinieerd zijn, is te vinden op: `http://en.cppreference.com/w/cpp/error/exception`.

Voor al deze details verwijs ik je naar [1, hoofdstuk 10] en [23, hoofdstuk 10].

7

Inleiding algoritmen en datastructuren

In het vervolg van dit dictaat gaan we de kennis die we hebben opgedaan betreffende objectgeoriënteerd programmeren in C++ toepassen bij het gebruiken en ontwikkelen van algoritmen en datastructuren.

Toen je begon met programmeren heb je kennis gemaakt met de *statische* datastructuren array en struct. De eerste hogere programmeertalen, zoals FORTRAN (1957) en ALGOL (1958) beschikten al over de ingebouwde statische datastructuur array. COBOL (1959) was de eerste hogere programmeertaal met de ingebouwde statische datastructuur struct (record). Ook de in het begin van de jaren '70 ontstane talen zoals Pascal en C hebben deze datastructuren ingebouwd. Al heel snel in de geschiedenis van het programmeren (eind jaren '50) werd duidelijk dat de in hogere programmeertalen ingebouwde statische datastructuren in de praktijk vaak niet voldoen.

De stand van een spelletjes competitie kan in C bijvoorbeeld in de volgende datastructuur opgeslagen worden:

```
typedef struct {
    int punten;
    char naam[80];
} Deelnemer;

typedef struct {
```

```
    int aantal_deelnemers;
    Deelnemer lijst[100];
} Stand;

Stand s;
```

De nadelen van het gebruik van de ingebouwde datastructuren struct en array blijken uit dit voorbeeld:

- De groottes van de arrays `lijst` en `naam` moeten bij het vertalen van het programma bekend zijn en kunnen niet aangepast worden (zijn statisch).
- Elke `Deelnemer` neemt hierdoor evenveel ruimte in onafhankelijk van de lengte van zijn `naam` (is statisch).
- Elke `Stand` neemt hierdoor evenveel ruimte in onafhankelijk van de waarde van `aantal_deelnemers` (is statisch).
- Het verwijderen van een `Deelnemer` uit de `Stand` is een heel bewerkelijke operatie.[120]

In paragraaf 1.10 heb je geleerd dat je beter de, in de standaard C++ library opgenomen, class `string` kunt gebruiken in plaats van een `char[]`. Een C++ standaard `string` is dynamisch en kan tijdens het uitvoeren van het programma, indien nodig, groeien of krimpen. In paragraaf 3.7 heb je geleerd dat je beter de, in de standaard C++ library opgenomen, class `vector` kunt gebruiken in plaats van een array. De array met 100 deelnemers genaamd `lijst` kan dus worden vervangen door een `vector<Deelnemer> lijst`. Het datatype `vector` is dynamisch en kan tijdens het uitvoeren van het programma, indien nodig, groeien of krimpen.

De stand van een spelletjes competitie kan in C++ bijvoorbeeld in de volgende datastructuur opgeslagen worden:

```
class Deelnemer {
public:
    Deelnemer(string_view n, int p);
    int punten() const;
    string naam() const;
    void verhoogpunten(int p);
```

[120] Alle `Deelnemers` die volgen op de `Deelnemer` die verwijderd moet worden moeten namelijk één plaatsje naar voren verplaatst worden. Dit probleem is te omzeilen door deelnemers niet echt te verwijderen maar te markeren als ongeldig. Bijvoorbeeld door de `punten` van de te verwijderen `Deelnemer` de waarde `-1` te geven. Dit maakt het verwijderen eenvoudig maar een aantal andere bewerkingen worden ingewikkelder. Bij het afdrukken van de lijst moet je bijvoorbeeld de gemarkeerde deelnemers overslaan en bij het invoegen van nieuwe deelnemers aan de `lijst` moet je ervoor zorgen dat de gemarkeerde deelnemers opnieuw gebruikt worden, anders passen er geen 100 deelnemers meer in de `lijst`.

```
private:
    int pnt;
    string nm;
};

class Stand {
public:
    void voegtoe(const Deelnemer& d);
    void verwijder(const string& n);
    int punten(const string& n) const;
    void verhoogpunten(const string& n, int p);
    vector<deelnemer>::size_type aantal_deelnemers() const;
private:
    vector<Deelnemer> lijst;
};

Stand s;
```

De lijst met deelnemers kan in C ook dynamisch gemaakt worden door het gebruik van `malloc` en `free` en door de ingebouwde datastructuur **`struct`** te combineren met pointers. Het fundamentele inzicht dat je moet krijgen is dat niet alleen code, maar ook data, *recursief* kan worden gedefinieerd.

Een lijst met deelnemers kan bijvoorbeeld als volgt gedefinieerd worden:

$$\text{lijst van deelnemers} = \begin{cases} \text{leeg of} \\ \text{deelnemer met een pointer naar een lijst van deelnemers.} \end{cases}$$

Deze definitie wordt recursief genoemd omdat de te definiëren term in de definitie zelf weer gebruikt wordt. Door nu dit idee verder uit te werken zijn er in de jaren '60 vele standaard manieren bedacht om data te structureren.

Een stamboom kan bijvoorbeeld als volgt gedefinieerd worden:

$$\text{stamboom van een persoon} = \begin{cases} \text{leeg of} \\ \text{persoon met} \begin{cases} \text{een pointer naar de stamboom van zijn moeder } \textbf{én} \\ \text{een pointer naar de stamboom van zijn vader.} \end{cases} \end{cases}$$

Het standaardwerk waarin de meest gebruikte datastructuren helemaal worden uitgekauwd is *The Art of Computer Programming* van Knuth [12, 13, 14]. Er zijn in de loop der jaren duizenden boeken over deze langzamerhand 'klassieke' datastructuren verschenen in allerlei verschillende talen (zowel programmeertalen als landstalen). Het was dus al heel snel duidelijk dat niet alleen het algoritme, waarmee de data bewerkt moet worden, belangrijk is bij het ontwerpen van programma's maar dat ook de structuur, waarin de data wordt opgeslagen, erg belangrijk is. Het zal je duidelijk zijn dat het algoritme en de datastructuur van een programma niet los van elkaar ontwikkeld kunnen worden, maar dat ze sterk met elkaar verweven zijn. De titel van een bekend boek op dit terrein luidt dan ook niet voor niets: Algorithms + Data Structures = Programs [26].

Het spreekt voor zich dat de klassieke datastructuren al snel door gebruik te maken van objectgeoriënteerde programmeertechnieken als herbruikbare *componenten* werden geïmplementeerd. Er zijn dan ook diverse componenten bibliotheken op het gebied van klassieke datastructuren verkrijgbaar. In 1998 is zelfs een componenten bibliotheek van elementaire datastructuren en algoritmen officieel in de ISO/ANSI C++-standaad opgenomen. Deze bibliotheek werd eerst de STL (Standard Template Library) genoemd maar sinds deze library in de C++-standaad is opgenomen spreken we over de standaard C++ library. Deze library is in latere versies van de C++-standaad [7, 8, 9] behoorlijk uitgebreid. In het vervolg van dit dictaat gaan we uitgebreid op deze standaard C++ library in. Ook worden in dit dictaat een aantal applicaties besproken waarin gebruik gemaakt wordt van standaard datastructuren.

7.1 Analyse van algoritmen

Bij het bespreken van algoritmen is het belangrijk om een maat te hebben om de executietijd (Engels: run time) van algoritmen met elkaar te kunnen vergelijken. De executietijd van een algoritmen is afhankelijk van vele factoren zoals, de gebruikte hardware en compiler maar ook van de zogenoemde computationele complexiteit[121] van het algoritme. De complexiteit van een algoritme kan van een bepaalde orde zijn. Deze orde wordt gespecificeerd met behulp van de zogenoemde *big-O-notatie*[122]. We spreken bijvoorbeeld over een algoritme met een complexiteit van $O(n^2)$ (dit wordt uitgesproken als "van de orde n-kwadraat"). $O(n^2)$ betekent dat de executietijd van het algoritme rechtevenredig toeneemt met het kwadraat van het aantal data-elementen (n). In tabel 7.1 kun je zien hoe een algoritme van een

[121] Zie eventueel: https://en.wikipedia.org/wiki/Computational_complexity_theory.

[122] Zie eventueel: https://en.wikipedia.org/wiki/Big_O_notation.

bepaalde orde zich gedraagt voor grote waarden van n. Er wordt hierbij vanuit gegaan dat alle algoritmen bij $n = 100$ net zo snel zijn (1 ms).

Tabel 7.1: Executietijd van een algoritme met een bepaalde orde bij verschillende waarden van n.

Orde	$n = 100$	$n = 10000$	$n = 1000000$	$n = 100000000$
$O(1)$	1 ms	1 ms	1 ms	1 ms
$O(\log n)$	1 ms	2 ms	3 ms	4 ms
$O(n)$	1 ms	0,1 s	10 s	17 min
$O(n \cdot \log n)$	1 ms	0,2 s	30 s	67 min
$O(n^2)$	1 ms	10s	28 uur	761 jaar
$O(n^3)$	1 ms	17 min	32 jaar	31710 eeuw
$O(10^n)$	1 ms	∞	∞	∞

Je ziet dat een algoritme met een complexiteit van $O(n^2)$ niet erg bruikbaar is voor grote hoeveelheden data en dat een algoritme met een executietijd $O(n^3)$ of $O(10^n)$ alleen bruikbaar is voor kleine hoeveelheden data. De meest voor de hand liggende sorteermethoden[123] (selection sort, insertion sort, bubble sort enz.) blijken allemaal een executietijd $O(n^2)$ te hebben. Deze algoritmen zijn dus onbruikbaar voor grotere datasets! Al in 1962 heeft Hoare het zogenaamde Quicksort algoritme [6] ontworpen dat gemiddeld een complexiteit van $O(n \cdot \log n)$ heeft. In elk boek over algoritmen en datastructuren kun je een implementatie van dit algoritme vinden. Maar op zich is dat niet zo interessant want de standaard C++ library heeft een efficiënte sorteermethode met een complexiteit van $O(n \cdot \log n)$. Wat geldt voor sorteeralgoritmen geldt voor de meeste standaard bewerkingen. De voor de hand liggende manier om het aan te pakken is meestal niet de meest efficiënte manier. Trek hieruit de volgende les: ga nooit zelf standaard algoritmen ontwerpen maar gebruik een implementatie waarvan de efficiëntie bewezen is. In de standaard C++ library die we later in dit dictaat leren kennen zijn vele algoritmen en datastructuren op een zeer efficiënte manier geïmplementeerd. Raadpleeg in geval van twijfel altijd een goed boek over algoritmen en datastructuren. Het standaardwerk op het gebied van sorteer- en zoekalgoritmen is *The Art of Computer Programming, Volume 3: Sorting and Searching* [14] van Knuth. Een meer toegankelijk boek is *Algorithms in C++, Parts 1-4: Fundamentals, Data Structure, Sorting, Searching* [19] van Sedgewick. Zeker als de data aan bepaalde voorwaarden voldoet (als

[123] In veel inleidende boeken over programmeertalen worden dit soort sorteeralgoritmen als voorbeeld gebruikt.

de data die gesorteerd moet worden bijvoorbeeld al gedeeltelijk gesorteerd is), zijn er vaak specifieke algoritmen die in dat specifieke geval uiterst efficiënt zijn. Ook hiervoor verwijs ik je naar de vakliteratuur op dit gebied.

7.2 Datastructuren

Elke professionele programmeur met enkele jaren praktijkervaring kent de feiten uit dit hoofdstuk uit zijn of haar hoofd. De hier beschreven datastructuren zijn verzamelingen van andere datastructuren en worden ook wel *containers* genoemd. Een voorbeeld van een datastructuur die je waarschijnlijk al kent is de stack. Een stack van integers is een verzameling integers waarbij het toevoegen aan en verwijderen uit de verzameling volgens het LIFO (Last In First Out) protocol gaat. De stack is maar één van de vele al lang bekende (klassieke) datastructuren. Het is erg belangrijk om de eigenschappen van de verschillende datastructuren goed te kennen, zodat je weet waarvoor je ze kunt toepassen. In dit hoofdstuk worden van de bekendste datastructuren de belangrijkste eigenschappen besproken, zodat je weet hoe je deze datastructuur kunt gebruiken. Het is daarbij niet nodig dat je weet hoe deze datastructuur geïmplementeerd moet worden. In tabel 7.2 vind je een poging de eigenschappen van de verschillende datastructuren samen te vatten. Elke datastructuur kent dezelfde drie basisbewerkingen: invoegen (Engels: insert), verwijderen (Engels: erase) en zoeken (Engels: find). In de tabel is voor elke datastructuur de orde van de executietijd (en de specifieke naam) van elke basisbewerking gegeven. Daarnaast zijn van elke datastructuur de belangrijkste applicaties en implementaties gegeven.

In 1998 is de zogenaamde STL (Standard Template Library) officieel in de C++-standaad opgenomen. Doordat nu een standaard implementatie van de klassieke datastructuren en algoritmen in elke standaard C++ library is opgenomen, is het minder belangrijk om zelf te weten hoe z'n datastructuur geïmplementeerd moet worden. Het belangrijkste is dat je weet wanneer je welke datastructuur kunt toepassen en hoe je dat dan moet doen.

Tabel 7.2: De meest bekende datastructuren.

naam	insert	erase	find	applicaties	implementaties
stack	push $O(1)$	pull $O(1)$ LIFO	top $O(1)$ LIFO	dingen omdraaien, is … gebalanceerd?, evaluatie van expressies	array (statisch + snel), linked list (dynamisch)

Deze tabel wordt vervolgd op de volgende pagina.

Tabel 7.2: (Vervolg) De meest bekende datastructuren.

naam	insert	erase	find	applicaties	implementaties
queue	enqueue $O(1)$	dequeue $O(1)$ FIFO	front $O(1)$ FIFO	printer queue, wachtrij	array (statisch + snel), linked list (dynamisch)
vector	$O(1)$	$O(n)$ zoeken	$O(n)$ op inhoud $O(1)$ op index	vaste lijst, code conversie	array (statisch[124], random access via operator[])
sorted vector	$O(n)$ zoeken + schuiven	$O(n)$	$O(\log n)$ op inhoud $O(1)$ op index	lijst waarin je veel zoekt en weinig muteert	array (statisch[124]) + binary search algoritme
linked list	$O(1)$	$O(n)$	$O(n)$	dynamische lijst waarin je weinig zoekt en verwijdert	linked list (dynamisch, sequential access via pointer)
sorted list	$O(n)$	$O(n)$	$O(n)$	dynamische lijst die je vaak gesorteerd afdrukt	
tree	$O(\log n)$	$O(n)$	$O(n)$	meerdimensionale lijst, file systeem, expressie boom	meer overhead in ruimte, minimal $n+1$ pointers met waarde 0
search tree	$O(\log n)$	$O(\log n)$	$O(\log n)$	dynamische lijst waarin je veel muteert en zoekt	sorted binary tree (meer overhead in ruimte dan linked list)
hash table	$O(1)$	$O(1)$	$O(1)$	symbol table (compiler), woordenboek	semi-statisch, minder efficiënt als overvol
priority queue	$O(\log n)$	$O(\log n)$	$O(1)$	event-driven simulatie	binary heap

[124] Een vector kan dynamisch gemaakt worden door op het moment dat de array vol is een nieuwe array aan te maken die twee keer zo groot is als de oorspronkelijke array. De executietijd van de insert operatie blijft dan gemiddeld $O(1)$, dit wordt constant amortized genoemd. Als we de array vergroten van N naar $2N$ elementen,

dan moeten er N elementen gekopieerd worden. De tijd die hiervoor nodig is kunnen we verdelen over de eerste N insert operaties (die nodig waren om de oorspronkelijke array vol te maken), hierdoor blijft de gemiddelde tijd per insert constant.

8

Gebruik van een stack

Als voorbeeld bespreken we in dit hoofdstuk enkele toepassingen van de bekende datastructuur *stack*. Een stack is een datastructuur waarin data-elementen kunnen worden opgeslagen. Als de elementen weer van de stack worden gehaald, dan kan dit alleen in de omgekeerde volgorde dan de volgorde waarin ze op de stack zijn geplaatst. Om deze reden wordt een stack ook wel een LIFO (Last In First Out) buffer genoemd. Het gedrag van een stack (de interface) kan worden gedefinieerd met behulp van een ABC (Abstract Base Class). Dit kan bijvoorbeeld als volgt, zie stack.cppm[125]:

```
export module stack;

export template <typename T> class Stack {
public:
    Stack() = default;
    Stack(const Stack&) = delete;
    virtual ~Stack() = default;
    void operator=(const Stack&) = delete;
    virtual void push(const T& t) = 0;
    virtual void pop() = 0;
    virtual const T& top() const = 0;
    virtual bool empty() const = 0;
    virtual bool full() const = 0;
};
```

[125] In de ISO/ANSI C++ standaard library is ook een type stack gedefinieerd, zie hoofdstuk 10.

Je ziet dat de ABC `Stack` bestaat uit 2 doe-functies (`push` en `pop`) en 3 vraag-functies (`top`, `empty` en `full`). Het van de stack afhalen van een element gaat bij deze interface in twee stappen. Met de vraag-functie `top` kan eerst het bovenste element van de stack gelezen worden waarna het met de doe-functie `pop` van de stack verwijderd kan worden.

Sinds C++ kun je automatisch gegenereerde memberfuncties, zie pagina's 59, 61 en 62 verwijderen door er `= delete` achter te plaatsen. De `operator=` en de copy constructor van de class `Stack` zijn met `= delete` gedeclareerd waardoor ze niet gebruikt kunnen worden. Het toekennen van de ene stack aan de andere en het kopiëren van een stack hebben beide een executietijd $O(n)$ en om die reden hebben we deze operaties verwijderd. Bijvoorbeeld om te voorkomen dat een stack per ongeluk als call by value argument aan een functie wordt doorgegeven. Omdat we een copy constructor hebben gedeclareerd moeten we zelf een default constructor (een constructor zonder parameters) definiëren omdat de compiler deze niet meer genereert als er één of meerdere constructors gedeclareerd zijn. In C++ kun je de compiler dwingen om een default memberfunctie te genereren door er `= default` achter te plaatsen. Omdat de class `Stack` als een base class gebruikt wordt is het belangrijk dat de destuctor `virtual` gedeclareerd wordt (zie eventueel paragraaf 5.5), door de toevoeging `= default` genereert de compiler zelf de default implementatie van deze destructor.

8.1 Haakjes balanceren

Als voorbeeld bekijken we nu een C++ programma `balance.cpp` dat controleert of alle haakjes in een tekst gebalanceerd zijn. We maken daarbij gebruik van de class `Stack_with_list<T>`, zie `stacklist.cppm`, die is afgeleid van de hierboven gedefinieerde `Stack<T>`.

```
// Gebruik een stack voor het controleren van de haakjes
// Invoer afsluiten met .

import std;
import stacklist;
using namespace std;

int main() {
    Stack_with_list<char> s;
    char c;
    println("Type een expressie met haakjes () [] of {{}} en sluit ↩
 ↪ af met .");
    cin.get(c);
    while (c != '.') {
```

```
        if (c == '(' || c == '{' || c == '[') {
            s.push(c);
        }
        else {
            if (c == ')' || c == '}' || c == ']') {
                if (s.empty()) {
                    println("Fout {} bijbehorend haakje openen ↩
↪ ontbreekt.", c);
                }
                else {
                    char d {s.top()};
                    s.pop();
                    if ((d == '(' && c != ')') || (d == '{' && c ↩
↪ != '}') || (d == '[' && c != ']')) {
                        println("Fout {} bijbehorend haakje openen ↩
↪ ontbreekt.", c);
                    }
                }
            }
        }
        cin.get(c);
    }
    while (!s.empty()) {
        char d {s.top()};
        s.pop();
        println("Fout {} bijbehorend haakje sluiten ontbreekt.", d);
    }
}
```

8.2 Eenvoudige rekenmachine

In veel programma's moeten numerieke waarden ingevoerd worden. Het zou erg handig zijn als we op alle plaatsen waar een getal ingevoerd moet worden ook een eenvoudige formule kunnen invoeren. Het rechtstreeks evalueren (interpreteren en uitrekenen) van een expressie zoals:

```
12 + 34 * (23 + 2) * 2
```

is echter niet zo eenvoudig.

8.2.1 Postfix-notatie

Wij zijn gewend om expressies in de zogenaamde *infix*-notatie op te geven. Infix wil zeggen dat de operator in het midden (tussen de operanden in) staat. Er zijn nog twee andere notatievormen mogelijk: *prefix* en *postfix*. In de prefix-notatie staat de operator voorop en in postfix-notatie staat de operator achteraan. De bovenstaande expressie wordt dan als volgt in postfix genoteerd:

```
12 34 23 2 + * 2 * +
```

Postfix-notatie wordt ook vaak “RPN” genoemd. Dit staat voor Reverse Polish Notation. Prefix-notatie is namelijk bedacht door een Poolse wiskundige met een moeilijke achternaam (Jan Lukasiewicz) waardoor prefix ook wel “polish notation” wordt genoemd. Omdat postfix de omgekeerde volgorde gebruikt ten opzichte van prefix wordt postfix dus “omgekeerde Poolse-notatie” genoemd.

In de infix-notatie hebben we zogenaamde prioriteitsregels nodig (Meneer Van Dale Wacht Op Antwoord) die de volgorde bepalen waarin de expressie geëvalueerd moet worden (Machtsverheffen Vermenigvuldigen, Delen, Worteltrekken, Optellen, Aftrekken). We moeten haakjes gebruiken om een andere evaluatievolgorde aan te geven. Bijvoorbeeld:

```
2 + 3 * 5 = 17
```

want vermenigvuldigen gaan voor optellen, maar

```
(2 + 3) * 5 = 25
```

want de haakjes geven aan dat je eerst moet optellen.

In de pre- en postfix-notaties zijn helemaal geen prioriteitsregels en dus ook geen haakjes meer nodig. De plaats van de operatoren bepaalt de evaluatievolgorde. De bovenstaande infix expressies worden in postfix als volgt geschreven:

```
2 3 5 * + = 17
```

en

```
2 3 + 5 * = 25
```

Postfix heeft de volgende voordelen ten opzichte van infix:

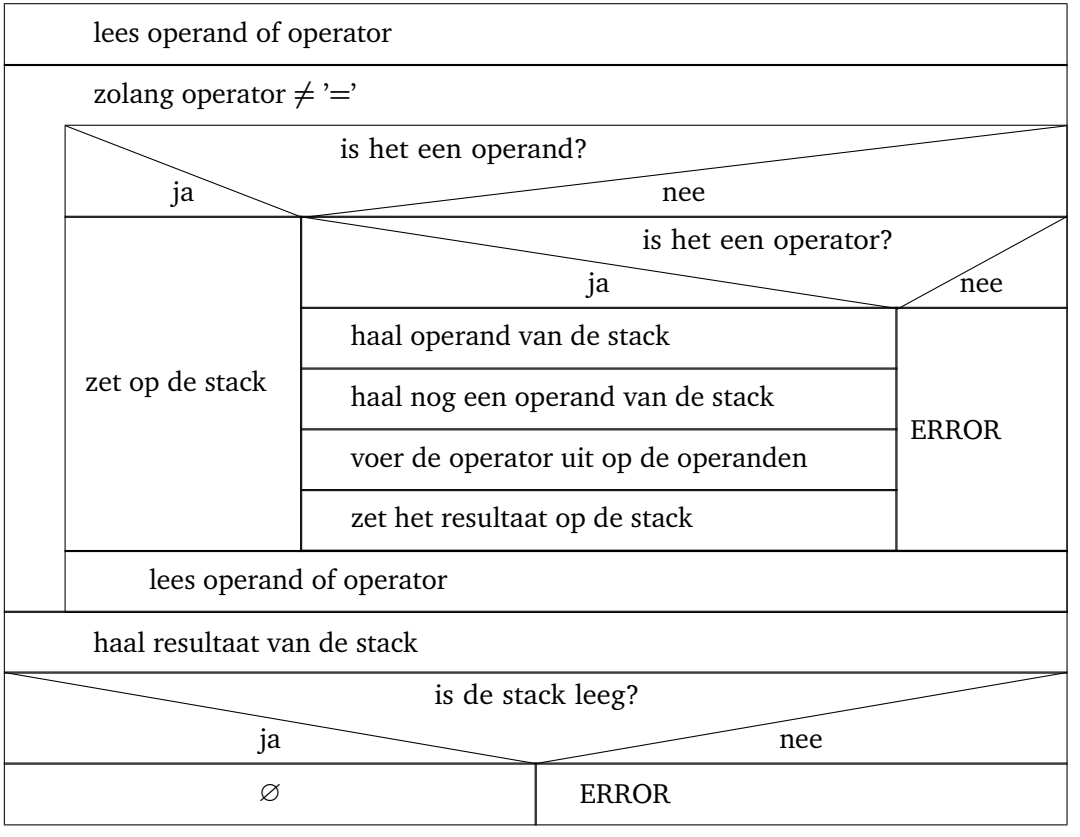
- Geen prioriteitsregel nodig.
- Geen haakjes nodig.

- Eenvoudiger te berekenen m.b.v. een stack.

Om deze redenen gebruikte de eerste handheld wetenschappelijke rekenmachine die in 1972 op de markt kwam (de HP-35[126]) RPN (Reverse Polish Notation). Tot op de dag van vandaag bieden veel rekenmachines van HP nog steeds de mogelijkheid om formules in RPN in te voeren.

8.2.2 Een postfix calculator

Een postfix expressie kan met het algoritme dat gegeven is in figuur 8.1 worden uitgerekend. Dit algoritme maakt gebruik van een stack en is weergegeven in een Nassi-Shneiderman diagram (NSD)[127].



Figuur 8.1: Algoritme voor het uitrekenen van een postfix expressie.

[126] Zie http://en.wikipedia.org/wiki/HP-35.

[127] Zie https://en.wikipedia.org/wiki/Nassi%E2%80%93Shneiderman_diagram.

Een calculator die postfix expressies kan uitrekenen is natuurlijk niet erg gebruiksvriendelijk. In de volgende paragraaf ga je zien dat op vrij eenvoudige wijze een infix naar postfix convertor te maken is. Door beide algoritmen te combineren ontstaat dan een infix calculator.

Vraag:

Implementeer nu zelf het algoritme uit figuur 8.1. Je mag jezelf beperken tot optellen en vermenigvuldigen.

Antwoord postfix.cpp:

```
import std;
import stacklist;
using namespace std;

int main() {
    Stack_with_list<int> s;
    char c;
    println("Type een postfix expressie (met + en * operator) en ↩
 ↪ sluit af met =");
    cin >> c;
    while (c != '=') {
        if (isdigit(c)) {
            cin.putback(c);
            int i;
            cin >> i;
            s.push(i);
        }
        else if (c == '+') {
            int op2 {s.top()}; s.pop();
            int op1 {s.top()}; s.pop();
            s.push(op1 + op2);
        }
        else if (c == '*') {
            int op2 {s.top()}; s.pop();
            int op1 {s.top()}; s.pop();
            s.push(op1 * op2);
        }
        else {
            println("Syntax error");
        }
```

```
            cin >> c;
        }
        println("= {}", s.top());
        s.pop();
        if (!s.empty()) {
            println("Fout operator ontbreekt.");
        }
}
```

8.2.3 Een infix-calculator

Een postfix-calculator is niet erg gebruiksvriendelijk. Een infix-calculator kan gemaakt worden door een infix naar postfix convertor te koppelen aan een postfix-calculator.

In 1961 heeft de Nederlander Dijkstra het volgende algoritme, dat bekend staat als het rangeerstationalgoritme[128] (shunting-yard algorithm), gepubliceerd [3] om een infix expressie om te zetten naar een postfix expressie:

- Dit algoritme maakt gebruik van een stack met karakters.
- Lees karakter voor karakter in.
- Als een ingelezen karakter geen haakje of operator is, dan kan dit meteen worden doorgestuurd naar de uitvoer. Een = teken wordt in dit geval niet als operator gezien.
- Een haakje openen wordt altijd op de stack geplaatst.
- Als we een operator inlezen, dan moeten we net zo lang operatoren van de stack halen en doorsturen naar de uitvoer totdat:
  - we een operator op de stack tegenkomen met een lagere prioriteit of
  - we een haakje openen op de stack tegenkomen of
  - de stack leeg is.
- Daarna moet de ingelezen operator op de stack worden geplaatst.
- Als we een haakje sluiten inlezen, dan moeten we net zo lang operatoren van de stack halen en doorsturen naar de uitvoer totdat we een haakje openen op de stack tegenkomen. Dit haakje openen moet wel van de stack verwijderd worden maar wordt niet doorgestuurd naar de uitvoer.
- Als we een = tegenkomen, moeten we alle operatoren van de stack halen en doorsturen naar de uitvoer.

[128] Zie http://en.wikipedia.org/wiki/Shunting-yard_algorithm.

In tabel 8.1 kun je zien hoe de expressie:

```
12 + 34 * (23 + 2) * 2 =
```

omgezet wordt in de postfix expressie:

```
12 34 23 2 + * 2 * + =
```

Tabel 8.1: Voorbeeld van een conversie van in- naar postfix.

gelezen karakter(s)	stack	uitvoer
12		12
+	+	12
34	+	12 34
*	+ *	12 34
(	+ * (	12 34
23	+ * (	12 34 23
+	+ * (+	12 34 23
2	+ * (+	12 34 23 2
)	+ *	12 34 23 2 +
*	+ *	12 34 23 2 + *
2	+ *	12 34 23 2 + * 2
=		12 34 23 2 + * 2 * +

Vraag:

Implementeer nu zelf een programma waarmee een infix-expressie kan worden omgezet in een postfix expressie. Je mag jezelf beperken tot optellen en vermenigvuldigen.

Antwoord infix_to_postfix.cpp:

```
import std;
import stacklist;
using namespace std;

bool has_lower_prio(char op1, char op2) {
    // geeft true als prio(op1) < prio(op2)
```

```
    // eenvoudig omdat alleen + en * toegestaan zijn
    return op1 == '+' && op2 == '*';
}

int main() {
    Stack_with_list<char> s;
    char c;
    println("Type een infix expressie (met + en * operator) en ↩
 ↪ sluit af met =");
    cin >> c;
    while (c != '=') {
        if (isdigit(c)) {
            cin.putback(c);
            int i;
            cin >> i;
            println("{} ", i);
        }
        else if (c == '(') {
            s.push(c);
        }
        else if (c == '+' || c == '*') {
            while (!s.empty() && s.top() != '(' && ↩
 ↪ !has_lower_prio(s.top(), c)) {
                println("{} ", s.top());
                s.pop();
            }
            s.push(c);
        }
        else if (c == ')') {
            while (s.top() != '(') {
                println("{} ", s.top());
                s.pop();
            }
            s.pop();
        }
        else {
            println("Syntax error");
        }
        cin >> c;
    }
    while (!s.empty()) {
```

```
        println("{} ", s.top());
        s.pop();
    }
}
```

Vraag:

Implementeer nu zelf een infix-calculator door de infix naar postfix converter te combineren met de postfix-calculator. Je mag jezelf beperken tot optellen en vermenigvuldigen.

Antwoord infix.cpp:

```
import std;
import stacklist;
using namespace std;

bool has_lower_prio(char op1, char op2) {
    return op1 == '+' && op2 == '*';
}

void process_operator(Stack_with_list<char>& s1, ↩
 ↪ Stack_with_list<int>& s2) {
    int op2 {s2.top()}; s2.pop();
    int op1 {s2.top()}; s2.pop();
    switch (s1.top()) {
        case '*': s2.push(op1 * op2); break;
        case '+': s2.push(op1 + op2); break;
    }
    s1.pop();
}

int main() {
    Stack_with_list<char> s1;
    Stack_with_list<int> s2;
    char c;
    println("Type een infix expressie (met + en * operator) en ↩
 ↪ sluit af met =");
    cin >> c;
    while (c != '=') {
        if (isdigit(c)) {
            cin.putback(c);
            int i;
```

```
            cin >> i;
            s2.push(i);
        }
        else if (c == '(') {
            s1.push(c);
        }
        else if (c == '+' || c == '*') {
            while (!s1.empty() && s1.top() != '(' && ↩
↪ !has_lower_prio(s1.top(), c)) {
                process_operator(s1, s2);
            }
            s1.push(c);
        }
        else if (c == ')') {
            while (s1.top() != '(') {
                process_operator(s1, s2);
            }
            s1.pop();
        }
        else {
            println("Syntax error");
        }
        cin >> c;
    }
    while (!s1.empty()) {
        process_operator(s1, s2);
    }
    println("= {}", s2.top());
    s2.pop();
    if (!s2.empty()) {
        println("Fout operator ontbreekt.");
        s2.pop();
    }
}
```

Het is natuurlijk netter om het geheel in een class `Calculator` in te kapselen zodat de calculator eenvoudig kan worden (her)gebruikt. Zie `infix_calculator.cpp`.

9

Implementaties van een stack

Een stack kan geïmplementeerd worden met behulp van een array maar ook met behulp van een gelinkte lijst. Elke methode heeft zijn eigen voor- en nadelen. Door beide applicaties over te erven van een gemeenschappelijke abstracte base class kunnen deze implementaties in een applicatie eenvoudig uitgewisseld worden. Ik bespreek in dit hoofdstuk beide implementaties.

9.1 Stack met behulp van een array

Inhoud van de file stackarray.cppm:

```
export module stackarray;
export import stack;
import std;
using namespace std;

export template <typename T> class Stack_with_array: public ↩
 ↪ Stack<T> {
public:
    explicit Stack_with_array(size_t size);
    void push(const T& t) override;
    void pop() override;
    const T& top() const override;
    bool empty() const override;
    bool full() const override;
private:
```

```
    unique_ptr<T[]> a; // pointer naar de array
    size_t s; // size van a (max aantal elementen op de stack)
    size_t i; // index in a van eerste vrije plaats op de stack
};

template <typename T> Stack_with_array<T>::Stack_with_array(size_t ↩
 ↪ size): a{nullptr}, s{size}, i{0} {
    if (s == 0)
        throw std::domain_error {"Stack size should be >0\n"};
    a = make_unique<T[]>(s);
}

template <typename T> void Stack_with_array<T>::push(const T& t) {
    if (full())
        throw std::overflow_error {"Can't push on an full stack\n"};
    a[i++] = t;
}

template <typename T> void Stack_with_array<T>::pop() {
    if (empty())
        throw std::underflow_error {"Can't pop from an empty ↩
 ↪ stack\n"};
    --i;
}

template <typename T> const T& Stack_with_array<T>::top() const {
    if (empty())
        throw std::underflow_error {"Can't top from an empty ↩
 ↪ stack\n"};
    return a[i - 1];
}

template <typename T> bool Stack_with_array<T>::empty() const {
    return i == 0;
}

template <typename T> bool Stack_with_array<T>::full() const {
    return i == s;
}
```

Er is hier gebruik gemaakt van een `unique_ptr<T[]>` en de functie `make_unique<T[]>` om de array a te benaderen en aan te vragen. Dit zorgt ervoor dat we niet kunnen vergeten om het geheugen weer vrij te geven met **`delete`**`[]`. De `unique_ptr<T[]>` zorgt ervoor dat het geheugen automatisch wordt vrijgegeven als het `Stack_with_array` object wordt vernietigd.

Het programma `stackarray.cpp` om deze implementatie te testen:

```
import std;
import stackarray;
using namespace std;

int main() {
    try {
        Stack_with_array<char> s {32};
        char c;
        println("Type een tekst en sluit af met .");
        cin.get(c);
        while (c != '.') {
            s.push(c);
            cin.get(c);
        }
        while (!s.empty()) {
            print("{}", s.top());
            s.pop();
        }
    } catch (const exception& e) {
        println(cerr, "Error: {}", e.what());
    }
}
```

9.2 Stack met behulp van een gelinkte lijst

Inhoud van de file `stacklist.cppm`:

```
export module stacklist;
export import stack;
import std;
using namespace std;

export template <typename T> class Stack_with_list: public ↩
 ↪ Stack<T> {
```

```
public:
    Stack_with_list();
    ~Stack_with_list() override;
    void push(const T& t) override;
    void pop() override;
    const T& top() const override;
    bool empty() const override;
    bool full() const override;
private:
    class Node {
    public:
        Node(const T& t, Node* n);
        T data;
        Node* next;
    };
    Node* p; // pointer naar de Node aan de top van de stack
};

template <typename T> Stack_with_list<T>::Stack_with_list(): ↩
 ↪ p(nullptr) {
}

template <typename T> Stack_with_list<T>::~Stack_with_list() {
    while (!empty())
        pop();
}

template <typename T> void Stack_with_list<T>::push(const T& t) {
    p = new Node(t, p);
}

template <typename T> void Stack_with_list<T>::pop() {
    if (empty())
        throw std::underflow_error {"Can't pop from an empty ↩
 ↪ stack\n"};
    Node* old(p);
    p = p->next;
    delete old;
}

template <typename T> const T& Stack_with_list<T>::top() const {
```

```
    if (empty())
        throw std::underflow_error {"Can't top from an empty ↩
 ↪ stack\n"};
    return p->data;
}

template <typename T> bool Stack_with_list<T>::empty() const {
    return p == nullptr;
}

template <typename T> bool Stack_with_list<T>::full() const {
    return false;
}

template <typename T> Stack_with_list<T>::Node::Node(const T& t, ↩
 ↪ Node* n):
         data(t), next(n) {
}
```

Er is hier bewust géén gebruik gemaakt van een smart pointer (shared_ptr<Node>) maar van een gewone pointer (Node*). Dit is gedaan omdat elke Node een pointer naar de volgende Node moet bevatten. Een shared_ptr<Node> neemt op een 64-bits pc twee keer zoveel ruimte in als een Node*. Het gebruik van een gewone pointer bespaart dus geheugen. Maar we moeten nu wel zelf zorgen dat het geheugen dat aangemaakt wordt in de functie pop ook weer vrijgeven wordt in de functie pop.

Het programma stacklist.cpp om deze implementatie te testen:

```
import std;
import stacklist;
using namespace std;

int main() {
    try {
        Stack_with_list<char> s;
        char c;
        println("Type een tekst en sluit af met .");
        cin.get(c);
        while (c != '.') {
            s.push(c);
            cin.get(c);
        }
```

```
        while (!s.empty()) {
            print("{}", s.top());
            s.pop();
        }
    } catch (const exception& e) {
        println(cerr, "Error: {}", e.what());
    }
}
```

9.3 Stack met array versus stack met gelinkte lijst

Beide implementaties `Stack_with_array` en `Stack_with_list` hebben voor- en nadelen. Het geheugengebruik van `Stack_with_array` is vast nadat de stack is aangemaakt (bij het aanmaken moet je de maximale grootte opgeven) terwijl het geheugengebruik van `Stack_with_list` dynamisch is. De `Stack_with_array` is erg snel omdat bij de push, top en pop bewerkingen slechts maximaal een array indexering en het bijwerken van een index nodig zijn. De `Stack_with_list` is veel trager omdat bij de push en pop bewerkingen een system call (respectievelijk **`new`** en **`delete`** wordt aangeroepen. De orde van de executietijd voor alle bewerkingen is natuurlijk bij beide implementaties wel gelijk: $O(1)$. De `Stack_with_array` gebruikt, als de stack niet vol is, meer geheugen dan eigenlijk nodig is. Het niet gevulde deel van de array is, op dat moment, eigenlijk niet nodig en kunnen we dus beschouwen als overhead. De `Stack_with_list` gebruikt, als de stack niet leeg is, meer geheugen dan eigenlijk nodig is. Voor elk element dat op de stack staat moet ook een `Node*` in het geheugen worden opgeslagen. De overhead is afhankelijk van de grootte van het data-element en de grootte van een `Node*`. Op een 64 bits machine (waar een pointer acht bytes groot is) heeft een `Stack_with_list` van **`char`**'s (die één byte groot zijn) dus een behoorlijke overhead. Een `Stack_with_list` met objecten die 1 Kbyte groot zijn heeft slechts een kleine overhead. We kunnen dus concluderen dat een `Stack_with_array` minder overhead heeft voor een stack met kleine objecten en dat een `Stack_with_list` minder overhead heeft voor een stack met grote objecten.

9.4 Dynamisch kiezen voor een bepaald type stack

We kunnen de keuze voor het type stack ook pas tijdens run time maken! Dit is een uitstekend voorbeeld van het gebruik van polymorfisme, zie `stack.cpp`.

```
import std;
```

```
import stacklist;
import stackarray;
using namespace std;

int main() {
    try {
        unique_ptr<Stack<char>> s {nullptr};

        print("Welke stack wil je gebruiken (l = list, a = array): ↩
 ↪ ");
        char c;
        do {
            cin.get(c);
            if (c == 'l' || c == 'L') {
                s = move(make_unique<Stack_with_list<char>>());
            }
            else if (c == 'a' || c == 'A') {
                print("Hoeveel elementen wil je gebruiken: ");
                int i;
                cin >> i;
                s = move(make_unique<Stack_with_array<char>>(i));
            }
        } while (s == nullptr);

        println("Type een tekst en sluit af met .");
        cin.get(c);
        while (c != '.') {
            s->push(c);
            cin.get(c);
        }
        while (!s->empty()) {
            print("{}", s->top());
            s->pop();
        }
    } catch (const exception& e) {
        println(cerr, "Error: {}", e.what());
    }
}
```

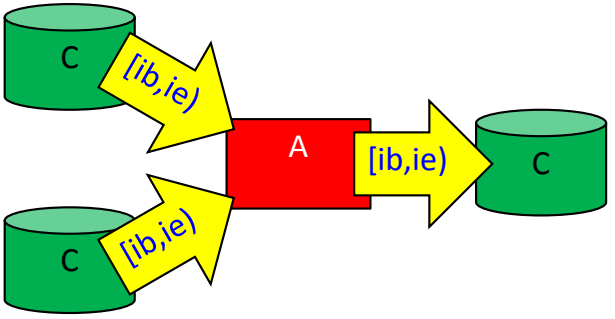
In de praktijk komt het niet vaak voor dat je de gebruiker van een applicatie de implementatie van de stack laat kiezen. Het kan echter wel voorkomen dat bij het vertalen van het

programma nog niet bekend is hoe groot de objecten zijn die op de stack geplaatst moeten worden. In dat geval is het handig dat je dynamisch (tijdens run time), afhankelijk van de grootte van de objecten, kunt kiezen welke implementatie van een stack gebruikt wordt om de overhead te minimaliseren.

10

De standaard C++ library

De standaard C++ library is zeer uitgebreid en beslaat 17 hoofdstukken en ruim 1400 pagina's in de C++-standaad [10]. In het vervolg van dit dictaat kijken we naar *containers*, *iteratoren* en *algoritmen*. De in paragraaf 7.2 geïntroduceerde datastructuren worden in de standaard C++ library *containers* genoemd. Een *iterator* kan gebruikt worden om de elementen in een container te benaderen. Je kunt in C++ met een iterator door een container heenlopen zoals je in C met een pointer door een array kunt lopen. Een iterator kan 'wijzen' naar een element uit een container op dezelfde wijze als een pointer kan wijzen naar een element uit een array. De algoritmen uit de standaard library werken op de data die door twee iteratoren wordt aangegeven. Deze twee iteratoren wijzen samen een opeenvolgende hoeveelheid elementen (het *bereik*) aan waar het betreffende algoritme op wordt uitgevoerd. Een iterator wijst naar het eerste data-element van het bereik en de andere iterator wijst naar de locatie net voorbij het laatste element van het bereik. De iteratoren zorgen ervoor dat de algoritmen generiek gehouden kunnen worden en dus op verschillende soorten containers uitgevoerd kunnen worden. In figuur 10.1 is de samenhang tussen containers, iteratoren en algoritmen weergegeven. Dit figuur laat de meest complexe situatie zien: een algoritme dat data uit twee verschillende containers leest en het resultaat in een derde container wegschrijft. Een algoritme kan ook data uit slechts één container lezen en het resultaat in dezelfde container wegschrijven.



Figuur 10.1: De samenhang tussen containers, iteratoren en algoritmen. A = Algoritme, C = Container, i = iterator, b = begin, e = end, [ib,ie) = bereik van ib tot ie.

10.1 Voorbeeldprogramma met `std::string`

Het voorbeeldprogramma `std_string.cpp` laat zien dat je met een iterator door een `std::string` kunt lopen op dezelfde manier als dat je met een pointer door een C-string kunt lopen. Een C-string is een array van karakters waarbij het einde van de string wordt aangegeven door het karakter NUL (`'\0'`).

Met een pointer kun je als volgt door een C-string heenlopen en de karakters één voor één benaderen:

```
char naam[] {"Roy"};
for (const char* p {naam}; *p != '\0'; ++p) {
    print("{} ", *p);
}
```

Je kunt ook het keyword **auto** gebruiken om de compiler zelf het type van de pointer te laten bepalen.

```
char naam[] {"Roy"};
for (const auto* p {naam}; *p != '\0'; ++p) {
    print("{} ", *p);
}
```

Met een iterator kun je als volgt door een `std::string` heenlopen en de karakters één voor één benaderen:

```
string naam {"Harry"};
for (string::const_iterator i {naam.cbegin()}; i != ↩
↪ naam.cend(); ++i) {
```

```
        print("{} ", *i);
    }
```

Met behulp van een `string::const_iterator` kun je de karakters uit de `string` alleen lezen. De memberfunctie `cbegin()` geeft een `const_iterator` terug naar het eerste karakter van de string. De memberfunctie `cend()` geeft een `const_iterator` terug net na het laatste karakter van de string. Met behulp van een `string::iterator` kun je de karakters in de `string` ook overschrijven. De memberfunctie `begin()` geeft een `iterator` terug naar het eerste karakter van de string. De memberfunctie `end()` geeft een `iterator` terug net na het laatste element van de string.

Je kunt ook het keyword **`auto`** gebruiken om de compiler zelf het type van de iterator te laten bepalen.

```
string naam {"Harry"};
for (auto i {naam.cbegin()}; i != naam.cend(); ++i) {
    print("{} ", *i);
}
```

10.2 Overzicht van de standaard library

De standaard C++ library [9] is zeer uitgebreid. In het vervolg van dit dictaat kijken we naar containers (zie hoofdstuk 11), iteratoren (zie hoofdstuk 12) en algoritmen (zie hoofdstuk 13). De library bevat daarnaast ook faciliteiten voor: strings, localization, numerics, input/output, regular expressions, threads en nog veel meer.

Alle informatie is beschikbaar op `https://en.cppreference.com/w/cpp`. Als je niet kunt vinden wat je zoekt in de standaard library dan kun je ook gebruik maken van vele open source libraries. Een overzicht vind je op `https://en.cppreference.com/w/cpp/links/libs`.

Ook in *C++ Annotations* [1] is veel informatie te vinden, onder andere over library-faciliteiten voor: strings [1, §5], smart pointers [1, §18.3, §18.4 en §18.5], containers [1, §12], iterators [1, §18.2], algorithms [1, §19], input/output [1, §6], regular expressions [1, §18.8], threads [1, §20], coroutines [1, §24], randomization [1, §18.9], date and time [1, §4.2] en filesystem [1, §4.3].

11

Containers

De standaard C++ library bevat verschillende containers. Een container waarmee je in paragraaf 3.7 al hebt kennisgemaakt is de `std::vector`. Elke container is als een template gedefinieerd zodat het type van de elementen die in de container opgeslagen kunnen worden, gevarieerd kan worden. Zo kun je b.v. een vector `vi` aanmaken met integers (`vector<int> vi;`) maar ook een vector vb met breuken (`vector<Breuk> vb;`). Een container zorgt voor zijn eigen geheugenbeheer. Om dit mogelijk te maken maakt de container kopietjes van de elementen die in de container worden opgeslagen. Als je dit niet wilt, moet je een containers met pointers aanmaken. In dat geval moet je er zelf voor zorgen dat de elementen waar de pointers in de container naar wijzen blijven bestaan zolang de pointers nog gebruikt kunnen worden. Alle elementen van een container moeten van hetzelfde type zijn. Dit kan overigens wel een polymorf pointertype zijn, bijvoorbeeld `vector<Hond*>`. In dit geval moeten er pointers in de containers worden opgeslagen omdat een object niet polymorf kan zijn, zie paragraaf 18.6.

De containers in de standaard C++ library zijn in te delen in sequentiële containers en associatieve containers. In een sequentiële container blijft de volgorde van de elementen ongewijzigd. In een associatieve container wordt de volgorde van de elementen bepaald door de container zelf. Hierdoor kan de zoekfunctie voor een associatieve container sneller werken dan voor een sequentiële container.

De standaard C++ library bevat de volgende sequentiële containers:

- `string`
- `array`

- `vector`
- `forward_list` (singly linked list)
- `list` (doubly linked list)
- `deque` (double ended queue, had eigenlijk double ended vector moeten heten)
- `bitset` (wordt in dit dictaat verder niet besproken)

Een gewone C-array (`[]`) is ook een sequentiële container.

Bepaalde sequentiële containers kunnen met behulp van zogenoemde adapter classes van een bepaalde interface worden voorzien. De standaard C++ library bevat de volgende adapters:

- `stack`
- `queue`
- `priority_queue`

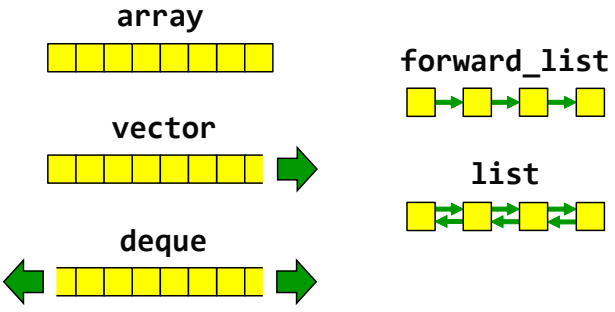
De standaard C++ library bevat de volgende associatieve containers:

- `set`
- `multiset`
- `map`
- `multimap`
- `unordered_set`
- `unordered_multiset`
- `unordered_map`
- `unordered_multimap`

11.1 Sequentiële containers

In figuur 11.1 zijn de belangrijkste sequentiële containers conceptueel weergegeven. Een `array` heeft een vaste grootte die als tweede template argument wordt opgegeven bij het definiëren van het type van de `array`, zie paragraaf 3.6. De grootte van de `array` kan niet wijzigen tijdens het uitvoeren van het programma. De `array` is dus een statische datastructuur. Alle overige sequentiële containers kunnen wel groeien en krimpen tijdens het uitvoeren van het programma en zijn dus dynamische datastructuren. Een `vector` kan groeien en krimpen aan de achterkant, door middel van de memberfuncties `push_back` en `pop_back`, zie paragraaf 3.7. Een deque kan net zoals een `vector` groeien en krimpen aan de achterkant maar kan bovendien groeien en krimpen aan de voorkant, door middel van

de memberfuncties `push_front` en `pop_front`. Een `forward_list` bestaat uit enkelvoudig gelinkte losse elementen en kan dus alleen van voor naar achter worden doorlopen. De `forward_list` kan achter elke willekeurige plek groeien en krimpen met behulp van de memberfuncties `push_front`, `pop_front`, `insert_after` en `erase_after`. Een `list` bestaat uit dubbelgelinkte losse elementen en kan dus zowel van voor naar achter als van achter naar voor doorlopen worden. De `list` kan op elke willekeurige plek groeien en krimpen met behulp van de memberfuncties `push_back`, `pop_back`, `push_front`, `pop_front`, `insert` en `erase`.



Figuur 11.1: Overzicht van de belangrijkste sequentiële containers.

Tabel 11.1: Overzicht van de belangrijkste sequentiële containers.

container	random access		element toevoegen en verwijderen					
			front		back		random	
`array`	`operator[]` `at()`	$O(1)$	–		–		–	
`vector`	`operator[]` `at()`	$O(1)$	–		`push_back` `pop_back`	$O(1)$	`insert` `erase`	$O(n)$
`deque`	`operator[]` `at()`	$O(1)$	`push_front` `pop_front`	$O(1)$	`push_back` `pop_back`	$O(1)$	`insert` `erase`	$O(n)$
`forward_list`	–		`push_front` `pop_front`	$O(1)$	–		`insert_after` `erase_after`	$O(1)$
`list`	–		`push_front` `pop_front`	$O(1)$	`push_back` `pop_back`	$O(1)$	`insert` `erase`	$O(1)$

In tabel 11.1 staat gegeven welke memberfuncties beschikbaar zijn voor de verschillende sequentiële containers en van welke orde hun executietijd is. De `array`, `vector` en deque bieden als enige random access op index aan. Elk element in deze containers kan via zijn index met een constante snelheid ($O(1)$) benaderd worden. Dit kan op twee manieren: met

`operator[]` en met de memberfunctie `at`. De `operator[]` voert *géén* controle uit op de als argument meegegeven index. Als de index buiten het bereik van de container ligt, dan wordt gewoon de geheugenplaats gebruikt waar het element met deze index zich zou bevinden als het had bestaan. Als je geluk hebt, dan levert dit een foutmelding van het operating system op omdat je een geheugenplaats gebruikt die niet van jou is. Als je pech hebt, wordt een willekeurige andere variabele uit je programma gebruikt! De memberfunctie `at` controleert de als argument meegegeven index wel. Als de index buiten het bereik van de container ligt, wordt de standaard exception `out_of_range` gegooid.

Bij de deque, `forward_list` en `list` is het mogelijk om aan de voorkant elementen toe te voegen aan, of te verwijderen uit, de container in een constante tijd ($O(1)$). Voor een deque geldt dat de `push_front` operatie *gemiddeld* van de orde 1 is maar als je net pech hebt en de capaciteit[129] van de deque moet vergroot worden dan kan die ene `push_front` wel eens langer duren. De executietijd van de `pop_front` van een deque is gemiddeld $O(1)$ omdat een deque zijn capaciteit uit zichzelf mag verlagen[130] . Als er aan de voorkant een element wordt toegevoegd aan een deque, dan krijgt dat nieuwe element de index 0 en het element dat voor de toevoeging index 0 had krijgt index 1 enz. Als aan de voorkant een element wordt verwijderd uit een deque, dan krijgt het element dat voor de verwijdering index 1 had index 0 enz.

Bij de `vector`, deque en `list` is het mogelijk om aan de achterkant elementen toe te voegen aan, of te verwijderen uit de container in een constante tijd ($O(1)$). Voor een `vector` en een deque geldt dat de `push_back` operatie *gemiddeld* van de orde 1 is maar als je net pech hebt en de capaciteit van de container moet vergroot, dan kan die ene `push_back` wel eens lang duren ($O(n)$). Een executietijd van de `pop_back` van een `vector` is altijd $O(1)$ omdat een vector nooit zijn capaciteit uit zichzelf verlaagd. Een executietijd van de `pop_back` van een deque is gemiddeld $O(1)$ omdat een deque zijn capaciteit uit zichzelf mag verlagen[130].

Alle sequentiële containers uit tabel 11.1 behalve de `array` hebben memberfuncties om op een willekeurige plaats elementen toe te voegen of te verwijderen. Bij een `vector` en een deque zijn deze operaties van de orde n omdat alle elementen vanaf de plaats waar moet worden ingevoegd tot aan het einde van de container opgeschoven moeten worden. De elementen moeten opgeschoven worden omdat de volgorde van de elementen in een sequentiële container niet mag wijzigen. Gemiddeld moeten $\frac{1}{2}\,n$ elementen opgeschoven

[129] Het aantal elementen waarvoor, op een bepaald moment, ruimte gereserveerd is wordt de capaciteit (Engels: capacity) van de container genoemd. Het aantal gevulde elementen in de container wordt de grootte (Engels: size) genoemd. Er geldt altijd size $\leq$ capacity.

[130] Dit is, volgens de C++-standaad, implementatie afhankelijk.

worden dus is de executietijd van deze operatie $O(n)$. Voor een `forward_list` en een `list` zijn deze operaties van de orde 1 omdat bij het invoegen of verwijderen van een element alleen enkele pointers verlegd moeten worden. In een enkel gelinkte lijst (`forward_list`) is het alleen mogelijk om een element toe te voegen of te verwijderen *achter* een willekeurige plaats. Dit komt omdat je bij het invoegen of verwijderen van een element de pointer in het element *voor* de plaats waar je wilt invoegen of verwijderen moet kunnen aanpassen. Dit element kan in een enkelvoudige lijst niet in een constante tijd ($O(1)$) benaderd worden. Bij een dubbel gelinkte lijst kun je wel een element toevoegen of verwijderen *op* een willekeurige plaats. Via de achterwaartse pointer kan het element voor de plaats waar ingevoegd moet worden snel ($O(1)$) worden bereikt.

11.1.1 Voorbeeldprogramma met `std::vector`

Dit voorbeeldprogramma `std_vector.cpp` laat zien:

- hoe een standaard vector gevuld kan worden;
- hoe door (een deel van) de vector heengelopen kan worden door middel van indexering[131];
- hoe door (een deel van) de vector heengelopen kan worden door middel van een *iterator*[132];
- hoe door de vector heengelopen kan worden door middel van een *range-based for*;
- hoe het gemiddelde van een rij getallen (opgeslagen in een vector) bepaald kan worden.

```
import std;
using namespace std;

// Verdubbel elementen van een vector<int> door middel van ↩
 ↪ indexering.
void verdubbel1(vector<int>& vec) {
    for (vector<int>::size_type index {0}; index != vec.size(); ↩
 ↪ ++index) {
        vec[index] *= 2;
    }
}
```

[131] Je kunt hierbij gebruik maken van een `decltype` om het type van de index variabele te definiëren. Zie paragraaf 15.11.

[132] Je kunt hierbij gebruik maken van een zogenoemde *auto-typed* variabele. Deze variabele wordt geïnitialiseerd met `begin()`. De memberfunctie geeft een `iterator` naar het eerste element in de container terug.

```
// Verdubbel elementen van een vector<int> door middel van ↩
 ↪ indexering met decltype.
void verdubbel2(vector<int>& vec) {
    for (decltype(vec.size()) index {0}; index != vec.size(); ↩
 ↪ ++index) {
        vec[index] *= 2;
    }
}

// Verdubbel elementen van een vector<int> door middel van iterator.
void verdubbel3(vector<int>& vec) {
    for (vector<int>::iterator iter {vec.begin()}; iter != ↩
 ↪ vec.end(); ++iter) {
        *iter *= 2;
    }
}

// Verdubbel elementen van een vector<int> door middel van ↩
 ↪ iterator met auto.
void verdubbel4(vector<int>& vec) {
    for (auto iter {vec.begin()}; iter != vec.end(); ++iter) {
        *iter *= 2;
    }
}

// Verdubbel elementen van een vector<int> door middel van ↩
 ↪ range-based for.
void verdubbel5(vector<int>& vec) {
    for (auto& elm: vec) {
        elm *= 2;
    }
}

// Berekenen van het gemiddelde door middel van iterator met auto.
double gem1(const vector<int>& vec) {
    if (vec.empty()) {
        return 0;
    }
    double som {0.0};
    for (auto iter {vec.cbegin()}; iter != vec.cend(); ++iter) {
        som += *iter;
```

```
    }
    return som / vec.size();
}

// Berekenen van het gemiddelde door middel van van range-based for.
double gem2(const vector<int>& vec) {
    if (vec.empty()) {
        return 0;
    }
    double som {0.0};
    for (auto elm: vec) {
        som += elm;
    }
    return som / vec.size();
}

int main() {
    vector<int> v;
    int i;
    print("Geef een aantal getallen (afgesloten door een 0): ");
    cin >> i;
    while (i != 0) {
        v.push_back(i);
        cin >> i;
    }
    println("v = {}", v);
    verdubbel1(v);
    println("v = {}", v);
    verdubbel2(v);
    println("v = {}", v);
    verdubbel3(v);
    println("v = {}", v);
    verdubbel4(v);
    println("v = {}", v);
    verdubbel5(v);
    println("v = {}", v);
    println("Het gemiddelde is: {:.3f}", gem1(v));
    println("Het gemiddelde is: {:.3f}", gem2(v));
}
```

In het bovenstaande programma worden maar liefst vijf verschillende manieren gedemonstreerd om de elementen in een `vector<int>` te verdubbelen. De eerste twee implementaties `verdubbel1` en `verdubbel2` gebruiken indexering om de elementen in de vector te benaderen. In plaats van de expressie `vec[index]` had de expressie `vec.at(index)` gebruikt kunnen worden zodat de waarde van de index gecontroleerd wordt op geldigheid. In dit geval vond ik dit niet zinvol omdat de expressie in een herhalingslus wordt gebruikt waarbij alleen geldige waarden van de index worden gebruikt. De volgende twee implementaties `verdubbel3` en `verdubbel4` gebruiken een iterator om de elementen in de vector te benaderen. De laatste implementatie `verdubbel5` gebruikt een range-based for om de elementen in de vector te benaderen en is het eenvoudigst.

11.2 Generieke verdubbel functie voor containers

Geïnspireerd door het bovenstaande voorbeeld zou je een generieke `verdubbel` functie kunnen bedenken waarmee de elementen uit elke willekeurige container gevuld met getallen verdubbeld kunnen worden. We kunnen voor deze generieke functie geen indexering gebruiken omdat indexering niet bij alle containers gebruikt kan worden. We zouden wel, als volgt, gebruik kunnen maken van een iterator, zie `generieke_verdubbel-container.cpp`:

```
template<typename C> void verdubbel(C& c)
requires Number<typename C::value_type> && !Const<C> && ↩
 ↪ output_iterator<typename C::iterator, typename C::value_type> {
    for (auto iter {c.begin()}; iter != c.end(); ++iter) {
        *iter *= 2;
    }
}
```

Bij het definiëren van deze generieke functie is gebruik gemaakt van het zelfgedefinieerde concept `Number<T>`, het zelfgedefinieerde concept `Const<T>`, zie paragraaf 17.4 en van het standaard concept `output_iterator<I, T>`, zie hoofdstuk 12 en paragraaf 3.10. Dit zorgt ervoor dat deze functietemplate alleen gebruikt kan worden voor containers die gevuld zijn met getallen en die aangepast kunnen worden.

Het is, ook in dit geval, eenvoudiger om gebruik te maken van een range-based for, zie `generieke_verdubbel-range-based-for.cpp`:

```
template<typename C> void verdubbel(C& c)
requires Number<typename C::value_type> && !Const<C> && ↩
 ↪ output_iterator<typename C::iterator, typename C::value_type> {
```

```
    for (auto& e: c) {
        e *= 2;
    }
}
```

In de bovenstaande functie is een **auto**& gebruikt om een onnodig kopietje te voorkomen. We weten immers niet hoe groot de elementen in de container zijn.

Bovenstaande twee generieke `verdubbel` functies verdubbelen altijd de waarden van de gehele container. Als we ook een gedeelte van een container willen kunnen verdubbelen met de generieke `verdubbel` functie, dan moeten we de functie twee iteratoren meegeven, zie generieke_verdubbel-iterator.cpp:

```
template <typename Iter> void verdubbel(Iter begin, Iter end)
requires Number<typename Iter::value_type> && ↩
 ↪ output_iterator<Iter, typename Iter::value_type> {
    for (Iter iter {begin}; iter != end; ++iter) {
        *iter *= 2;
    }
}
```

De eerste iterator `begin` wijst naar het eerste element dat moet worden verdubbeld en de tweede iterator `end` wijst net voorbij het laatste element dat moet worden verdubbeld. Als de iteratoren `i1` en `i2` aan deze functie worden meegegeven, dan worden alle elementen vanaf `i1` tot (en dus *niet* tot en met) `i2` verdubbeld. In de wiskunde zouden we dat noteren als het halfopen interval `[i1,i2)`. In C++ wordt zo'n halfopen interval van twee iteratoren een *bereik* (Engels: *range*) genoemd. Algoritmen uit de standaard C++ library gebruiken twee iteratoren als parameters zodat ze ook op een deel van een container uitgevoerd kunnen worden. Daarnaast zijn in namespace `ranges` algoritmen beschikbaar die op een hele container uitgevoerd kunnen worden en waarbij de container als argument doorgegeven kan worden, zie paragraaf 12.6.

De bovenstaande generieke `verdubbel` kan gebruikt worden om de vector `v` volledig te verdubbelen:

```
verdubbel(v.begin(), v.end());
```

Dezelfde functie kan ook gebruikt worden om alle elementen van de vector `v` te verdubbelen *behalve* de eerste en de laatste:

```
verdubbel(v.begin() + 1, v.end() - 1);
```

11.3 Container adapters

Bepaalde sequentiële containers kunnen met behulp van zogenoemde adapter classes van een bepaalde interface worden voorzien. De standaard C++ library bevat de volgende adapters:

- `stack`
- `queue`
- `priority_queue`

Bij het aanmaken van een `stack` kun je aangeven (als tweede template argument) of voor de implementatie een `vector`, deque of `list` gebruikt moet worden:

```
stack<int, vector<int>> s1; // stack implemented with vector
stack<int, deque<int>> s2;  // stack implemented with deque
stack<int, list<int>> s3;   // stack implemented with list
stack<int> s4;              // using deque by default
```

Als je niets opgeeft, wordt (default) een deque gebruikt. Merk op dat tijdens compile time bepaald wordt hoe de `stack` geïmplementeerd wordt. Je kunt dat niet dynamisch (tijdens het uitvoeren van het programma) kiezen zoals bij de zelfgemaakte stack die in paragraaf 9.4 besproken is.

Bij het aanmaken van een queue kun je aangeven (als tweede template argument) of voor de implementatie een deque (default) of `list` gebruikt moet worden. Je kunt een queue niet implementeren met een `vector` omdat een `vector` slechts aan een kant kan groeien en krimpen, zie figuur 11.1, terwijl bij een queue aan één kant elementen moeten worden toegevoegd en aan de *andere* kant elementen moeten worden verwijderd.

Bij het aanmaken van een `priority_queue` kun je aangeven (als tweede template argument) of voor de implementatie een `vector` (default) of deque gebruikt moet worden. Een `priority_queue` wordt geïmplementeerd als een binary heap[133] waarbij gebruik gemaakt wordt van indexering. Je kunt een `priority_queue` dus niet implementeren met een `list` omdat de elementen in een `list` niet door middel van indexering benaderd kunnen worden, zie tabel 11.1.

[133] Een binary heap is een gedeeltelijk gesorteerde binaire boom die opgeslagen kan worden in een array omdat de boom altijd volledig gevuld is. Er wordt geen gebruik gemaakt van pointers om de kinderen van een node te vinden maar van indexering. Zie http://en.wikipedia.org/wiki/Binary_heap.

11.3.1 Voorbeeldprogramma met std::stack

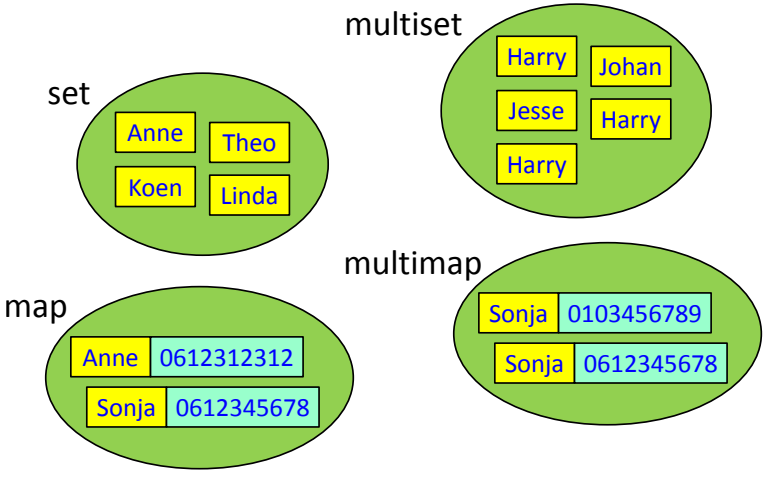
Het onderstaande voorbeeldprogramma std_balanced.cpp is identiek aan het programma uit paragraaf 8.1 maar in plaats van de zelfgemaakte Stack_with_list wordt nu de standaard stack gebruikt.

```
// Gebruik een stack voor het controleren van de haakjes
// Invoer afsluiten met .

import std;
using namespace std;

int main() {
    stack<char> s;
    char c;
    println("Type een expressie met haakjes () [] of {{}} en sluit af met .");
    cin.get(c);
    while (c != '.') {
        if (c == '(' || c == '{' || c == '[') {
            s.push(c);
        }
        else {
            if (c == ')' || c == '}' || c == ']') {
                if (s.empty()) {
                    println("Fout {} bijbehorend haakje openen ontbreekt.", c);
                }
                else {
                    char d {s.top()};
                    s.pop();
                    if ((d == '(' && c != ')') || (d == '{' && c != '}') || (d == '[' && c != ']')) {
                        println("Fout {} bijbehorend haakje openen ontbreekt.", c);
                    }
                }
            }
        }
        cin.get(c);
    }
    while (!s.empty()) {
```

```
            char d {s.top()};
            s.pop();
            println("Fout {} bijbehorend haakje sluiten ontbreekt.", d);
        }
}
```

11.4 Associatieve containers

In een sequentiële container blijft de volgorde van de elementen ongewijzigd, zie paragraaf 11.1. In een associatieve container wordt de volgorde van de elementen bepaald door de container zelf. Hierdoor kan de zoekfunctie voor een associatieve container sneller werken dan voor een sequentiële container. In figuur 11.2 zijn de belangrijkste associatieve containers conceptueel weergegeven.



Figuur 11.2: Overzicht van de belangrijkste associatieve containers.

In een `set` kan elk element slechts één maal voorkomen, net zoals in een wiskundige verzameling (Engels: set). De waarden die in een set zijn opgeslagen worden vaak sleutels (Engels: keys) genoemd. De `set` in figuur 11.2 bevat de roepnamen van mijn kinderen. Voor de meeste mensen geldt dat de roepnamen van hun kinderen een, mogelijk lege, `set` vormen, dat wil zeggen dat elk van hun kinderen een unieke roepnaam heeft. Er zijn echter wel uitzonderingen. Ken jij zo'n uitzondering?[134]. Deze sleutels worden zodanig in de `set` opgeslagen dat je snel kunt opzoeken of een sleutel zich in de `set` bevindt.

[134] Zie http://nl.wikipedia.org/wiki/Flodder

In een multiset kunnen sleutels meerdere keren voorkomen, in de wiskunde wordt dit ook wel een zak (Engels: bag) genoemd. De multiset in figuur 11.2 bevat de roepnamen van een bepaalde groep mensen. De roepnaam Harry komt zoals je ziet drie maal voor in deze multiset. Deze sleutels worden zodanig in de multiset opgeslagen dat je snel kunt opzoeken hoe vaak een sleutel zich in de multiset bevindt.

Een map is vergelijkbaar met een set maar in plaats van uit losse sleutels bestaan de elementen in een map uit zogenoemde sleutel/waarde-paren (Engels: key-value pairs). Aan elke sleutel is een waarde gekoppeld. Hiervoor wordt het standaard type pair gebruikt. Dit type heeft twee public datamembers first en second. Er is een standaardfunctie make_pair beschikbaar waarmee een paar eenvoudig kan worden aangemaakt. In de map in figuur 11.2 is de sleutel bijvoorbeeld een roepnaam en de waarde een telefoonnummer. Elke sleutel in een map moet uniek zijn. Deze sleutel/waarde-paren worden zodanig in de map opgeslagen dat je de bij een sleutel behorende waarde snel kunt opzoeken. Hierdoor is het niet mogelijk om een sleutel te wijzigen zolang deze zich in de map bevindt. De bij de sleutel behorende waarde kan echter wel eenvoudig worden gewijzigd. De elementen in een map zijn dus van het type pair<**const** Key, Value> waarbij Key het type van de sleutel is en Value het type van de aan deze sleutel gekoppelde waarde is. De map heeft een indexeringsoperator (**operator**[]) waardoor de sleutel als een index kan worden gebruikt om de bijbehorende waarde te benaderen.

Een map met een string als sleutel en een **unsigned int** als waarde kun je als volgt aanmaken:

```
map<string, unsigned int> telefoonnummer;
```

Met behulp van indexering kun je nu het telefoonnummer[135] van Sonja in deze map zetten.

```
telefoonnummer["Sonja"] = 612345678;
```

Het telefoonnummer van Sonja kan nu, wederom met behulp van indexering, als volgt worden opgezocht en afgedrukt.

```
println("{}: {}", "Sonja", telefoonnummer["Sonja"]);
```

Je ziet dat een map gebruikt kan worden alsof het een soort array (of vector) is waarbij de index geen integer hoeft te zijn. Dit soort arrays worden in veel andere programmeertalen

[135] De nul waarmee het telefoonnummer begint, wordt niet opgeslagen. Als je per ongeluk toch de nul invoert telefoonnummer["Sonja"] = 0612345678;, dan interpreteert de C++-compiler dit als een getal in het octale talstelsel en dat levert in dit geval een foutmelding op omdat een octaal getal alleen de cijfers 0 tot en met 7 mag bevatten.

(waaronder JavaScript en PHP) *associatieve arrays* genoemd. In Python worden dit soort arrays *dictionaries* genoemd.

In een `multimap` mag een sleutel wel meerdere keren voorkomen. In de `multimap` in figuur 11.2 komt de sleutel `Sonja` twee keer voor. Dit kan al snel tot verwarring leiden: gaat het hier om dezelfde `Sonja` die twee telefoonnummers heeft, mobiel en vast, of gaat het hier om twee verschillende `Sonja`'s? Omdat er meerdere verschillende waarden met dezelfde sleutel geassocieerd kunnen zijn ondersteund de `multimap` geen indexering. Om die reden wordt de container `map` in de praktijk vaker gebruikt dan de container `multimap`.

De standaard library bevat twee verschillende implementaties van deze vier associatieve containers. De containers `set`, `multiset`, `map` en `multimap` worden geïmplementeerd met behulp van een zoekboom (Engels: search tree). Zoals je in tabel 7.2 kunt zien zijn de executietijden van de operaties `insert`, `erase` en `find` voor een zoekboom allemaal $O(\log n)$. De sleutels worden van laag naar hoog gesorteerd (met behulp van de standaard comperatorfunctie `less`) opgeslagen. Als je een andere sorteervolgorde wilt, dan kun je zelf een andere comperatorfunctie opgeven. De containers `unordered_set`, `unordered_multiset`, `unordered_map` en `unordered_multimap` worden geïmplementeerd met behulp van een hashtabel (Engels: hash table). Zoals je in tabel 7.2 kunt zien zijn de executietijden van de operaties `insert`, `erase` en `find` voor een hashtabel allemaal $O(1)$. Als de hashtabel echter te vol raakt, of als een slechte hashfunctie wordt gebruikt, dan kunnen deze executietijden $O(n)$ worden. De in de standaard opgenomen `unsorted_...` containers groeien automatisch als het aantal elementen in de container gedeeld door het aantal plaatsen in de hashtabel (de zogenoemde `load_factor`) groter wordt dan de `max_load_factor`, die een default waarde heeft van 1.0. De sleutels worden niet gesorteerd opgeslagen (vandaar de namen `unsorted_...`). Er moet een hashfunctie voor het als sleutel gebruikte type beschikbaar zijn. Voor een aantal types zoals `int` en `string` is een standaard hashfunctie aanwezig in de library. Als je een zelf gedefinieerd type als sleutel wilt gebruiken dan moet je zelf een hashfunctie definiëren[136], zie paragraaf 20.7.

11.4.1 Voorbeeldprogramma met `std::set`

In een `set` mag elk element ten hoogste één keer voorkomen. In het onderstaande voorbeeldprogramma `set.cpp` wordt het gebruik van een `set` gedemonstreerd. Een `set` heeft, onder andere, de volgende memberfuncties:

[136] Zie `http://en.cppreference.com/w/cpp/utility/hash`

- `insert`; Deze memberfunctie kan gebruikt worden om een sleutel aan de `set` toe te voegen. De sleutels worden automatisch op de goede plaats ingevoegd. Het returntype van deze memberfunctie is `pair<iterator, bool>`. De `iterator` geeft de plek aan waar ingevoegd is en de **`bool`** geeft aan of het invoegen gelukt is. Het invoegen mislukt als de sleutel die ingevoegd moet worden al in de `set` aanwezig is.
- `erase`; Deze memberfunctie kan gebruikt worden om een sleutel uit de `set` te verwijderen. De sleutel die je als argument meegeeft wordt opgezocht en uit de set verwijderd. De returnwaarde waarde geeft aan hoeveel sleutels verwijderd zijn. Bij een `set` kan dit uitsluitend 0 of 1 zijn. Een `multiset` heeft dezelfde memberfunctie en in dat geval kan de returnwaarde wel groter dan 1 zijn. Je kunt ook een iterator, die refereert naar een sleutel in de `set`, als argument meegeven. De sleutel waar de iterator naar refereert wordt dan uit de `set` verwijderd. Je kunt ook twee iteratoren, die refereren naar sleutels in de `set`, als argument meegeven. Alle sleutels vanaf degene waar de eerste iterator naar refereert tot (dus *niet* tot en met) degene waar de tweede iterator naar refereert worden uit de `set` verwijderd.
- `find`; Deze memberfunctie kan gebruikt worden om een sleutel in de `set` te zoeken. De returnwaarde is een iterator die refereert naar de gevonden sleutel of is gelijk aan de iterator `end()` van de `set` als de sleutel niet gevonden kan worden.
- `count`. Deze memberfunctie kan gebruikt worden om het aantal maal te tellen dat een sleutel in de `set` voorkomt. Bij een `set` kan dit uitsluitend 0 of 1 zijn. Een `multiset` heeft dezelfde memberfunctie en in dat geval kan de returnwaarde wel groter dan 1 zijn.

```
import std;
using namespace std;

int main() {
    set<string> docenten {"Ron", "Daniël", "Roy", "Harry"};
    docenten.insert("Elles");
    println("De set bevat: {:n:s}", docenten);
    auto result {docenten.insert("Harry")};
    if (!result.second)
        println("1 Harry is genoeg.");
    println("Er is {} Ron.", docenten.count("Ron"));
    docenten.erase("Harry");
    println("De set bevat: {:n:s}", docenten);
}
```

De uitvoer van dit programma:

```
De set bevat: Daniël Elles Harry Ron Roy
1 Harry is genoeg.
Er is 1 Ron.
De set bevat: Daniël Elles Ron Roy
```

De inhoud van een `set` kan eenvoudig afgedrukt worden met behulp van de standaard functie `println`. In de format specifier `{:n:s}` zorgt de `:n` ervoor dat er geen accolades om de set heen geprint worden[137] en de `:s` zorgt ervoor dat de elementen van de set als `strings` zonder aanhalingstekens eromheen worden afgedrukt[138]. Als de standaard format specifier `{}` wordt gebruikt, dan wordt de eerste regel van de output van het bovenstaande programma:

```
De set bevat: "Daniël", "Elles", "Harry", "Ron", "Roy"
```

11.4.2 Voorbeeldprogramma met `std::multiset` (bag)

Vergelijk het onderstaande programma `multiset.cpp` met het vorige en let op de verschillen tussen een `set` en een `multiset`. Een `mutiset` heeft dezelfde memberfuncties als een `set`. De memberfunctie `insert` geeft echter alleen een iterator als returnwaarde in plaats van een `pair` van een iterator en een **`bool`**. Het invoegen in een multiset kan namelijk niet mislukken. Als bij de memberfunctie `erase` een sleutel als argument wordt weergegeven, dan worden alle sleutels met die specifieke waarde uit de `multiset` verwijderd. Als je slechts één sleutel wil verwijderen dan moet je een iterator als argument aan `erase` meegeven.

```
import std;
using namespace std;

int main() {
    multiset<string> docenten {"Ron", "Daniël", "Roy", "Harry"};
    docenten.insert("Elles");
    println("De multiset bevat: {:n:s}", docenten);
    docenten.insert("Harry");
    println("De multiset bevat: {:n:s}", docenten);
    docenten.insert("Harry");
    println("De multiset bevat: {:n:s}", docenten);
    println("Er zijn {} Harry's.", docenten.count("Harry"));
```

[137] Zie https://en.cppreference.com/w/cpp/utility/format/range_formatter.html.

[138] Zie https://en.cppreference.com/w/cpp/utility/format/spec.html.

```
    docenten.erase(docenten.find("Harry"));
    println("De multiset bevat: {:n:s}", docenten);
    docenten.erase("Harry");
    println("De multiset bevat: {:n:s}", docenten);
    docenten.erase(docenten.find("Elles"));
    println("De multiset bevat: {:n:s}", docenten);
    docenten.erase(docenten.find("Ron"), docenten.end());
    println("De multiset bevat: {:n:s}", docenten);
}
```

De uitvoer van dit programma:

```
De bag bevat: Daniël Elles Harry Ron Roy
De bag bevat: Daniël Elles Harry Harry Ron Roy
De bag bevat: Daniël Elles Harry Harry Harry Ron Roy
Er zijn 3 Harry's.
De bag bevat: Daniël Elles Harry Harry Ron Roy
De bag bevat: Daniël Elles Ron Roy
De bag bevat: Daniël Ron Roy
De bag bevat: Daniël
```

11.4.3 Voorbeeldprogramma met `std::unordered_set`

Vergelijk het onderstaande programma unordered_set.cpp met het programma uit paragraaf 11.4.1 en let op de verschillen tussen een set en een unordered_set.

```
import std;
using namespace std;

int main() {
    unordered_set<string> mensen {"Ron", "Daniël", "Roy", "Harry"};
    mensen.insert("Elles");
    println("De set bevat: {:n:s}", mensen);
    auto result {mensen.insert("Harry")};
    if (!result.second)
        println("1 Harry is genoeg.");
    println("Er is {} Elles.", mensen.count("Elles"));
    mensen.erase("Harry");
    println("De set bevat: {:n:s}", mensen);
    println("De hash table heeft {} buckets.", ↩
 ↪ mensen.bucket_count());
```

```
    println("Elke bucket bevat gemiddeld {} elementen.", ↩
 ↪ mensen.load_factor());
    println("De maximale load_factor is {}.", ↩
 ↪ mensen.max_load_factor());
    println("We voegen nu nog 21 namen toe.");
    mensen.insert({"Amir", "Martijn", "Ashwin", "Nicholas", ↩
 ↪ "Deniz", "Mitchel", "Sietse", "Robin", "Bálint", "Dewin", ↩
 ↪ "Connor", "Nino", "Alexander", "Jennifer", "Leendert", ↩
 ↪ "Maurice", "Thomas", "Bas", "Wesley", "Finley", "Jasper"});
    println("De set bevat: {:n:s}", mensen);
    println("De hash table heeft {} buckets.", ↩
 ↪ mensen.bucket_count());
    println("Elke bucket bevat gemiddeld {} elementen.", ↩
 ↪ mensen.load_factor());
}
```

Een mogelijke[139] uitvoer van dit programma:

```
De set bevat: Elles, Harry, Roy, Daniël, Ron
1 Harry is genoeg.
Er is 1 Elles.
De set bevat: Elles, Roy, Daniël, Ron
De hash table heeft 13 buckets.
Elke bucket bevat gemiddeld 0.30769232 elementen.
De maximale load_factor is 1.
We voegen nu nog 21 namen toe.
De set bevat: Jasper, Wesley, Thomas, Maurice, Leendert, ↩
 ↪ Alexander, Nino, Connor, Ron, Martijn, Deniz, Bas, Daniël, ↩
 ↪ Bálint, Roy, Sietse, Elles, Amir, Ashwin, Jennifer, Nicholas, ↩
 ↪ Mitchel, Finley, Dewin, Robin
De hash table heeft 29 buckets.
Elke bucket bevat gemiddeld 0.86206895 elementen.
```

11.4.4 Voorbeeldprogramma met `std::map`

Het onderstaande voorbeeldprogramma `count_words.cpp` gebruikt een `map` om de woordfrequentie te tellen. Van een aantal belangrijke C/C++ keywords wordt het aantal maal dat ze voorkomen afgedrukt. Een `map` heeft dezelfde memberfuncties als een `set`, zie para-

[139] De volgorde van de elemeneten in de set kan variëren afhankelijk van de gebruikte compiler. In dit geval is GCC 15.2 gebruikt.

graaf 11.4.1. Je kunt daarnaast ook gebruik maken van indexering waarbij de sleutel als index wordt gebruikt.

```
import std;
using namespace std;

int main() {
    string w;
    map<string, int> freq;
    print("Geef filenaam: ");
    cin >> w;
    ifstream fin {w};
    while (fin >> w) {
        ++freq[w];
    }
    println("Woordfrequentie: {}", freq);
    println("Enkele belangrijke keywords:");
    println("if: {}", freq["if"]);
    println("else: {}", freq["else"]);
    println("for: {}", freq["for"]);
    println("while: {}", freq["while"]);
}
```

12

Iteratoren

In figuur 10.1 heb je gezien dat iteratoren de verbinding vormen tussen containers en algoritmen. In het ideale geval zou je op deze manier elk algoritme op elke container kunnen toepassen. Als je daar echter even over nadenkt, kom je tot de conclusie dat dit praktisch niet mogelijk is. Sommige algoritmen maken namelijk gebruik van bepaalde bewerkingen die niet door alle containers worden ondersteund. Het standaard `sort` algoritme maakt bijvoorbeeld gebruik van indexering en die bewerking wordt niet ondersteund door een `list`. Als je probeert om `sort` uit te voeren op een `list`, dan geeft de compiler een foutmelding.

```
list l {12, 18, 6};
sort(l.begin(), l.end()); // error: no match for 'operator-'
```

Om deze reden zijn er meerder soorten iteratoren in de C++-standaad gedefinieerd. Elke soort ondersteund bepaalde bewerkingen. Een container levert (Engels: provides) een specifieke iteratorsoort en elk algoritme heeft bepaalde iteratorsoorten nodig (Engels: requires). Op deze manier kun je een algoritme op elke container toepassen die een iterator levert die de door het algoritme benodigde bewerkingen ondersteund. Elk object dat de operaties van een bepaalde soort iterator ondersteund is te gebruiken als zo'n soort iterator. Een iteratorsoort is dus niets anders dan een afspraak (of concept). Er zijn in de C++-standaad vijf iteratorsoorten gedefinieerd:

- Een *input iterator* kan gebruikt worden om uit een container te lezen. Deze iteratorsoort ondersteund de volgende operaties:
  - met de * operator kan de data waar de iterator naar refereert worden *gelezen*.
  - met de ++ operator kan de iterator op het *volgende* element worden geplaatst.

- met de == en != operator kan bepaald worden of de iterator naar *hetzelfde* element refereert als een andere iterator.[140]

Een input iterator is een zogenoemde single pass iterator. Dat wil zeggen dat nadat de ++ operator is gebruikt de vorige data verdwenen kan zijn. Een typisch voorbeeld van een container met een input iterator is een keyboard. Als een ingetypt karakter is ingelezen (met de * operator) en als de iterator eenmaal naar het volgende karakter is verplaatst (met de ++ operator), dan kan dit karakter niet nogmaals ingelezen worden.

- Een *output iterator* kan gebruikt worden om in een container te schrijven. Deze iteratorsoort ondersteund de volgende operaties:
  - met de * operator kan het element waar de iterator naar refereert worden *overschreven*.
  - met de ++ operator kan de iterator op het *volgende* element worden geplaatst.

  Een output iterator is een zogenoemde single pass iterator. Dat wil zeggen dat nadat de ++ operator is gebruikt het vorige element niet nogmaals overschreven kan worden. Een typisch voorbeeld van een container met een output iterator is een lijnprinter. Als een karakter geprint is (met de * operator) en als de iterator eenmaal naar het volgende positie is verplaatst (met de ++ operator), dan kan het karakter in de vorige positie niet meer overschreven worden.
- Een *forward iterator* kan gebruikt worden om de elementen van een container van voor naar achter te doorlopen (zowel om te schrijven als om te lezen). Deze iteratorsoort ondersteund de volgende operaties:
  - met de * operator kan de data waar de iterator naar refereert worden *gelezen* of *overschreven*.
  - met de ++ operator kan de iterator op het *volgende* element worden geplaatst.
  - met de == en != operator kan bepaald worden of de iterator naar *hetzelfde* element refereert als een andere iterator.

  Een forward iterator is een zogenoemde multi pass iterator. Dat wil zeggen dat de iterator gebruikt kan worden om de container meerdere malen te doorlopen. Als het einde van de container is bereikt dan kan de iterator weer op het eerste element worden

[140] De expressie `i1 == i2` is waar als iterator `i1` en iterator `i2` beide refereren naar dezelfde positie in een container. Let op het verschil met `*i1 == *i2`. De expressie `*i1 == *i2` is waar als de data waar iterator `i1` naar refereert gelijk is aan de data waar iterator `i2` naar refereert. Als `i1 == i2` waar is, dan is `*i1 == *i2` ook altijd waar maar als `*i1 == *i2` waar is, dan hoeft `i1 == i2` niet waar te zijn. De iteratoren `i1` en `i2` kunnen in dat geval namelijk refereren naar twee verschillende elementen die dezelfde data bevatten.

geplaatst en de container opnieuw worden doorlopen.[141] Een typisch voorbeeld van een container met een forward iterator is een `forward_list`.

- Een *bidirectional iterator* kan alles wat een forward iterator kan maar kan bovendien gebruikt worden om de elementen van een container van achter naar voor te doorlopen (zowel om te schrijven als om te lezen). Deze iteratorsoort ondersteunt alle operaties die de forward iterator ondersteunt plus:
  - met de `--` operator kan de iterator op het *vorige* element worden geplaatst.

  Een bidirectional iterator is vanzelfsprekend een multi pass iterator. Een typisch voorbeeld van een container met een bidirectional iterator is een `list`.
- Een *random access* iterator kan alles wat een bidirectional iterator kan maar kan bovendien gebruikt worden om de elementen van een container random access te doorlopen (zowel om te schrijven als om te lezen). Deze iteratorsoort ondersteunt alle operaties die de bidirectional iterator ondersteunt plus:
  - met de `+=`, `-=`, `+` of `-` operator kan de iterator een aantal elementen verder of terug worden geplaatst.
  - met de `>`, `>=`, `<` of `<=` operator kunnen twee iteratoren met elkaar vergeleken worden. De expressie `i > j` is waar als iterator `i` en iterator `j` beide naar een element in dezelfde container refereren én als de iterator `i` ongelijk is aan iterator `j` maar wel gelijk gemaakt geworden aan `j` door `i` een aantal elementen verder te plaatsen.
  - met de `-` operator kan het aantal elementen dat zich tussen twee iteratoren bevindt worden bepaald. Voorwaarde daarbij wel is dat bij de bewerking `i - j` iterator `i` en iterator `j` beide naar een element in dezelfde container refereren. Als `i >= j` is, dan is `i - j` positief maar als `i < j` is, dan is `i - j` negatief.
  - met de `[index]` operator kan een element dat zich `index` elementen verder bevindt worden benaderd.[142] [143]

[141] In feite kan een forward iterator naar elke onthouden iterator worden verplaatst en daarvandaan weer verder stappen in voorwaartse richting.

[142] Met behulp van de operator `[]` kun je indexering toepassing vanaf het element dat wordt aangewezen door een iterator. De index mag ook negatief zijn. Als de iterator `i` naar een bepaald element wijst dan kun je met de expressie `i[2]` twee elementen verder lezen of schrijven zonder dat je de iterator `i` hoeft te verplaatsen. Met de expressie `i[-1]` kun je het voorgaande element lezen of schrijven. Je bent er als programmeur zelf verantwoordelijk voor dat de elementen die je met indexering leest of schrijft bestaan anders is het gedrag niet gedefinieerd.

[143] De expressie `i[n]` is gelijk aan de expressie `*(i + n)`.

Een random access iterator is vanzelfsprekend een multi pass iterator. Een typisch voorbeeld van een container met een random access iterator is een `vector`.

In tabel 12.1 is aangegeven welke bewerkingen door de verschillende iteratorsoorten ondersteund worden.

Tabel 12.1: Standaard iteratorsoorten en de bewerkingen die ze leveren.

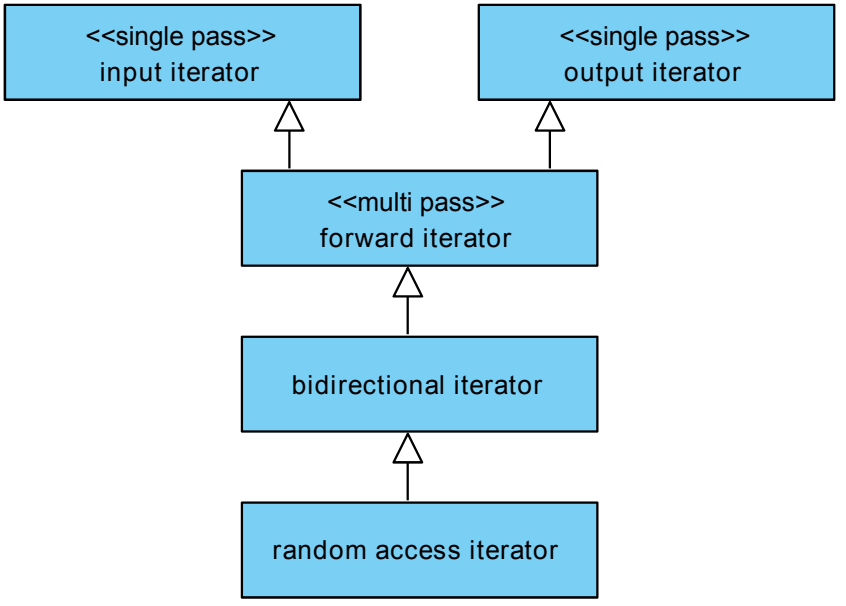
iteratorsoort	* (lezen)	* (schrijven)	++	== en !=	--	+=, -=, +, -, >, >=, <, <= en []
input (single pass)	✓	✗	✓	✓	✗	✗
output (single pass)	✗	✓	✓	✗	✗	✗
forward	✓	✓	✓	✓	✗	✗
bidirectional	✓	✓	✓	✓	✓	✗
random access	✓	✓	✓	✓	✓	✓

Je ziet in tabel 12.1 dat een forward iterator ook alle operaties levert die door een input en een output iterator worden geleverd. Een bidirectional iterator levert ook alle operaties die door een forward iterator worden geleverd. Een random access iterator levert ook alle operaties die door een bidirectional iterator worden geleverd. Deze relatie tussen de verschillende iterator concepten is weergegeven in figuur 12.1.

Als een algoritme bijvoorbeeld een input iterator nodig heeft om zijn werk te kunnen doen, dan kun je dit algoritme *ook* uitvoeren op een container die een bidirectional iterator levert. Het is namelijk niet erg als de betreffende iterator meer operaties levert dan het algoritme nodig heeft. Als een algoritme bijvoorbeeld een random access iterator nodig heeft om zijn werk te kunnen doen, dan kun je dit algoritme *niet* uitvoeren op een container die een bidirectional iterator levert. Het dan namelijk zo dat de betreffende iterator minder operaties levert dan het algoritme nodig heeft. Voor alle containers is in de C++-standaad aangegeven welke soort iterator ze leveren:

- `forward_list`, `unordered_set`, `unordered_multiset`, `unordered_map` en `unordered_multimap` leveren een forward iterator;
- `list`, `set`, `multiset`, `map` en `multimap` leveren een bidirectional iterator;
- `vector` en deque leveren een random access iterator.

Voor alle algoritmen is in de C++-standaad aangegeven welke soort iterator ze nodig hebben. Bijvoorbeeld:



Figuur 12.1: De relaties tussen de verschillende iterator concepten.

- `find` heeft een input iterator nodig;
- `copy` heeft een input en een output iterator nodig;
- `replace` heeft een bidirectional iterator nodig;
- `sort` heeft een random access iterator nodig.

Op deze manier kun je dus zien op welke containers een bepaald algoritme uitgevoerd kan worden. We kunnen nu begrijpen waarom de code die aan het begin van deze paragraaf is gegeven een compilerfout oplevert. Het `sort` algoritme heeft een random access iterator nodig maar de container `list` levert slechts een (minder krachtige) bidirectional iterator.

12.1 Voorbeeldprogramma's met verschillende iterator soorten

De implementatie van een algoritme kan afhankelijk zijn van de soort iterator die beschikbaar is. Als voorbeeld bekijken we een functie die een iterator teruggeeft die refereert naar het middelste element in een als argument meegegeven bereik [begin, end). Als het bereik een even aantal elementen bevat dan is er niet echt een middelste element (er zijn dan twee

elementen die samen het midden van het bereik vormen), in dit geval wordt een iterator die refereert naar het laatste van deze twee elementen teruggegeven.

Als er random access iteratoren beschikbaar zijn, dan is de implementatie van deze functie erg eenvoudig, zie `find_upper_middle_random_access.cpp`:

```
template <random_access_iterator I>
I find_upper_middle(I begin, I end) {
    return begin + (end - begin)/2;
}
```

Door een *constrained* template, zie paragraaf 3.10 te gebruiken kan deze functie aangeroepen worden voor elke container die random access iteratoren levert, bijvoorbeeld een `vector`. De executietijd van deze functie is $O(1)$. Als deze functie echter aangeroepen wordt om het midden van een `list` te vinden dan geeft de compiler een foutmelding:

```
list l = {12, 18, 6};
println("{}", *find_upper_middle(l.begin(), l.end())); // error: no ↩
 ↪ matching function ...
```

Als er bidirectional iteratoren beschikbaar zijn, dan kan de functie als volgt geïmplementeerd worden, zie `find_upper_middle_bidirectional.cpp`:

```
template <bidirectional_iterator I>
I find_upper_middle(I begin, I end) {
    while (begin != end) {
        --end;
        if (begin != end) {
            ++begin;
        }
    }
    return begin;
}
```

Door een *constrained* template te gebruiken kan deze functie aangeroepen worden voor elke container die bidirectional iteratoren levert[144], bijvoorbeeld een `list`. De executietijd van deze functie is $O(n)$. Als deze functie echter aangeroepen wordt om het midden van een `forward_list` te vinden dan geeft de compiler een foutmelding:

[144] Deze functie kan ook aanroepen worden voor elke container die random access iteratoren levert, maar in dat geval kan beter de template `find_upper_middle<random_access_iterator I>` aangeroepen worden want die is sneller.

```
forward_list l {12, 18, 6};
println("{}", *find_upper_middle(l.begin(), l.end())); // error: no ↩
 ↪ matching function ...
```

Als er forward iteratoren beschikbaar zijn, dan kan de functie als volgt geïmplementeerd worden, zie find_upper_middle_forward.cpp:

```
template <forward_iterator I>
I find_upper_middle(I begin, I end) {
    I i {begin};
    while (begin != end) {
        ++begin;
        if (begin != end) {
            ++begin;
            ++i;
        }
    }
    return i;
}
```

Door een *constrained* template te gebruiken kan deze functie aangeroepen worden voor elke container die forward iteratoren levert[145], bijvoorbeeld een `forward_list`. De executietijd van deze functie is $O(n)$.

Doordat er gebruik gemaakt is van *constrained* templates zal de compiler zelf de meeste efficiënte versie van `find_upper_middle` kiezen afhankelijk van de gebruikte iterator[146].

Zie find_upper_middle.cpp:

```
import std;
using namespace std;

template <forward_iterator I>
I find_upper_middle(I begin, I end) {
    println("forward iterator used");
```

[145] Deze functie kan ook aanroepen worden voor elke container die bidirectional of random access iteratoren levert, maar in dat geval kan beter de functie `find_upper_middle<bidirectional_iterator I>` respectievelijk `find_upper_middle<random_access_iterator I>` aangeroepen worden want die zijn sneller.

[146] Als je een C++versie gebruikt van voor de C++17 standaard, dan kun je hetzelfde bereiken door gebruik te maken van *iterator tags*, zie find_upper_middle_with_iterator_tags. Als je een C++versie gebruikt van voor de C++20 standaard, dan kun je hetzelfde bereiken door gebruik te maken van *compile time if's*, zie paragraaf 15.13 en find_upper_middle_with_compile_time_if.cpp.

```
    I i {begin};
    while (begin != end) {
        ++begin;
        if (begin != end) {
            ++begin;
            ++i;
        }
    }
    return i;
}

template <bidirectional_iterator I>
I find_upper_middle(I begin, I end) {
    println("bidirectional iterator used");
    while (begin != end) {
        --end;
        if (begin != end) {
            ++begin;
        }
    }
    return begin;
}

template <random_access_iterator I>
I find_upper_middle(I begin, I end) {
    println("random access iterator used");
    return begin + (end - begin)/2;
}

int main() {
    forward_list fl {1, 2};
    println("find_upper_middle called on forward_list");
    if (*find_upper_middle(fl.begin(), fl.end()) != 2) {
        println(cerr, "Test 1 failed!");
        return 1;
    }
    list l {1, 2, 3};
    println("find_upper_middle called on list");
    if (*find_upper_middle(l.begin(), l.end()) != 2) {
        println(cerr, "Test 2 failed!");
        return 2;
```

```
    }
    vector v {1, 2, 3, 4};
    println("find_upper_middle called on vector");
    if (*find_upper_middle(v.begin(), v.end()) != 3) {
        println(cerr, "Test 3 failed!");
        return 3;
    }
    println("All tests passed!");
}
```

De uitvoer van het bovenstaande programma is:

```
find_upper_middle called on forward_list
forward iterator used
find_upper_middle called on list
bidirectional iterator used
find_upper_middle called on vector
random access iterator used
All tests passed!
```

Je ziet dat de compiler de meest effectieve variant van `find_upper_middle` aanroept, afhankelijk van het bij de container beschikbare iterator soort.

12.2 Reverse-iteratoren

Alle containers hebben de memberfuncties `begin()` en `end()` waarmee een iterator naar het eerste respectievelijk één voorbij het laatste element opgevraagd kan worden. Deze iteratoren kunnen gebruikt worden om alle elementen van een container van voor naar achter te doorlopen, zie paragraaf 10.1. De functies `cbegin()` en `cend()` zijn vergelijkbaar maar geven in plaats van een `iterator` een `const_iterator` terug die alleen gebruikt kan worden om elementen uit de container te lezen. Als je bijvoorbeeld de eerste 5 in een vector met integers wilt vervangen door een 7 dan kan dit als volgt:

```
    auto eerste_vijf {find(v.begin(), v.end(), 5)};
    if (eerste_vijf != v.end()) {
        *eerste_vijf = 7;
    }
```

Het standaard algoritme `find` heeft drie parameters: een iterator naar de plaats waar begonnen wordt met zoeken, een iterator naar de plaatst tot waar gezocht wordt en de

waarde die gezocht moet worden. Dit algoritme geeft een iterator terug die refereert naar het eerste element met de gezochte waarde. Als zo'n element niet gevonden is dan wordt de iterator die als tweede argument is meegegeven teruggegeven.

Alle tot nu toe behandelde containers behalve de `forward_list` hebben bovendien de memberfuncties `rbegin()` en `rend()` waarmee een zogenoemde *reverse-iterator* naar het laatste respectievelijk één voor het eerste element opgevraagd kan worden. Deze iteratoren kunnen gebruikt worden om alle elementen van een container van achter naar voor te doorlopen. De functies `crbegin()` en `crend()` zijn vergelijkbaar maar geven in plaats van een `reverse_iterator` een `const_reverse_iterator` terug die alleen gebruikt kan worden om elementen uit de container te lezen. Als je bijvoorbeeld de laatste 5 in een vector met integers wilt vervangen door een 7 dan kan dit als volgt:

```
auto laatste_vijf {find(v.rbegin(), v.rend(), 5)};
if (laatste_vijf != v.rend()) {
    *laatste_vijf = 7;
}
```

Het volledige voorbeeldprogramma `reverse_iterator.cpp` kun je online vinden.

12.3 Insert-iteratoren

Als een iterator `itr` refereert naar een element en je schrijft met behulp van de dereference operator `*itr = ...`, dan wordt het betreffende element overschreven. Soms wil je vóór een bepaalde plaats in een container die wordt aangewezen met een iterator elementen toevoegen. Dit kan met behulp van zogenoemde insert-iteratoren.

Er zijn drie functietemplates beschikbaar waarmee zulke insert-iteratoren eenvoudig aangemaakt kunnen worden:

- `back_inserter(c)`; Deze functie levert een zogenoemde `back_insert_iterator` waarmee elementen toegevoegd kunnen worden aan het einde van de container `c` door te schrijven via deze iterator met behulp van de `*` operator. De container `c` moet een sequentiële container zijn en beschikken over een memberfunctie push_back, zie tabel 11.1. Dus `c` moet een `vector`, deque of `list` zijn.
- `front_inserter(c)`; Deze functie levert een zogenoemde `front_insert_iterator` waarmee elementen toegevoegd kunnen worden aan het begin van de container `c` door te schrijven via deze iterator met behulp van de `*` operator. De container `c` moet

een sequentiële container zijn en beschikken over een memberfunctie `push_front`, zie tabel 11.1. Dus `c` moet een `deque`, `forward_list` of `list` zijn.

- `inserter(c, i)`; Deze functie levert een zogenoemde `insert_iterator` waarmee elementen toegevoegd kunnen worden voor het element dat aangewezen wordt door de iterator `i` in de container `c` door te schrijven via deze iterator met behulp van de operator `*`.[147] De container `c` moet een container zijn die beschikt over een memberfunctie `insert`. Dus `c` kan elke, tot nu toe besproken, container zijn behalve een `array`. Als `c` een associatieve container is dan is het gek dat je, door middel van de parameter `i`, aan moet geven waar de elementen toegevoegd moeten worden omdat een associatieve container zelf bepaalt waar de elementen geplaatst worden. In dit geval moet je `i` opvatten als een hint.

12.4 Stream-iteratoren

Met behulp van stream-iteratoren kunnen C++ streams worden gelezen of geschreven met behulp van algoritmen uit de C++ standaard library. Er zijn twee stream-iteratoren beschikbaar:

- `istream_iterator<T>` kan gebruikt worden om elementen van type `T` via deze iterator uit een input stream te lezen. Dit is een input iterator. De betreffende input stream moet als argument aan de constructor van de `istream_iterator` worden meegegeven. De default constructor levert een `istream_iterator` die gebruikt kan worden om te bepalen of het einde van de input stream bereikt is.
- `ostream_iterator<T>` kan gebruikt worden om elementen van type `T` via deze iterator naar een output stream te schrijven. Dit is een output iterator. De betreffende output stream moet als argument aan de constructor van de `ostream_iterator` worden meegegeven. Als tweede argument kan een C-string worden meegegeven die na elk element wordt afgedrukt.

[147] Door deze schrijfoperatie wordt in feite de memberfunctie `insert` van de container `c` aangeroepen met de iterator `i` als argument. De executietijd van deze schrijfoperatie is dus gelijk aan de executietijd van de betreffende `insert` memberfunctie. Zie voor sequentiële containers tabel 11.1 en voor de associatieve containers paragraaf 11.4. Elke schrijfoperatie via een `insert_iterator` in bijvoorbeeld een `vector` heeft dus een executietijd van de $O(n)$.

12.5 Voorbeeldprogramma's met iteratoren

In het onderstaande voorbeeldprogramma stream_iterator.cpp wordt het gebruik van stream- en insert-iteratoren gedemonstreerd:

```
import std;
using namespace std;

int main() {
    vector<int> rij;
    ifstream fin {"getallen_ongesorteerd.txt"};
    if (!fin)
        return 1; // kan fin niet openen
    istream_iterator<int> iin {fin}, einde;
    copy(iin, einde, back_inserter(rij));
    ranges::sort(rij);
    ofstream fout {"getallen_gesorteerd.txt"};
    if (!fout)
        return 2; // kan fout niet openen
    ostream_iterator<int> iout {fout, " "};
    ranges::copy(rij, iout);
}
```

De integers uit het bestand getallen_ongesorteerd.txt worden ingelezen in de vector genaamd rij, gesorteerd en vervolgens weggeschreven naar het bestand getallen_gesorteerd.txt. Het gebruikte algoritme copy heeft drie parameters: een iterator naar de plaats waar begonnen wordt met lezen, een iterator naar de plaatst tot waar gelezen wordt en een iterator naar de plaats waar begonnen wordt met schrijven.

In het onderstaande voorbeeldprogramma iteratoren.cpp wordt het gebruik van stream-, insert- en reverse-iteratoren gedemonstreerd:

```
import std;
using namespace std;

int main() {
    set<string> namen;
    ifstream inf {"namen.txt"};
    if (inf) {
        copy(istream_iterator<string> {inf}, ↩
 ↪ istream_iterator<string> {}, inserter(namen, namen.begin()));
```

```
        copy(namen.crbegin(), namen.crend(), ↩
 ↪ ostream_iterator<string> {cout, "\n"});
    }
}
```

De strings uit het bestand `namen.txt` worden ingelezen in de `set` genaamd `namen`. Vervolgens wordt de inhoud van deze `set` van achter naar voor op het scherm afgedrukt, één string per regel. In tegenstelling tot het vorige programma worden in dit programma geen variabelen van het type `istream_iterator` en `ostream_iterator` aangemaakt maar worden de constructors van deze classes aangeroepen op de plaats waar deze objecten nodig zijn (als argumenten van de functie copy).

12.6 Ranges

TODO: nog toevoegen!

13

Algoritmen

De C++-standaard bevat vele generiek algoritmen. Een generiek algoritme werkt op één of meerdere bereiken [i1, i2), zie figuur 10.1. Een bereik wordt aangegeven door twee iteratoren i1 en i2. Het bereik loopt van i1 *tot* (dus niet tot en met) i2. In de wiskunde zouden we dit noteren als het halfopen interval [i1, i2).

Daarnaast zijn in de namespace `ranges` ook algoritmen beschikbaar die werken op het gehele bereik van een container (dus van `c.begin()` tot `c.end()`), zie paragraaf 12.6.

Een generiek algoritme werkt op meerdere container types. Sommige containers hebben voor bepaalde bewerkingen specifieke memberfuncties:

- omdat het generieke algoritme krachtigere iteratoren nodig heeft dan de container kan leveren;
- omdat de specifieke memberfunctie sneller is dan het generieke algoritme.

Je hebt in hoofdstuk 12 gezien dat het `sort` algoritme niet gebruikt kan worden om een `list` te sorteren. Om deze reden heeft een `list` de memberfunctie `sort` waarmee de lijst wel gesorteerd kan worden.

```
list l {12, 18, 6};
l.sort(); // Ok
```

Het `find` algoritme voert een lineaire zoekactie uit en de executietijd is dus $O(n)$. Je hebt in paragraaf 11.4 gezien dat de gewone associatieve containers een zoekactie kunnen uitvoeren met executietijd $O(\log n)$ en de ongesorteerde associatieve containers een zoekactie kunnen uitvoeren met executietijd $O(1)$, als ze niet te vol zijn en als een goede hashfunctie gebruikt

wordt. Om deze reden hebben alle associatieve containers de memberfunctie `find` waarmee snel in de container gezocht kan worden.

```
set priem {2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, ↩
 ↪ 53, 59, 61, 67, 71, 73, 79, 83, 89, 97};
auto i1 {ranges::find(priem, 91)}; // O(n)
auto i2 {priem.find(91)}; // O(log n)
```

Kijk dus voordat je een generiek algoritme gebruikt bij een bepaalde container altijd eerst of er een specifieke memberfunctie bij die container beschikbaar is.
Zie `algorithm_vs_memberfunction.cpp`.

Er zijn heel veel algoritmen beschikbaar in de C++ library. In dit dictaat worden ze niet allemaal behandeld, kijk voor een volledig overzicht op http://en.cppreference.com/w/cpp/algorithm. De algoritmen uit de standaard C++ library gebruiken twee iteratoren als parameters zodat ze ook op een deel van een container uitgevoerd kunnen worden. Daarnaast zijn in namespace `ranges` algoritmen beschikbaar die op een hele container uitgevoerd kunnen worden (dus van `c.begin()` tot `c.end()`) en waarbij de container `c` als argument doorgegeven kan worden, zie paragraaf 12.6.

13.1 Zoeken, tellen, testen, bewerken en vergelijken

De volgende algoritmen kun je gebruiken om een bepaald element te zoeken in een range[148]:

- `find`; Zoek het eerste element met een bepaalde waarde.
- `find_if`; Zoek het eerste element dat voldoet aan een bepaalde voorwaarde.[149]
- `find_if_not`; Zoek het eerste element dat niet voldoet aan een bepaalde voorwaarde.
- `find_first_of`[150]; Zoek het eerste element met een waarde uit een bepaalde range van waarden.
- `adjacent_find`; Zoek de eerste twee opeenvolgende elementen met dezelfde waarde.

De volgende algoritmen kun je gebruiken om bepaalde elementen te tellen in een range:

[148] Vanaf hier wordt in dit dictaat het Engelse woord *range* gebruikt in plaats van het Nederlandse woord *bereik*. Een range kan worden doorgegeven als twee iteratoren (dat geldt voor algoritmen in de namespace `std` of als een gehele container (dat geldt voor algoritmen in de namespace `ranges`).

[149] Bijvoorbeeld: zoek het eerste element > 0.

[150] Bedenk zelf waarom er geen algoritmen nodig zijn om het laatste in plaats van het eerste element te zoeken. Kijk indien nodig ter inspiratie in paragraaf 12.2.

- `count`; Tel alle elementen met een bepaalde waarde.
- `count_if`; Tel alle elementen die voldoen aan een bepaalde voorwaarde.

De volgende algoritmen kun je gebruiken om te testen of bepaalde elementen voorkomen in een range:

- `all_of`; Test of alle elementen aan een bepaalde voorwaarde voldoen.
- `any_of`; Test of ten minste één element aan een bepaalde voorwaarde voldoet.
- `none_of`; Test of géén enkel element aan een bepaalde voorwaarde voldoet.

Deze `..._of` algoritmen zijn sneller dan een `count_if` omdat ze niet altijd alle elementen hoeven af te gaan. Bijvoorbeeld om te testen of géén enkel element aan een bepaalde voorwaarde voldoet zou je de elementen die aan deze voorwaarde voldoen kunnen tellen, met `count_if` en vervolgens kunnen kijken of dit nul oplevert. De `count_if` moet dan echter altijd alle elementen in de range doorlopen maar een `none_of` kan meteen stoppen, en **`false`** teruggeven, zodra een element gevonden wordt dat wel aan de bepaalde voorwaarde voldoet en hoeft bovendien geen teller bij te houden. Beiden algoritmen zijn van dezelfde orde maar de `none_of` is altijd sneller.

Het volgende algoritmen kun je gebruiken om een zelf te specificeren bewerking uit te voeren op elementen in een range:

- `for_each`; Voer een bewerking uit op elk element.[151]

De volgende algoritmen kun je gebruiken om te zoeken naar een bepaalde subrange van elementen in een range:

- `search`; Zoek naar een bepaalde subrange.
- `search_n`; Zoek naar een subrange van n opeenvolgende elementen met een bepaalde waarde.
- `find_end`[152]; Zoek, van achter naar voren, naar een bepaalde subrange.

De volgende algoritmen kun je gebruiken om ranges met elkaar te vergelijken:

- `equal`; Vergelijk twee ranges. Geeft een **`bool`** terug.
- `mismatch`; Zoek naar het eerste verschil in twee ranges. Geeft een iterator terug.

De executietijd van de in deze paragraaf genoemde algoritmen is $O(n)$ behalve de executietijd van `search` en `find_end`. De executietijd van `search` is namelijk $O(n \times m)$, waarbij n het

[151] Bijvoorbeeld: vermenigvuldig elk element met 2.

[152] De naam van dit algoritme is inconsequent gekozen. Een betere naam zou zijn `search_end`.

aantal elementen is van de range die wordt doorzocht en m het aantal elementen is van de subrange die wordt gezocht. De executietijd van `find_end` is $O(m \times (n-m))$.

13.1.1 Voorbeeldprogramma met `std::find`

In het onderstaande voorbeeldprogramma `find.cpp` wordt het `find` algoritme gebruikt om het bekende spelletje galgje te implementeren.

Dit programma werkt als volgt. De string `w` bevat het woord dat geraden moet worden. De `vector<bool>` genaamd gevonden wordt gevuld met n maal de waarde **`false`**, waarbij n het aantal karakters in het te raden woord is.[153] In de **`for`**-loop worden alle tot dusver gevonden letters van het te raden woord afgedrukt. Voor elke nog niet gevonden letter wordt een punt afgedrukt.

Als de gebruiker een letter heeft ingevoerd[154], worden de bijbehorende posities in de `vector` gevonden **`true`** gemaakt. Dit gebeurt door de ingevoerde letter te zoeken in het te raden woord met behulp van de `find` functie. Als de returnwaarde van `find` ongelijk is aan het tweede argument dat aan `find` is meegegeven, dan refereert de iterator `itr` naar de gevonden letter. De index van deze letter is te berekenen met de expressie `itr - w.cbegin()`. Op deze index wordt in de `vector` gevonden **`true`** geschreven om te markeren dat deze letter gevonden is. Vervolgens wordt de iterator `itr` een karakter verder geplaatst en vanaf deze positie wordt opnieuw naar de ingevoerde letter gezocht. Tevens wordt in de variabele `count` bijgehouden hoe vaak de letter gevonden is.

Het zoeken en markeren herhaald zich net zolang tot deze letter niet meer gevonden wordt. Zolang de waarde **`false`** nog in de `vector` gevonden voorkomt, zijn nog niet alle letters van het te raden woord geraden. We maken hier gebruik van `count` om het aantal maal te tellen dat de waarde **`false`** in de `vector` gevonden voorkomt. Later, op pagina 226, wordt besproken hoe je dit met behulp van `any_of` kunt doen.

---

[153] Je ziet dat hier *geen* gebruik gemaakt wordt van een universele initialisatie, zie paragraaf 1.4. Als je bij een container een universele initialisatie gebruikt dan worden de elementen in de initialisatielijst in de container geplaatst. Als je een van de andere constructors van een container wilt gebruiken, zoals de constructor `vector(size_type count, const T& value)`, zie https://en.cppreference.com/w/cpp/container/vector/vector, dan moet je ronde haakjes gebruiken in plaats van accolades bij het aanroepen van de constructor.

[154] Je ziet dat de functie `cin.get()` gebruikt wordt om het door de gebruiker ingevoerde karakter in te lezen. De **`static_cast`**, zie paragraaf 15.15, is nodig omdat deze functie geen **`char`** teruggeeft maar een integer. Hierdoor kan deze functie de speciale waarde `Traits::eof()` teruggeven als het einde van de invoer bereikt is. Als je deze cast wilt voorkomen, dan kun je de variabele `c` ook als volgt initialiseren: `c = cin.get();`. De tweede `cin.get();` is nodig om de door de gebruiker ingetoetste 'Enter' in te lezen (en weg te gooien).

```
import std;
using namespace std;

int main() {
    string w {"galgje"};
    vector<bool> gevonden (w.size(), false);
    do {
        for (string::size_type i {0}; i < w.size(); ++i) {
            print("{}", gevonden[i] ? w[i] : '.');
        }
        print("\nRaad een letter: ");
        char c {static_cast<char>(cin.get())}; cin.get();
        auto itr {w.cbegin()};
        int count {0};
        while ((itr = find(itr, w.cend(), c)) != w.cend()) {
            gevonden[(itr - w.cbegin())] = true;
            ++itr;
            ++count;
        }
        println("De letter '{}' komt {} keer voor in het te raden ↩
 ↪ woord.", c, count);
    }
    while (ranges::count(gevonden, false) != 0);
    println("Je hebt het woord \"{}\" geraden.", w);
}
```

Mogelijke invoer[155] en uitvoer van dit programma:

```
......
Raad een letter: h
De letter 'h' komt 0 keer voor in het te raden woord
......
Raad een letter: a
De letter 'a' komt 1 keer voor in het te raden woord
.a....
Raad een letter: r
De letter 'r' komt 0 keer voor in het te raden woord
.a....
Raad een letter: r
```

[155] De invoer is groen en onderstreept weergegeven.

```
De letter 'r' komt 0 keer voor in het te raden woord
.a....
Raad een letter: y
De letter 'y' komt 0 keer voor in het te raden woord
.a....
Raad een letter: g
De letter 'g' komt 2 keer voor in het te raden woord
ga.g..
Raad een letter: l
De letter 'l' komt 1 keer voor in het te raden woord
galg..
Raad een letter: j
De letter 'j' komt 1 keer voor in het te raden woord
galgj.
Raad een letter: e
De letter 'e' komt 1 keer voor in het te raden woord
Je hebt het woord "galgje" geraden.
```

Dit is zeker niet de meest efficiënte en meest eenvoudige implementatie van het spelletje galgje. Een eenvoudigere en efficiëntere versie vind je in `galgje_strings.cpp`.

13.1.2 Voorbeeldprogramma met `std::find_if`

Het onderstaande voorbeeldprogramma `find_if.cpp` laat zien hoe je het standaard algoritme `find_if` kunt gebruiken om positieve getallen te zoeken. De zoekvoorwaarde (condition) wordt op drie verschillende manieren opgegeven:

- door middel van een *functie* die een **`bool`** teruggeeft die aangeeft of aan de voorwaarde wordt voldaan;
- door middel van een *functie-object*[156] met een overloaded **`operator`**`()` die een **`bool`** teruggeeft die aangeeft of aan de voorwaarde wordt voldaan;
- door middel van een *lambda-functie*[157] die een **`bool`** teruggeeft die aangeeft of aan de voorwaarde wordt voldaan.

```
import std;
```

[156] Een functie-object is een object dat zich voordoet als een functie door middel van operator overloading.

[157] Een lambda-functie is een anoniem functie-object dat wordt gedefinieerd op de plaats waar dit functie-object nodig is. Dit wordt later, in paragraaf 13.2, nog uitgebreid besproken.

```
using namespace std;

bool is_pos(int i) {
    return i >= 0;
}

template<totally_ordered T>
class Is_greater_equal {
public:
    Is_greater_equal(T r): right(r) {
    }
    bool operator()(T left) const {
        return left >= right;
    }
private:
    T right;
};

int main() {
    list l {-3, -4, 3, 4};
//  Zoeken met behulp van een functie als zoekvoorwaarde.
//  Nadeel: voor elke zoekvoorwaarde moet een aparte
//  functie worden geschreven.
    auto r {ranges::find_if(l, is_pos)};
    if (r != l.cend()) {
        println("Het eerste positieve element is: {}", *r);
    }
//  Zoeken met behulp van een functie-object als zoekvoorwaarde.
//  Voordeel: flexibeler dan een functie.
//  Nadeel: voor elke vergelijkings operator moet een
//  apart functie-object worden geschreven.
    r = ranges::find_if(l, Is_greater_equal(0));
    if (r != l.cend()) {
        println("Het eerste positieve element is: {}", *r);
    }
//  Zoeken met behulp van een lambda functie als zoekvoorwaarde.
//  Voordeel: handige oplossing als zoekvoorwaarde uniek is.
//  Nadeel: lambda funties hebben een speciale, niet heel ↩
 ↪ leesbare, syntax.
    r = ranges::find_if(l, [](int i) {
        return i >= 0;
```

```
    });
    if (r != l.end()) {
        println("Het eerste positieve element is: {}", *r);
    }
}
```

De uitvoer van dit programma:

```
Het eerste positieve element is: 3
Het eerste positieve element is: 3
Het eerste positieve element is: 3
```

Nu je gezien hebt hoe, met behulp van een lambda expressie, een bepaalde voorwaarde kan worden opgegeven kunnen we de conditie van de do-while-lus in het programma uit paragraaf 13.1.1 verbeteren.

De regel:

```
    while (ranges::count(gevonden, false) != 0);
```

kan vervangen worden door (zie galgje_find.cpp):

```
    while (ranges::any_of(gevonden, [](auto b) {
        return b == false;
    }));
```

Het any_of algoritme is sneller dan het count algoritme omdat het niet altijd alle elementen hoeft af te gaan. Om te testen of één of meer elementen in de vector gevonden de waarde **false** hebben werden de elementen die **false** zijn geteld, met count en vervolgens werd gekeken of dit een waarde ongelijk aan nul opleverde. De count moet echter altijd alle elementen van de vector doorlopen maar een any_of kan meteen stoppen, en **true** teruggeven, zodra een element gevonden wordt dat **false** is en hoeft bovendien geen teller bij te houden. Beiden algoritmen zijn O(n) maar de any_of is altijd sneller.

13.1.3 Voorbeeldprogramma met std::find_first_of

Het onderstaande voorbeeldprogramma find_first_of laat zien hoe je het standaard algoritme find_first_of kunt gebruiken om het eerste priemgetal kleiner dan honderd in een bestand gevuld met integers te zoeken.

```
imoprt std;
using namespace std;
```

```
int main() {
    list priem {2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, ↩
 ↪ 43, 47, 53, 59, 61, 67, 71, 73, 79, 83, 89, 97};
    ifstream getallen {"getallen_gesorteerd.txt"};
    if (!getallen)
        return 1;
    auto eerste_priem {find_first_of(istream_iterator<int> ↩
 ↪ {getallen}, istream_iterator<int> {}, priem.begin(), ↩
 ↪ priem.end())};
    if (eerste_priem != istream_iterator<int> {}) {
        println("Het eerste priemgetal kleiner dan 100 is: {}", ↩
 ↪ *eerste_priem);
    }
}
```

13.1.4 Voorbeeldprogramma met std::for_each

Het onderstaande voorbeeldprogramma for_each.cpp laat zien hoe je het standaard algoritme for_each kunt gebruiken om elk element van een vector dubbel af te drukken. De bewerking die voor elk element uitgevoerd moet worden, wordt op twee verschillende manieren opgegeven:

- door middel van een *functie* die de bewerking uitvoert;
- door middel van een *lambda-functie* die de bewerking uitvoert.

```
import std;
using namespace std;

void print_dubbel(int i) {
    print("{0} {0} ", i);
}

int main() {
    vector v {-3, -4, 3, 4};
    println("{:n}", v);
//  Bewerking opgeven met een functie.
//  Nadeel: voor elke bewerking moet een aparte functie worden ↩
 ↪ geschreven.
    ranges::for_each(v, print_dubbel);
    println();
//  Bewerking opgeven met een lambda functie.
```

```
//  Voordeel: handige oplossing als zoekvoorwaarde uniek is.
//  Nadeel: lambda funties hebben een speciale, niet heel ↩
 ↪ leesbare, syntax.
    ranges::for_each(v, [](int i) {
        print("{0} {0} ", i);
    });
    println();
}
```

van dit programma:

```
-3 -4 3 4
-3 -3 -4 -4 3 3 4 4
-3 -3 -4 -4 3 3 4 4
```

We kunnen alle elementen van een vector ook dubbel afdrukken met behulp van een range-based for:

```
    for (auto i: v) {
        print("{0} {0} ", i);
    };
    println();
```

Het gebruik van een range-based for heeft dezelfde voordelen en nadelen als het gebruik van een `for_each` met een lambda-functie. De syntax van de range-based for is echter eenvoudiger. De range-based for kan echter alleen gebruikt worden als de hele container moet worden doorlopen. Met een `for_each` kan ook een deel van een container doorlopen worden.

13.2 Lambda-functies

In paragraaf 13.1.2 heb je gezien dat een lambda-functie gebruikt kan worden om de voorwaarde van een `find_if` te specificeren. In paragraaf paragraaf 13.1.4 heb je gezien dat een lambda-functie gebruikt kan worden om de bewerking van een `for_each` te specificeren. Een lambda-functie is een anoniem functie-object dat wordt gedefinieerd op de plaats waar dit functie-object nodig is.

13.2.1 Closure

Soms wil je bij het definiëren van een lambda-functie in de code van deze functie gebruik maken van variabelen die buiten de lambda-functie gedefinieerd zijn. Het object dat door een lambda functie wordt aangemaakt en waarin waarden van of verwijzingen naar variabelen van buiten de functie kunnen worden opgenomen wordt een *closure* genoemd (Nederlands: insluiting). De variabelen die ingesloten moeten worden in de closure worden tussen [en] opgegeven. Deze ingesloten variabelen kunnen in de code van de closure gebruikt worden. Enkele voorbeelden:

- `[a, &b]` a wordt 'by value' en b wordt 'by reference' ingesloten.
- `[&]` elke in de lambda-functie gebruikte variabele wordt 'by reference' ingesloten.
- `[=]` elke in de lambda-functie gebruikte variabele wordt 'by value' ingesloten.
- `[=, &c]` c wordt 'by reference' ingesloten en alle andere in de lambda-functie gebruikte variabelen worden 'by value' ingesloten.
- `[]` er worden géén variabelen ingesloten.

Het onderstaande programma `closure.cpp` laat zien hoe je met behulp van een `for_each` algoritme en een closure de som van alle even getallen in een range kunt bepalen.[158]

```
import std;
using namespace std;

int main() {
    vector v {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
    int som_even {0};
    ranges::for_each(v, [&som_even](auto i) {
        if (i % 2 == 0) {
            som_even += i;
        }
    });
    println("De som van alle even getallen in v = {}", som_even);
}
```

In plaats van `[&som_even]` kan ook `[&]` gebruikt worden. Het expliciet specificeren dat de variabele `som_even` ingesloten moet worden in de closure heeft als voordeel dat de

[158] Als alternatief zou je ook een `accumulate` algoritme met een gewone lambda-functie kunnen gebruiken. Zie online: `accumulate.cpp`.

compiler een foutmelding geeft als je per ongeluk een andere variabele in de code van de lambda-functie gebruikt.

13.2.2 Returntype van een lambda-functie

Bij alle tot nu toe gebruikte lambda-functies kan de compiler zelf het returntype van de functie bepalen. Er zijn situaties waarbij dit niet mogelijk is. Als de compiler niet in staat is om zelf het returntype te bepalen, dan kan het returntype gespecificeerd worden na de parameterlijst van de lambda-functie `-> typenaam`.

In het onderstaande programma `resultaat_naar_geheel.cpp` wordt een `list` met toetsresultaten met één decimaal cijfer getransformeerd[159] naar een `vector` met gehele toetsresultaten. Daarbij moeten de oorspronkelijke resultaten worden afgerond. Het minimaal te behalen gehele toetsresultaat is echter 1, dus alle oorspronkelijke toetsresultaten < 1.5 moeten omgezet worden naar 1.

```
import std;
using namespace std;

int main() {
    list resultaten {0.4, 1.4, 1.5, 4.9, 5.0, 8.9, 9.1, 9.5, 10.0};
    println("Resultaten: {}", resultaten);
    vector<int> gehele_resultaten;
    // Alle resultaten < 1.5 moeten worden afgerond tot 1
    // Alle overige resultaten moeten worden afgerond
    ranges::transform(resultaten, ↩
 ↪ back_inserter(gehele_resultaten), [](auto r) -> int {
        if (r < 1.5) {
            return 1;
        }
        return round(r);
    });
    println("Gehele resultaten: {}", gehele_resultaten);
}
```

In dit geval is het noodzakelijk om het returntype van de lambda-functie expliciet op te geven. De compiler is niet in staat om zelf het returntype te bepalen omdat de `round` functie een **`double`** teruggeeft. Het returntype van de lambda-functie zou dus **`int`** of **`double`** kunnen zijn

[159] Zie voor informatie over het `transform` algoritme http://en.cppreference.com/w/cpp/algorithm/transform of verderop in dit dictaat: paragraaf 13.3.1.

en alleen de programmeur weet wat de bedoeling is. Als je het returntype niet specificeert krijg je de volgende foutmelding: `error: inconsistent types 'int' and 'double' deduced for lambda return type`.

13.2.3 Lambda-functieparametertype met `auto`

Het is ook mogelijk om het parametertype van een lambda-function als **`auto`** te definiëren.De compiler bepaalt dan zelf het juiste type.

In paragraaf 13.1.2 is de volgende lambda-functie gebruikt:

```
r = ranges::find_if(l, [](int i) {
    return i >= 0;
});
```

Je kunt het type van de parameter van de lambda-functie ook als **`auto`** declareren en door de compiler laten bepalen:

```
r = ranges::find_if(l, [](auto i) {
    return i >= 0;
});
```

De compiler bepaalt nu zelf dat het parametertype van de lambda functie **`int`** moet zijn omdat de elementen van de `list l` van het type **`int`** zijn. Het voordeel hiervan is dat je deze code niet aan hoeft te passen als het type van de variabele `l` van `list<int>` wordt gewijzigd in `list<double>`. Je ziet dat de code door het gebruik van **`auto`** om het parametertype van de lambda-functie te definiëren beter aanpasbaar wordt.

Een lambda met een **`auto`** parameter lijkt op een template functie, zie paragraaf 3.1 en wordt een *generieke lambda* genoemd.

13.3 Kopiëren, bewerken, vervangen, roteren, schudden, verwisselen, verwijderen en vullen

In paragraaf 13.1 heb je al kennis gemaakt met een aantal algoritmen uit de standaard C++ library. In dit dictaat worden echter niet alle algoritmen behandeld, kijk voor een volledig overzicht op `http://en.cppreference.com/w/cpp/algorithm`.

De volgende algoritmen kun je gebruiken om een zelf te specificeren bewerking op elementen in een range uit te voeren:

- `for_each`; Voer bewerking uit op elk element van een range. Dit algoritme hebben we al besproken in paragraaf 13.1.4.
- `transform`; Voer bewerking uit op elk element van één of twee ranges en schrijf het resultaat naar een andere range. Met dit algoritme heb je al kennis gemaakt in paragraaf 13.2.2.

De volgende algoritmen kun je gebruiken om ranges te kopiëren:

- `copy`; Kopieer alle elementen van de ene range naar de andere. Dit algoritme hebben we in dit dictaat al diverse keren gebruikt, zie bijvoorbeeld paragraaf 12.5.
- `copy_if`; Kopieer alle elementen die voldoen aan een bepaalde voorwaarde van de ene range naar de andere.

De volgende algoritmen kun je gebruiken om elementen in een range te vervangen:

- `replace`; Vervang alle elementen met een bepaalde waarde.
- `replace_if`; Vervang alle elementen die voldoen aan een bepaalde voorwaarde.
- `replace_copy`; Effectief hetzelfde als een `copy` gevolgd door een `replace`.[160]
- `replace_copy_if`; Effectief hetzelfde als een `copy` gevolgd door een `replace_if`.

De volgende algoritmen kun je gebruiken om elementen in een range te roteren:

- `rotate`; Roteer alle elementen.
- `rotate_copy`; Effectief hetzelfde als een `copy` gevolgd door een `rotate`.

Het volgende algoritme kun je gebruiken om elementen in een range te husselen:

- `random_shuffle`; Hussel alle elementen door elkaar. Zoals bij het schudden van een stok kaarten.

Het volgende algoritme kun je gebruiken om ranges te verwisselen:

- `swap_range`; Verwissel ranges.

De volgende algoritmen kun je gebruiken om de volgorde van alle elementen in een range om te keren:

- `reverse`; Keer de volgorde van alle elementen om.
- `reverse_copy`; Effectief hetzelfde als een `copy` gevolgd door een `reverse`.

[160] Alle algoritmen met _copy in de naam voeren effectief een copy uit gevolgd door het betreffende algoritme. Deze combinatie kan vaak sneller geïmplementeerd worden dan de twee afzonderlijke bewerkingen.

De volgende algoritmen kun je gebruiken om elementen uit een range te verwijderen[161]:

- `remove`; Verwijder alle elementen met een bepaalde waarde.
- `remove_if`; Verwijder alle elementen die voldoen aan een bepaalde voorwaarde.
- `remove_copy`; Effectief hetzelfde als een `copy` gevolgd door een `remove`.
- `remove_copy_if`; Effectief hetzelfde als een `copy` gevolgd door een `remove_if`.
- `unique`; Verwijder alle elementen die gelijk zijn aan hun voorganger.
- `unique_copy`; Effectief hetzelfde als een `copy` gevolgd door een `unique`.

De volgende algoritmen kun je gebruiken om elementen in een range te overschrijven[162] met een bepaalde waarde:

- `fill`; Maak alle elementen gelijk aan een bepaalde waarde.
- `fill_n`; Maak de eerste n elementen gelijk aan een bepaalde waarde.
- `generate`; Maak alle elementen gelijk aan de uitvoer van een bepaalde functie.
- `generate_n`; Maak de eerste n elementen gelijk aan de uitvoer van een bepaalde functie.

De executietijd van de in deze paragraaf genoemde algoritmen is $O(n)$.

13.3.1 Voorbeeldprogramma met `std::ranges::transform`

Het onderstaande voorbeeldprogramma `transform.cpp` laat zien hoe je het standaard algoritme `transform` kunt gebruiken om een vector bij een andere vector op te tellen. De transformatie (bewerking) wordt op twee[163] verschillende manieren opgegeven:

- door middel van een *functie* die de transformatie uitvoert;
- door middel van een *lambda-functie* die de transformatie uitvoert.

[161] De elementen worden door deze algoritmen niet daadwerkelijk verwijderd (deze algoritmen geven een iterator terug die gebruikt kan worden om de elementen daadwerkelijk te verwijderen met behulp van de memberfunctie `erase` van de betreffende container. Zie paragraaf 13.3.2).

[162] Door deze algoritmen te combineren met een insert-iterator, zie paragraaf 12.3, kunnen ze ook gebruikt worden om containers te vullen (of aan te vullen).

[163] Er is nog een derde manier, zie paragraaf 20.9

```
import std;
using namespace std;

int telop(int i, int j) {
    return i + j;
}

int main() {
    vector v {-3, -4, 3, 4};
    vector w {1, 2, 3, 4};
    println("v: {}", v);
    println("w: {}", w);
//  Bewerking opgeven met een functie.
//  Nadeel: voor elke bewerking moet een aparte functie worden ↩
 ↪ geschreven.
    ranges::transform(v, w, v.begin(), telop);
    println("v na transform: {}", v);
//  Bewerking opgeven met een lambda functie.
//  Voordeel: handige oplossing als zoekvoorwaarde uniek is.
//  Nadeel: lambda funties hebben een speciale, niet heel ↩
 ↪ leesbare, syntax.
    ranges::transform(v, w, v.begin(), [](int i1, int i2) {
        return i1 + i2;
    });
    println("v na transform: {}", v);
}
```

De uitvoer van dit programma:

```
v: [-3, -4, 3, 4]
w: [1, 2, 3, 4]
v na transform: [-2, -2, 6, 8]
v na transform: [-1, 0, 9, 12]
```

Je kunt ook **auto** gebruiken om de parametertypes van de lambda-functie te definiëren, zoals besproken in paragraaf 13.2.3:

```
    ranges::transform(v, w, v.begin(), [](auto i1, auto i2) {
        return i1 + i2;
    });
```

De compiler bepaalt nu zelf de parametertypes van de lambda-functie. Het type van parameter `i1` moet **`int`** zijn omdat de elementen van de `vector` `v` van het type **`int`** zijn. Het type van parameter `i2` moet, toevallig ook, **`int`** zijn omdat de elementen van de `vector` `w` van het type **`int`** zijn.

In het programma `galgje_strings_transform.cpp` kun je zien dat het standaard algoritme `transform` ook kunt gebruiken om het bekende spelletje galgje te implementeren.

13.3.2 Voorbeeldprogramma met `std::remove`

Na `remove` is nog een `erase` nodig om de elementen echt te verwijderen. Het is namelijk niet mogelijk om vanuit een generiek algoritme elementen daadwerkelijk uit een range te verwijderen omdat het algoritme niet weet wat het type van de container is. Het verwijderen van elementen uit een `vector` gaat natuurlijk heel anders dan het verwijderen van elementen uit een `list`. De `remove` algoritmen plaatsen alle elementen die niet verwijderd moeten worden voorin de range en geven een iterator terug naar het eerste daadwerkelijk te verwijderen element. Daarna moet de memberfunctie `erase` van de betreffende container aangeroepen worden om deze elementen daadwerkelijk te verwijderen. Dit wordt gedemonstreerd in het onderstaande programma `remove.cpp`:

```
import std;
using namespace std;

int main() {
    vector<int> v;
    for (int i {0}; i < 10; ++i) {
        v.push_back(i * i);
    }
    vector w {v};

//  code om te laten zien hoe remove werkt:
    println("Na initialisatie: {}", v);
    auto end_range {ranges::remove_if(v, [](auto i) {
            return i % 2 == 0;
        })
    };
    println("Na remove (tot returned iterator): {}", end_range);
    println("Na remove (hele vector): {}", v);
    v.erase(end_range.cbegin(), v.cend());
    println("Na erase (hele vector): {}", v);
```

```
//  in de praktijk gebruiken we een remove altijd binnen een erase:
    w.erase(ranges::remove_if(w, [](auto i) {
        return i % 2 == 0;
    }).cbegin(), w.cend());
    println("Na remove binnen erase: {}", w);
}
```

De uitvoer van dit programma:

```
Na initialisatie: [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
Na remove (tot returned iterator): [1, 9, 25, 49, 81]
Na remove (hele vector): [1, 9, 25, 49, 81, 25, 36, 49, 64, 81]
Na erase (hele vector): [1, 9, 25, 49, 81]
Na remove binnen erase: [1, 9, 25, 49, 81]
```

13.3.3 Voorbeeldprogramma met `std::generate_n`

In paragraaf 13.3.2 werd de `vector v` gevuld met kwadraten door een **`for`**-lus te gebruiken. Als alternatief kunnen we ook gebruik maken van het algoritme `generate_n`, een insert-iterator (zie paragraaf 12.3) en een closure (zie paragraaf 13.2.1. Al kun je jezelf in dit geval natuurlijk wel afvragen of de **`for`**-lus niet eenvoudiger en duidelijker is.

```
import std;
using namespace std;

int main() {
    vector<int> kwadraten;
    int n {0};
    generate_n(back_inserter(kwadraten), 10, [&n]() {
        ++n; return n * n;
    });
    println("De eerste 10 kwadraten: {:n}", kwadraten);
}
```

13.4 Sorteren en bewerkingen op gesorteerde ranges

In paragraaf 13.1 en paragraaf 13.3 heb je al kennis gemaakt met een aantal algoritmen uit de standaard C++ library. In dit dictaat worden echter niet alle algoritmen behandeld, kijk voor een volledig overzicht op http://en.cppreference.com/w/cpp/algorithm.

De volgende algoritmen kun je gebruiken om een range te sorteren:

- `sort`; Sorteer alle elementen van een range.
- `stable_sort`; Sorteer alle elementen van een range waarbij gelijke elementen ten opzichte van elkaar niet van volgorde veranderen.

De executietijd van beide algoritmen is $O(n \times \log n)$, maar `sort` is sneller dan `stable_sort`.

Het volgende algoritme kun je gebruiken om in een *gesorteerde* range te zoeken:

- `binary_search`; Zoekt in een gesorteerde range naar een bepaalde waarde. De executietijd van deze functie is $O(\log n)$ en het returntype is **`bool`**.

De volgende algoritmen kun je gebruiken om bewerkingen op verzamelingen[164] uit te voeren:

- `set_intersection`; Bepaal de doorsnede van twee verzamelingen: $S = S1 \cap S2$.
- `set_union`; Bepaal de vereniging van twee verzamelingen: $S = S1 \cup S2$.
- `set_difference`; Bepaal het verschil van twee verzamelingen: $S = S1 \setminus S2$.
- `set_symmetric_difference`; Bepaal het symmetrische verschil van twee verzamelingen: $S = S1 \triangle S2$.
- `includes`; Bepaal of verzameling $S1$ een deelverzameling is van $S2$: $S1 \subseteq S2$.

Je kunt deze bewerkingen niet alleen uitvoeren op containers van het type `set` maar op alle gesorteerde ranges. De executietijd van deze algoritmen is $O(n1 + n2)$, waarbij $n1$ staat voor het aantal elementen in verzameling $S1$ en $n2$ staat voor het aantal elementen in verzameling $S2$

13.4.1 Voorbeeldprogramma met `std::sort`

In paragraaf 12.5 heb je al gezien hoe het `sort` algoritme gebruikt kan worden om een `vector` met integers te sorteren. Het `sort` algoritme maakt default gebruik van de **`operator`**< om de elementen met elkaar te vergelijken. De elementen worden dus default van laag naar hoog gesorteerd. Je kunt echter ook indien gewenst zelf een vergelijkingsfunctie definiëren. Deze vergelijkingsfunctie moet als derde argument aan `sort` worden meegegeven. Dit is nodig als je in een andere volgorde wilt sorteren of als voor de te sorteren objecten geen **`operator`**< gedefinieerd is.

[164] Kijk, als je niet bekend bent met verzamelingenleer of als je een opfrisser nodig hebt, op: http://nl.wikipedia.org/wiki/Verzameling_(wiskunde).

Als je de getallen in het programma stream_iterator.cpp uit paragraaf 12.5 van hoog naar laag wilt sorteren dan kun je de regel:

```
ranges::sort(rij);
```

vervangen door (zie sort_lambda.cpp)[165]:

```
ranges::sort(rij, [](auto i, auto j) {
    return i > j;
});
```

Een stable_sort is nodig als een range dubbel gesorteerd moet worden. Stel dat bij een spelletje, na afloop van elk spel, de naam van de speler en het behaalde aantal punten opgeslagen worden in een vector met objecten van de class Score. Dan kun je een alfabetische 'high score list' maken door de vector eerst van laag naar hoog te sorteren op naam en daarna van hoog naar laag te sorteren op het behaalde aantal punten. Bij de tweede sorteeroperatie mogen scores met een gelijk aantal punten niet in volgorde verwisseld worden zodat spelers met hetzelfde aantal punten op naam gesorteerd blijven. Voor de tweede sorteeroperatie moet dus een stable_sort gebruikt worden. Zie sort_deelnemers.cpp.

```
import std;
using namespace std;

class Score {
public:
    Score(string_view n, int p);
    int punten() const;
    string naam() const;
private:
    string nm;
    int pnt;
};

Score::Score(string_view n, int p): nm{n}, pnt{p} {
}
int Score::punten() const {
    return pnt;
}
string Score::naam() const {
    return nm;
```

[165] Er is een nog eenvoudigere manier, zie paragraaf 20.9.

```
}

int main() {
    vector scores {
        Score {"Theo", 300},
        Score {"Marie-louise", 300},
        Score {"Koen", 300},
        Score {"Linda", 300},
        Score {"Marie-louise", 400},
        Score {"Anne", 300},
        Score {"Marie-louise", 50}
    };
    ranges::sort(scores, [](const auto& d1, const auto& d2) {
        return d1.naam() < d2.naam();
    });
    ranges::stable_sort(scores, [](const auto& d1, const auto& d2) {
        return d1.punten() > d2.punten();
    });
    println("Scores van hoog naar laag:");
    for (const auto& s: scores) {
        println("{:>5} {}", s.punten(), s.naam());
    }
}
```

De uitvoer van dit programma:

```
Scores van hoog naar laag:
  400 Marie-louise
  300 Anne
  300 Koen
  300 Linda
  300 Marie-louise
  300 Theo
   50 Marie-louise
```

13.4.2 Voorbeeldprogramma met `std::includes`

In paragraaf 13.1.1 is het `find` algoritme gebruikt om het bekende spelletje galgje te implementeren. Dit spel kan ook geïmplementeerd worden door gebruik te maken van verzamelin-

gen, zie galgje_sets.cpp. Het is dan ook mogelijk om te controleren of een letter al eerder is geprobeerd.

Dit programma werkt als volgt. De string w bevat het woord dat geraden moet worden. De set genaamd te_raden wordt gevuld met de letters die voorkomen in het te raden woord. Dit wordt gedaan met behulp van het copy algoritme en een insert-iterator (zie paragraaf 12.3). De set genaamd geraden wordt gebruikt om de door de gebruiker geraden letters in op te slaan. In de **for**-loop worden alle tot dusver gevonden letters van het te raden woord afgedrukt. Voor elke nog niet gevonden letter wordt een punt afgedrukt. Als de gebruiker een letter heeft ingevoerd, wordt deze letter toegevoegd aan de verzameling gevonden. Als dit niet lukt[166], doordat de letter al aanwezig is in deze verzameling dan wordt net zo lang om een nieuwe letter gevraagd tot het toevoegen wel lukt. Zolang de verzameling te_raden nog *geen* deelverzameling is van de verzameling geraden, zijn nog niet alle letters van het te raden woord geraden.

```
import std;
using namespace std;

int main() {
    string w {"galgje"};
    set<char> te_raden, geraden;
    ranges::copy(w, inserter(te_raden, te_raden.begin()));
    do {
        for (auto c: w) {
            print("{}", geraden.count(c) ? c : '.');
        }
        print("\nRaad een letter: ");
        char c {static_cast<char>(cin.get())}; cin.get();
        while (!geraden.insert(c).second) {
            print("De letter {} had je al geraden...", c);
            print("\nRaad een andere letter: ");
            c = cin.get(); cin.get();
        }
        println("De letter '{}' komt {} keer voor in het te raden ↩
 ↪ woord.", c, ranges::count(w, c));
    } while (!ranges::includes(geraden, te_raden));
    println("Je hebt het woord {} geraden.", w);
}
```

[166] In paragraaf 11.4.1 is de insert memberfunctie van set behandeld.

Invoer en uitvoer van dit programma:

```
......
Raad een letter: h
De letter 'h' komt 0 keer voor in het te raden woord.
......
Raad een letter: a
De letter 'a' komt 1 keer voor in het te raden woord.
.a....
Raad een letter: g
De letter 'g' komt 2 keer voor in het te raden woord.
ga.g..
Raad een letter: a
De letter a had je al geraden...
Raad een andere letter: h
De letter h had je al geraden...
Raad een andere letter: e
De letter 'e' komt 1 keer voor in het te raden woord.
ga.g.e
Raad een letter: i
De letter 'i' komt 0 keer voor in het te raden woord.
ga.g.e
Raad een letter: j
De letter 'j' komt 1 keer voor in het te raden woord.
ga.gje
Raad een letter: l
De letter 'l' komt 1 keer voor in het te raden woord.
Je hebt het woord galgje geraden.
```

14

Toepassingen van datastructuren

In dit hoofdstuk worden algoritmen en datastructuren toegepast om de computer het spelletje Boter, Kaas en Eieren te laten spelen. Een andere toepassing is het berekenen van het kortste pad in een graph.

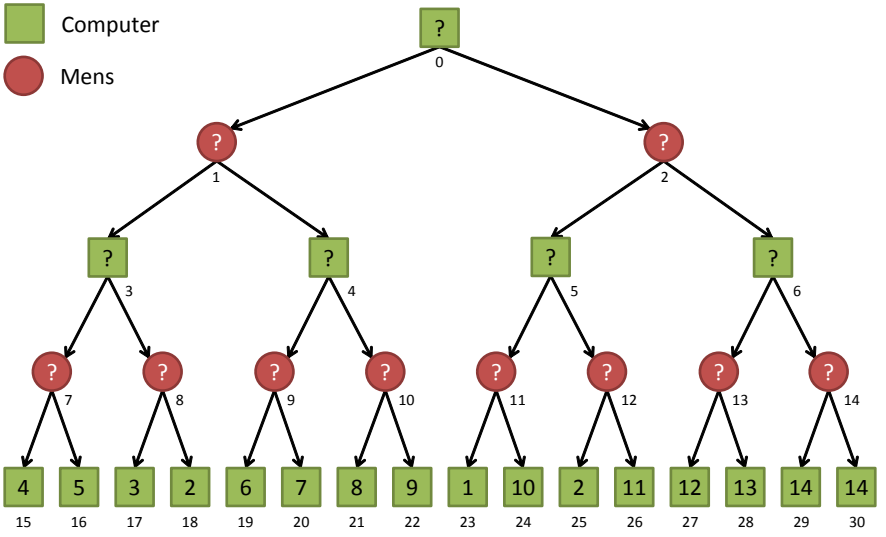
14.1 Boter, Kaas en Eieren

In deze paragraaf wordt besproken hoe je een computer het spelletje Boter, Kaas en Eieren kunt laten spelen. Maar eerst wordt uitgelegd hoe je de beste zet kunt vinden in een spelboom (Engels: game tree) door het minimax algoritme toe te passen. In de volgende paragraaf wordt uitgelegd hoe je dit algoritme kunt implementeren in C++. Daarna wordt uitgelegd hoe je dit algoritme kunt versnellen door de zogenoemde 'alpha-beta pruning' toe te passen. Tot slot wordt verwezen naar programma's waarin het minimax algoritme en alpha-beta pruning zijn toegepast om een computer Boter, Kaas en Eieren te laten spelen.

14.1.1 Minimax algoritme

Als je de beste zet in een spel (met twee spelers, die elk afwisselend aan de beurt zijn) wilt vinden, kun een boom tekenen met alle mogelijke posities. Stel dat deze boom er (voor een denkbeeldig spel) uitziet zoals in figuur 14.1 is gegeven.

In figuur 14.1 is een vierkant getekend voor een positie waarbij de computer aan de beurt is en een rondje als de tegenstander (mens) aan de beurt is. De waarde van een positie is 15



Figuur 14.1: Voorbeeld van een spelboom.

als de computer wint en 0 als de mens wint. De computer moet dus een positie met een zo hoog mogelijke waarde zien te bereiken.

De posities zijn genummerd van links naar rechts en van boven naar beneden. Voor elke positie $15 \leq p < 31$ geldt dat de waarde van deze positie bekend is. Voor elke positie $0 \leq p < 15$ geldt dat de twee mogelijke volgende posities $2p+1$ en $2p+2$ zijn. Bijvoorbeeld: positie 4 heeft als mogelijke volgende posities $2 \cdot 4 + 1 = 9$ en $2 \cdot 4 + 2 = 10$.

De computer moet de hoogst haalbare waarde zien te bereiken. In de figuur 14.1 lijkt dit een van de twee posities met de waarde 14 maar deze zijn, bij sterk spel van de menselijke tegenspeler, onbereikbaar. Denk maar even na. Bij zijn eerste beurt gaat de computer naar rechts (richting 14). De tegenstander (mens) gaat dan echter naar links. De computer gaat nu bij zijn volgende beurt naar rechts (richting 11) maar zijn tegenstander kiest bij de volgende beurt weer voor links waardoor de bereikte positie de waarde 2 heeft (niet best voor de computer dus).

Is dit de hoogst haalbare waarde voor de computer?

De hoogst haalbare waarde kan bepaald worden door van onder naar boven door de boom te wandelen en de waarde van elk knooppunt als volgt te bepalen:

- Als de computer aan de beurt is, moet je voor de positie kiezen met de *hoogste* waarde (de computer wil zo sterk mogelijk spelen).

- Als de mens aan de beurt is (de tegenstander), moet je voor de positie kiezen met de *laagste* waarde (er van uitgaande dat de tegenstander zo sterk mogelijk speelt).

Omdat je dus afwisselend kiest voor de maximale (als de computer aan de beurt is) en de minimale (als de tegenstander aan de beurt is) waarde wordt dit algoritme het minimax algoritme genoemd.

In het onderstaande programma `minimax0.cpp` wordt de boom uit figuur 14.1 gebruikt en wordt de hoogst haalbare waarde berekend met het minimax algoritme. Er wordt daarbij gebruik gemaakt van 2 functies die elkaar (recursief) aanroepen.

```
import std;
using namespace std;

int value(int pos);
int choose_computer_move(int pos);
int choose_human_move(int pos);

constexpr int UNDECIDED {-1};

int value(int pos) {
    static const int value[16] {4, 5, 3, 2, 6, 7, 8, 9, 1, 10, 2, ↩
 ↪ 11, 12, 13, 14, 14};
    if (pos >= 15 && pos < 31)
        return value[pos - 15]; // return known value
    return UNDECIDED;
}

int choose_computer_move(int pos) {
    int best_value {value(pos)};
    if (best_value == UNDECIDED) {
        best_value = 0;
        for (int i {1}; i < 3; ++i) {
            int value {choose_human_move(2 * pos + i)};
            if (value > best_value) {
                best_value = value;
            }
        }
    }
    return best_value;
}
```

```
int choose_human_move(int pos) {
    int best_value {value(pos)};
    if (best_value == UNDECIDED) {
        best_value = 15;
        for (int i {1}; i < 3; ++i) {
            int value {choose_computer_move(2 * pos + i)};
            if (value < best_value) {
                best_value = value;
            }
        }
    }
    return best_value;
}

int main() {
    int value {choose_computer_move(0)};
    println("Minimaal te behalen Maximale waarde = {}", value);
}
```

De functie value geeft UNDECIDED terug als de waarde van de positie berekend moet worden. Als de positie zich in de onderste laag van de boom bevindt en de waarde dus bekend is, geeft value deze waarde terug.

De functie choose_computer_move wordt aangeroepen om te bepalen wat de waarde van de positie pos is als de computer aan de beurt is in deze positie. Deze functie moet dus het *maximum* bepalen van de (in dit geval twee) posities die vanuit de positie pos bereikt kunnen worden. In deze binaire boom zijn dat de posities: `2 * pos + 1` en `2 * pos + 2`. Bij binnenkomst van de functie choose_computer_move wordt eerst de functie value aangeroepen. De returnwaarde van deze functie wordt opgeslagen in de lokale variabele best_value. Als de waarde van best_value ongelijk is aan UNDECIDED, kan de functie choose_computer_move deze waarde meteen teruggeven. De positie pos bevindt zich in dat geval in de onderste laag van de boom. Als de positie pos waarmee de functie choose_computer_move is aangeroepen zich niet in de onderste laag van de boom bevindt, geeft de functie value de waarde UNDECIDED terug en wordt de waarde van de variabele best_value gelijk gemaakt aan het *absolute minimum* (in dit geval `0`). Het is logisch dat een functie die het maximum moet bepalen begint met het (tot dan toe gevonden) maximum gelijk te maken aan het absolute minimum. Vervolgens wordt in de **for**-lus de waarde van de (in dit geval twee) posities die vanuit de positie pos bereikt kunnen worden bepaald. Dit gebeurt door het aanroepen van de functie choose_human_move met als argument `2 * pos + i` waarbij i

achtereenvolgens de waarde 1 en 2 heeft. We moeten hier de functie choose_human_move aanroepen omdat als in positie pos de computer aan de beurt is, in de posities die vanuit de positie pos bereikt kunnen worden de tegenstander (mens) aan de beurt is. De returnwaarde van deze functie wordt opgeslagen in de lokale variabele value. Als de waarde van value groter is dan het tot nu toe gevonden maximum (opgeslagen in de variabele best_value), dan wordt het tot nu toe gevonden maximum gelijk gemaakt aan value. Aan het einde van de **for**-lus bevat de variabele best_value dus het maximum van de (in dit geval twee) posities die vanuit de positie pos bereikt kunnen worden. De waarde van best_value wordt teruggegeven aan de aanroepende functie.

De functie choose_human_move wordt aangeroepen als de mens aan de beurt is en lijkt erg veel op de functie choose_computer_move, in plaats van het maximum wordt echter het minimum bepaald.

De functie choose_human_move wordt aangeroepen om te bepalen wat de waarde van de positie pos is als de mens aan de beurt is in deze positie. Deze functie moet dus het *minimum* bepalen van de (in dit geval twee) posities die vanuit de positie pos bereikt kunnen worden. In deze binaire boom zijn dat de posities: 2 * pos + 1 en 2 * pos + 2. Bij binnenkomst van de functie choose_human_move wordt eerst de functie value aangeroepen. De returnwaarde van deze functie wordt opgeslagen in de lokale variabele best_value. Als de waarde van best_value ongelijk is aan UNDECIDED, kan de functie choose_human_move deze waarde meteen teruggeven. De positie pos bevindt zich in dat geval in de onderste laag van de boom. Als de positie pos waarmee de functie choose_human_move is aangeroepen zich niet in de onderste laag van de boom bevindt, geeft de functie value de waarde UNDECIDED terug en wordt de waarde van de variabele best_value gelijk gemaakt aan het *absolute maximum* (in dit geval 15). Het is logisch dat een functie die het minimum moet bepalen begint met het (tot dan toe gevonden) minimum gelijk te maken aan het absolute maximum. Vervolgens wordt in de **for**-lus de waarde van de (in dit geval twee) posities die vanuit de positie pos bereikt kunnen worden bepaald. Dit gebeurt door het aanroepen van de functie choose_computer_move met als argument 2 * pos + i waarbij i achtereenvolgens de waarde 1 en 2 heeft. We moeten hier de functie choose_computer_move aanroepen omdat als in positie pos de mens aan de beurt is, in de posities die vanuit de positie pos bereikt kunnen worden de computer aan de beurt is. De returnwaarde van deze functie wordt opgeslagen in de lokale variabele value. Als de waarde van value kleiner is dan het tot nu toe gevonden minimum (opgeslagen in de variabele best_value), dan wordt het tot nu toe gevonden minimum gelijk gemaakt aan value. Aan het einde van de **for**-lus bevat de variabele best_value dus het

minimum van de (in dit geval twee) posities die vanuit de positie pos bereikt kunnen worden. De waarde van best_value wordt teruggegeven aan de aanroepende functie.

In de functie main wordt ervan uitgegaan dat de computer in positie 0 aan de beurt is.

De uitvoer van dit programma is:

```
Minimaal te behalen Maximale waarde = 4
```

Om de werking van dit minimax algoritme helemaal te doorgronden is het heel zinvol om het uitvoeren van dit programma stap voor stap te volgen, eventueel met behulp van een debugger.

1. Vanuit main wordt de functie choose_computer_move(0) aangeroepen. De parameter pos krijgt dus de waarde 0. Vervolgens wordt de functie value(0) aangeroepen. De parameter pos van de functie value krijgt dus de waarde 0 en deze functie geeft UNDECIDED terug. Deze waarde wordt toegekend aan de lokale variabel best_value van de functie choose_computer_move. De voorwaarde van de **if** is dus **true** en de code binnen de **if** wordt uitgevoerd. De lokale variabele best_value wordt gelijk gemaakt aan 0 en de **for**-lus wordt uitgevoerd. De variabele i krijgt de waarde 1 en de functie choose_human_move wordt aangeroepen met 2 * 0 + 1 is 1 als argument.

1.1. Vanuit de functie choose_computer_move(0) wordt dus de functie choose_human_move(1) aangeroepen. Vervolgens wordt de functie value(1) aangeroepen en deze functie geeft UNDECIDED terug. Deze waarde wordt toegekend aan de lokale variabel best_value van de functie choose_human_move. De voorwaarde van de **if** is dus **true** en de code binnen de **if** wordt uitgevoerd. De lokale variabele best_value wordt gelijk gemaakt aan 15 en de **for**-lus wordt uitgevoerd. De variabele i krijgt de waarde 1 en de functie choose_computer_move wordt aangeroepen met 2 * 1 + 1 is 3 als argument.

1.1.1. Vanuit de functie choose_human_move(1)wordt dus de functie choose_computer_move(3) aangeroepen. Vervolgens wordt de functie value(3) aangeroepen en deze functie geeft UNDECIDED terug. Deze waarde wordt toegekend aan de lokale variabel best_value van de functie choose_computer_move. De code binnen de **if** wordt uitgevoerd. De lokale variabele best_value wordt gelijk gemaakt aan 0 en de **for**-lus wordt uitgevoerd. De variabele i krijgt de waarde 1 en de functie choose_human_move wordt aangeroepen met 2 * 3 + 1 is 7 als argument.

1.1.1.1. Vanuit de functie choose_computer_move(3) wordt dus de functie choose_human_move(7) aangeroepen. Vervolgens wordt de functie value(7) aangeroepen

en deze functie geeft UNDECIDED terug. Deze waarde wordt toegekend aan de lokale variabel best_value van de functie choose_human_move. De code binnen de **if** wordt uitgevoerd. De lokale variabele best_value wordt gelijk gemaakt aan 15 en de **for**-lus wordt uitgevoerd. De variabele i krijgt de waarde 1 en de functie choose_computer_move wordt aangeroepen met 2 * 7 + 1 is 15 als argument.

1.1.1.1.1. Vanuit de functie choose_human_move(7) wordt dus de functie choose_computer_move(15) aangeroepen. Vervolgens wordt de functie value(15) aangeroepen en deze functie geeft array-element value[pos - 15] is value[0] is 4 terug. Deze waarde wordt toegekend aan de lokale variabel best_value van de functie choose_computer_move. De voorwaarde van de **if** is nu **false** en de waarde 4 wordt teruggegeven.

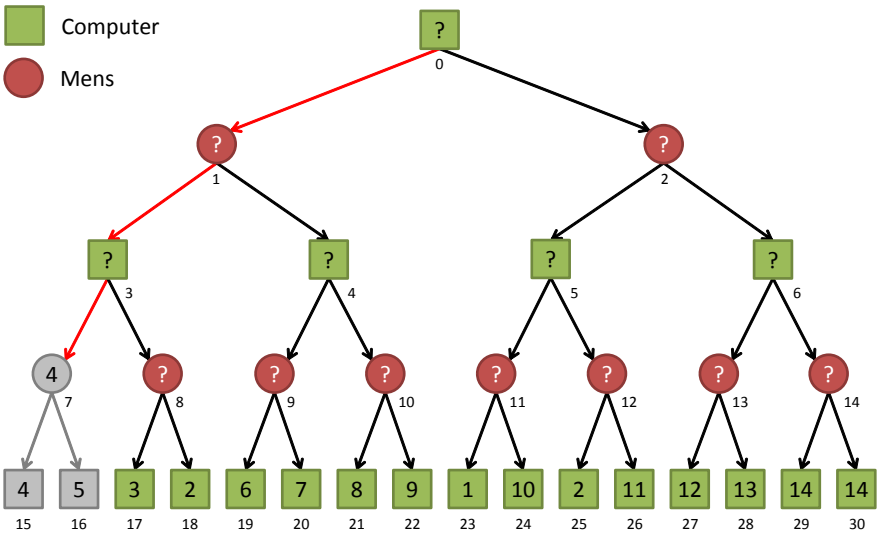
1.1.1.2. We keren terug naar de functie choose_human_move(7). De lokale variabele value krijgt de waarde 4 en omdat de waarde van value kleiner is dan de waarde van best_value (4 is kleiner dan 15) krijgt best_value de waarde 4. De **for**-lus wordt vervolgd. De variabele i krijgt de waarde 2 en de functie choose_computer_move wordt aangeroepen met 2 * 7 + 2 is 16 als argument.

1.1.1.2.1. Vanuit de functie choose_human_move(7) wordt dus de functie choose_computer_move(16) aangeroepen. Vervolgens wordt de functie value(16) aangeroepen en deze functie geeft array-element value[pos - 15] is value[1] is 5 terug. Deze waarde wordt toegekend aan de lokale variabel best_value van de functie choose_computer_move. De voorwaarde van de **if** is nu **false** en de waarde 5 wordt teruggegeven.

1.1.1.3. We keren terug naar de functie choose_human_move(7). De lokale variabele value krijgt de waarde 5 en omdat de waarde van value niet kleiner is dan de waarde van best_value (5 is niet kleiner dan 4) behoudt best_value de waarde 4. De **for**-lus is nu voltooid. Deze situatie is weergegeven in figuur 14.2. De functie choose_human_move(7) geeft de waarde 4 terug.

1.1.2. We keren terug naar de functie choose_computer_move(3). De lokale variabele value krijgt de waarde 4 en omdat de waarde van value groter is dan de waarde van best_value (4 is groter dan 0) krijgt best_value de waarde 4. De **for**-lus wordt vervolgd. De variabele i krijgt de waarde 2 en de functie choose_human_move wordt aangeroepen met 2 * 3 + 2 is 8 als argument.

1.1.2.1. Vanuit de functie choose_computer_move(3) wordt dus de functie choose_human_move(8) aangeroepen. Vervolgens geeft de functie value(8) UNDECIDED



Figuur 14.2: Spelboom net voor het einde van het uitvoeren van `choose_human_move(7)`.

terug. Deze waarde wordt toegekend aan de lokale variabel `best_value` van de functie `choose_human_move` en de code binnen de **`if`** wordt uitgevoerd. De lokale variabele `best_value` wordt gelijk gemaakt aan 15 en de **`for`**-lus wordt uitgevoerd. De variabele `i` krijgt de waarde 1 en de functie `choose_computer_move` wordt aangeroepen met `2 * 8 + 1` is 17 als argument.

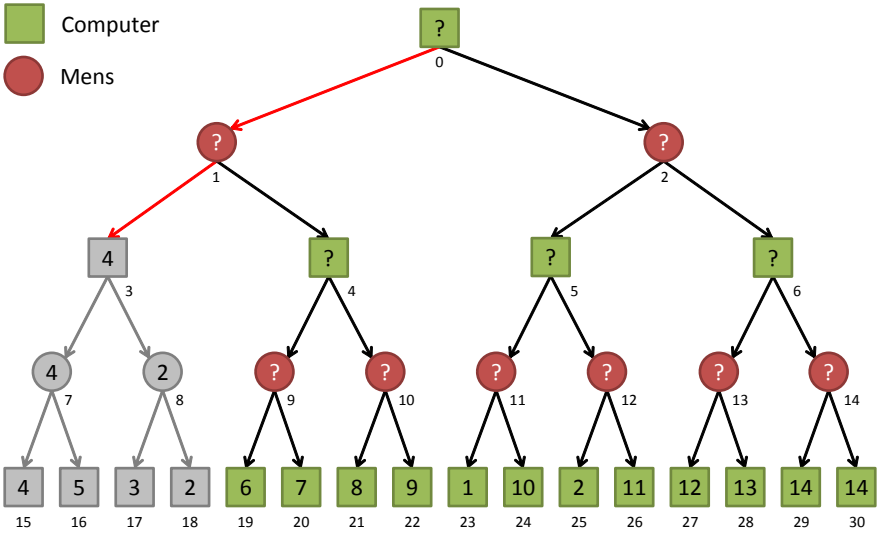
1.1.2.1.1. Vanuit de functie `choose_human_move(8)` wordt dus de functie `choose_computer_move(17)` aangeroepen. Vervolgens wordt de functie `value(17)` aangeroepen en die geeft array-element `value[17 - 15]` is `value[2]` is 3 terug. Deze waarde wordt teruggegeven vanuit de functie `choose_computer_move`.

1.1.2.2. We keren terug naar de functie `choose_human_move(8)`. De lokale variabele `value` krijgt de waarde 3 en omdat de waarde van `value` kleiner is dan de waarde van `best_value` (3 is kleiner dan 15) krijgt `best_value` de waarde 4. De **`for`**-lus wordt vervolgd. De variabele `i` krijgt de waarde 2 en de functie `choose_computer_move` wordt aangeroepen met `2 * 8 + 2` is 18 als argument.

1.1.2.2.1. Vanuit de functie `choose_human_move(8)` wordt dus de functie `choose_computer_move(18)` aangeroepen. Vervolgens wordt de functie `value(18)` aangeroepen en die geeft array-element `value[18 - 15]` is `value[3]` is 2 terug. Deze waarde wordt teruggegeven vanuit de functie `choose_computer_move`.

1.1.2.3. We keren terug naar de functie `choose_human_move(8)`. De lokale variabele `value` krijgt de waarde `2` en omdat de waarde van `value` kleiner is dan de waarde van `best_value` (`2` is kleiner dan `3`) krijgt `best_value` de waarde `2`. De **`for`**-lus is nu voltooid. De functie `choose_human_move(8)` geeft de waarde `2` terug.

1.1.3. We keren terug naar de functie `choose_computer_move(3)`. De lokale variabele `value` krijgt de waarde `2` en omdat de waarde van `value` niet groter is dan de waarde van `best_value` (`2` is niet groter dan `2`) behoudt `best_value` de waarde `4`. De **`for`**-lus is nu voltooid. Deze situatie is weergegeven in figuur 14.3. De functie `choose_computer_move(3)` geeft de waarde `4` terug.



Figuur 14.3: Spelboom net voor het einde van het uitvoeren van `choose_computer_move(3)`.

1.2. We keren terug naar de functie `choose_human_move(1)`. Omdat de geretourneerde waarde (`4`) kleiner is dan de waarde van `best_value` (`15`) krijgt `best_value` de waarde `4`. De **`for`**-lus wordt vervolgd. De functie `choose_computer_move` wordt aangeroepen met `2 * 1 + 2` is `4` als argument.

1.2.1. Vanuit de functie `choose_human_move(1)`wordt dus de functie `choose_computer_move(4)` aangeroepen. De lokale variabele `best_value` wordt gelijk gemaakt aan `0` en de **`for`**-lus wordt uitgevoerd. De functie `choose_human_move` wordt aangeroepen met `2 * 4 + 1` is `9` als argument.

1.2.1.1. Vanuit de functie `choose_computer_move(4)` wordt dus de functie `choose_human_move(9)` aangeroepen. De lokale variabele `best_value` wordt gelijk

gemaakt aan 15 en de **for**-lus wordt uitgevoerd. De functie choose_computer_move wordt aangeroepen met 2 * 9 + 1 is 19 als argument.

1.2.1.1.1. Vanuit de functie choose_human_move(9) wordt dus de functie choose_computer_move(19) aangeroepen die de waarde 6 teruggeeft.

1.2.1.2. We keren terug naar de functie choose_human_move(9). Omdat de geretourneerde waarde (6) kleiner is dan de waarde van best_value (15) krijgt best_value de waarde 6. De **for**-lus wordt vervolgd. De functie choose_computer_move wordt aangeroepen met 2 * 9 + 2 is 20 als argument.

1.2.1.2.1. Vanuit de functie choose_human_move(9) wordt dus de functie choose_computer_move(20) aangeroepen die de waarde 7 teruggeeft.

1.2.1.3. We keren terug naar de functie choose_human_move(9). Omdat de geretourneerde waarde (7) niet kleiner is dan de waarde van best_value (6) geeft de functie choose_human_move(9) de waarde 6 terug.

1.2.2. We keren terug naar de functie choose_computer_move(4). Omdat de geretourneerde waarde (6) groter is dan de waarde van best_value (0) krijgt best_value de waarde 6. De **for**-lus wordt vervolgd. De functie choose_human_move wordt aangeroepen met 2 * 4 + 2 is 10 als argument.

1.2.2.1. Vanuit de functie choose_computer_move(4) wordt dus de functie choose_human_move(10) aangeroepen. De lokale variabele best_value wordt gelijk gemaakt aan 15 en de **for**-lus wordt uitgevoerd. De functie choose_computer_move wordt aangeroepen met 2 * 10 + 1 is 21 als argument.

1.2.2.1.1. De functie choose_computer_move(21) geeft de waarde 8 terug.

1.2.2.2. Omdat de geretourneerde waarde (8) kleiner is dan de waarde van best_value (15) krijgt best_value de waarde 8. De **for**-lus wordt vervolgd. De functie choose_computer_move wordt aangeroepen met 2 * 10 + 2 is 22 als argument.

1.2.2.2.1. De functie choose_computer_move(22) geeft de waarde 9 terug.

1.2.2.3. Omdat de geretourneerde waarde (9) niet kleiner is dan de waarde van best_value (8) geeft de functie choose_human_move(9) de waarde 8 terug.

1.2.3. We keren terug naar de functie choose_computer_move(4). Omdat de geretourneerde waarde (8) groter is dan de waarde van best_value (6) geeft de functie choose_computer_move(3) geeft de waarde 8 terug.

1.3. We keren terug naar de functie `choose_human_move(1)`. Omdat de geretourneerde waarde (`8`) niet kleiner is dan de waarde van `best_value` (`4`) geeft de functie `choose_human_move(1)` de waarde `4` terug.

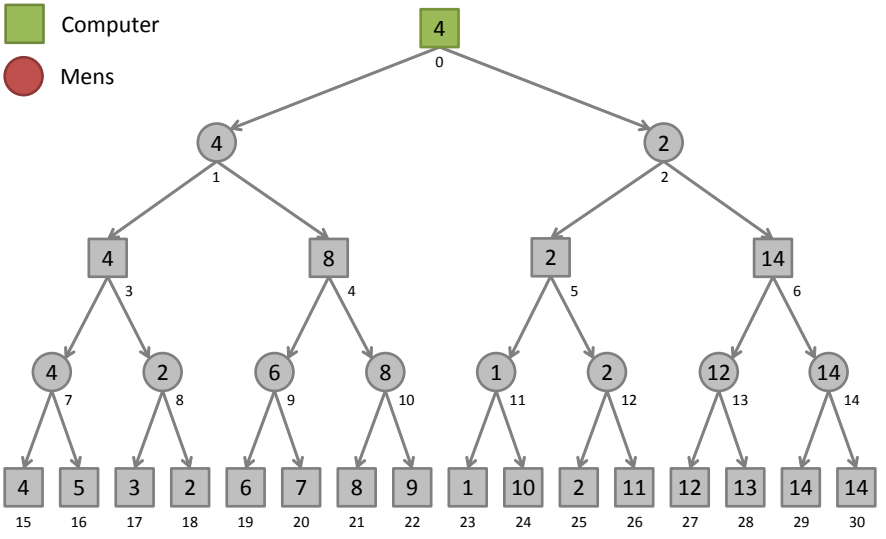
2. We keren terug naar de functie `choose_computer_move(0)`. Omdat de geretourneerde waarde (`4`) groter is dan de waarde van `best_value` (`0`) krijgt `best_value` de waarde `4`. De **`for`**-lus wordt vervolgd. De functie `choose_human_move` wordt aangeroepen met `2 * 0 + 2` is `2` als argument.

2.1 ···

2.2 ···

2.3 ··· De functie `choose_human_move(2)` geeft de waarde `2` terug.

3. We keren terug naar de functie `choose_computer_move(0)`. Omdat de geretourneerde waarde (`2`) niet groter is dan de waarde van `best_value` (`4`) geeft de functie `choose_computer_move(0)` de waarde `4` terug. Deze situatie is weergegeven in figuur 14.4.



Figuur 14.4: Spelboom net voor het einde van het uitvoeren van `choose_computer_move(0)`.

In de figuur 14.4 kun je zien dat met behulp van het minimax algoritme de best bereikbare positie voor de computer wordt gevonden.

In het programma `minimax0_print_tree.cpp` wordt de spelboom weergegeven in figuur 14.1 geprint met behulp van de recursieve functie `print_tree`. De berekende waarde voor elke positie weergegeven in figuur 14.4 wordt geprint met behulp van de recursieve functie `print_calculated_Tree`. Beide functies zijn hieronder weergegeven:

```
    if (level != 5) {
        print_tree(2 * pos + 2, level + 1);
        println("{:>{}}:{}", pos, level * 5, value(pos));
        print_tree(2 * pos + 1, level + 1);
    }
}

void print_calculated_tree(int pos, int level) {
    if (level != 5) {
        print_calculated_tree(2 * pos + 2, level + 1);
        println("{:>{}}:{}", pos, level * 5, (level % 2 == 0 ? ↩
 ↪ choose_computer_move(pos) : choose_human_move(pos)));
        print_calculated_tree(2 * pos + 1, level + 1);
    }
}
```

Het is ook mogelijk om de functies value_move_computer en value_move_human te combineren tot 1 functie, zie minimax1.cpp:

```
enum class Side {HUMAN, COMPUTER};
using enum Side;

int choose_move(Side s, int pos) {
    int best_value {value(pos)};
    if (best_value == UNDECIDED) {
        best_value = s == COMPUTER ? 0 : 15;
        for (int i {1}; i < 3; ++i) {
            int value {choose_move(s == COMPUTER ? HUMAN : ↩
 ↪ COMPUTER, 2 * pos + i)};
            if ((s == COMPUTER && value > best_value) || (s == ↩
 ↪ HUMAN && value < best_value)) {
                best_value = value;
            }
        }
    }
    return best_value;
}
```

Er zijn echter twee redenen om dit *niet* te doen:

- De functie wordt moeilijker te begrijpen en dus moeilijker te debuggen en/of aan te passen.

- Het programma wordt minder snel (omdat steeds extra condities moeten worden bepaald). In de praktijk wordt het minimax algoritme gebruikt voor spelletjes waarbij het aantal door te rekenen posities veel groter is dan in dit voorbeeld. Snelheid is dan belangrijk.

In het bovenstaande programma wordt alleen de waarde van de positie bepaald. Dit is uiteraard niet voldoende. We moeten ook weten welke zet we moeten doen om deze waarde te bereiken! Dit kan eenvoudig worden bereikt door bij het zoeken naar de maximale of minimale waarde de beste positie op te slaan in de uitvoerparameter `best_next_pos`. De functies `choose_computer_move` en `choose_human_move` moeten dan als volgt worden aangepast:

```
int choose_computer_move(int pos, int& best_next_pos) {
    int best_value {value(pos)};
    if (best_value == UNDECIDED) {
        best_value = 0;
        for (int i {1}; i < 3; ++i) {
            int dummy_pos;
            int value {choose_human_move(2 * pos + i, dummy_pos)};
            if (value > best_value) {
                best_value = value;
                best_next_pos = 2 * pos + i;
            }
        }
    }
    return best_value;
}

int choose_human_move(int pos, int& best_next_pos) {
    int best_value {value(pos)};
    if (best_value == UNDECIDED) {
        best_value = 15;
        for (int i {1}; i < 3; ++i) {
            int dummy_pos;
            int value {choose_computer_move(2 * pos + i, dummy_pos)};
            if (value < best_value) {
                best_value = value;
                best_next_pos = 2 * pos + i;
            }
        }
    }
```

```
    return best_value;
}
```

In de functie `main` kunnen we deze functies als volgt gebruiken om beste zet voor de computer respectievelijk de gebruiker te bepalen:

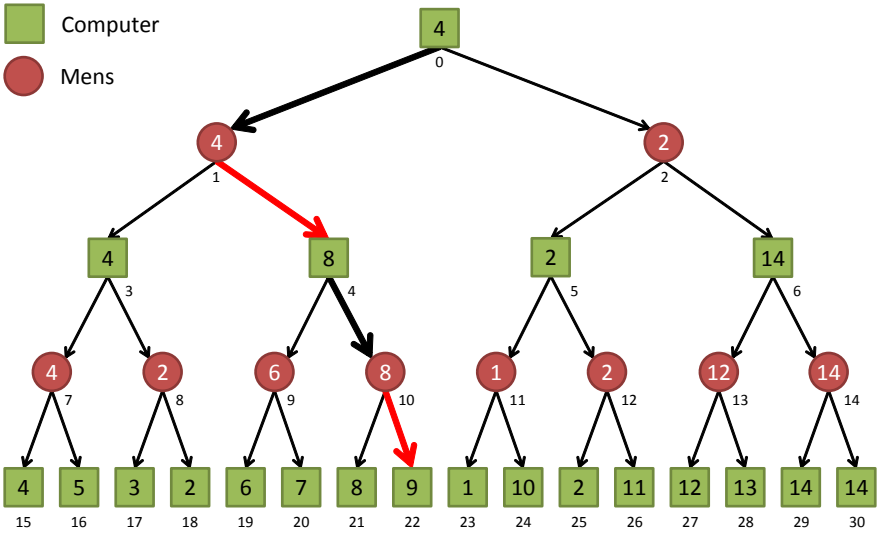
```
int main() {
    int pos {0}, best_next_pos;
    while (pos < 15) {
        int best_value {choose_computer_move(pos, best_next_pos)};
        println("Minimaal te behalen Maximale waarde = {}", ↩
 ↪ best_value);
        pos = best_next_pos;
        println("Computer kiest positie: {}", pos);
        if (pos < 15) {
            int pos_l {2 * pos + 1};
            int pos_r {2 * pos + 2};
            println("Je kunt kiezen voor positie {} of positie ↩
 ↪ {}", pos_l, pos_r);
            choose_human_move(pos, best_next_pos);
            println("Pssst, {} is de beste keuze.", best_next_pos);
            do {
                print("Maak je keuze: ");
                cin >> pos;
            } while (pos != pos_l && pos != pos_r);
        }
    }
    println("Behaalde waarde = {}", value(pos));
}
```

Een mogelijke uitvoer van dit programma `minimax2.cpp` is als volgt:

```
Minimaal te behalen Maximale waarde = 4
Computer kiest positie: 1
Je kunt kiezen voor positie 3 of positie 4
Pssst, 3 is de beste keuze.
Maak je keuze: 4
Minimaal te behalen Maximale waarde = 8
Computer kiest positie: 10
Je kunt kiezen voor positie 21 of positie 22
Pssst, 21 is de beste keuze.
```

```
Maak je keuze: 22
Behaalde waarde = 9
```

De bijbehorende weg door de spelboom is weergegeven in figuur 14.5. De computer behaald in dit geval de waarde 9. Deze waarde is hoger dan de minimaal te behalen maximale waarde van 4. Dit komt omdat de mens, ondanks de tips die hij/zij krijgt, steeds de zwakste keuze maakt.

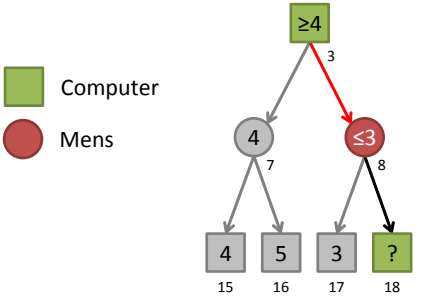


Figuur 14.5: Mogelijke weg door de spelboom bij het uitvoeren van minimax2.cpp.

14.1.2 Alpha-beta pruning

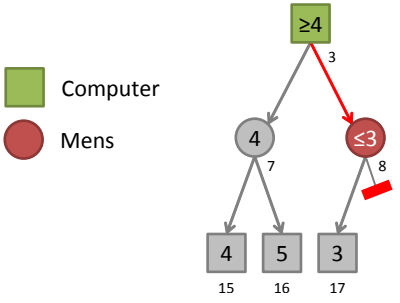
Bij het zoeken naar een *minimale* waarde op een bepaalde positie kun je stoppen zodra je een waarde hebt gevonden die *kleiner of gelijk* is aan de tot dan toe gevonden *maximale* waarde in de *bovenliggende* positie.

In figuur 14.6 is de waarde van positie 3 gelijk aan het maximum van de waarden van de posities 7 en 8. De waarde van positie 7 is het minimum van de waarden van de posities 15 en 16. De waarde van positie 7 is dus 4 (het minimum van 4 en 5). Het tijdelijke maximum in positie 3 wordt dus (na het berekenen van de waarde van positie 7) gelijk gemaakt aan 4. De waarde van positie 8 is het minimum van positie 17 en 18. Bij het zoeken naar dit minimum wordt eerst de waarde 3 voor positie 17 gevonden. Omdat deze waarde kleiner of gelijk is aan het tot nu toe gevonden maximum in de bovenliggende positie (de waarde 4) kunnen we meteen stoppen! Het is dus niet nodig om de waarde van positie 18 te bepalen. Denk



Figuur 14.6: In deze spelboom is het niet nodig om de waarde van positie 18 te bepalen.

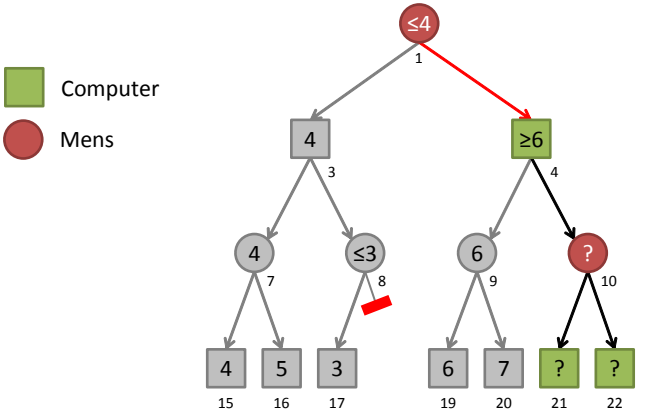
maar mee. Als deze waarde <3 (b.v. 0) is, dan wordt deze waarde gekozen als minimale waarde. Echter in de bovenliggende positie (positie 3) wordt toch voor de maximale waarde 4 gekozen. Als de waarde van positie 18 >3 (b.v. 15) is, dan wordt de waarde 3 (van positie 17) gekozen als minimale waarde. Echter in de bovenliggende positie (positie 3) wordt toch voor de maximum waarde 4 gekozen. De waarde van positie 18 hoeft dus helemaal niet bepaald te worden. Dit deel van de boom hoeft niet doorzocht te worden en we kunnen als het ware de spelboom snoeien (Engels: to prune), zie figuur 14.7



Figuur 14.7: De tak naar positie 18 kan worden gesnoeid uit de boom.

Om dit mogelijk te maken moet wel de tot dan toe gevonden maximale waarde van de bovenliggende positie worden doorgegeven aan de onderliggende posities. Dit gebeurt in het onderstaande programma `alpha_beta0.cpp` door middel van de parameter alpha. Deze naam is toevallig zo gekozen door de bedenkers van dit algoritme en heeft verder geen betekenis [15].

Bij het zoeken naar een maximale waarde op een bepaalde positie kun je stoppen zodra je een waarde hebt gevonden die *groter of gelijk* is aan de tot dan toe gevonden *minimale* waarde in de *bovenliggende* positie.

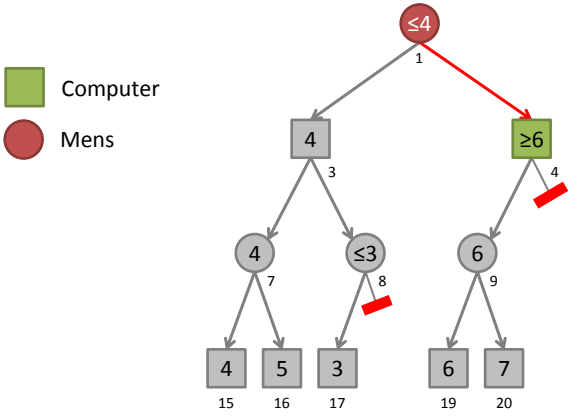


Figuur 14.8: In deze spelboom is het niet nodig om de waarde van posities 10, 21 en 22 te bepalen.

In figuur 14.8 is de waarde van positie 1 gelijk aan het minimum van de waarden van de posities 3 en 4. De waarde van positie 3 is hiervoor al bepaald (4). Het tijdelijke minimum in positie 1 wordt dus (na het berekenen van de waarde van positie 3) gelijk gemaakt aan 4. De waarde van positie 4 is het maximum van positie 9 en 10. Bij het zoeken naar dit maximum wordt eerst de waarde 6 voor positie 9 gevonden (door het minimum te bepalen van positie 19 en 20). Omdat het tot nu toe gevonden maximum in positie 4 (de waarde 6) groter of gelijk is aan het tot nu toe gevonden minimum in de bovenliggende positie (de waarde 4) kunnen we meteen stoppen! Het is dus niet nodig om de waarden van posities 10, 21 en 22 te bepalen. Denk maar mee. Als de waarde van positie 10 >6 (b.v. 15) is, dan wordt deze waarde gekozen als maximale waarde. Echter in de bovenliggende positie (positie 1) wordt toch als minimale waarde 4 gekozen. Als de waarde van positie 10 <6 (b.v. 0) is, dan wordt de waarde 6 (van positie 9) gekozen als maximale waarde. Echter in de bovenliggende positie (positie 1) wordt toch als minimale waarde 4 gekozen. De waarden van posities 10, 21 en 22 hoeven dus helemaal niet bepaald te worden. Dit deel van de boom hoeft niet doorzocht te worden en we kunnen de spelboom weer snoeien, zie figuur 14.7

Om dit mogelijk te maken moet wel de tot dan toe gevonden minimale waarde van de bovenliggende positie worden doorgegeven aan de onderliggende posities. Dit gebeurt in het onderstaande programma `alpha_beta0.cpp` door middel van de parameter beta. Deze naam is toevallig zo gekozen door de bedenkers van dit zogenoemde alpha-beta pruning algoritme en heeft verder geen betekenis [15].

```
import std;
using namespace std;
```



Figuur 14.9: De tak naar positie 10 kan worden gesnoeid uit de boom.

```
int value(int pos);
int choose_computer_move(int pos, int alpha = 0, int beta = 15);
int choose_human_move(int pos, int alpha = 0, int beta = 15);

constexpr int UNDECIDED {-1};

int value(int pos) {
    static const int value[16] {4, 5, 3, 2, 6, 7, 8, 9, 1, 10, 2, ↩
 ↪ 11, 12, 13, 14, 14};
    if (pos >= 15 && pos < 31)
        return value[pos - 15]; // return known value
    return UNDECIDED;
}

int choose_computer_move(int pos, int alpha, int beta) {
    int best_value {value(pos)};
    if (best_value == UNDECIDED) {
        best_value = alpha;
        for (int i {1}; best_value < beta && i < 3; ++i) {
            int value {choose_human_move(2 * pos + i, alpha, beta)};
            if (value > best_value) {
                best_value = value;
                alpha = best_value;
            }
        }
    }
```

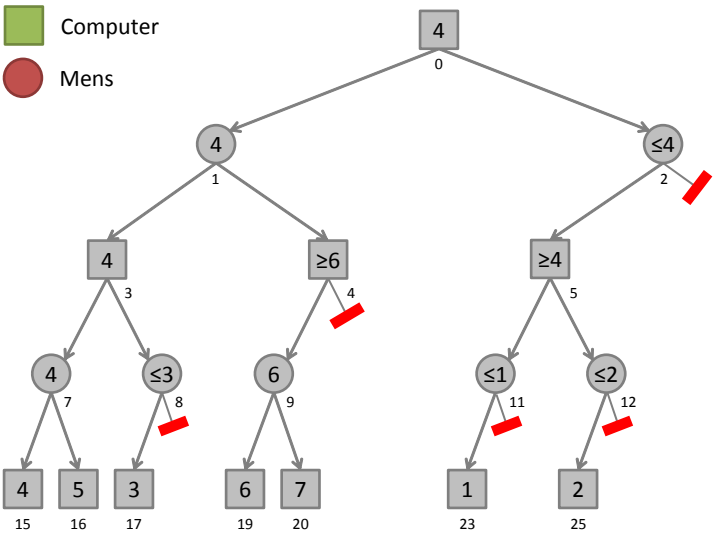
```
    return best_value;
}

int choose_human_move(int pos, int alpha, int beta) {
    int best_value {value(pos)};
    if (best_value == UNDECIDED) {
        best_value = beta;
        for (int i {1}; best_value > alpha && i < 3; ++i) {
            int value {choose_computer_move(2 * pos + i, alpha, ↩
 ↪ beta)};
            if (value < best_value) {
                best_value = value;
                beta = best_value;
            }
        }
    }
    return best_value;
}

int main() {
    int value {choose_computer_move(0)};
    println("Minimaal te behalen Maximale waarde = {}", value);
}
```

In een uitgebreidere versie van het bovenstaande programma, genaamd `alpha_beta0_verbose.cpp`, wordt output naar het beeldscherm gestuurd als er gesnoeid wordt. De uitvoer van dit programma is als volgt:

```
snoei node 18
snoei node 10
snoei node 24
snoei node 26
snoei node 6
Minimaal te behalen Maximale waarde = 4
```

In figuur 14.10 is te zien welke takken het alpha-beta algoritme de spelboom uit figuur 14.1 heeft gesnoeid.

Bij de eerste aanroep van de functie `value_move_computer` moet voor `alpha` het absolute minimum (in dit geval 0) en voor `beta` het absolute maximum (in dit geval 15) worden ingevuld. Dit is logisch want `alpha` is het tot nu toe gevonden maximum in de bovenliggende



Figuur 14.10: De gesnoeide boom.

posities (we hebben in dit geval geen bovenliggende posities dus het tot nu toe gevonden maximum is het absolute minimum) en `beta` is het tot nu toe gevonden minimum in de bovenliggende posities (we hebben in dit geval geen bovenliggende posities dus het tot nu toe gevonden minimum is het absolute maximum).

Je ziet dat bij het bepalen van de maximale waarde in een positie de waarde van `best_value` wordt geïnitialiseerd met `alpha` en niet met het absolute minimum (`0`). Een, in een bovenliggende positie gevonden, tijdelijk maximum mag namelijk *niet* in een (2 lagen dieper) onderliggende positie op een *lagere* waarde worden gezet! Als je `best_value` initialiseert met `0`, wordt de positie 26 *niet* gesnoeid! Probeer maar.

Om dezelfde reden moet bij het bepalen van de minimale waarde in een positie de waarde van `best_value` worden geïnitialiseerd met `beta` en niet met het absolute maximum (`15`). Een, in een bovenliggende positie gevonden, tijdelijk minimum mag namelijk *niet* in een (2 lagen dieper) onderliggende positie op een *hogere* waarde worden gezet!

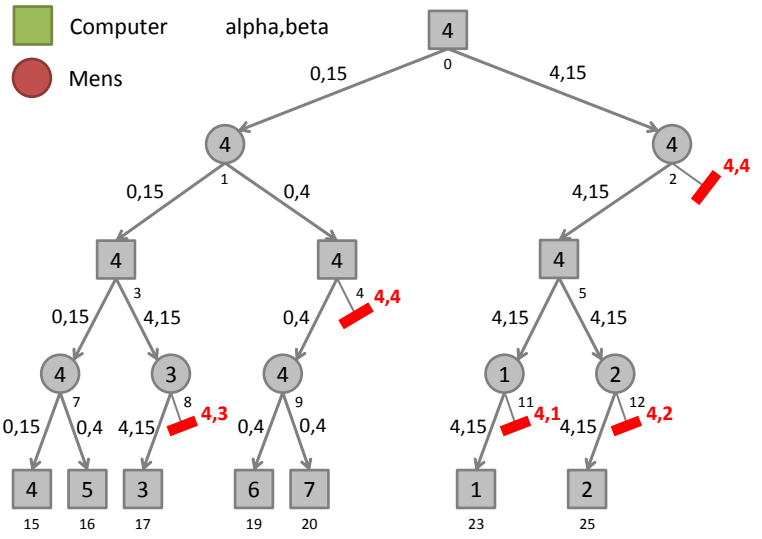
Merk op dat in de functie `choose_computer_move` de expressie `best_value < beta` vervangen kan worden door `alpha < beta` omdat `best_value` en `alpha` in de **`for`**-lus altijd dezelfde waarde hebben. In de **`for`**-lus is het gebruik van `best_value` dus overbodig. Merk op dat in de functie `choose_human_move` de expressie `best_value > alpha` vervangen kan worden door `beta > alpha` omdat `best_value` en `beta` in de **`for`**-lus altijd dezelfde waarde hebben. In de **`for`**-lus is het gebruik van `best_value` dus overbodig. De expressie `beta > alpha` kan

vervangen worden door `alpha < beta` die exact hetzelfde betekent. In het onderstaande programma alpha_beta1.cpp zijn deze wijzigingen doorgevoerd:

```
int choose_computer_move(int pos, int alpha, int beta) {
    int best_value = value(pos);
    if (best_value == UNDECIDED) {
        for (int i {1}; alpha < beta && i < 3; ++i) {
            int value {choose_human_move(2 * pos + i, alpha, beta)};
            if (value > alpha) {
                alpha = value;
            }
        }
        best_value = alpha;
    }
    return best_value;
}

int choose_human_move(int pos, int alpha, int beta) {
    int best_value = value(pos);
    if (best_value == UNDECIDED) {
        for (int i {1}; alpha < beta && i < 3; ++i) {
            int value {choose_computer_move(2 * pos + i, alpha, ↩
 ↪ beta)};
            if (value < beta) {
                beta = value;
            }
        }
        best_value = beta;
    }
    return best_value;
}
```

De voorwaarde om te snoeien is dus in de functies choose_computer_move en choose_human_move exact hetzelfde, namelijk `alpha` $\geq$ `beta`. In figuur 14.11 zijn de waarden van `alpha` en `beta` bij elke aanroep van choose_computer_move en choose_human_move gegeven. Om de werking van dit alpha-beta algoritme helemaal te doorgronden is het heel zinvol om het uitvoeren van het programma alpha_beta1_verbose.cpp stap voor stap te volgen, eventueel met behulp van een debugger.

In het bovenstaande programma wordt alleen de waarde van de positie bepaald. Dit is uiteraard niet voldoende. We moeten ook weten welke zet we moeten doen om deze



Figuur 14.11: De waarde van alpha en beta bij elke aanroep van choose_computer_move en choose_human_move.

waarde te bereiken! Dit kan weer worden bereikt door bij het zoeken naar de maximale of minimale waarde de beste positie op te slaan in de uitvoerparameter best_next_pos. De functies choose_computer_move en choose_human_move moeten dan worden aangepast zoals weergegeven in het onderstaande programma alpha_beta2.cpp.

```
int choose_computer_move(int pos, int& best_next_pos, int alpha, ↩
 ↪ int beta) {
    int best_value {value(pos)};
    if (best_value == UNDECIDED) {
        for (int i {1}; alpha < beta && i < 3; ++i) {
            int dummy_pos;
            int value {choose_human_move(2 * pos + i, dummy_pos, ↩
 ↪ alpha, beta)};
            if (value > alpha) {
                alpha = value;
                best_next_pos = 2 * pos + i;
            }
        }
        best_value = alpha;
    }
    return best_value;
}
```

```
int choose_human_move(int pos, int& best_next_pos, int alpha, int ↩
 ↪ beta) {
    int best_value {value(pos)};
    if (best_value == UNDECIDED) {
        for (int i {1}; alpha < beta && i < 3; ++i) {
            int dummy_pos;
            int value {choose_computer_move(2 * pos + i, ↩
 ↪ dummy_pos, alpha, beta)};
            if (value < beta) {
                beta = value;
                best_next_pos = 2 * pos + i;
            }
        }
        best_value = beta;
    }
    return best_value;
}
```

De volgorde waarin de mogelijke volgende posities worden onderzocht heeft invloed op het gedrag van het alpha-beta algoritme. In het programma `alpha_beta1_verbose.cpp` wordt deze volgorde bepaald door de regel:

```
        for (i = 1; alpha < beta && i < 3; ++i) {
```

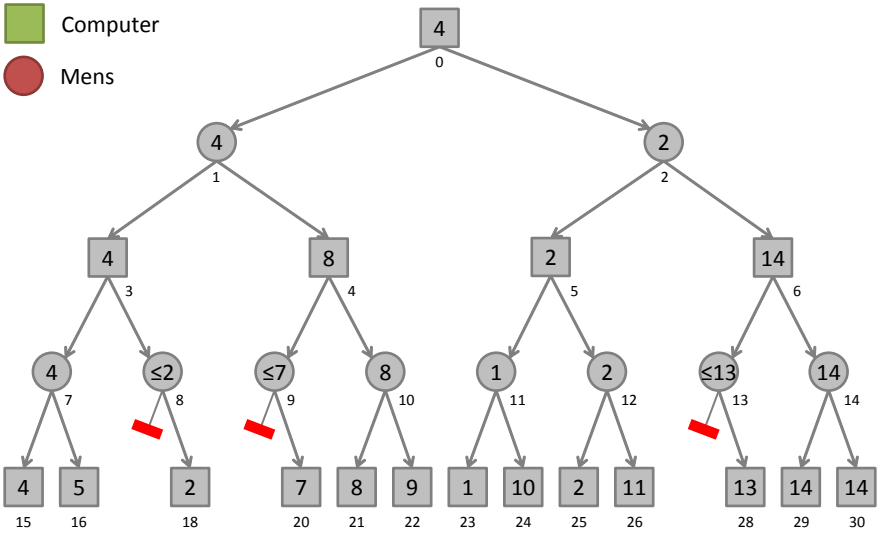
De mogelijke posities worden dus van links naar rechts doorzocht. We kunnen de mogelijke posities ook van rechts naar links doorzoeken door de bovenstaande regel te vervangen door de regel:

```
        for (i = 2; alpha < beta && i > 0; --i) {
```

In het resulterende programma `alpha_beta3_verbose.cpp` kan heel wat minder worden gesnoeid. De uitvoer van dit programma is als volgt:

```
snoei node 27
snoei node 19
snoei node 17
Minimaal te behalen Maximale waarde = 4
```

In figuur 14.12 is te zien welke takken het alpha-beta algoritme de spelboom uit figuur 14.1 heeft gesnoeid als de boom van rechts naar links wordt doorlopen.



Figuur 14.12: De gesnoeide boom als de boom van rechts naar links wordt doorlopen.

14.1.3 Verschillende implementaties van Boter, Kaas en Eieren

In deze paragraaf wordt verwezen naar programma's die Boter, Kaas en Eieren (Engels: Tic-Tac-Toe) spelen. Deze programma's zijn geïnspireerd door paragraaf 11.2 van het boek *Data Structures and Problem Solving Using C++* van Weiss[25]. Al deze programma's maken gebruik van een matrix class (zie `Matrix.cppm`) om de rondjes en kruisjes in op te slaan.

Online zijn de volgende programma's beschikbaar:

- Tic-Tac-Toe programma met minimax algoritme: `Tic-Tac_slow.cpp`.
- Tic-Tac-Toe programma met alpha-beta pruning: `Tic-Tac_a-b.cpp`.
- Tic-Tac-Toe programma met alpha-beta pruning en transposition table met behulp van `std::map`: `Tic-Tac_map.cpp`.

De bovenstaande programma's kiezen geen zet als alle mogelijke zetten tot verlies leiden. Dit is bij Tic-Tac-Toe geen probleem omdat de computer niet kan verliezen maar bij ingewikkeldere spellen is dit wel een probleem! Dit probleem wordt gedemonstreerd en opgelost in het programma `Tic-Tac_a-b_loser.cpp` (met dank aan Marc Cornet).

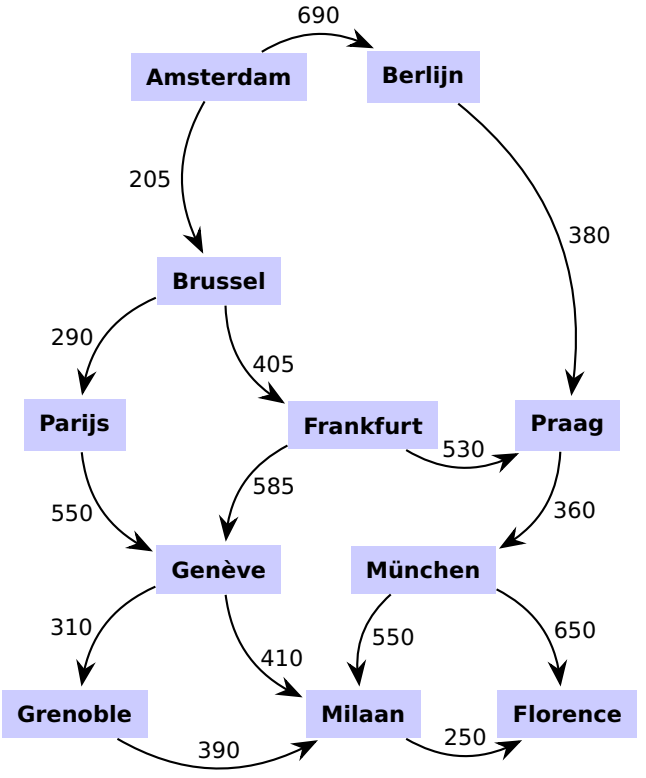
Bij het gebruik van een transpositietabel kun je de diepte waarop de tabel gebruik wordt limiteren, zie `Tic-Tac_map_restricted_depth.cpp`. Daarbij moet de optimale waarde van `MAX_TABLE_DEPTH` worden bepaald. Dit kun je doen met het programma `Tic-Tac_determine_MAX_TABLE_DEPTH.cpp`.

14.2 Het zoeken van het kortste pad in een graph

In deze paragraaf wordt verwezen naar een programma dat het kortste pad in een graph kan vinden. Dit programma is geïnspireerd door hoofdstuk 15 van het boek *Data Structures and Problem Solving Using C++* van Weiss[25].

Een graaf (Engels: graph) bestaat uit een verzameling punten met lijnen ertussen. De punten worden knopen (Engels: vertices or nodes) en de lijnen worden kanten (Engels: edges) genoemd. Ik gebruik in het vervolg van dit dictaat de Engelse benamingen. In figuur 14.13 is een graph weergegeven waarin de vertices Europese steden zijn. De verzameling van alle vertices wordt V genoemd en in figuur 14.13 geldt $V = \{$Amsterdam, Berlijn, Brussel, Parijs, Frankfurt, Praag, Genève, München, Grenoble, Milaan, Florence$\}$. De kardinaliteit (het aantal elementen) van de verzameling V wordt aangeduid met $|V|$. Dus in figuur 14.13 geldt $|V| = 11$. Elke edge vormt de verbinding tussen twee vertices. Als je vanuit vertex v via één edge vertex w kunt bereiken dan zeggen we dat w aanliggend is aan (adjacent to) v. De verzameling van alle edges in een graph wordt E genoemd. De edges in figuur 14.13 hebben geen naam, wel kunnen we vaststellen dat voor de graph in deze figuur geldt $|E| = 15$. Aan een edge kan een zogenoemd gewicht (Engels: weight) gekoppeld worden. Dit gewicht staat voor de moeite die het kost om via deze edge vanuit de ene vertex de andere vertex te bereiken. In figuur 14.13 staat het gewicht bij de edges voor de afstand (in kilometer) tussen de twee steden die door de betreffende edge verboden worden. Als alle edges van een graph een bepaalde richting hebben dan noemen we zo'n graph een directed graph (Nederlands: gerichte graaf), vaak afgekort als digraph. Een edge met een richting wordt aangegeven met een pijl in plaats van met een gewone lijn en meestal ook gewoon pijl (Engels: arc) genoemd. Je kunt dan alleen van de ene naar de andere vertex reizen in de richting van de arc. Je ziet dat figuur 14.13 een directed graph is. Blijkbaar kun je in die graph wel vanuit Amsterdam Berlijn bereiken maar is het niet mogelijk om vanuit Berlijn Amsterdam te bereiken.

Een rijtje vertices die afgelopen kunnen worden door één of meer edges (achtereenvolgens) te volgen wordt een wandeling (Engels: walk) genoemd. In figuur 14.13 is de rij vertices (Parijs, Genève, Grenoble, Milaan) een walk van Parijs naar Milaan. Het gewicht van de walk is gelijk aan de som van alle in de walk gevolgde edges. Het gewicht van de walk (Parijs, Genève, Grenoble, Milaan) is dus $550 + 310 + 390 = 1250$. Een walk van een vertex naar zichzelf wordt een cykel (Engels: cycle) genoemd. Een walk zonder cycle wordt een pad (Engels: path) genoemd. De zojuist als voorbeeld genoemde walk is dus ook een path. Al je goed kijkt, zie je dat in figuur 14.13 geen enkele cycle voorkomt. Een directed graph zonder cycles wordt een directed acyclic graph genoemd, vaak afgekort tot DAG.



Figuur 14.13: Wat is de kortste weg van Amsterdam naar Florence in deze DAG?

Tussen twee vertices kunnen meerdere paden bestaan. Naast het hierboven gegeven path is er nog een path van Parijs naar Milaan namelijk: (Parijs, Genève, Milaan). Het kortste pad tussen twee vertices in een weighted graph is gedefinieerd als het pad met een lager gewicht dan alle andere paden tussen deze twee vertices (en dus niet als het pad met de minste edges). Het zoeken van het kortste pad in een graph is een algoritme dat veel wordt toegepast[167]. In deze paragraaf wordt verwezen naar een programma waarin vier verschillende algoritmen geïmplementeerd zijn om het kortste pad te vinden.

Een graph waarin elke vertex rechtstreeks met elke andere vertex is verbonden wordt een volledige (Engels: complete) graph genoemd. Een complete graph heeft $\frac{1}{2} \cdot |V| \cdot (|V| - 1)$ edges[168]. Voor een complete digraph geldt: $|E| = |V| \cdot (|V| - 1)$, er is dan namelijk tussen elk

[167] graphs kunnen in verrassend veel vakgebieden worden toegepast, zie http://en.wikipedia.org/wiki/graph_theory#Applications.

[168] Dit is eenvoudig te bewijzen, zie http://nl.wikipedia.org/wiki/Grafentheorie#De_volledige_graaf.

paar vertices één edge nodig voor de heenweg en één edge nodig voor de terugweg. In veel praktische toepassingen is het aantal edges veel kleiner dan het maximale aantal edges. In figuur 14.13 geldt dat het maximale aantal edges gelijk is aan: $11 \cdot (11-1) = 110$ terwijl er maar 15 aanwezig zijn. Een graph waarin het aantal edges veel kleiner is dan het maximaal aantal mogelijke edges wordt een schaarse (Engels: sparse) graph genoemd.

In het programma waarnaar in deze paragraaf wordt verwezen, wordt ervan uitgegaan dat er sprake is van een sparse graph. Het is voor sparse graphs verstandiger om in elke vertex een lijst met uitgaande edges (met hun gewicht) op te slaan in plaats van een matrix te gebruiken waarin voor elke combinatie van vertices het gewicht van de bijbehorende edge is opgeslagen. Als de edges met hun gewicht opgeslagen worden in lijsten per vertex dan is daar voor een sparce graph minder geheugen voor nodig ($O(|E|)$) dan voor het opslaan van een matrix met de gewichten van alle mogelijke edges ($O(|V|^2)$).

In het programma `paths.cpp` worden vier verschillende algoritmen geïmplementeerd om het kortste pad te vinden:

- Breadth-first search; Dit algoritmen kan alleen gebruikt worden als de edges van de graph geen gewicht hebben (of allemaal hetzelfde gewicht hebben). Het is geen probleem als de graph cycles bevat. De executietijd van dit algoritme is $O(|E|)$.
- Topological search; Dit algoritmen kan alleen gebruikt worden als er geen enkele cycle in de graph voorkomt. Het is geen probleem als de edges een gewicht hebben, ook niet als dit gewicht negatief is. De executietijd van dit algoritme is $O(|E|)$.
- Dijkstra; Dit algoritmen kan alleen gebruikt worden als er geen enkele edge met een negatief gewicht in de graph voorkomt. Het is geen probleem als de edges een gewicht hebben en het is ook geen probleem als er cycles in de graph voorkomen. De executietijd van dit algoritme is $O(|E| \cdot \log |V|)$.
- Bellman-Ford; Dit algoritmen kan gebruikt worden als er cycles in de graph voorkomen en als er edges met een negatief gewicht in de graph voorkomen. Er mag echter geen enkele cycle met een negatief gewicht in de graph voorkomen. De kortste weg is in dat geval namelijk $-\infty$. Omdat elke keer, als de cycle met een negatief gewicht wordt doorlopen, er een nog kortere weg wordt gevonden. De executietijd van dit algoritme is $O(|E| \cdot |V|)$.

Als we dit programma gebruiken om het kortste pad van Amsterdam naar Florence in de graph die gegeven in figuur 14.13 te berekenen dan ziet de uitvoer er als volgt uit:

```
p = positive (graph_positive.png),
n = negative (graph_negative.png),
```

```
a = acyclic  (graph_acyclic.png),
s = steden   (graph_steden.png) or
q = quit.
Choose graph: s
Show graph? (y/n): n
Reading file ..............
Enter start node: Amsterdam
Enter destination node: Florence
Enter algorithm u = unweighted, d = dijkstra, n = negative or
a = acyclic: a
(Costs are: 1705) Amsterdam to Brussel to Parijs to Genève to Milaan
to Florence
```

15

Van C naar C++ (de details)

In hoofdstuk 1 hebben we enkele kleine verbeteringen van C++ ten opzichte van zijn voorganger C besproken. In dit hoofdstuk bespreken we nog een aantal kleine verbeteringen.

15.1 Binaire getallen

In een C++ en C23 [11] programma kun je constante getallen niet alleen meer in het decimale, octale[169] en hexadecimale[170] talstelsel invoeren maar ook in het binaire talstelsel. Binaire getallen kun je invoeren door ze vooraf te laten gaan door `0b` of `0B`. Dus de code `int x = 0b10111101;` initialiseert de variabele `i` met de waarde 189 want 10111101_2 is 189_{10}.

15.2 Het type bool

Als je in een C-programma (van vóór de C23 standaard maar na de C99 standaard) een variabele van het type **bool** wilt definiëren, dan moet je de preprocessor directive **#include** `stdbool.h` gebruiken. In C++ en C23 [11] is een preprocessor directive niet nodig, als je het type **bool** wilt gebruiken. In C++ en C23 is **bool** een keyword en hoef je dus geen **#include**

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

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

te gebruiken. Met dit keyword kun je booleaanse variabelen definiëren van het type **bool**. Een variabele van dit type heeft slechts twee mogelijke waarden: **true** en **false**.

15.3 Het type byte

Type byte is bedoeld voor het opslaan van een byte (8 bits data). Deze data wordt in tegenstelling tot een int8_t of een **char** niet geïnterpreteerd als een getal of een karakter maar als een ongestructureerde verzameling van bits. Alleen de bitwise logische operatoren en de schuifoperatoren kunnen gebruikt worden bij dit type. Bijvoorbeeld (byte.cpp)

```
    byte b1 {234};
    println("b1 = {}", to_integer<unsigned int>(b1));
    byte b2 {b1 >> 3};
    println("b2 = {}", to_integer<uint8_t>(b2));
    byte b3 {b1 + b2}; // error:: no match for 'operator+' (operand ↩
↪ types are 'std::byte' and 'std::byte)'
```

Dit datatype komt vooral van pas als je met 'ruwe' data werkt, bijvoorbeeld bij het samenstellen van een I^2C bericht. In het verleden werd hier meestal het type **char** of uint8_t voor gebruikt maar dat is niet ideaal omdat deze types worden geïnterpreteerd als karakters respectievelijk getallen. Met behulp van de template functie to_integer<T> kan een byte worden omgezet naar een getal van het type T, bijvoorbeeld **int** of uint8_t.

15.4 Meer floating point types

De types **float** en **double** zijn types die implementatie afhankelijk zijn[171], net zoals **short**, **int** enzovoort[172]. in C++11 zijn de types int8_t, int16_t, int32_t, int64_t geïntroduceerd waarvan het aantal bits vast gedefinieerd is. Een int32_t gebruikt altijd 32 bits, onafhankelijk van de implementatie, voor het opslaan van een geheel getal. Sinds C++23 kun je ook floating point getallen met een vast aantal bits gebruiken. float16_t, float32_t, float64_t en

[171] Volgens de C++-standaard moet een **double** minstens dezelfde precisie hebben als een **float**. De meeste compilers definiëren **float** als een 32-bits single-precision floating-point getal en **double** als een 64-bits double-precision floating-point getal volgens de IEEE 754 standaard. De compiler voor de Arduino Uno (die een 8-bit ATmega328P processor bevat) definieert **float** en **double** beide als 32-bits single-precision floating-point getallen, zie https://docs.arduino.cc/language-reference/en/variables/data-types/double/.

[172] Volgens de C++-standaard moet een **int** minstens hetzelfde aantal bits hebben als een **short**. De meeste compilers definiëren **short** als een 16-bits signed integer en **int** als een 32-bits signed integer. De compiler voor de Arduino Uno definieert **short** en een **int** beide als een 16-bits signed integer, zie https://docs.arduino.cc/language-reference/en/variables/data-types/int/.

`float128_t` zijn types die respectievelijk 16, 32, 64 en 128 bits gebruiken voor het opslaan van een floating point getal volgens de IEEE 754 standaard[173]. `bfloat16_t` brain floating point is een type dat 16 bits gebruikt voor het opslaan van een floating point getal dat wordt gebruikt in bepaalde AI-processors (o.a. in NVIDIA-GPU's).[174].

15.5 Digit separators

Om lange getallen beter leesbaar te maken mag in C++ en C23 [11] een enkel aanhalingsteken (') gebruikt worden om groepjes cijfers te onderscheiden.

```
uint16_t uart_settings = 0b1011'1001'000'1011;
int32_t miljard = 1'000'000'000;
```

15.6 Zero-initialisatie

Zero-initialisatie van een variabele kan worden gedaan met behulp van de syntax `T t {};`. Deze syntax werkt voor alle types. De variabele `t` krijgt de nul-waarde van het type `T`. Voor **`int`** is dit `0`, voor **`double`** is dit `0.0`, voor pointers is dit **`nullptr`**, zie voetnoot 62, enz. Voor user-defined types is dit de waarde die wordt verkregen door het aanroepen van de constructor zonder argumenten van het type `T` (de default constructor), zie paragraaf 2.4.

15.7 Aangewezen initialisatie

Met behulp van aangewezen initialisatie (Engels designated initializers) kun je datamembers initialiseren met hun naam. Deze syntax is vooral handig bij het initialiseren van structuren met veel datamembers, omdat het de leesbaarheid van de code verbetert en het risico op fouten bij het initialiseren van datamembers vermindert.

Als de struct `Persoon` bijvoorbeeld de volgende datamembers heeft:

```
struct Persoon {
    string roepnaam;
    vector<string> doopnamen;
    string tussenvoegsel {""};
    string familienaam;
```

[173] Zie eventueel https://en.wikipedia.org/wiki/IEEE_754.

[174] Zie eventueel https://en.wikipedia.org/wiki/Bfloat16_floating-point_format.

```
    int geboortedag {0};
    int geboortemaand {0};
    int geboortejaar {0};
    int lengte {0};
};
```

Dan kun je een object van het type `Persoon` als volgt initialiseren met behulp van aangewezen initialisatie, zie `designated_initializers.cpp`:

```
    Persoon harry {
        .roepnaam="Harry",
        .doopnamen={"Johannis", "Zacharias", "Maria"},
        .familienaam="Broeders",
        .geboortedag=24,
        .geboortemaand=12,
        .geboortejaar=1962,
        .lengte=184
    };
```

Dit is beter leesbaar en minder gevoeliger voor fouten dan de traditionele manier van initialiseren, waarbij je de waarden in de juiste volgorde moet plaatsen zonder de namen van de datamembers te vermelden:

```
    Persoon harry {"Harry", {"Johannis", "Zacharias", "Maria"}, ↩
↪ "", "Broeders", 24, 12, 1962, 184};
```

De volgorde van de designated initializers moet overeenkomen met de volgorde van de datamembers in de struct. Een designated initializer kan overgeslagen worden als het datamember een defaultwaarde heeft.

15.8 Zelf gedefinieerde module

Je kunt ook zelf een module definiëren. De functies, classes, templates, enz. die in deze module gedefinieerd worden en geëxporteerd worden kunnen dan in meerdere programma's of in andere modules gebruikt (geïmporteerd) worden.

In het onderstaande voorbeeld wordt een module aangemaakt die een functie genaamd `hello()` exporteerd.

De interface van deze module is gedefinieerd in hello_world_module_interface.cppm[175]:

```
export module helloworld;

export void hello();
```

De implementatie van deze module is gedefinieerd in hello_world_module_implementation.cpp:

```
module helloworld;
import std;
using namespace std;

void hello() {
    println("Hello world!");
}
```

Het programma dat gebruik maakt van deze module is gedefinieerd in hello_world_module_main.cpp:

```
import helloworld;

int main() {
    hello();
}
```

Je kunt de module interface compileren met het commando:

```
g++ -std=c++23 -fmodules -fmodule-only -c hello_world_module_interface.cppm
```

Dit genereerd een precompiled module interface file genaamd helloworld.gcm in het directory gcm.cache.

Je kunt de module implementatie compileren met het commando:

```
g++ -std=c++23 -fmodules -c hello_world_module_implementation.cpp
```

Dit genereerd een een object file genaamd helloworld.o.

[175] Er is geen standaard extensie voor module interface bestanden, maar de extensie .cppm wordt ondersteund door de GCC en Clang compilers. De Microsoft C++-compiler gebruikt de extensie .ixx.

Je kunt het programma dat gebruik maakt van deze module compileren met het commando:

```
g++ -std=c++23 -fmodules -c hello_world_module_main.cpp
```

Dit genereerd een een object file genaamd `hello_world_module_main.o`.

Beide object files kunnen vervolgens gelinkt worden tot een uitvoerbaar programma met het commando:

```
g++ hello_world_module_implementation.o hello_world_module_main.o -o hello
```

Dit programma kan ten slotte uitgevoerd worden met het commando[176]:

```
./hello
```

Het is ook mogelijk om de interface en implementatie van de module beide in één file te plaatsen (`hello_world_module.cppm`):

```
export module helloworld;
import std;
using namespace std;

export void hello() {
    println("Hello world!");
}
```

Deze module kan vervolgens gecompileerd worden met het commando:

```
g++ -std=c++23 -fmodules -c hello_world_module_interface.cppm
```

Dit commando maakt dan zowel het bestand `helloworld.gcm` in het directory `gcm.cache` als het object file `hello_world_module.o` aan. De object file kan gelinkt worden met de al eerder aangemaakte `hello_world_main.o` met het commando:

```
g++ hello_world_module.o hello_world_module_main.o -o hello
```

Hoewel het dus mogelijk is om de interface en implementatie te combineren in één bestand, is het overzichtelijker om ze te splitsen in twee apparte bestanden.

[176] Onder Windows moet je het commando `hello.exe` gebruiken in plaats van `hello`.

15.9 Namespaces

Bij grote programma's kunnen verschillende classes 'per ongeluk' dezelfde naam krijgen. In C++ kun je classes (en functies etc.) groeperen in zogenoemde *namespaces*:

```
namespace BroJZ {
   void f(int);
   double sin(double x);
}

// andere file zelfde namespace:
namespace BroJZ {
   class string {
   //  ...
   };
}

// andere namespace:
namespace VersD {
   class string {
   //  ...
   };
}
```

Je ziet dat in de namespace `BroJZ` een functie `sin` is opgenomen die ook in de standaard library is opgenomen. De in de namespace `BroJZ` gedefinieerde class `string` is ook in de namespace `VersD` en ook in de standaard library gedefinieerd. Je hebt op pagina 21 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:

```
BroJZ::string s1 {"Harry"};
VersD::string s2 {"Daniël"};
std::string s3 {"Standard"};
```

In het bovenstaande codefragment worden drie objecten gedefinieerd van drie verschillende classes. Het object `s1` is van de class `string` die gedefinieerd is in de namespace `BroJZ`, het object `s2` is van de class `string` die gedefinieerd is in de namespace `VersD` 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 `BroJZ` wilt gebruiken, kun je dat opgeven met behulp van een *using declaration*.

```
using BroJZ::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 `BroJZ`. De usingdeclaratie blijft net zolang geldig als een gewone variabeledeclaratie. Tot de bijbehorende accolade sluiten dus.

Als je in een stuk code steeds classes en functies uit de namespace `BroJZ` wilt gebruiken, kun je dat opgeven met behulp van een *using directive*.

```
using namespace BroJZ;
string s6 {"Hallo"};
double d {sin(0,785398163397448)};
```

Het object `s6` is nu van de class `string` die gedefinieerd is in de namespace `BroJZ`. De functie `sin` die wordt aangeroepen is nu de in de namespace `BroJZ` gedefinieerde functie. De using directive blijft net zolang geldig als een gewone variabeledeclaratie. Tot de bijbehorende accolade sluiten dus.

15.10 using in plaats van typedef

In C en C++ kun je met behulp van een **typedef** een alias (andere naam) voor een bestaand type definiëren, bijvoorbeeld:

```
typedef map<string, int> Frequenties;
```

De typenaam Frequenties is nu een alias voor het type map<string, int>. In C++ kunnen we hetzelfde bereiken met behulp van het keyword **using**:

```
using Frequenties = map<string, int>;
```

De meeste programmeurs vinden deze vorm leesbaarder. Deze vorm maakt bovendien template argument binding mogelijk, zie paragraaf 17.2.

15.11 decltype type

In C++ kun je het type van een expressie opvragen met behulp van het keyword **decltype**. Dit kan soms handig zijn. Zo zou je bijvoorbeeld de volgende code kunnen schrijven:

```
auto i {42};
auto& ri1 {i};
```

De variabele i is nu van het type **int** en de variabele ri1 is een reference naar i. Als je nu een tweede referentie wilt definiëren ri2 die een kopietje is van ri1, dan kan dat *niet* als volgt:

```
auto ri2 {ri1};
```

Het type van de variabele ri2 is nu namelijk **int** omdat **auto** nooit een reference type oplevert. Je zou dit als volgt op kunnen lossen:

```
auto& ri2 {ri1};
```

Maar dat is niet erg elegant omdat je eigenlijk wilt zeggen dat ri2 van hetzelfde type moet zijn als ri1. Dit kan met de volgende code:

```
decltype(ri1) ri2 {ri1};
```

De type van de variabele ri2 is gedefieerd als hetzelfde type als ri1, dus ri2 is ook een reference naar i. De naam ri1 komt nu twee keer voor in deze code, wat ook weer niet zo elegant is. De mooiste manier om aan te geven dat ri2 van hetzelfde type moet zijn als ri1 is:

```
int i {42};
int& ri1 {i};
decltype(auto) ri2 {ri1};
```

In paragraaf 1.7 hebben we de volgende code besproken:

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

Als het type van de variabele `rij` gewijzigd wordt, dan wordt het type van de variabele `element` automatisch aangepast. Je kunt er nu met behulp van **`decltype`** voor zorgen dat het type van de variabele `som` ook automatisch wordt bepaald:

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

Als de rij nu als volgt wordt gedefinieerd:

```
double rij[] = {12.5, 2, 17, 32, 1, 18};
```

Dan wordt het type van `som` en `element` ook automatisch aangepast naar **`double`**.

De bovenstaande code is waarschijnlijk ingewikkelder dan je had gedacht. De volgende definitie van de variabele `som` lijkt meer voor de hand te liggen:

```
decltype(rij[0]) som {0};
```

Dit werkt echter niet omdat `rij[0]` van het type **`int`**& is en niet van het beoogde type **`int`**. De GCC C++-compiler geeft de volgende foutmelding:

```
error: cannot bind non-const lvalue reference of type 'int&' to an ↩
 ↪ rvalue of type 'int'
```

In de standaard C++ library zijn een aantal functietemplates opgenomen die gebruikt kunnen worden om types te manipuleren[177]. Je kunt de functietemplate `remove_reference_t`

[177] Zie https://en.cppreference.com/w/cpp/header/type_traits.

gebruiken om van een reference type een gewoon type te maken. Dus je kunt de variabele som als volgt definiëren:

```
remove_reference_t<decltype(rij[0])> som1 {0};
```

Maar je kunt het type van som nu ook afleiden van het type van rij waarbij je de functietemplate remove_extent_t gebruikt om van een eendimensionaal arraytype een gewoon type te maken.

```
remove_extent_t<decltype(rij)> som {0};
```

15.12 Compile time functies

Stel dat je een functie hebt geschreven om n^m te bereken waarbij n en m gehele positieve getallen zijn, je bent er daarbij vanuit gegaan dat het resultaat past in een **unsigned int**:

```
unsigned int power(unsigned int n, unsigned int m) {
    unsigned int result = 1;
    for (unsigned int i {0}; i < m; ++i) {
        result *= n;
    }
    return result;
}
```

Deze functie kun je bijvoorbeeld als volgt aanroepen:

```
println("3 tot de macht 5 = {}", power(3, 5));
```

De uitvoer van deze regel is dan:

```
3 tot de macht 5 = 243
```

Als je een array wilt definiëren met 3^5 integers dan kun je die *niet* definiëren met de zojuist besproken functie power.

```
    array<int, power(3, 5)> rij;
//  error: call to non-constexpr function 'unsigned int ↩
 ↪ pow(unsigned int, unsigned int)'
```

Nu kun je natuurlijk je rekenmachine pakken en berekenen dat 3^5 gelijk is aan 243. De array kan dan als volgt gedefinieerd worden:

```
array<int, 243> rij; // 243 = 3 tot de macht 5
```

Nu is in commentaar weergegeven hoe het aantal elementen van de `array` berekend is. Deze code is echter niet goed onderhoudbaar. Als het aantal elementen in de `array` bijvoorbeeld gewijzigd moet worden in 3^6, dan moet je eerst uitrekenen dat dit gelijk is aan 729 en vervolgens moet je 243 vervangen door 729 en tot slot moet je niet vergeten om ook het commentaar aan te passen. Het zou natuurlijk veel beter zijn voor de onderhoudbaarheid als de compiler deze waarde zelf zou kunnen berekenen. In C++ is dat inderdaad mogelijk door de functie vooraf te laten gaan door het keyword **`constexpr`**. Zo'n functie wordt een compile time functie genoemd[178] omdat een aanroep naar deze functie niet alleen tijdens het uitvoeren van het programma maar ook tijdens het vertalen van het programma kan worden uitgevoerd[179].

```
constexpr unsigned int power(unsigned int n, unsigned int m) {
    unsigned int result = 1;
    for (unsigned int i {0}; i < m; ++i) {
        result *= n;
    }
    return result;
}
```

De `array` met 3^5 integers kan nu als volgt gedefinieerd worden (`constexpr_functie.cpp`):

```
    array<int, power(3, 5)> rij;
```

De aanroep van een compile time functie levert echter alleen een compile time constante op als de parameters waarmee de functie wordt aangeroepen ook compile time constanten zijn. Dit betekent dat een compile time functie toch tijdens run time uitgevoerd wordt, als de functie wordt aangeroepen met variabelen. Dit voorkomt dat er twee identieke functies gedefinieerd moeten worden: één met en één zonder **`constexpr`**.

Dit is handig omdat het nu dus niet nodig is om een tweede `power`-functie te schrijven (die gedefinieerd is zonder **`constexpr`**).

Het is dus aan te bevelen om een functie die nuttig zou kunnen zijn bij het berekenen van een run time constante met **`constexpr`** te definiëren[180].

[178] Een compile time functie is vergelijkbaar met een met **`#define`** gedefinieerde macro voor het berekenen van een constante waarde in de programmeertaal C.

[179] Zo zijn bijvoorbeeld de functies en operatoren van het standaard type `complex` zijn als `constexpr` gedeclareerd, zodat je tijdens het compileren complexe rekenkundige bewerkingen kunt uitvoeren.

[180] Dit wordt ook aangeraden in de C++ Core Guideline F.4.

Het is ook mogelijk om een functie te definiëren die *alleen* maar tijdens compile time uitgevoerd kan worden. Dit doe je door de functie te laten voorafgaan door het keyword **consteval** (in plaats van **constexpr**). Zie constevalfunctie.cpp.

15.13 Compile time if

Vanaf C++17 kun je gebruik maken van een zogenoemde *compile time if*. Dit is een **if**-instructie die niet tijdens het uitvoeren van het programma (tijdens run time), maar tijdens het vertalen van het programma (compile time) wordt uitgevoerd. De syntax van deze compile time if is: **if constexpr** `(conditie) ...`. De expressie die als conditie gebruikt wordt moet uiteraard tijdens compile time uit te voeren zijn en kan dus bijvoorbeeld geen variabelen bevatten.

15.14 Read-only pointers met const

Je hebt in paragraaf 1.9 gezien hoe je in C++ de **const** qualifier kunt gebruiken om read-only variabelen (met een constante waarde) te definiëren.

Voorbeeld met **const**:

```
const int aantal_regels {80};
```

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

15.14.1 const *

```
int i {3};
const int j {4};
const int* p {&i};[181]
```

Dit betekent: `p` wijst naar `i` en je kunt `i` via `p` alleen lezen. Je kunt `i` dus *niet* wijzigen via `p`. Dit is zinvol als je een pointer als argument aan een functie wilt meegeven en als je niet wilt dat de variabele waar deze pointer naar wijst in de functie gewijzigd wordt. Je gebruikt dan als parameter een **const** `T*` in plaats van een `T*`. Dat de functie de variabele waar de pointer naar wijst niet mag wijzigen zouden we natuurlijk ook gewoon kunnen afspreken (en bijvoorbeeld in het commentaar van de functie vermelden) maar door deze afspraak expliciet als een **const**-declaratie vast te leggen kan deze afspraak door de compiler gecontroleerd

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

worden. Dit vermindert de kans op het maken van fouten bij het implementeren of wijzigen van de functie. Let op: je kunt `i` zelf wel rechtstreeks wijzigen!

```
    const int* p {&i};
    // p wijst naar i en je kunt i niet via p wijzigen.
    // Let op: je kunt i zelf wel rechtstreeks wijzigen!
    i = 4;
    *p = 5;
//  error: assignment of read-only location '* p'
    p = &j;
```

De pointer `p` is in dit geval zelf *geen* read-only variabele. Zoals je hierboven ziet, kun je `p` wijzigen en bijvoorbeeld naar de variabele `j` laten wijzen. Merk op dat de variabele `j` een read-only variabele is, dat is geen probleem omdat `j` via `p` niet gewijzigd kan worden. Een gewone `int*` kun je *niet* naar `j` laten wijzen omdat `j` niet gewijzigd mag worden. Een `const int*` kan dus wijzen naar een `int` en naar een `const int` maar een `int*` kan alleen maar naar een `int` wijzen.

15.14.2 `* const`

```
int* const q {&i};
```

Dit betekent: `q` wijst naar `i` en je mag `q` alleen lezen. Je kan `q` dus nergens anders meer naar laten wijzen. Dit is zinvol als je een pointer altijd naar dezelfde variabele wilt laten wijzen. Let op: je kunt `i` wel via `q` (of rechtstreeks) wijzigen.

```
    int* const q {&i};
    // q wijst naar i en je kunt q nergens anders meer naar laten ↩
 ↪  wijzen.
    // Let op: je kunt i wel via q (of rechtstreeks) wijzigen.
    i = 4;
    *q = 5;
    q = &j;
//  error: assignment of read-only variable 'q'
```

De pointer `q` is in dit geval een read-only variabele.

15.14.3 const * const

```
const int* const r {&i};
```

Dit betekent: r wijst naar i, je kunt i via r alleen lezen en je kunt r alleen lezen. Je kunt i niet via r wijzigen en je kunt r nergens anders meer naar laten wijzen. Let op: je kunt i zelf wel wijzigen!

```
    const int* const r {&i};
    // r wijst naar i en je kunt i niet via r wijzigen en je kunt ↩
 ↪ r nergens anders meer naar laten wijzen.
    // Let op: je kunt i zelf wel rechtstreeks wijzigen!
    i = 4;
    *r = 5;
//  error: assignment of read-only location '*(const int*)r'
    r = &j;
//  error: assignment of read-only variable 'r'
```

15.15 Casting

Door middel van *casting*[182] kun je variabelen van het ene type omzetten naar een ander type en kun je objecten van de ene class omzetten naar een andere class. Dit zijn 'gevaarlijke' bewerkingen die alleen moeten worden gebruikt als het echt niet anders kan.

In C kun je met een eenvoudige vorm van casting typeconversies doen. Stel dat je het adres van een C string in een integer wilt opslaan[183], dan kun je dit als volgt proberen (casting1.cpp):

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

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

```
    i = "Hallo";
//  error: invalid conversion from 'const char*' to 'int'
```

[182] Het Engelse woord casting betekent onder andere in een mal gieten. Als je een variabele cast naar een ander type, giet je die variabele als het ware in een andere mal.

[183] 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!

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 op een 32-bit microcontroller wilt gebruiken) dan kun je de compiler 'dwingen' met behulp van een zogenoemde *cast*:

```
i = (int)"Hallo";
```

In C++ mag je deze cast ook als volgt coderen:

```
i = int("Hallo");
```

Het vervelende van beide vormen van casting is dat een cast erg moeilijk te vinden is. Omdat een cast in bijna alle gevallen code oplevert die niet portable is, is het echter wel belangrijk om alle casts in een programma op te kunnen sporen.

In de C++-standaad 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 MSP430-microcontroller. Als je de output poort B van deze controller wilt aansturen, kan dat via adres 0x38. Dit kun je in C++ als volgt doen:

```
volatile[184]  uint8_t* portb = reinterpret_cast<uint8_t*>(0x38);
*portb = 0xFE; // schrijf 0xFE (hex) naar adres 0x38 (hex)
```

Als we een cast willen doen die wel portable is, kan dat met **static_cast**.

Vraag:

Wat is de uitvoer van het volgende programmadeel (casting2.cpp)?

```
int i1 {1};
int i2 {2};
println("1/2 = {}", i1 / i2);
```

Antwoord:

```
1/2 = 0
```

[184] Zie voor uitleg over het gebruik van **volatile** https://harrybroeders.bitbucket.io/KOF01/volatile.htm.

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.

Als je wilt dat het bovenstaande programma `0.5` afdrukt, dan moet je ervoor zorgen dat in plaats van een integer deling een floating-point deling wordt gebruikt. De computer gebruikt een floating-point deling als één van de twee argumenten een floating-point getal is. Door het gebruik van een **`static_cast`** kun je een van de twee getallen omzetten naar een **`double`**:

```
    println("1/2 = {}", static_cast<double>(i1) / i2);
```

De output van deze regel is:

```
1/2 = 0.5
```

Er bestaat ook een speciale cast om een **`const`** weg te casten de zogenoemde **`const_cast`**. Voorbeeld (casting3.cpp):

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

int main() {
    string s {"Dag"};
    println("s = {}", s);
    stiekem(s);
    println("s = {}", s);
}
```

Uitvoer:

```
s = Dag
s = Hallo
```

Het zal duidelijk zijn dat je het gebruik van **`const_cast`** zoveel mogelijk moet beperken. Als de reference a in het bovenstaande programma naar een **`const`** `string` verwijst, is het resultaat onbepaald omdat een compiler **`const`** objecten in het ROM geheugen kan plaatsen (dit gebeurt veel bij embedded systems).

15.16 static_assert

In paragraaf 6.1 heb je al kennis gemaakt met de standaard functie `assert`. Met deze functie wordt tijdens het uitvoeren van het programma gecheckt of aan een bepaalde voorwaarde wordt voldaan. In C++ kun je ook tijdens het compileren van je programma al checken of aan een bepaalde voorwaarde is voldaan. Dit doe je met behulp van de standaard functie **`static_assert`**. De code **`static_assert`**`(test, melding)` geeft de compiler-error `melding` als de `test` **`false`** oplevert.

Stel bijvoorbeeld dat je een stuk code hebt geschreven waarbij gebruik gemaakt wordt van het type **`int`** dat alleen maar werkt als dit type 32 bits bevat. Het aantal bits in een **`int`** is, zoals je waarschijnlijk wel weet, afhankelijk van de compiler. Je kunt nu de volgende code in je programma opnemen om te checken of een **`int`** 32 bits (oftewel 4 bytes) breed is:

```
static_assert(sizeof(int) == 4, "int to small");
```

Als we deze code compileren voor een processor die 16-bit integers heeft[185], dan geeft de compiler op deze regel een errormelding: `static assertion failed: int to small`.

In dit geval had je natuurlijk in plaats van het type **`int`** ook het type `int32_t` kunnen gebruiken dat altijd 32 bits breed is.

15.17 Structured binding

In C++ kun je zogenoemde 'structured binding' toepassen. Hiermee kun je de waarden van een samengesteld type (bijvoorbeeld een **`struct`**) in één statement toekennen aan meerdere variabelen. Bijvoorbeeld, stel dat je de volgende **`struct`** hebt gedefinieerd:

```
struct Adres {
    string straatnaam;
    int huisnummer;
    string plaatsnaam;
};
```

Als de functie `adreslocatie_HR` dan het adres van een locatie van de Hogeschool Rotterdam teruggeeft, dan kan dit adres met behulp van structured binding toegekend worden aan drie variabelen:

```
auto [straat, nr, plaats] {adreslocatie_HR("AP")};
```

[185] De meeste 8- of 16-bit microprocessoren, zoals de MSP430, gebruiken 16-bit **`int`**s.

```
    println("Het adres van de locatie AP van de HR is: {} {} {}", ↩
↪ straat, nr, plaats);
```

Het gebruik hiervan is soms handig, maar vaak is het duidelijker om de waarden van het samengestelde type één voor één te gebruiken:

```
    auto ap {adreslocatie_HR("AP")};
    println("Het adres van de locatie AP van de HR is: {} {} {}", ↩
↪ ap.straatnaam, ap.huisnummer, ap.plaatsnaam);
```

Vooral bij gebruik van sommige standaard library types zoals `set` en `map` is het gebruik van structured binding handig. In paragraaf 11.4.1 heb je gezien dat de memberfunctie `insert` van `set` het returntype `pair<iterator, bool>` heeft. De `iterator` geeft de plek aan waar ingevoegd is en de `bool` geeft aan of het invoegen gelukt is. In het voorbeeldprogramma werd de volgende code gebruikt om te kijken of het invoegen van `"Harry"` in de set `docenten` gelukt was:

```
set<string> docenten {"Ron", "Daniël", "Roy", "Harry"};
auto result {docenten.insert("Harry")};
if (!result.second)
    println("1 Harry is genoeg.");
```

Dit zou je nu ook met behulp van structured binding als volgt kunnen coderen:

```
set<string> docenten {"Ron", "Daniël", "Roy", "Harry"};
auto [itr, gelukt] {docenten.insert("Harry")};
if (!gelukt)
    println("1 Harry is genoeg.");
```

15.18 std::string_view

In paragraaf 1.10 heb je de class `std::string` leren kennen. Operaties op variabelen van het type `string` zijn altijd veilig omdat de class `string` het geheugen beheert dat nodig is om de string op te slaan. Bij het uitvoeren van operaties op een string, zoals het opvragen van een substring, wordt er een nieuw object van het type `string` gemaakt dat een kopie van de substring bevat.

Bijvoorbeeld:

```
string s1 {"Hello, World!"};
// s1 bevat "Hello, World!"
string s2 {s1.substr(0, s1.find(","))};
```

```
    // s2 bevat "Hello"
    println("{:?}", s1); // "Hello, World!"
    println("{:?}", s2); // "Hello"
```

De format specifier :? zorgt ervoor dat de inhoud van de string met dubbele aanhalingstekens eromheen wordt afgedrukt.

Een `string_view` is een lichtgewicht object dat een read-only pointer naar een tekenreeks en een lengte bevat. Een `string_view` biedt een interface die bijna gelijk is aan de *read-only* interface van `string`. Hierdoor kun je bovenstaande code efficiënter schrijven door een `string_view` te gebruiken in plaats van een `string`, omdat er dan geen nieuw object van het type `string` hoeft te worden gemaakt, zie proglinkstring_view.cpp:

```
    string_view sv1 {"Hello, World!"};
    // s1 verwijst naar de string literal "Hello, World!"
    string_view sv2 {sv1.substr(0, sv1.find(","))};
    // s2 verwijst naar "Hello" binnen de string literal "Hello, ↩
↪ World!"
    println("{:?}", sv1); // "Hello, World!"
    println("{:?}", sv2); // "Hello"
```

Er worden nu helemaal geen objecten van het type `string` aangemaakt in het geheugen. De objecten van het type `string_view` bevatten enkel een pointer naar de string literal en een lengte. Maar omdat de `string_view` in tegenstelling tot een `string` de karakters niet zelf opslaat en beheert, moet je wel oppassen bij het gebruik ervan. Als de tekst waarnaar de `string_view` verwijst, wordt gewijzigd of verwijderd, dan kan dit leiden tot onverwacht of zelfs ongedefinieerd gedrag als je daarna de `string_view` gebruikt. Bijvoorbeeld:

```
    string s1 {"Hello, World!"};
    // s1 bevat "Hello, World!";
    string_view sv1 {s1};
    // sv1 verwijst naar "Hello, World!" in s1
    string_view sv2 {sv1.substr(0, sv1.find(","))};
    // sv2 verwijst naar "Hello" in s1
    s1.replace(0, 5, "Hi");
    // s1 bevat nu "Hi, World!"
    println("{:?}", s1); // "Hi, World!"
    println("{:?}", sv1); // "Hi, World!\u{0}d!"
    println("{:?}", sv2); // "Hi, W"
```

In de bovenstaande code wordt de `string_view` `sv2` na het aanpassen van de `string` `s1` afgedrukt:

```
    s1.replace(0, 5, "Hi");
    println("{:?}", sv2); // "Hi, W"
```

Dit geeft, misschien onverwacht, als output `"Hi, W"` omdat de `string_view` `sv2` nog steeds verwijst naar de eerste vijf karakters van de `string` `s1`.

Ook wordt de `string_view` `sv1` afgedrukt, dat heeft in principe ongedefinieerd gedrag tot gevolg omdat de `string_view` `sv1` nog steeds verwijst naar de eerst dertien karakters van de `string` `s1`. Maar deze `string` bevat nog maar tien karakters. Het afdrukken van dertien karakters uit deze `string` leidt tot ongedefinieerd gedrag[186].

`string_view` voegt de memberfuncties `remove_prefix(size_t)` en `remove_suffix(size_t)` toe, die een `string_view` inkorten. Zie `string_view.cpp`.

Als je een stukje tekst aan een functie wilt meegeven, dan kun je dat doen door een **`const`** `string&` parameter te gebruiken. Als je nu echter een string literal (bijvoorbeeld `"Hallo"`) als argument doorgeeft, dan creëert de compiler stilzwijgend een tijdelijk object van het type `string` dat misschien niet nodig is. Je zou een overloaded functie, zie paragraaf 1.12, kunnen schrijven die een **`const char*`** accepteert. Er hoeft dan geen object van het type `string` gemaakt te worden. In dit geval kun je ook één functie schrijven die een `string_view` accepteert, omdat een `string_view` zowel een **`const char*`** als een `string` kan accepteren zonder dat er een tijdelijk object van het type `string` hoeft te worden gemaakt.

```
void f1(const char* s) {
    println("f1(const char* s): s = {}", s);
}

void f2(const string& s) {
    println("f2(const string& s): s = {}", s);
}

void f3(const char* s) {
    println("f3(const char* s): s = {}", s);
}

void f3(const string& s) {
```

[186] Er kan dus van alles gebeuren, zoals het afdrukken van willekeurige tekens, een crash van het programma, enzovoort.

```
    println("f3(const string& s): s = {}", s);
}

void f4(string_view s) {
    println("f4(string_view s): s = {}", s);
}
```

In het bovenstaande codefragment zijn vijf functies gedefinieerd waarmee een string afgedrukt kan worden.

- `f1(const char* s)` kan aangeroepen worden met een string literal:

  ```
  f1("Hallo wereld!");
  ```

  `f1` kan niet aangeroepen worden met een `string`:

  ```
  string hallo {"Hallo daar!"};
  f1(hallo); // error:: cannot convert 'std::string' to 'const ↩
   ↪ char*'
  ```

  Je kunt de `string` wel omzetten naar een `const char*` door de `c_str()` memberfunctie van de `string` te gebruiken:

  ```
  f1("Hallo wereld!");
  ```

  Maar dit is onhandig en in de functie `f1` kun je natuurlijk geen gebruik maken van de memberfuncties van de class `string`.

- `f2(const string& s)` kan aangeroepen worden met een `string` en met een string literal:

  ```
  f2("Hallo wereld!");
  f2(hallo);
  ```

  Maar als je `f2` aanroept met een string literal, dan wordt er een tijdelijk object van het type `string` gemaakt en dat is niet efficiënt als je de memberfuncties van de class `string` niet nodig hebt.

- De twee functies `f3` zijn overloaded zodat deze zowel een `const char*` als een `const string&` kunnen accepteren:

  ```
  f3("Hallo wereld!");
  f3(hallo);
  ```

Maar het is natuurlijk niet handig om steeds twee functies te definiëren om de code zo efficiënt mogelijk te maken.

- `f4(string_view s)` kan ook aangeroepen worden met een `string` en met een string literal, maar er wordt geen tijdelijk object van het type `string` gemaakt als je een string literal doorgeeft. Dit is dus de meest efficiënte manier om een string aan een functie door te geven als je de memberfuncties die een `string` aanpassen niet nodig hebt. De `string_view` heeft dezelfde read-only interface als een `string`, waar je gebruik van kunt maken in de functie, ook als je een string literal doorgeeft. Een `string_view` wordt meestal als call by value parameter gebruikt, omdat een `string_view` efficiënt gekopieerd kan worden.

Zie `string_view_parameter.cpp`.

Als een functie een `string_view` als parameter heeft en in de functie moet de `string_view` worden geconverteerd naar een `string`, dan is het beter om de functie een `const string&` parameter te geven. Dit is dan efficienter omdat er, als de functie met een string literal als argument wordt aangeroepen, uiteindelijk toch een `string` aangemaakt moet worden.

Een `string_view` wordt niet impliciet geconverteerd naar een `string` om onbewuste kopietjes te voorkomen, want dat was namelijk het idee achter `string_view`). Deze conversie moet expliciet worden gemaakt door een `string` te construeren met de `string_view` als argument:

```
string_view sv {"Hallo Wereld!"};
string s {sv}; // expliciete conversie van string_view naar string
```

15.19 Attributen

Attributen (Engels: *attributes*) zijn een gestandaardiseerde manier, sinds C++11 en C23, om extra informatie of instructies mee te geven aan de C/Cpp-compiler. Een attribuut heeft de volgende syntax `[[naam]]`. Er zijn een aantal standaard attributen gedefinieerd. Zie `https://en.cppreference.com/w/cpp/language/attributes`. Hier volgen enkele voorbeelden:

- De hoofdfunctie in veel embedded systemen bevat een `while(1)`-loop omdat deze functie 'altijd' moet blijven draaien. Sommige compilers geven in dit geval een warning. Je kunt aangeven dat het juist de bedoeling is dat de functie nooit terugkeert door de functie te voorzien van het attribuut `[[noreturn]]`.
- In `switch`-statements gebruik je meestal aan het einde van elke `case` een `break`-statement. Soms wil je echter dat het programma na uit uitvoeren van de `case`

verdergaat met het uitvoeren van de volgende **case**. In dit geval eindigt de **case** niet met een **break**. Veel compilers geven in dit geval een warning. Je kunt aangeven dat je bewust de **break** hebt weggelaten door het attribuut `[[fallthrough]]` te gebruiken. Zie `attributes.cpp`.

- Als je een variabele definieert maar niet gebruikt in een programma, geven veel compilers een warning. Deze warning kun je onderdrukken met het attribuut `[[maybe_unused]]`, zie `maybe_unused.cpp`.
- Als je de return waarde van een functie negeert, dan geven veel compilers *geen* warning. Wil je juist *wel* een warning als de return waarde van een functie genegeert wordt dan kun je bij het definiëren van de functie het attribuut `[[nodiscard]]` gebruiken, zie `nodiscard.cpp`. Dit is vooral nuttig als de functie een foutcode teruggeeft die niet genegeerd mag worden, zie paragraaf 6.2.

16

Objects and classes (de details)

In hoofdstuk 2 heb je geleerd wat een UDT is. Als voorbeeld van een UDT hebben we de class `Breuk` behandeld. In paragraaf 2.20 heb je gezien hoe je de `operator+=` kunt overloaden voor de class `Breuk`. In dit hoofdstuk leer je hoe je andere operatoren kunt overloaden voor de class `Breuk` zodat je een object van deze class op dezelfde manier kunt gebruiken als een variabele van het type `int`.

In C heb je de `enum` leren kennen. In C++ is een verbeterde versie, zogenoemde `enum class`, toegevoegd. Deze verbeterde versie wordt in dit hoofdstuk besproken.

16.1 Operator overloading (deel 2)

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

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

```
const Breuk Breuk::operator+(const Breuk& rechts) const {
    Breuk copy_links {*this}; // maak een kopietje van de receiver
    copy_links += rechts;     // tel daar het object rechts bij op
    return copy_links;        // geef deze waarde terug
}
int main() {
    Breuk b1 {2, 3}, b2 {4, 5}, b3;
    b3 = b1 + b2;
    println("{} + {} = {}", b1, b2, b3);
```

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

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

Zie `operator+_member.cpp` voor een compleet voorbeeldprogramma.

16.2 `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 [17] en [18]. Als `a`, `b` en `c` van het type `int` zijn, levert de expressie `(a + b) += c` de volgende (onduidelijke) foutmelding op `error: lvalue required as left operand of assignment`. 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 wordt de expressie `(a + b) += c` zonder foutmelding vertaald. 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: `error: passing 'const Breuk' as 'this' argument discards qualifiers`. 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.[187]

[187] Dat dit nogal subtiel is blijkt wel uit het feit dat Bjarne Stroustrup (de ontwerper van C++) in een vergelijkbaar voorbeeld [23, blz. 528] geen `const` bij het return type gebruikt.

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

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

Q: Kun je in de `operator+` de benodigde lokale variabele niet met `new`, zie paragraaf 5.1 aanmaken zodat we toch een reference kunnen teruggeven? De met `new` aangemaakte variabele blijft immers na afloop van de `operator+` memberfunctie gewoon bestaan.

A: Ja dit kan wel, maar je kunt het beter *niet* doen. De met `new` aangemaakte variabele wordt namelijk (zo lang het programma draait) nooit meer vrijgegeven. 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 'kijkt' dan 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 'kijkt' vervolgens of de `int` omgezet kan worden naar een `Breuk`. Dit is in dit geval mogelijk door gebruik te maken van de constructor `Breuk(int)`. De compiler maakt dan met deze constructor een tijdelijke variabele van het type `Breuk` aan en initialiseert deze variabele met `1`. Vervolgens wordt een reference naar deze tijdelijke variabele als argument aan de `operator+` memberfunctie meegegeven. De tijdelijke variabele wordt na het uitvoeren van de `operator+` memberfunctie weer opgeruimd.

Q: Kun je bij een `int` ook een `Breuk` optellen?
```
Breuk a 1, 2;
Breuk b 1 + a;
```

A: Nee, nu niet. De expressie `1 + a` wordt geïnterpreteerd als `1.operator+(a)`. De compiler 'kijkt' dan 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 'kijkt' vervolgens of de `Breuk` omgezet kan worden naar een **`int`**. Dit is in dit geval ook niet mogelijk[188]. De GCC C++-compiler genereert de volgende foutmelding: `no match for 'operator+' (operand types are 'int' and 'Breuk')`. 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.

16.3 Operator overloading (deel 3)

Naast het definiëren van operatoren als memberfuncties van zelfgedefinieerde types kun je ook de globale operatoren overloaden[189]. 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` wordt dus, als `a` en `b` beide van het type **`int`** zijn, geïnterpreteerd als **`operator`**`+(a, b)`. Je kunt nu door gebruik te maken van operator overloading zelf zoveel globale **`operator`**+ functies definiëren als je maar wilt.

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

```
const Breuk operator+(int links, const Breuk& rechts) {
    return rechts + links;
}
```

Deze implementatie roept simpelweg de `Breuk::operator`+ memberfunctie aan! Optellen is namelijk commutatief, dat wil zeggen dat $a + b$ gelijk is aan $b + a$. Als we van een **`int`**

[188] In paragraaf 16.5 bespreek ik hoe je deze type conversie indien gewenst zelf kunt definiëren.

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

een Breuk willen aftrekken, moet een andere aanpak worden gekozen want aftrekken is niet commutatief, $a - b$ is namelijk ongelijk aan $b - a$.

```
const Breuk operator-(int links, const Breuk& rechts) {
    Breuk copy_links {links};
    copy_links -= rechts;
    return copy_links;
}
```

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

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

Zie operator+_global.cpp voor een compleet voorbeeldprogramma.

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

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

```
class Breuk {
public:
    Breuk(int t);
    Breuk& operator+=(const Breuk& rechts);
    // ...
};
const Breuk operator+(const Breuk& links, const Breuk& rechts) {
    Breuk copy_links {links};
    copy_links += rechts;
    return copy_links;
}
```

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

```
const Breuk operator+(const Breuk& links, const Breuk& rechts) {
    return Breuk {links} += rechts;
```

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

```
const Breuk operator+(Breuk links, const Breuk& rechts) {
    return links += rechts;
```

Doordat de eerste parameter `left` hier als call by value (zie pagina 70) is gedeclareerd wordt bij aanroep 'van zelf' een kopietje van het linker argument van de **operator**+ gemaakt.

Zie `Breuk3.cpp` voor een compleet voorbeeldprogramma.

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

16.4 Overloaden operator++ en operator--

Bij het overloaden van de **operator**++ en **operator**-- ontstaat een probleem omdat beide zowel een *prefix* als een *postfix* operator variant kennen[190]. Dit probleem is opgelost door de postfix versie te voorzien van een (dummy) **int** parameter.

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

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

De implementatie van deze memberfuncties wordt in paragraaf 16.6 gegeven.

[190] 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;`.

Het gebruik van het return type `Breuk&` in plaats van `const Breuk&` bij de prefix `operator++` zorgt ervoor dat de expressie `a = ++++b` als `a` en `b` van het type `Breuk` zijn gewoon werkt[191] (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[192] geeft (net zoals bij het type `int`).

16.5 Conversie operatoren

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

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

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

Pas op bij conversies: definieer geen conversie waarbij informatie verloren gaat! Is het dan wel verstandig om een `Breuk` automatisch te laten converteren naar een `int`?[193]

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

[192] `a = b++++` wordt geïnterpreteerd als `a = (b++)++`. De postfix versie van de `operator++` maakt een kopietje van `b`, verhoogd de waarde van `b` en geeft tot slot de waarde van het kopietje (de *oude* waarde van `b`) terug. Het is niet mogelijk om de operator ++ op deze returnwaarde uit te voeren. De GCC C++-compiler geeft de error: `error: lvalue required as increment operand`. Met `lvalue` wordt een expressie bedoeld die links van een = teken gebruikt kan worden.

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

16.6 Voorbeeld class Breuk (vierde versie)

Dit voorbeeld is een uitbreiding van de in paragraaf 2.11 gegeven class Breuk (derde versie). In deze versie zijn de operator == en de operator != toegevoegd. We leren een nieuwe vriend (*friend*) kennen die ons helpt bij het implementeren van deze operatoren. Ook leer je 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 presenteer nu eerst de complete broncode van het programma Breuk3.cpp waarin het type Breuk gedeclareerd, geïmplementeerd en gebruikt wordt. Meteen daarna ga ik, één voor één, op bovengenoemde punten in.

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

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

istream& operator>>(istream& in, Breuk& b);
const Breuk operator+(const Breuk& links, const Breuk& rechts);
```

```
// ...
// Er zijn nog veel uitbreidingen mogelijk
// ...
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& rechts) {
    boven = boven * rechts.onder + onder * rechts.boven;
    onder *= rechts.onder;
    normaliseer();
    return *this;
}

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

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

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

```
const Breuk operator+(const Breuk& links, const Breuk& rechts) {
    Breuk copy_links {links};
    copy_links += rechts;
    return copy_links;
}

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

istream& operator>>(istream& in, Breuk& b) {
    int teller;
    if (in >> teller)
        if (in.peek() == '/') {
            in.get();
            int noemer;
            if (in >> noemer) b = Breuk{teller, noemer};
            else b = Breuk{teller};
        }
        else b = Breuk{teller};
    else b = Breuk{};
    return in;
}

template<>
struct std::formatter<Breuk>: public formatter<string> {
    auto format(const Breuk& breuk, auto& context) const {
        ostringstream ss;
        ss << breuk;
        return formatter<string>::format(ss.str(), context);
    }
};

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

int main() {
    Breuk b1, b2;               // definiëren van variabelen
    print("Geef Breuk: ");
```

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

Ik heb ervoor gekozen om de globale **operator**== te overloaden in plaats van een member-functie `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. Sinds C++20 is dit niet meer nodig, zie paragraaf 16.9. Maar ik ben er toch voor gekozen om de globale **operator**== te overloaden omdat dit een goed voorbeeld is van het gebruik van een friend function.

De implementatie van deze globale **operator** == is als volgt:

```
bool operator==(const Breuk& links, const Breuk& rechts) {
    return links.boven == rechts.boven && links.onder == rechts.onder;
}
```

Dit levert bij compilatie echter de volgende fouten op: `error: 'int Breuk::boven' is private within this context` en `error: 'int Breuk::onder' is private within this context`.

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*.

16.7 `friend` functions

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

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

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

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

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

Vanaf C++20 zal de compiler automatisch een **operator**!= genereren voor een UDT als er een **operator**== gedefinieerd is[194]. De gegenereerde **operator**!= zal de uitkomst van de **operator**== inverteren.

16.8 Default operator==

Sinds C++20 kun je de compiler ook een default **operator**== voor een UDT laten genereren. Deze voert dan een **operator**== voor alle datamembers uit. De default **operator**== geeft alleen **true** terug als alle datamembers gelijk zijn. In het geval van de class Breuk is dat alleen correct als de breuken altijd genormaliseerd zijn.

Omdat er in dit dictaat voor is gekozen om de breuken altijd genormaliseerd te houden, kan de default **operator**== worden gebruikt en is het dus niet nodig om zelf een operator== te definiëren zoals in paragraaf 16.6 gedaan is.

De **operator**== kan als volgt =**default** worden gedefinieerd, zie Breuk_default_operator==.cpp.

```
class Breuk {
public:
    Breuk(int t = 0, int n = 1);
    bool operator==(const Breuk& rechts) const = default;
private:
    // Invariant: Breuken zijn altijd genormaliseerd.
    int boven;
    int onder;
    void normaliseer();
};
```

Je kunt nu zonder zelf een **operator**== en **operator**!= te definiëren, toch breuken vergelijken. Doordat een **int** automatisch kan worden omgezet in een Breuk, kun je ook een Breuk vergelijken met een **int** en vice versa.

[194] Als je gebruik maakt van een C++ versie van vóór C++20, dan moet je zelf een globale operator!= overloaden voor het type Breuk:

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

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

16.9 Drieweg-vergelijkingsoperator <=>

In C++20 is de drieweg-vergelijkingsoperator geïntroduceerd <=>, ook wel spaceship operator genoemd omdat het enigzins op een ruimteschip uit Star Wars lijkt[195]. Met behulp van deze operator kan in één keer bepaald worden of een object kleiner, groter of gelijk is aan een ander object. Bijvoorbeeld, zie spaceship.cpp:

```
int i {5};
auto result { i <=> 5 };
if (result < 0) { println("less"); }
if (result > 0) { println("greater"); }
if (result == 0) { println("equal"); }
```

Het nut van deze operator blijkt vooral bij operator overloading, omdat, als de **operator**<=> overloaded wordt, de compiler automatisch de vergelijkingsoperatoren <, <=, > en >= definieert.

De spaceship operator kan een strong_ordering, een weak_ordering of een partial_ordering teruggeven:

- Een strong_ordering geeft aan dat de twee objecten die vergeleken worden volledig geordend zijn, dat wil zeggen dat ze ofwel gelijk (strong_ordering::equal) zijn, of dat het ene object kleiner (strong_ordering::less) or groter (strong_ordering::greater) is dan het andere.
- Een partial_ordering heeft de volgende mogelijke waarden: partial_ordering::less, partial_ordering::greater, partial_ordering::equivalent of partial_ordering::unordered. Twee objecten zijn partial_ordering::unordered als ze niet geordend kunnen worden, bijvoorbeeld als een object van het type **double** de waarde NaN (Not an Number)[196] heeft. Twee objecten a en b zijn partial_ordering::equivalent als ze niet partial_ordering::unordered zijn en als a niet kleiner en ook niet groter is dan b.
- Een weak_ordering heeft de volgende mogelijke waarden: partial_ordering::less, partial_ordering::greater of partial_ordering::equivalent. Twee objecten die weak_ordering::equivalent zijn, zijn niet noodzakelijkerwijs gelijk, maar worden wel als equivalent beschouwd voor de doeleinden van sorteren en vergelijken. Als je bijvoorbeeld objecten van de class string case insensitive vergelijkt, dan kunnen twee

[195] Zie eventueel https://www.starwars.com/databank/darth-vader-s-tie-fighter

[196] Zie https://en.wikipedia.org/wiki/NaN.

strings die verschillen in hoofdletters en kleine letters als `weak_ordering::equivalent` worden beschouwd, ook al zijn ze niet exact gelijk.

De drie verschillende soorten ordeningen ondersteunen bepaalde impliciete conversies. Een `strong_ordering` kan impliciet worden geconverteerd naar een `partial_ordering` of een `weak_ordering`. Een `weak_ordering` kan impliciet worden geconverteerd naar een `partial_ordering`.

Door de spaceship operator te definiëren voor een UDT, worden automatisch de operatoren <, <=, > en >= gedefinieerd. Door de operator == te definiëren voor een UDT, worden automatisch de operatoren != gedefinieerd. Zie Breuk_spaceship.cpp:

```
class Breuk {
public:
    Breuk();
    Breuk(int t);
    Breuk(int t, int n);
    bool operator==(const Breuk& rechts) const;
    strong_ordering operator<=>(const Breuk& rechts) const;
private:
    // Invariant: Breuken zijn altijd genormaliseerd.
    int boven;
    int onder;
    void normaliseer();
friend ostream& operator<<(ostream& out, const Breuk& b);
};

bool Breuk::operator==(const Breuk& rechts) const {
    return boven == rechts.boven && onder == rechts.onder;
}

strong_ordering Breuk::operator<=>(const Breuk& rechts) const {
    if (*this == rechts) {
        return strong_ordering::equal;
    }
    if (boven * rechts.onder < rechts.boven * onder) {
        return strong_ordering::less;
    }
    return strong_ordering::greater;
}
```

Het is nu mogelijk de objecten b1 en b2 van het type Breuk te vergelijken met alle vergelijkingsoperatoren, bijvoorbeeld:

```
if (b1 != b2) {
    println("{} is niet gelijk aan {}.", b1, b2);
}
if (b1 < b2) {
    println("{} is kleiner dan {}.", b1, b2);
}
```

Omdat een **int** automatisch kan worden omgezet in een Breuk, via de constructor Breuk(**int**), kunnen ook vergelijkingen tussen een Breuk b1en een **int** iworden gemaakt, bijvoorbeeld:

```
if (b1 != i) {
    println("{} is niet gelijk aan {}.", b1, i);
}
if (b1 < i) {
    println("{} is kleiner dan {}.", b1, i);
}
```

Ook kan een **int** i worden vergeleken met een Breuk b2, bijvoorbeeld:

```
if (i != b2) {
    println("{} is niet gelijk aan {}.", i, b2);
}
if (i < b2) {
    println("{} is kleiner dan {}.", i, b2);
}
```

Merk op dat het in dit geval *niet* nodig is om de **operator**== en de **operator**<=> als globale friend functies te definiëren, zoals dat bij **operator**+ wel nodig is, zie paragraaf 16.3. De compiler snapt, sinds C++20, zelf dat a>b gelijk is aan b<a, dat a==b gelijk is aan b==a, en dat a>=b gelijk is aan b<=a, enzovoort. De overige binaire operatoren zoals **operator**+, **operator**- enzovoorts moeten echter nog steeds als globale friend functies worden gedefinieerd, a-b is immers niet gelijk aan b-a.

In oudere versies van C++ (pre C++20) moest je zelf alle vergelijkingsoperatoren ==, !=, <, <=, > en >= apart definiëren. Daarbij moest je er zelf voor zorgen dat deze opeatoren onderling consistent waren, bijvoorbeeld dat a>b gelijk was aan b<a en dat a!=b waar was als a>b waar was, enzovoort. Tevens moesten deze operatoren als globale functies gedefinieerd worden om er voor te zorgen dat impliciete conversies zowel op het linker als het rechter argument toegepast werden. Zie Breuk_compare_pre_C++20.cpp.

16.10 Default operator<=>

Sinds C++20 kun je de compiler ook een default **operator**<=> voor een UDT laten genereren. Deze voert dan één voor één een **operator**<=> op de datamembers uit, in de volgorde waarin ze zijn gedeclareerd. Zodra de vergelijking van een datamember een ongelijk resultaat geeft, zal de default **operator**<=> dit ongelijk resultaat teruggeven. Als alle datamembers gelijk zijn, zal de default **operator**<=> `std::strong_ordering::equal` teruggeven. Als de **operator**<=> =**default** gedefinieerd is, dan zal de compiler ook automatisch de operatoren ==, !=,<, >, <= en >= genereren.

Voor de class `Breuk` geeft dit echter niet het gewenste resultaat, omdat als de teller van de ene breuk groter is dan de teller van de andere breuk, dan wil dat niet zeggen dat de ene breuk groter is dan de andere breuk, zie `Breuk_spaceship_default.cpp`:

```
class Breuk {
public:
    Breuk(int t = 0, int n = 1);
    strong_ordering operator<=>(const Breuk& rechts) const = default;
private:
    // Invariant: Breuken zijn altijd genormaliseerd.
    int boven;
    int onder;
    void normaliseer();
};
```

De operatoren == en != worden nu automatisch, *correct* gegenereerd door de compiler. De operatoren <, >, <= en >= worden nu automatisch, maar *incorrect* gegenereerd door de compiler. Als je kijkt of $^3/_8$ kleiner is dan $^1/_2$, dan zal dit **false** opleveren, terwijl dit natuurlijk **true** moet zijn.

Het programmafragment:

```
    Breuk b1 {3, 8};
    Breuk b2 {1, 2};
    assert(b1 != b2);
    assert(b1 < b2);
```

Geeft als output:

```
assertion "b1 < b2" failed
```

Het is in dit geval dus noodzakelijk om zelf de **operator**<=> te definiëren voor Breuk zoals gegeven is in paragraaf 16.9.

Bij een UDT waarbij een tijdsduur wordt bijgehouden in uren en minuten en waarin de tijdsduur altijd genormaliseerd is, werkt de default **operator**<=> wel correct, zie Tijdsduur_spaceship_default.cpp.

16.11 Operator overloading (deel 4)

Sinds C++23 kunnen de print en println functies gebruikt worden om variabelen af te drukken, zie paragraaf 1.2. Daarvoor werd de **operator**<< gebruikt om variabelen af te drukken, zie paragraaf 1.3. Om het gebruik van een UDT zo flexibel mogelijk te maken is het aan te bevelen om het mogelijk te maken om objecten van het UDT op beide manieren af te kunnen drukken.

Het object cout dat in de standaard library 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[197], zie Breuk4.cpp:

```
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, zie paragraaf 16.7. Als eerste parameter is een ostream& gebruikt omdat de operator het als argument meegegeven object (in ons geval cout) moet kunnen aanpassen. Als return type is het type ostream& gebruikt. De als parameter meegegeven ostream& wordt ook weer teruggegeven. Dit heeft tot gevolg dat je verschillende << operatoren achter elkaar kunt 'rijgen'. Bijvoorbeeld:

```
    cout << "De breuk a = " << a << " en de breuk b = " << b << '\n';
```

[197] 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 parameter een `ostream&` deze reference ook weer als return waarde teruggeven kun je de bovenstaande **`operator`**<< vereenvoudigen tot:

```
ostream& operator<<(ostream& out, const Breuk& b) {
    return out << b.boven << '/' << b.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.

Om het mogelijk te maken om een `Breuk` af te drukken met de `print` functies, moet de template `std::formatter` gespecialiseerd worden voor het de class `Breuk`, zie paragraaf 17.6.

```
template<>
struct std::formatter<Breuk>: public formatter<string> {
    auto format(const Breuk& breuk, auto& context) const {
        ostringstream ss;
        ss << breuk;
        return formatter<string>::format(ss.str(), context);
    }
};
```

In dit geval is er voor gekozen om de `formatter<Breuk>` over te erven van `formatter<string>`. Dit zorgt ervoor dat de format specifiers die bij een `string` gebruikt kunnen worden, nu ook bij een `Breuk` gebruikt kunnen worden. De implementatie van de `format` functie maakt gebruik van de al eerder gedefinieerde **`operator`**<< om een `Breuk` af te drukken naar de `ostringstream` `ss` die vervolgens omgezet kan worden naar een `string` door middel van `ss.str()`.

De volgende code:

```
    Breuk b(30, 94);
    println("Breuk b = {}", b);
    println("Breuk b = {:_^16}", b);
    // println("Breuk breuk = {:0X}", breuk);
}

/* Output:
```

```
b = 15/47
```

Geeft de volgende uitvoer:

```
b = 15/47
      b =
   15/47
"b ="
"15/47"
______b =_______
_____15/47______
```

ls je deze laatste versie van `Breuk` vergelijkt met de eerste versie op pagina 51, dan is eigenlijk het enige belangrijke verschil dat het zelfgedefinieerde type `Breuk` nu op precies dezelfde wijze als de ingebouwde types te gebruiken is. Je hebt er heel wat 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 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 geef ik hier nogmaals antwoord op de vraag: "Is het al die inspanningen wel waard om een `Breuk` zo te maken dat je hem net zoals een ingebouwd type kan gebruiken?" Ja! Het maken van de class `Breuk` is dan wel een hele inspanning maar iedere programmeur kan vervolgens dit zelfgedefinieerde type, als hij of zij de naam `Breuk` maar kent, als vanzelf (intuïtief) gebruiken. Er is daarbij geen handleiding of helpfunctie nodig is, omdat het type `Breuk` zich net zo gedraagt als een ingebouwd type. De class `Breuk` hoeft maar één keer gemaakt te worden, maar kan talloze malen gebruikt worden.

Natuurlijk is het nog beter en slimmer om een goedgeteste implementatie van het type `Breuk` te zoeken en te hergebruiken, bijvoorbeeld `boost::rational`[198]. In dit dictaat heb ik het type `Breuk` stap voor stap ontwikkeld om je te leren hoe je zoiets in C++ kunt doen. In de praktijk moet je deze kennis alleen toepassen om een UDT mee te definiëren die je nergens kunt vinden.

16.12 `static` class members

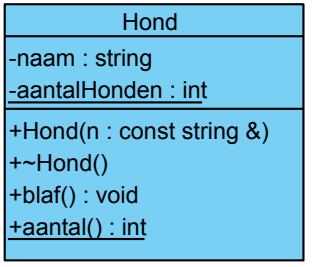
Je hebt geleerd dat elk object zijn eigen datavelden heeft terwijl de memberfuncties door alle objecten van een bepaalde class gedeeld worden. Stel nu dat je wilt tellen hoeveel objecten van een bepaalde class 'levend' zijn. Dit zou kunnen door een globale teller te definiëren

[198] Zie https://www.boost.org/doc/libs/1_72_0/libs/rational/rational.html.

die in de constructor van de class met 1 wordt verhoogd en in de destructor weer met 1 wordt verlaagd. Het gebruik van een globale variabele maakt het programma echter slecht onderhoudbaar.

Een **static** dataveld is een onderdeel van de class en wordt door alle objecten van de class gedeeld. Z'n **static** dataveld kan bijvoorbeeld gebruikt worden om het aantal 'levende' objecten van een class te tellen. In UML worden **static** datavelden en **static** memberfuncties onderstreept weergegeven zoals in figuur 16.1 te zien is. Zie Honden_teller.cpp:

```
import std;
using namespace std;

class Hond {
public:
    Hond(string_view n);
    ~Hond();
    void blaf() const;
    static int aantal();
private:
    string naam;
    static int aantal_honden;
};

int Hond::aantal_honden {0};

Hond::Hond(string_view n): naam {n} {
    ++aantal_honden;
}
Hond::~Hond() {
    --aantal_honden;
}
int Hond::aantal() {
    return aantal_honden;
}
void Hond::blaf() const {
    println("{} zegt: WOEF", naam);
}

int main() {
    println("Er zijn nu {} honden.", Hond::aantal());
    {
```



Figuur 16.1: Class Hond met **static** members.

```
        Hond h1 {"Boris"};
        h1.blaf();
        println("Er zijn nu {} honden.", Hond::aantal());
        Hond h2 {"Fikkie"};
        h2.blaf();
        println("Er zijn nu {} honden.", Hond::aantal());
    }
    println("Er zijn nu {} honden.", Hond::aantal());
}
```

Uitvoer:

```
Er zijn nu 0 honden.
Boris zegt: WOEF
Er zijn nu 1 honden.
Fikkie zegt: WOEF
Er zijn nu 2 honden.
Er zijn nu 0 honden.
```

Je kunt een **static** memberfunctie op twee manieren aanroepen:

- via een object van de class: `object_naam.memberfunctie_naam(parameters);`
- via de classnaam: `class_naam::memberfunctie_naam(parameters)`.

De laatste methode heeft de voorkeur omdat die ook gebruikt kan worden als er nog geen objecten zijn.

Een **static** memberfunctie heeft ten opzichte van een normale memberfunctie de volgende beperkingen:

- Een **static** memberfunctie heeft geen receiver (ook niet als hij via een object aangeroepen wordt).
- Een **static** memberfunctie heeft dus geen **this**-pointer.
- Een **static** memberfunctie kan dus geen gewone memberfuncties aanroepen en ook geen gewone datavelden gebruiken.

16.13 Initialiseren van datavelden

De datavelden van een class mogen ook direct geïnitialiseerd worden. Als dezelfde datavelden ook in een member initialization list (zie paragraaf 2.4) worden geïnitialiseerd, wordt de

directe initialisatie genegeerd.
Zie Breuk1_direct_member_init.cpp:

```
class Breuk {
public:
    Breuk();
    Breuk(int t);
    Breuk(int t, int n);
    int teller() const;
    int noemer() const;
private:
    int boven {0}; // directe initialisatie
    int onder {1}; // directe initialisatie
    void normaliseer();
};

Breuk::Breuk() { // geen initialization list
}

Breuk::Breuk(int t): boven{t} { // de datamember onder wordt niet ↩
 ↪ in de initialization list geïnitialiseerd
}

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

//  ...

int main() {
    Breuk b1;
    println("b1 = {}/{}", b1.teller(), b1.noemer());
    Breuk b2 {4};
    println("b2 {{4}} = {}/{}", b2.teller(), b2.noemer());
    Breuk b3 {23, -5};
    println("b3 {{23, -5}} = {}/{}", b3.teller(), b3.noemer());
}
```

Uitvoer:

```
b1 = 0/1
b2 4 = 4/1
b3 23, -5 = -23/5
```

16.14 Compile time constanten in een class

Met behulp van het keyword **constexpr** kun je compile time constanten definiëren. Globale constanten komen de onderhoudbaarheid niet ten goede. Als je alle objecten van een class dezelfde compile time constante wilt laten delen, kun je deze constanten **static**[199] definiëren.

Voorbeeld met **static** compile time constanten (Color.cpp):

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

ostream& operator<<(ostream& out, Color c) {
    return out << setw(6) << setfill('0') << hex << c.get_value();
}
```

[199] Zie paragraaf 16.12

```
Color::Color(): value{BLACK} {
}

Color::Color(int v): value{v} {
}

int Color::get_value() const {
    return value;
}

void Color::set_value(int v) {
    value = v;
}
```

Deze constanten kunnen als volgt gebruikt worden:

```
    Color c {Color::YELLOW};
    println("c = {}", c);
    c.set_value(Color::BLUE);
    println("c = {}", c);
```

Uitvoer:

```
c = ffff00
c = 0000ff
```

Als alternatief kun je ook een **enum** gebruiken.

Voorbeeld met anonymous (naamloze) **enum**:

```
class Color {
public:
    Color();
    Color(int c);
    int get_value() const;
    void set_value(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};
    println("c = {}", c);
    c.set_value(Color::BLUE);
    println("c = {}", c);
```

Uitvoer:

```
c = ffff00
c = 0000ff
```

16.15 Inline memberfuncties

Sommige programmeurs denken dat het gebruik van eenvoudige memberfuncties, zoals de memberfuncties `teller` en `noemer` van de class Breuk, zie paragraaf 2.3, 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. Dit heeft (mogelijk) wel tot gevolg dat de code omvangrijker wordt. Memberfuncties mogen ook in de classdeclaratie gedefinieerd worden. Ze zijn dan vanzelf inline. Als de memberfunctie in de classdeclaratie alleen maar gedeclareerd is, kan de definitie van die functie voorafgegaan worden door het keyword **`inline`** om de functie inline te maken. Veel compilers plaatsen, als je aangeeft het programma voor snelheid te willen optimaliseren, korte functies automatisch inline.

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;
}
```

16.16 User-defined literals

Voor de ingebouwde types in C++ kun je zogenoemde literals definiëren. Een literal is een constante waarde. Voorbeelden van literals van het type **int** zijn `42`, `0b0101010` en `0x2A`. **true** is een voorbeeld van een literal van het type **bool**. `'a'` en `'\n'` zijn voorbeelden van literals van het type **char**. `1.2` en `-3.234e-17` zijn voorbeelden van literals van het type **double**. `"Hallo daar"` is een voorbeeld van een C-string literal van het type **const char**`[11]`. Het is ook mogelijk om literals (constante getallen) te definiëren voor user-defined datatypes (UDT's) zoals `Breuk`. Zie `user_defined_literals.cpp`.

Dit is bijvoorbeeld ook gebruikt in de volgende standaard library classes:

- `string`
  `"Hallo daar"s` is een literal van het type `string`.
- `complex`
  `2i` is een literal van het type `complex<double>`.
- `chrono::duration`
  `2ms` is een literal van het type `chrono::duration`.

16.17 enum class

In C heb je de **enum** leren kennen, zie eventueel [5, paragraaf 5.6.2] of https://en.cppreference.com/w/c/language/enum. In C++ is een verbeterde versie, zogenoemde **enum class**, toegevoegd.

In C en C++ kun je een gewoon enumeratie-type als volgt definiëren en gebruiken[200]:

```
enum Stoplicht {groen, oranje, rood};
Stoplicht s = rood;
println("s = {}", s);
```

De namen die je gebruikt in de **enum** zijn niets anders dan namen voor integer constanten. Ze worden automatisch genummerd vanaf `0`[201]. De uitvoer van de bovenstaande code is dus:

```
s = 2
```

De naam `rood` wordt, zoals je ziet automatisch omgezet naar een integer. De naam `rood` kun je overal waar de **enum** `Stoplicht` zichtbaar is gebruiken. Omdat een **enum** vaak globaal gedefinieerd wordt, leidt dit bij grotere programma's tot problemen als er ook een **enum** `Kleur` is gedefinieerd die ook de naam `rood` definieert. Als je in plaats van een **enum** een **enum class** gebruikt dan wordt zogenoemde 'strong typing' gebruikt en zijn de namen 'scoped'. Bijvoorbeeld:

```
enum class Stoplicht {groen, oranje, rood};
Stoplicht s = Stoplicht::rood;
if (s == Stoplicht::rood) println("s = rood");
```

Het gebruik van 'strong typing' zorgt ervoor dat een variabele van het type `Stoplicht` *niet* meer automatisch omgezet wordt naar een integer. Je kunt zo'n variabele dus niet zonder meer printen[202]. Het feit dat de namen 'scoped' zorgt ervoor dat je moet aangeven van welk type een enumeratie-waarde (naam) is. Je moet dus `Stoplicht::rood` gebruiken in plaats van `rood`. Dit zorgt ervoor dat je de naam `rood` meerdere keren als enumeratie-waarde kunt gebruiken. Bijvoorbeeld ook voor `Kleur::rood`. Omdat het gebruik van **enum class** geen

[200] In C heb je een **typedef** nodig om hetzelfde te bereiken: **typedef enum** `groen, oranje, rood Stoplicht;`. In C++ is de naam van een enumeratie automatisch een type.

[201] Je kunt indien gewenst de waarde die bij elke naam hoort ook opgeven. Bijvoorbeeld: **typedef enum** `groen = 7, oranje = 13, rood Stoplicht;` De naam `groen` heeft nu de waarde `7`, `oranje` de waarde `13` en `rood` de waarde 14. Als je geen waarde opgeeft, wordt er doorgenummerd.

[202] Als je dit wel wilt, dan moet je operator overloading gebruiken, zie paragraaf 16.11 en `enum_class.cpp`.

onverwachtse verrassingen op kan leveren gebruiken we in dit dictaat **enum class** in plaats van **enum**[203].

Als je de **enum class** in een beperkte scope gebruikt, dan kun je ook gebruik maken van **using** om de namen van de enumeratie-waarden in die scope beschikbaar te maken. Bijvoorbeeld (enum_class_using.cpp):

```
enum class Stoplicht {groen, oranje, rood};
using enum Stoplicht;
Stoplicht s = rood;
if (s == rood) println("s = rood");
```

Het onderliggende type van **enum class** is standaard **int**, maar kan worden aangepast (enum_class_underlying.cpp):

```
enum class Stoplicht: int8_t {groen, oranje, rood};
```

Je kunt ook de onderliggende waarde van een enum class constante opvragen:

```
println("rood = {}", to_underlying(Stoplicht::rood));
```

Geeft als uitvoer:

```
rood = 2
```

[203] Dit wordt ook aangeraden in de C++ Core Guideline Enum.3.

17

Templates (de details)

17.1 Variable-template

Ook variabelen en constanten kun je als template definiëren.

Sinds C++20 zijn in de namespace `std::numbers` een aantal wiskundige constanten gedefinieerd als templates, zoals `pi_v<T>` en `e<v>`. De constanten `pi` en `e` zijn in diezelfde namespace gedefinieerd als respectievelijk `pi_v<double>` en `e_v<double>`.

Je kunt deze constanten als volgt gebruiken:

```
auto d1 {numbers::pi_v<double>}; // d1 is een double met de waarde ↩
 ↪ π
auto d2 {numbers::pi}; // d2 is een double met de waarde π
auto f {numbers::pi<float>}; // f is een float met de waarde π
```

De constante τ (tau)[204] is niet gedefinieerd in de namespace `std::numbers`.

Je kunt de constante `tau` zelf definiëren als template zodat je naar keuze een **`float`**, **`double`** of **`long double`** met de waarde τ kunt aanmaken:

```
template<floating_point T> constexpr T tau = ↩
 ↪ atan(static_cast<T>(1)) * 8;
```

[204] De constante τ is gelijk aan 2π, zie https://en.wikipedia.org/wiki/Tau_(mathematics).

Je ziet dat het concept `floating_point` gebruikt is als template parameter constraint, zie paragraaf 3.10. Dit zorgt ervoor dat de compiler een duidelijke foutmelding geeft als de template met een niet floating point type wordt gebruikt.

Je kunt nu de constante `tau` als volgt gebruiken:

```
auto f {tau<float>}; // f is een float met de waarde τ
auto d {tau<double>}; // d is een double met de waarde τ
```

Zie `tau.cpp`.

17.2 Template argument binding

Met **`using`** kun je een parametertype van een template 'invullen'. In het Engels wordt dit *argument binding* genoemd.

```
template<typename Value> using String_map = map<string, Value>;
```

Je kunt nu de template `String_map` als volgt gebruiken:

```
String_map<int> m1; // m1 is van het type map<string, int>
String_map<Breuk> m2; // m2 is van het type map<string, Breuk>
```

17.3 Template parameters automatisch bepalen

In sommige gevallen kunnen template parameters automatisch door de C++-compiler bepaald worden[205]. Bijvoorbeeld het standaard type `vector` kan als volgt gebruikt worden:

```
vector v1 {1, 2, 3}; // v1 is van het type vector<int>
vector v2 {v1}; // v2 is ook van het type vector<int>
```

Een zogenoemde *deduction guide* kan gespecificeerd worden om de compiler hierbij te helpen. Voor de in paragraaf 19.8 gedefinieerde class `Array` kun je bijvoorbeeld de volgende deduction guide specificeren, zie `Array_template_deduction_guide.cpp`:

```
template<input_iterator Iter> Array(Iter, Iter) -> ↩
 ↪ Array<iter_value_t<Iter>>;
```

Dit zorgt ervoor dat type van onderstaand object `a2` door de compiler bepaald kan worden.

[205] Dit is mogelijk vanaf C++17.

```
    Array a1 {1, 2, 3, 4, 5}; // a1 is van het type Array<int>
    Array a2 (v.cbegin() + 1, v.cend() - 1); // a2 is van het type ↩
↪ Array<int>
```

17.4 Zelf concepts definiëren

In paragraaf 3.10 heb je al kennis gemaakt met standaard concepten. Standaard concepten kun je vinden in de concepts library[206], iterators library[207], algorithms library[208] en ranges library[209]. Ook heb je gezien dat deze concepten met behulp van logische expressies gecombineerd kunnen worden. Het is daarnaast ook mogelijk om zelf concepten te definiëren. In paragraaf 11.2 heb je al een voorbeeld gezien waarin de zelfgedefinieerde concepten `Number<T>` en `Const<T>` gebruikt werden. Hierbij moet `Number<T>` waar (**`true`**) zijn voor een type `T` waarmee je kunt rekenen (zoals **`int`** of **`double`** maar ook `complex<double>` of `Breuk`). Het concept `Const<T>` moet waar (**`true`**) zijn voor een type `T` dat read-only, dus als **`const`** `T`, is gedefinieerd.

Het definiëren van een **`concept`** dat waar (**`true`**) is voor types waarmee je kunt rekenen, lijkt in eerste instantie eenvoudig:

```
concept Number = integral<T> || floating_point<T>;
```

Maar dit is niet correct want `integral<bool>` is **`true`**, maar met een **`bool`** kun je *niet* rekenen. En zowel `integral<complex<double>>` als `floating_point<complex<double>>` zijn **`false`**, maar met een `complex<double>` kun je *wel* rekenen. Ook voor een zelfgedefinieerd UDT zoals `Breuk` zal `Number<Breuk>` nu **`false`** zijn, ook als alle rekenkundige operatoren voor het UDT `Breuk` gedefinieerd zijn.

Het is mogelijk om zelf een concept te definiëren door in een zogenoemde **`requires`** clausule aan te geven welke operaties het type moet ondersteunen om aan het concept te voldoen. Je kunt bijvoorbeeld het concept `Arithmetic<T>` definiëren dat waar (**`true`**) is voor types `T` waarvoor de rekenkundige operatoren `+`, `-`, `*` en `/` gedefinieerd zijn (`concept_number.cpp`):

```
template <typename T>
concept Arithmetic = requires(T x, T y) {
    x=x;
```

[206] https://en.cppreference.com/cpp/concepts

[207] https://en.cppreference.com/cpp/iterator#Iterator_concepts

[208] https://en.cppreference.com/cpp/iterator#Algorithm_concepts_and_utilities

[209] https://en.cppreference.com/cpp/ranges#Range_concepts

```
    x+y; x-y; x*y; x/y;
    x+=y; x-=y; x*=y; x/=y;
};
```

Na het keyword **requires** volgt een soort parameterlijst waarin variabelen gedefinieerd kunnen worden. Daarna volgt een codeblok. De code in het codeblok wordt nooit uitgevoerd, alleen gecontroleerd door de compiler. Als deze code *zonder* errors gecompileerd kan worden, dan is het concept waar (**true**). Als de code *niet* zonder errors gecompileerd kan worden, dan geeft de comiler *geen* foutmelding, maar is het concept onwaar (**false**).

De volgende code:

```
    println("Arithmetic<int> = {}", Arithmetic<int>);
    println("Arithmetic<double> = {}", Arithmetic<double>);
    println("Arithmetic<complex<double>> = {}", ↩
↪ Arithmetic<complex<double>>);
    println("Arithmetic<Breuk> = {}", Arithmetic<Breuk>);
    println("Arithmetic<vector<int>> = {}", Arithmetic<vector<int>>);
```

Geeft, zoals verwacht, de uitvoer:

```
Arithmetic<int> = true
Arithmetic<double> = true
Arithmetic<complex<double>> = true
Arithmetic<Breuk> = true
Arithmetic<vector<int>> = false
```

Maar het zal je misschien verrassen dat de volgende code:

```
    println("Arithmetic<bool> = {}", Arithmetic<bool>);
    println("Arithmetic<char> = {}", Arithmetic<char>);
```

De volgende uitvoer geeft:

```
Arithmetic<bool> = true
Arithmetic<char> = true
```

Dit komt omdat de types **bool** en **char** automatisch kunnen worden geconverteerd naar **int**, en daardoor dus voldoen aan de eisen van het concept `Arithmetic<T>`. Maar **bool** en **char** zijn natuurlijk geen types die je gebruikt om rekenkundige bewerkingen mee uit te voeren. Dus een concept dat waar (**true**) is voor types waarmee je kunt rekenen, moet deze types niet accepteren.

Omdat er in C++ verschillende types zijn waarin een character kan worden opgeslagen, is het handig om eerst het concept `Character<T>` te definiëren dat bepaald of `T` een character type is:

```
template <typename T>
concept Character =
    same_as<T, char> ||
    same_as<T, signed char> ||
    same_as<T, unsigned char> ||
    same_as<T, wchar_t> ||
    same_as<T, char8_t> ||
    same_as<T, char16_t> ||
    same_as<T, char32_t>;
```

Er is hier gebruik gemaakt van het standaard concept `same_as<T, U>` dat bepaalt of `T` en `U` hetzelfde type zijn[210].

Vervolgens kan dan het concept `Number<T>` als volgt gedefinieerd worden:

```
template<typename T>
concept Number = Arithmetic<T> && !Character<T> && !same_as<T, bool>;
```

De volgende code:

```
    println("Number<int> = {}", Number<int>);
    println("Number<double> = {}", Number<double>);
    println("Number<complex<double>> = {}", Number<complex<double>>);
    println("Number<Breuk> = {}", Number<Breuk>);
    println("Number<char> = {}", Number<char>);
    println("Number<bool> = {}", Number<bool>);
    println("Number<vector<int>> = {}", Number<vector<int>>);
```

Geeft, zoals verwacht, de uitvoer:

```
Number<int> = true
Number<double> = true
Number<complex<double>> = true
Number<Breuk> = true
Number<char> = false
Number<bool> = false
Number<vector<int>> = false
```

[210] Zie https://en.cppreference.com/cpp/concepts/same_as.

Zie paragraaf 11.2 voor een toepassing van het concept `Number<T>`. In datzelfde voorbeeld is ook het concept `Const<T>` gebruikt dat waar (**`true`**) is voor types die read-only, dus als **`const`** `T`, zijn gedefinieerd.

Tot mijn verbazing is dit concept niet opgenomen in de C++23 standaard en ook niet in de C++26 standaard, maar gelukkig is het eenvoudig zelf te definiëren:

```
template <typename T>
concept Const = is_const_v<T>;
```

Hierbij is gebruik gemaakt van een zogenoemde *type trait*, namelijk `is_const`, die in C++11 gedefinieerd is en de bijbehorende *helper variabele template* `is_const_v` die in C++17 gedefinieerd is[211]. Deze type traits waren een manier om informatie over types te verkrijgen voordat concepten in C++20 werden geïntroduceerd. In dit geval geeft `is_const_v<T>` aan of `T` read-only, dus als **`const`** `T`, gedefinieerd is. Precies wat we nodig hebben om het concept `Const<T>` te definiëren.

17.5 Class template specialisatie

In paragraaf 3.9 is de functietemplate `number_of_zero_bits` besproken, zie `number_of_zero_bits1.cpp`. Stel dat je deze functie in een programma heel vaak moet aanroepen voor het type `uint8_t` en dat dit programma niet snel genoeg uitgevoerd wordt en dat uit een analyse blijkt dat deze functie de bottleneck is en dus versneld moet worden.

Je zou dan op het idee kunnen komen dat een `uint8_t` maar 256 verschillende waarden kan hebben en dat je dus een *lookup table* kunt maken met 256 elementen, waarin voor elke mogelijke waarde van een `uint8_t` het aantal nullen in de binaire representatie van die waarde opgeslagen is[212]. In dat geval zou je *function name overloading*, zie paragraaf 1.12, kunnen gebruiken om een overload van `number_of_zero_bits` te definiëren die een `uint8_t` als parameter heeft en die de waarde uit de lookup table teruggeeft, zie `number_of_zero_bits9.cpp`:

```
int number_of_zero_bits(uint8_t n) {
    static const int lookup[256] = {
        8, 7, 7, 6, 7, 6, 6, 5, 7, 6, 6, 5, 6, 5, 5, 4,
        7, 6, 6, 5, 6, 5, 5, 4, 6, 5, 5, 4, 5, 4, 4, 3,
```

[211] Zie `https://en.cppreference.com/cpp/types/is_const`.

[212] Deze manier van het versnellen van een functie door gebruik te maken van een *lookup table* is een veelgebruikte optimalisatietechniek, zie `https://en.wikipedia.org/wiki/Lookup_table` die *memoization* genoemd wordt, zie `https://en.wikipedia.org/wiki/Memoization`.

```
        // zie voor de rest van de lookup table: number_of_zero_bits9.cpp
        4, 3, 3, 2, 3, 2, 2, 1, 3, 2, 2, 1, 2, 1, 1, 0
    };
    return lookup[n];
}
```

De volgende code:

```
    uint8_t n1 {42};
    println("number_of_zero_bits({:0{}b}) = {}", n1, sizeof n1 * ↩
 ↪ 8, number_of_zero_bits(n1));
    unsigned int n2 {42};
    println("number_of_zero_bits({:0{}b}) = {}", n2, sizeof n2 * ↩
 ↪ 8, number_of_zero_bits(n2));
```

Geeft de volgende uitvoer:

```
number_of_zero_bits(00101010) = 5
number_of_zero_bits(00000000000000000000000000101010) = 29
```

Met de debugger kun je zien dat de eerste aanroep van `number_of_zero_bits` de functie `number_of_zero_bits(uint8_t n)` aanroept en dat de tweede aanroep de functietemplate `number_of_zero_bits<unsigned int>` aanroept.

Net zoals het wenselijk kan zijn om een functietemplate te overloaden voor een bepaald type, kan het ook wenselijk zijn om een aparte versie van een class template te definiëren voor een bepaald type. Je kunt een class template echter *niet* overloaden, maar je kunt een class template *wel* specialiseren[213]. Dit wordt *class template specialisation* genoemd.

In paragraaf 3.3 is de class template `Dozijn` besproken, die een dozijn objecten van een willekeurig type kan opslaan. Stel dat je een dozijn **bool**'s wilt opslaan, dan zou je op het idee kunnen komen om een aparte class voor `Dozijn<bool>` te definiëren, die in plaats van een array van 12 **bool**'s een enkele `uint16_t` gebruikt, waarbij elk bit van de `uint16_t` een **bool** voorstelt. Een **bool** moet door de compiler in een hele byte opgeslagen worden[214], dus een array van 12 **bool**'s zou 12 bytes geheugen gebruiken, terwijl een enkele `uint16_t` maar 2 bytes geheugen gebruikt. Deze optimalisatie is natuurlijk in de praktijk alleen zinvol als er

[213] Een functietemplate kan ook gespecialiseerd worden, maar in de C++ Core Guideline T.144 wordt aangeraden om dit *niet* te doen en in plaats daarvan overloading te gebruiken.

[214] Dit is noodzakelijk omdat je volgens de C++-standaad het adres van een **bool** object op moet kunnen vragen en dus ook een pointer naar een **bool** object moet kunnen maken, en dat kan alleen als een **bool** object minstens 1 byte geheugen gebruikt. De GCC C++-compiler gebruikt 1 byte per **bool** object, maar er zijn ook compilers die 4 bytes per **bool** object gebruiken.

heel veel dozijnen met **bool**'s opgeslagen moeten worden, wat erg onwaarschijnlijk is. Maar een class template specialisatie voor Dozijn<**bool**> is wel een mooi voorbeeld om te laten zien hoe je een class template kunt specialiseren voor een bepaald type.

Een class template specialisatie voor Dozijn<**bool**> kan als volgt gedefinieerd worden, zie Dozijn_template_specialisation.cpp:

```
template<> class Dozijn<bool> {
public:
    void zet_in(int index, bool waarde);
    bool lees_uit(int index) const;
private:
    uint16_t data; // 16 bits, genoeg voor 12 booleans
};

void Dozijn<bool>::zet_in(int index, bool waarde) {
    if (index >= 0 && index < 12) {
        if (waarde) {
            data |= (1 << index);
        } else {
            data &= ~(1 << index);
        }
    }
}

bool Dozijn<bool>::lees_uit(int index) const {
    if (index < 0)
        index = 0;
    if (index > 11)
        index = 11;
    return data & (1 << index);
}
```

Je ziet dat de class template specialisatie van de class template **template**<**typename** T> **class** Dozijn voor het type **bool** wordt gedefinieerd met de syntax **template**<> **class** Dozijn<**bool**>.

De volgende code:

```
    println("sizeof(int) = {}", sizeof(int));
    println("sizeof(Dozijn<int>) = {}", sizeof(Dozijn<int>));
    println("sizeof(bool) = {}", sizeof(bool));
```

```
    println("sizeof(Dozijn<bool>) = {}", sizeof(Dozijn<bool>));
```

Geeft de volgende uitvoer:

```
sizeof(int) = 4
sizeof(Dozijn<int>) = 48
sizeof(bool) = 1
sizeof(Dozijn<bool>) = 2
```

Je ziet dat voor een dozijn **int**'s inderdaad $12 \times 4 = 48$ bytes geheugen gebruikt worden, terwijl voor een dozijn **bool**'s geen $12 \times 1 = 12$ maar slechts 2 bytes geheugen gebruikt worden.

De class template specialisatie van `Dozijn` voor **bool** wordt een volledige (Engels: *full*) class template specialisatie genoemd, omdat er geen template parameters meer zijn. Je kunt ook een class template specialisatie definiëren die nog wel template parameters heeft, maar die voor een deel gespecialiseerd zijn, dit wordt een *partial class template specialisation* genoemd.

In paragraaf 3.4 is de class template **template**<**typename** `T, size_t N`> **class** `Rij` besproken. In een `Rij` kunnen `N` elementen van type `T` opgeslagen worden. Stel dat je, als voorbeeld, een aparte versie voor deze template wilt definiëren voor `N` gelijk aan 1. Deze gespecialiseerde class kan (iets) sneller werken omdat er geen array meer nodig is om de elementen in op te slaan.

Dan kan dat als volgt gedaan worden, zie `Rij_partial_specialisation.cpp`:

```
template<typename T> class Rij<T, 1> {
public:
    void zet_in(size_t index, const T& waarde);
    const T& lees_uit(size_t index) const;
    constexpr size_t aantal_plaatsen();
private:
    T data;
};

template<typename T> void Rij<T, 1>::zet_in(size_t index, const T& ↩
 ↪ waarde) {
    if (index < 1)
        data = waarde;
}

template<typename T> const T& Rij<T, 1>::lees_uit(size_t) const {
```

```
    return data;
}

template<typename T> constexpr size_t Rij<T, 1>::aantal_plaatsen() {
    return 1;
}
```

Je ziet dat de gedeeltelijk gespecialiseerde class template nu nog steeds een template parameter heeft, namelijk `T`, maar dat de template parameter `N` nu 'vastgezet' is op 1.

De interface is bewust gelijk gehouden aan de interface van de oorspronkelijke class template, zodat de code die een `Rij` gebruikt, zoals bijvoorbeeld de functietemplate `naar_string(const Rij<T, N>& r)` niet aangepast hoeft te worden als er een `Rij` met maar 1 element gebruikt wordt.

De volgende code:

```
    Rij<int, 1> kampioen_schappen_Sparta;
    kampioen_schappen_Sparta.zet_in(0, 1959);
    println("Sparta was landskampioen in {}", ↩
↪ kampioen_schappen_Sparta.lees_uit(0));
    println("kampioen_schappen_Sparta = {}", ↩
↪ naar_string(kampioen_schappen_Sparta));
```

Geeft de volgende uitvoer:

```
Sparta was landskampioen in 1959
kampioen_schappen_Sparta = 1959
```

Met gedeeltelijke class template specialisatie kun je bijvoorbeeld ook een aparte versie van een class template definiëren voor pointer types. Stel dat je niet wilt dat de template class `Dozijn` gebruikt mag worden met pointer types, dan zou je de volgende *partial class template specialisation* kunnen definiëren voor pointer types:

```
template<typename T> class Dozijn<T*> {
public:
    Dozijn() = delete;
};
```

In deze template class is de standaard constructor verwijderd, zodat er geen objecten van deze class gemaakt kunnen worden. De volgende code:

```
    Dozijn<int*> d;
```

Geeft bij het compileren de volgende foutmelding:

```
error: use of deleted function 'Dozijn<T*>::Dozijn() [with T = int]'
```

17.6 Zelf een `formatter` definiëren

Om het mogelijk te maken om een object van een UDT af te drukken met de `print` functies of te formatteren met de `format` functie, moet de template `std::formatter` gespecialiseerd, zie paragraaf 17.5, worden voor het betreffende UDT.

Een template specialisatie moet altijd in dezelfde namespace gedefinieerd worden als de oorpronkelijke template. In dit geval is dat de namespace `std`. Een `formatter` is een **`class`** template met twee public members: een memberfunctietemplate[215] `parse` en een memberfunctietemplate `format`. Deze **`class`** kan als **`struct`** gedefinieerd worden, zodat de members standaard **`public`** zijn.

Als voorbeelden worden in deze paragraaf achtereenvolgens formatters besproken voor de classes `Dobbelsteen` (zie paragraaf 20.2), `Breuk` (zie paragraaf 16.6) en `complex<T>`[216].

17.6.1 `formatter<Dobbelsteen>`

In paragraaf 20.2 wordt verwezen naar de class `Dobbelsteen`, zie `Dobbelsteen.cpp`. Het enige wat je van de class `Dobbelsteen` hoeft te weten, is dat deze class een memberfunctie `rol()` heeft die een willekeurig getal vanaf 1 tot en met 6 teruggeeft. Je kunt bijvoorbeeld twintig worpen met een `Dobbelsteen` afdrukken met de volgende code:

```
Dobbelsteen d;
for (int i {0}; i < 19; ++i) {
    print("{}, ", d.rol());
}
println("{}", d.rol());
```

De uitvoer is bijvoorbeeld als volgt:

```
6, 4, 2, 1, 4, 5, 5, 5, 5, 4, 2, 4, 6, 2, 2, 2, 6, 5, 6, 1
```

[215] Een memberfunctie van een class of van een class template kan zelf ook een template zijn. In dat geval wordt deze memberfunctie een *memberfunctietemplate* genoemd.

[216] Zie eventueel `https://en.cppreference.com/w/cpp/numeric/complex`.

Je kunt een formatter voor `Dobbelsteen` definiëren zodat je een `Dobbelsteen` object meteen kunt afdrukken met de `print` functies of formatteren met de `format` functie. Dat maakt het mogelijk om de bovenstaande code te vervangen door de volgende code:

```
    Dobbelsteen d;
    for (int i {0}; i < 9; ++i) {
        print("{}, ", d);
    }
    println("{}", d);
```

In dit geval is het misschien een beetje overdreven om een formatter te definiëren, maar het is een eenvoudig voorbeeld om te laten zien hoe een formatter voor een UDT gedefinieerd kan worden.

Een eenvoudige formatter voor `Dobbelsteen`, die zeker nog voor verbetering vatbaar is, kan als volgt gedefinieerd worden:

```
template<>
struct std::formatter<Dobbelsteen> {
    constexpr auto parse(auto& context) {
        auto iter {context.begin()};
        if (iter!= context.end() && *iter != '}') {
            throw format_error {"Ongeldige format specifier voor ↩
 ↪ Dobbelsteen."};
        }
        return iter;
    }
    auto format(Dobbelsteen& d, auto& context) const {
        return format_to(context.out(), "{}", d.rol());
    }
};
```

De memberfunctietemplate `parse` is verantwoordelijk voor het interpreteren of ontleden (Engels: to parse) van de format specifier, die zich tussen de { en } tekens bevindt in een format string. De memberfunctietemplate `format` is verantwoordelijk voor het formatteren van een `Dobbelsteen` object volgens de format specifier die geïnterpreteerd is door de memberfunctietemplate `parse`. De memberfunctietemplate `parse` moet als **`constexpr`**, zie paragraaf 1.8, gedefinieerd zijn zodat de compiler de format specifier al tijdens het compileren kan interpreteren. Het argument van de memberfunctietemplate `parse` is een instantie van de

template `basic_format_parse_context`[217]. Dit object bevat een iterator, zie hoofdstuk 12, op te vragen met `begin()` die wijst naar het karakter dat volgt op de dubbele puntin de format specifier en een iterator, op te vragen met `end()`, die wijst naar het einde van de format specifier.

Als de dobbelsteen `d` bijvoorbeeld als volgt geprint wordt:

```
println("{:>20}", d);
```

Dan wordt de memberfunctietemplate `parse` van `formatter<Dobbelsteen>` aangeroepen met een `context` object waarvan de iterator `begin()` wijst naar het teken `>` in de format specifier `{:>20}` en waarvan de iterator `end()` wijst

Als de dobbelsteen `d` bijvoorbeeld als volgt geprint wordt:

```
println("{}", d);
```

Dan wijst de iterator `begin()` van het `context` object dat aan de memberfunctietemplate `parse` wordt doorgegeven naar het teken `}` in de format specifier of is `begin()` gelijk aan `end()`. De memberfunctietemplate `parse` moet controleren of de format specifier geldig is en de relevante informatie uit deze format specifier opslaan in data members van de formatter class, zodat deze informatie later gebruikt kan worden door de memberfunctietemplate `format`. De memberfunctietemplate `parse` moet een iterator teruggeven die wijst naar het einde van de geïnterpreteerde format specifier.

In de eenvoudige formatter voor `Dobbelsteen` wordt alleen de lege (`{}`) format specifier ondersteund. Er wordt dus alleen gecontroleerd of de iterator `context.begin()` gelijk is aan `context.end()` of dat `*context.begin()` gelijk is aan `}`. Als dit niet het geval is, dan wordt een `format_error` exception gegooid[218]. Dit is een beetje vreemd omdat de memberfunctietemplate `parse` tijdens compile time wordt uitgevoerd, maar een exception alleen tijdens run-time kan worden gegooid. Maar dit zorgt wel voor een duidelijke foutmelding als er een ongeldige format specifier gebruikt wordt, bijvoorbeeld als de dobbelsteen `d` als volgt geprint wordt:

```
println("{:>20}", d);
```

Wordt door de compiler de volgende foutmelding gegenereerd:

[217] Zie eventueel https://en.cppreference.com/cpp/utility/format/basic_format_parse_context

[218] Zie eventueel https://en.cppreference.com/cpp/utility/format/format_error.

```
error: call to consteval function 'std::basic_format_string<char, ↩
 ↪ Dobbelsteen&>("{:>20}")' is not a constant expression
error: expression '<throw-expression>' is not a constant expression
       throw format_error {"Ongeldige format specifier voor ↩
 ↪ Dobbelsteen."};
```

De memberfunctietemplate `format` moet als **`const`** gedefinieerd zijn omdat deze functie het formatter object niet mag wijzigen. Het eerste argument van de memberfunctietemplate `format` is een referentie naar het `Dobbelsteen` object dat geformatteerd moet worden (`d`). Het tweede argument van de memberfunctietemplate `format` is een instantie van de template `basic_format_context`[219]. Dit object bevat een iterator, op te vragen met `out()` die wijst naar de plek waar de geformateerde uitvoer weggeschreven moet worden. De memberfunctietemplate `format` moet de geformatteerde uitvoer naar deze plek schrijven en een iterator teruggeven die wijst net voorbij het einde van de geformateerde uitvoer. In de eenvoudige formatter voor `Dobbelsteen` wordt de standaard functie `format_to`[220] gebruikt om de returnwaarde van de aanroep `d.rol()` te formatteren en naar `context.out()` te schrijven. Deze functie geeft een iterator terug die net voorbij het einde van de geformatteerde uitvoer wijst. Deze iterator kan dus meteen teruggegeven worden door de memberfunctietemplate `format`.

Natuurlijk wil je de eenvoudige formatter voor `Dobbelsteen` verbeteren zodat ook format specifiers ondersteund worden. Het meest logisch is om dezelfde format specifiers te ondersteunen als de formatter voor **`int`**. Dit blijkt eenvoudig, door het parsen en formatteren uit te besteden (Engels: to delegate) aan de formatter voor **`int`**, zie `format_Dobbelsteen_delegate.cpp`.

```
template<>
struct std::formatter<Dobbelsteen> {
    constexpr auto parse(auto& context) {
        return int_formatter.parse(context);
    }
    auto format(Dobbelsteen& d, auto& context) const {
        return int_formatter.format(d.rol(), context);
    }
private:
    formatter<int> int_formatter;
};
```

[219] Zie eventueel https://en.cppreference.com/cpp/utility/format/basic_format_context

[220] Zie eventueel https://en.cppreference.com/cpp/utility/format/format_to

De code:

```
    Dobbelsteen d;
    println("|{:<8}|", d);
    println("|{:>8}|", d);
    println("|{:^8}|", d);
    println("|{:0>8b}|", d);
```

Geeft nu als uitvoer bijvoorbeeld het volgende:

```
|2       |
|       5|
|   3    |
|00000110|
```

Je ziet dat je de format specifiers die ondersteund worden door de formatter voor **int**[221], zoals alignment en binaire notatie, nu ook kunt gebruiken voor `Dobbelsteen` objecten.

Het is zelfs nog eenvoudiger om een formatter voor `Dobbelsteen` te definiëren als je overerft van de formatter voor **int**, zie `format_Dobbelsteen_derived.cpp`.

```
template<>
struct std::formatter<Dobbelsteen>: public formatter<int> {
    auto format(Dobbelsteen& d, auto& context) const {
        return formatter<int>::format(d.rol(), context);
    }
};
```

Zoals je ziet is het nu niet meer nodig om zelf een memberfunctietemplate `parse` te definiëren, omdat deze al overgeërfd worden van de formatter voor **int**. De memberfunctietemplate `format` van de formatter voor `Dobbelsteen` roept de memberfunctietemplate `format` van de formatter voor **int** aan, waarbij het resultaat van de aanroep `d.rol()` als argument wordt doorgegeven.

Je kunt ook zelf een format specifier bedenken. Omdat je in C++23 ook gebruik kunt maken van UTF-8 tekens kun je een dobbelsteen ook als tekstsymbool (⚀⚁⚂⚃⚄⚅) formatteren.

[221] Zie eventueel https://en.cppreference.com/cpp/utility/format/spec.

Het idee is dat als de format specifier begint met een s, symbolen gebruikt moeten worden in plaats van getallen. De formatter voor `Dobbelsteen` die de dobbelsteem als getal of als symbool kan formatteren kan als volgt gedefinieerd worden, zie `format_Dobbelsteen_symbolen.cpp`:

```
template<>
struct std::formatter<Dobbelsteen> {
    constexpr auto parse(auto& context) {
        auto iter {begin(context)};
        if (*iter == 's') {
            gebruik_symbolen = true;
            context.advance_to(iter+1);
            return string_formatter.parse(context);
        }
        gebruik_symbolen = false;
        return int_formatter.parse(context);
    }
    auto format(Dobbelsteen& d, auto& context) const {
        if (gebruik_symbolen) {
            const array symbols {"⚀", "⚁", "⚂", "⚃", "⚄", "⚅"};
            return string_formatter.format(symbols[d.rol() - 1], ↩
 ↪ context);
        }
        return int_formatter.format(d.rol(), context);
    }
private:
    bool gebruik_symbolen;
    formatter<int> int_formatter;
    formatter<string> string_formatter;
};
```

Als de format specifier begint met een s, dan wordt de rest van de format specifier geparsed door de formatter voor `string` en worden de dobbelsteen symbolen gebruikt. Als de format specifier niet begint met een s, dan wordt de rest van de format specifier geparsed door de formatter voor **`int`** en worden getallen gebruikt.

De code:

```
    Dobbelsteen d;
    println("{:>3}", d);
    println("{:s>3}", d);
    println("{:>03b}", d);
    println("{:s>3?}", d);
```

Geeft nu als uitvoer bijvoorbeeld het volgende:

```
  6
  ⣿
101
"⊡"
```

Je ziet dat je de format specifiers die ondersteund worden door de formatter voor `string`[222], kunt gebruiken als de formatspecifier begint met een `s` voor `Dobbelsteen` objecten. Als de format specifier niet begint met een `s`, dan kun je de format specifiers die ondersteund worden door de formatter voor **`int`** gebruiken.

17.6.2 `formatter<Breuk>`

Voor een UDT die geprint kan worden met een overloaded **`operator`**<< is er een simpele manier om een formatter te definiëren zodat de UDT ook geprint kan worden met de `print`-functies en geformateerd kan worden met de `format`-functies.

In paragraaf 16.11 heb je al gezien hoe de formatter voor `Breuk` gedefinieerd is met behulp van de overloaded **`operator`**<< voor `Breuk`. Het is eenvoudig mogelijk om een formatter voor `Breuk` te definiëren die als friend van de class `Breuk` is gedefinieerd en dus rechtstreeks de private members van `Breuk` kan benaderen, zie `Breuk_format_inherit_from_string.cpp`. Doordat deze formatters overerven van `formatter<string>` kunnen alle format specifiers die bij een `string` gebruikt kunnen worden, nu ook bij een `Breuk` gebruikt worden. Je kunt een breuk dus bijvoorbeeld centraal uitlijnen op een veld van zestien tekens met een underscore als opvulteken, zie `format_Breuk.cpp`:

```
Breuk b(30, 94);
println("Breuk b = {:_^16}", b);
```

Geeft als output:

```
Breuk b = _____15/47______
```

Het is echter niet mogelijk om een `Breuk` in het hexadecimale of binaire talstelsel te printen, omdat deze format specifiers niet gedefinieerd zijn voor een `string`. Om dit wel mogelijk te maken, kun je een formatter voor `Breuk` definiëren die overerft van `formatter<int>`, zie `Breuk_format_inherit_from_int.cpp`:

```
template <>
```

[222] Zie eventueel https://en.cppreference.com/cpp/utility/format/spec.

```
struct std::formatter<Breuk>: public formatter<int> {
    auto format(const Breuk& breuk, auto& context) const {
        context.advance_to(formatter<int>::format(breuk.boven, ↩
 ↪ context));
        context.advance_to(format_to(context.out(), "/"));
        return formatter<int>::format(breuk.onder, context);
    }
};
```

Je kunt een breuk nu dus afdrukken in verschillende talstelsels:

```
    println("Breuk b = {}", b);
    println("Breuk b = {:#0X}", b);
    println("Breuk b = {:#0b}", b);
```

Geeft als output:

```
Breuk b = 15/47
Breuk b = 0XF/0X2F
Breuk b = 0b1111/0b101111
```

Maar als je nu probeert om een breuk centraal uit te lijnen op een veld van zestien tekens met een underscore als opvulteken, dan werkt dat niet zoals verwacht:

```
    println("Breuk b = {:_^16}", b);
```

Geeft als output:

```
Breuk b = _______15_______/_______47_______
```

Maar de output zou moeten zijn:

```
Breuk b = _____15/47______
```

Dus de teller en de noemer worden apart uitgelijnd op een veld van zestien tekens, terwijl de hele breuk als één geheel uitgelijnd zou moeten worden op een veld van zestien tekens.

Om dit probleem op te lossen kun je een formatter definiëren die een format specifier gebruikt die is geïnspireerd op de format specifier voor ranges[223].

[223] Zie https://en.cppreference.com/cpp/utility/format/range_formatter#Range_format_specification.

De format specifier bestaat dan uit twee delen, gescheiden door een dubbele punt, waarbij het eerste deel de format specifier is voor de gehele breuk en het tweede deel de format specifier is voor de teller en noemer.

De volgende code:

```
    Breuk b {30, 94};
    println("Breuk b = {}", b);
    println("Breuk b = {::#0X}", b);
    println("Breuk b = {::#0b}", b);
    println("Breuk b = {:_^16}", b);
    println("Breuk b = {:_^16:#0X}", b);
```

Geeft dan als output:

```
Breuk b = 15/47
Breuk b = 0XF/0X2F
Breuk b = 0b1111/0b101111
Breuk b = _____15/47______
Breuk b = ____0XF/0X2F____
```

Deze formatter heeft de volgende structuur:

```
template<>
struct std::formatter<Breuk> {
    constexpr auto parse(format_parse_context& context) {
    // ...
    }
    auto format(const Breuk& breuk, auto& context) const {
    // ...
    }
private:
    string string_format;
    string int_format;
};
```

De **struct** bevat twee private members, `string_format` en `int_format`, die de format specifiers bevatten voor respectievelijk de gehele breuk en de teller en noemer. In de memberfunctietemplate `parse` worden deze format specifiers uit de format string gehaald en in deze private members opgeslagen, zodat ze later in de memberfunctietemplate `format` gebruikt kunnen worden.

De memberfunctietemplate parse is als volgt geïmplementeerd, zie Breuk_format1.cpp:

```
    constexpr auto parse(format_parse_context& context) {
        int number_of_parsed_colons {0};
        string_format.clear();
        int_format.clear();
        auto iter {context.begin()};
        for (; iter != context.end(); ++iter) {
            if (*iter == '}') { break; }
            if (number_of_parsed_colons == 0) {
                if (*iter == ':') {
                    ++number_of_parsed_colons;
                }
                else {
                    string_format += *iter;
                }
            }
            else {
                int_format += *iter;
            }
        }
        return iter;
    }
```

Beide private variabelen worden leeg gemaakt aan het begin van de memberfunctietemplate parse, vervolgens wordt alles tot de eerste dubbele punt in de format specifier opgeslagen in string_format en alles na de eerste dubbele punt wordt opgeslagen in int_format.

De memberfunctietemplate format is als volgt geïmplementeerd, zie Breuk_format1.cpp:

```
    auto format(const Breuk& breuk, auto& context) const {
        string int_fmt {int_format.empty() ? "{}" : "{:" + ↩
↪ int_format + "}"};
        string teller_text {vformat(int_fmt, ↩
↪ make_format_args(breuk.boven))};
        string noemer_text {vformat(int_fmt, ↩
↪ make_format_args(breuk.onder))};
        string inner_text {teller_text + "/" + noemer_text};
        string string_fmt {string_format.empty() ? "{}" : "{:" + ↩
↪ string_format + "}"};
        string final_text {vformat(string_fmt, ↩
↪ make_format_args(inner_text))};
```

```
        return format_to(context.out(), "{}", final_text);
    }
```

In de memberfunctietemplate `format` wordt eerst de teller en noemer van de breuk geformatteerd en opgeslagen in de strings `teller_text` respectievelijk `noemer_text`, met behulp van de `int_frt` format specifier, die wordt opgebouwd met behulp van de opgeslagen string `int_format`. Omdat de format string *geen* compile time constante is, moet de functie `vformat` gebruikt worden, zie paragraaf 20.11.

Vervolgens worden de strings `teller_text` en `noemer_text` met een schuine streep ertussen samengevoegd tot een string `inner_text`. Deze `inner_text` wordt vervolgens geformatteerd en opgeslagen in de string `final_text`, met behulp van de `string_frt` format specifier, die wordt opgebouwd met behulp van de opgeslagen string `string_format`. Deze string wordt ten slotte naar `context.out()` geschreven.

Bij het gebruik van deze formatter wilde ik bij het printen van een `Breuk` soms de teller anders formatteren dan de noemer. Om dit mogelijk te maken is een nieuwe versie van de formatter voor `Breuk` gedefinieerd. Waarbij de format specifier uit maximaal drie delen bestaat, gescheiden door dubbele punten. Het eerste deel is de format specifier voor de gehele breuk, het tweede deel is de format specifier voor de teller en het derde deel is de format specifier voor de noemer. Als een derde deel niet aanwezig is, dan wordt de tweede deel gebruikt als de format specifier voor zowel de teller als de noemer.

De volgende code:

```
Breuk b {30, 94};
println("Breuk b = {}", b);
println("Breuk b = {::#0X}", b);
println("Breuk b = {::#0b}", b);
println("Breuk b = {:_^16}", b);
println("Breuk b = {:_^16:#0X}", b);
println("Breuk b = {:_^16:#0b:#0x}", b);
Breuk b1 {60, 142};
Breuk b2 {1001, 30011};
println("Breuk b1 = {:>21:>10:<10}", b1);
println("Breuk b2 = {:>21:>10:<10}", b2);
```

Geeft dan als output:

```
Breuk b = 15/47
Breuk b = 0XF/0X2F
```

```
Breuk b = 0b1111/0b101111
Breuk b = _____15/47______
Breuk b = ____0XF/0X2F____
Breuk b = __0b1111/0x2f___
Breuk b1 =          30/71
Breuk b2 =       1001/30011
```

Deze formatter is te vinden in Breuk_format2.cpp.

17.6.3 formatter<std::complex>>

Er is in de C++-standaard geen formatter voor complex<T> gedefinieerd[224]. Wat een beetje vreemd is omdat er wel een overloaded **operator**<<, zie paragraaf 16.11, voor complex<T> is gedefinieerd. Je kunt een complex getal bijvoorbeeld als volgt afdrukken (de uitvoer staat in het commentaar):

```
complex<double> z {3.14E4, 6.28E-4};
cout << "z = " << z << '\n';     // z = (31400,0.000628)
```

Het ligt voor de hand om een formatter voor complex<T> te definiëren die als standaard format dezelfde uitvoer geeft als de overloaded **operator**<<. Dus:

```
complex<double> z {3.14E4, 6.28E-4};
cout << "z = " << z << '\n';     // z = (31400,0.000628)
println("z = {}", z);            // z = (31400,0.000628)
```

In de wiskunde wordt een complex getal geschreven met een plus of min teken als scheiding tussen het reële en imaginaire deel in plaats van een komma en wordt het imaginaire deel aangeduid met een i. In de onderstaande formatter kan dit aangegeven worden door als eerste letter in de format specifier een m te gebruiken. Daarnaast worden de haakjes ook vaak weggelaten. In de onderstaande formatter kan dit aangegeven worden door als eerste letter in de format specifier een i te gebruiken. In de Elektrotechniek wordt vaak de letter j gebruikt in plaats van i, omdat i al gebruikt wordt voor stroom. In de onderstaande formatter kan dit aangegeven worden door als eerste letter in de format specifier een j te gebruiken.

Bij het formatteren van een complex getal wil je het format van het gehele getal kunnen bepalen (zoals bijvoorbeeld de veldbreedte en uitlijning), maar ook het format van het reële en imaginaire deel apart kunnen bepalen (zoals bijvoorbeeld het aantal decimalen). Dit zou

[224] Er zijn wel verschillende voorstellen gedaan, maar die hebben het bljkbaar niet gehaald, zie https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2023/p2930r0.html#complex

op dezelfde manier aangepakt kunnen worden als bij de formatter voor `Breuk`, zie **??**, door een dubbele punt als scheiding tussen deze format specifiers te gebruiken.

Een complex getal kan met behulp van de formatter, gegeven in `format_complex.cpp`, als volgt worden afgedrukt (de uitvoer staat in het commentaar):

```
    complex<double> z {3.14E4, 6.28E-4};
    println("z = {}", z);            // z = (31400,0.000628)
    println("z = {:m}", z);          // z = (31400+0.000628i)
    println("z = {:i}", z);          // z = 31400+0.000628i
    println("z = {:j}", z);          // z = 31400+0.000628j
    println("z = {:j:.4e}", z);      // z = 3.1400e+04+6.2800e-04j
    println("z = {:j:.4E}", z);      // z = 3.1400E+04+6.2800E-04j
    println("z = {:j:f}", z);        // z = 31400.000000+0.000628j
    println("z = {:j:g}", z);        // z = 31400+0.000628j
    println("z = {:j:G}", z);        // z = 31400+0.000628j
    println("z = {:j:.3f}", z);      // z = 31400.000+0.001j
    println("z = {:j_>25:.3f}", z); // z = _________31400.000+0.001j
    println("z = {:j_>25:.4f}", z); // z = _______31400.0000+0.0006j
    println("z = {:j_>25:.5f}", z); // z = _____31400.00000+0.00063j
    println("z = {:j_>25:.6f}", z); // z = ___31400.000000+0.000628j
    println("z = {:j_>25:f}", z);   // z = ___31400.000000+0.000628j
```

Een veel eenvoudigere formatter voor `complex<T>` kun je definiëren door over te erven van `formatter<T>`. Zie `format_complex_simple.cpp`:

```
template<typename T>
struct std::formatter<complex<T>>: public formatter<T> {
    auto format(const complex<T>& z, auto& context) const {
        context.advance_to(formatter<T>::format(z.real(), context));
        if (z.imag() >= 0) {
            context.advance_to(format_to(context.out(), "+"));
        } else {
            context.advance_to(format_to(context.out(), "-"));
        }
        context.advance_to(formatter<T>::format(abs(z.imag()), ↩
 ↪ context));
        return format_to(context.out(), "j");
    }
};
```

Complexe getallen van het type `complex<T>` kunnen met behulp van deze eenvoudige formatter worden afgedrukt met behulp van de format specifiers van het betreffende type `T`[225] (de uitvoer staat in het commentaar):

```
complex<double> z1 {1.2345, -2.9876};
complex<double> z2 {3.14E4, 6.28E-4};
cout << "z1 = " << z1 << '\n'; // z1 = (1.2345,-2.9876)
cout << "z2 = " << z2 << '\n'; // z2 = (31400,0.000628)
println("z1 = {}", z1);        // z1 = 1.2345-2.9876j
println("z2 = {}", z2);        // z2 = 31400+0.000628j
println("z1 = {:.3f}", z1);    // z1 = 1.234-2.988j
println("z2 = {:.3f}", z2);    // z2 = 31400.000+0.001j
println("z1 = {:.3e}", z1);    // z1 = 1.234e+00-2.988e+00j
println("z2 = {:.3e}", z2);    // z2 = 3.140e+04+6.280e-04j
println("z1 = {:.3g}", z1);    // z1 = 1.23-2.99j
println("z2 = {:.3g}", z2);    // z2 = 3.14e+04+0.000628j
// dit werkt misschien anders dan verwacht:
println("z1 = {:>10.3f}", z1); // z1 =      1.234-     2.988j
println("z2 = {:>10.3e}", z2); // z2 =  3.140e+04+ 6.280e-04j
```

Deze eenvoudige formatter is in alle programma's in dit dictaat waarbij gewerkt wordt met complexe getallen gebruikt, maar heeft vergeleken met de meer geavanceerde versie de volgende nadelen:

- de veldbreedte en uitlijning zowel op het reële als op het imaginaire deel wordt toegepast en niet op het gehele complexe getal;
- het complexe getal wordt altijd afgedrukt in het formaat dat in de Elektrotechniek gebruikelijk is, dus met een `j` en zonder haakjes.

Het voordeel van deze eenvoudige formatter is dat je de percisie tijdens het uitvoeren van het programma kunt bepalen:

```
int precision;
print("Geef de gewenste precisie op: ");
cin >> precision;
println("z1 = {:>.{}f}", z1, precision);
println("z2 = {:>.{}f}", z2, precision);
```

Geeft bijvoorbeeld de volgende uitvoer:

[225] De format specifiers voor het type **double** vind je hier: https://en.cppreference.com/cpp/utility/format/spec.

```
Geef de gewenste precisie op: 6
z1 = 1.234500-2.987600j
z2 = 31400.000000+0.000628j
```

In [2] wordt uitgelegd hoe je zelf een formatter kunt definiëren waarbij de veldbreedte tijdens het uitvoeren van het programma kan worden bepaald. Het toepassen van deze techiek op de hierboven beschreven geavanceerde formatter voor `complex<T>` is een mooie oefening voor de lezer 😉.

17.7 Nog meer template details

Over templates valt nog veel meer te vertellen, onder andere:

- Template memberfuncties;
  Een class (die zelf geen template hoeft te zijn) kan memberfuncties hebben die als templates gedefinieerd zijn. Dit wordt dan een memberfunctietemplate genoemd. Een **`virtual`** memberfunctie kan echter geen memberfunctietemplate zijn.
- Default waarden voor template-parameters.
  Net zoals voor gewone functie parameters kun je voor template-parameters ook default waarden (of types) specificeren.
- Templates met een variabel aantal argumenten.
  Een template kan zo worden gedefinieerd dat deze een willekeurig aantal argumenten van willekeurige typen accepteert. Een dergelijk template wordt een “variadic template” genoemd. Zie `variadic_template.cpp`. Een parameter die door . . . wordt gevolgd, wordt een *parameter pack* genoemd.
-

Voor al deze details verwijs ik je naar [1, hoofdstuk 21, 22 en 23], [23, hoofdstuk 23 t/m 29] en [24].

18

Inheritance (de details)

18.1 Overloading en overriding van memberfuncties

Het onderscheid tussen memberfunctie *overloading* en memberfunctie *overriding* is van groot belang. In paragraaf 1.12 heb je gezien dat een functienaam meerdere keren gebruikt (overloaded) kan worden. De compiler selecteert aan de hand van de gebruikte argumenten de juiste functie. Dit maakt deze functies eenvoudiger te gebruiken omdat de gebruiker (de programmeur die deze functies aanroept) slechts één naam hoeft te onthouden. Elke functienaam, dus ook een memberfunctienaam, kan overloaded worden. Een memberfunctie die een andere memberfunctie *overload* heeft dus dezelfde naam. Bij een aanroep van de memberfunctienaam wordt de juiste memberfunctie door de compiler aangeroepen door naar de argumenten bij aanroep te kijken.

Voorbeeld[226] van het gebruik van overloading van memberfuncties (`overload.cpp`):

```
class Class {
public:
    void f() const {
        println("Ik ben f()");
    }
    void f(int i) const { // overload f()
        println("Ik ben f(int i): i = {}", i);
```

[226] In dit voorbeeld (en ook in enkele volgende voorbeelden) heb ik de memberfunctie in de class zelf gedefinieerd. Dit zogenaamd *inline* definiëren (zie ook paragraaf 16.15) heeft als voordeel dat ik iets minder hoef te typen, maar het nadeel is dat de class niet meer afzonderlijk gecompileerd kan worden (zie paragraaf 2.21).

```
    }
};

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 i): i = 3
```

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 (hide.cpp):

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

class Base {
    public:
    void f() const {
        println("Ik ben Base::f()");
    }
};

class Derived: public Base {
    public:
    void f(int i) const[227] { // Verberg f() !! Geen goed idee !!!
        println("Ik ben Derived::f(int i): i = {}", i);
    }
};

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

int main() {
```

```
    Base b;
    Derived d;

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

    d.f();
    //  error: no matching function for call to 'Derived::f()'[228]
    d.Base::f();[229]
    // ...
```

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

```
Ik ben Base::f()
Ik ben Derived::f(int i): i = 3
Ik ben Base::f()
```

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

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

// Code van Dewi
class Derived: public Base {
public:
    void f(double d) const {
        println("Ik ben Derived::f(double d): d = {:.2f}", d);
    }
```

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

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

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

```
};

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

Uitvoer:

```
Ik ben Derived::f(double d): d = 3.00
```

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 {
        println("Ik ben Base::f(int i): i = {}", i);
    }
};
```

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

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

In paragraaf 4.3 heb je gezien dat een *virtual* gedefinieerde memberfunctie in een derived class overridden kan worden. Een memberfunctie die in een derived class een memberfunctie uit de base class *override* moet dezelfde naam en dezelfde parameters hebben[230]. 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 paragraaf 4.2) aangeroepen wordt, dan wordt tijdens het uitvoeren van het programma bepaald

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

naar welk type object de pointer wijst (of de reference verwijst). Pas daarna wordt de in deze class gedefinieerde memberfunctie aangeroepen.

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

- door de compiler aangeroepen door naar het statische type van het object waarop de memberfunctie wordt uitgevoerd te kijken als de memberfuncties niet virtual zijn (er is dan sprake van overloading);
- tijdens run time aangeroepen door naar het dynamische type van het object waarop de memberfunctie wordt uitgevoerd te kijken als de memberfunctie wel virtual zijn (er is dan sprake van overriding).[231]

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

```
class Base {
public:
    void f(int i) const {
        println("Base::f(int) called: i = {}", i);
    }
    virtual void g(int i) const {
        println("Base::g(int) called: i = {}", i);
    }
// ...
};

class Derived: public Base {
public:
    void f(int i) const { // f is overloaded!
        println("Derived::f(int) called: i = {}", i);
    }
    void g(int i) const override { // g is overridden
        println("Derived::g(int) called: i = {}", i);
    }
// ...
```

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

```
};

int main() {
    Base b;
    Derived d;
    Base& rb{d};
    Base* pb{&d};
    b.f(1);
    d.f(2);
    rb.f(3);
    pb->f(4);
    b.g(5);
    d.g(6);
    rb.g(7);
    rb.Base::g(8);
    pb->g(9);
    pb->Base::g(10);
}
```

Uitvoer:

```
Base::f(int) called: i = 1
Derived::f(int) called: i = 2
Base::f(int) called: i = 3
Base::f(int) called: i = 4
Base::g(int) called: i = 5
Derived::g(int) called: i = 6
Derived::g(int) called: i = 7
Base::g(int) called: i = 8
Derived::g(int) called: i = 9
Base::g(int) called: i = 10
```

18.2 Dynamic binding werkt niet in constructors en destructors

Als in een memberfunctie een andere memberfunctie van dezelfde class wordt aangeroepen en als die laatste memberfunctie virtual gedefinieerd is, dan wordt een message verstuurd naar de receiver van de eerste memberfunctie. Bij het zoeken van de method voor deze message wordt eerst gekeken in de class van de receiver en daarna in de base class van de

receiver enz. Dit wordt *dynamic binding* genoemd. Hieronder is een voorbeeld van dynamic binding gegeven (DynamicBinding.cpp):

```
class Fruit {
public:
    virtual void print() {
        println("{}", soort());
    }
private:
    virtual string soort() const {
        return "Fruit";
    }
//  ...
};

class Appel: public Fruit {
private:
    string soort() const override {
        return "Appel";
    }
//  ...
};

class Peer: public Fruit {
private:
    string soort() const override {
        return "Peer";
    }
//  ...
};

int main() {
    Fruit f;
    f.print();
    Appel a;
    a.print();
    Peer p;
    p.print();
//  ...
```

Uitvoer:

```
Fruit
Appel
Peer
```

Laten we eens kijken hoe de regel `a.print();` wordt uitgevoerd. De message `print()` wordt verstuurd naar het object `a`. Er wordt in de class van de receiver (de class `Appel`) gekeken of er een method `print` is gedefinieerd. Dit is niet het geval. Vervolgens wordt in de base class van de class `Appel` (de class `Fruit`) gekeken of er een method `print` is gedefinieerd. Dit is het geval en de betreffende method heeft de juiste parameter. Deze method `Fruit::print` wordt dus uitgevoerd. In deze method wordt de message `soort()` verstuurd naar de receiver. De receiver van de method `print` is het object `a`. Er wordt dus in de class van de receiver (de class `Appel`) gekeken of er een method `soort` is gedefinieerd. Dit is het geval. De method `Appel::soort` wordt dus uitgevoerd en de string `"Appel"` wordt teruggegeven. Deze string wordt vervolgens in de method `Fruit::print` afgedrukt. De method die wordt uitgevoerd bij de ontvangst van een message is dus afhankelijk van de receiver en wordt tijdens run time opgezocht[232], dit wordt *dynamic binding* (ook wel *late binding*) genoemd. Maar dit wist je natuurlijk allemaal al lang (lees paragraaf 2.1 en paragraaf 4.2 nog maar eens door).

In een constructor en in een destructor werkt dynamic binding echter niet! Op het eerste gezicht is dit vreemd maar als je er goed over nadenkt dan blijkt het toch logisch te zijn. Dit wordt uitgelegd aan de hand van het volgende voorbeeld. Als we in het bovenstaande programma de class `Fruit` als volgt wijzigen (`DynamicBindingInConstructorEnDestructor.cpp`):

```
class Fruit {
public:
    Fruit() {
        println("Er is een {} aangemaakt.", soort());
    }
    virtual ~Fruit() {
        println("Er is een {} verwijderd.", soort());
    }
    virtual void print() {
```

[232] De method wordt niet echt opgezocht tijdens run time. Late binding wordt in C++ geïmplementeerd door middel van een zogenoemde virtuele functietabel die uit pointers naar memberfuncties bestaat. Elke class (die virtuele functies bevat) heeft zo'n virtuele functietabel en elk object van zo'n class bevat een (verborgen) pointer naar deze tabel. Via deze tabel met memberfunctiepointers kan late binding op een zeer snelle manier (zonder te zoeken) worden geïmplementeerd.

```
        println("{}", soort());
    }
private:
    virtual string soort() const {
        return "Fruit";
    }
};
```

dan wordt de uitvoer:

```
Er is een Fruit aangemaakt.
Fruit
Er is een Fruit aangemaakt.
Appel
Er is een Fruit aangemaakt.
Peer
Er is een Fruit verwijderd.
Er is een Fruit verwijderd.
Er is een Fruit verwijderd.
```

Deze uitvoer is anders dan je (uitgaande van dynamic binding) zou verwachten. Toch is deze uitvoer goed te verklaren.

Bij het aanmaken van een object a van de class `Appel` door middel van `Appel a;` wordt namelijk eerst de constructor van de base class (de class `Fruit`) aangeroepen. De constructor van de derived class (de class `Appel`) is nog niet uitgevoerd. Je zou dus kunnen zeggen dat de appel nog niet af is, het is al wel een stuk fruit maar het heeft nog niet de specifieke eigenschappen van een appel. Het is dus logisch dat vanuit de constructor `Fruit::soort` wordt aangeroepen en niet `Appel::soort` want het betreffende object is (nog) geen appel. Stel dat de memberfunctie `Appel::soort` zou worden aangeroepen vanuit de constructor `Fruit::Fruit` dan zou die memberfunctie gebruik kunnen maken van nog niet geïnitialiseerde datavelden van de class `Appel`, de constructor van `Appel` is namelijk nog niet uitgevoerd. Het zal duidelijk zijn dat dit ongewenst gedrag kan veroorzaken en daarom vermeden moet worden. Als vanuit de constructor van een class een message wordt verstuurd, wordt geen dynamic binding maar *static binding* toegepast. Dat wil zeggen dat de memberfunctie die in dezelfde class zelf is gedefinieerd wordt aangeroepen.

Bij het verwijderen van een object a van de class `Appel` wordt eerst de destructor van de derived class (de class `Appel`) uitgevoerd. Daarna wordt pas de destructor van de base class (de class `Fruit`) aangeroepen. Je zou dus kunnen zeggen dat de appel dan al (gedeel-

telijk) verwijderd is, het is nog wel een stuk fruit maar het heeft niet meer de specifieke eigenschappen van een appel. Het is dus logisch dat vanuit de destructor `Fruit::soort` wordt aangeroepen en niet `Appel::soort` want het betreffende object is geen appel (meer). Stel dat de memberfunctie `Appel::soort` zou worden aangeroepen vanuit de destructor `Fruit::~Fruit` dan zou die memberfunctie gebruik kunnen maken van al verwijderde datavelden van de class `Appel`, de destructor van `Appel` is namelijk al uitgevoerd. Het zal duidelijk zijn dat dit ongewenst gedrag kan veroorzaken en daarom vermeden moet worden. Als vanuit de destructor van een class een message wordt verstuurd, wordt geen dynamic binding maar *static binding* toegepast. Dat wil zeggen dat de memberfunctie die in dezelfde class zelf is gedefinieerd wordt aangeroepen.

18.3 Expliciet overridden van memberfuncties

In paragraaf 4.3 hebben we je aangeraden om *expliciet* aan te geven dat je een memberfunctie wilt overridden. Dit voorkomt dat je een functie 'per ongeluk' overloadt. Omdat het keyword **override** pas in C++11 is toegevoegd, is het niet verplicht om het keyword **override** te gebruiken. In deze paragraaf ga je zien wat er fout kan gaan als je het keyword **override** *niet* gebruikt.

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

```
class Base {
public:
    void f(int i) const {
        println("Base::f(int) called: i = {}", i);
    }
    virtual void g(int i) const {
        println("Base::g(int) called: i = {}", i);
    }
};
```

Een (niet zo snuggere) programmeur wil nu van deze base class een class afleiden en daarbij zowel de functie `f` als de functie `g` *overridden*:

```
class Derived: public Base {
public:
    void f(int i) const {
        println("Derived::f(int) called: i = {}", i);
    }
    void g(int i) {
```

```
        println("Derived::g(int) called: i = {}", i);
    }
};
```

De programmeur heeft het volgende testprogramma geschreven:

```
int main() {
    Derived d;
    Base* pb {&d};
    pb->f(1);
    pb->g(2);
    // ...
```

De programmeur, die denkt zowel `f` als g overridden te hebben, verwachtte de volgende uitvoer:

```
Derived::f(int) called: i = 1
Derived::g(int) called: i = 2
```

De werkelijke uitvoer is echter:

```
Base::f(int) called: i = 1
Base::g(int) called: i = 2
```

Dit komt omdat de programmeur beide functies `f` en g *overloaded* heeft in plaats van overridden. De functie `f` is in de base class niet virtual gedefinieerd en kan dus niet overridden worden. De functie g in de derived class heeft een andere definitie dan in de basis class (in de basis class is g een **`const`** memberfunctie maar in de derived class ontbreekt het woordje **`const`**). De functie g wordt dus overloaded in plaats van overridden[233].

We kunnen deze problemen eenvoudig voorkomen door expliciet aan te geven dat we een functie willen overridden. Dit kun je doen door het woord **`override`** achter de declaratie van de memberfunctie te plaatsen.

De class `Derived` kan dan als volgt gedefinieerd worden (`explicit_override.cpp`):

```
class Derived: public Base {
public:
    void f(int i) const override {
        println("Derived::f(int) called: i = {}", i);
```

[233] De GCC compiler 15.2.0 geeft wel de volgende waarschuwing: `warning: 'virtual void Base::g(int) const' was hidden.`

```
    }
    void g(int i) override {
        println("Derived::g(int) called: i = {}", i);
    }
};
```

De programmeur geeft hier expliciet aan dat het de bedoeling is om `f` en `g` te overridden. Als deze class vertaald wordt, geeft de compiler een foutmelding:

```
class Derived: public Base {
public:
    void f(int i) const override {
//  error: 'void Derived::f(int) const' marked override, but does ↩
 ↪ not override
        println("Derived::f(int) called: i = {}", i);
    }
    void g(int i) override {
//  error: 'void Derived::g(int)' marked override, but does not ↩
 ↪ override
        println("Derived::g(int) called: i = {}", i);
    }
};
```

Het is dus, om fouten te voorkomen, het beste om elke functie die je override expliciet als zodanig te declareren[234] .

18.4 `final` overriding van memberfuncties

Een functie die in een derived class overridden is kan, in een class die weer afgeleid is van deze class, opnieuw overridden worden. Normaal is dit geen probleem, maar in sommige gevallen is dit ongewenst. Je kunt ook expliciet aangeven dat een functie die overridden is niet verder overridden mag worden. Stel dat we in een bibliotheekapplicatie een basis class `Uitleenbaar_item` hebben met een puur virtuele memberfunctie `id` waarmee het id van het betreffende item kan worden opgevraagd. Van deze basis class zijn verschillende classes zoals `Boek` en `DVD` afgeleid elk met een overridden memberfunctie `id`. In de class `Boek` gebruiken we het ISBN nummer als id. Stel dat we niet willen dat deze implementatie van de functie `id` in classes die afgeleid worden van `Boek` overridden wordt dan kunnen we

[234] Dit wordt ook aangeraden in de C++ Core Guideline C.128.

de functie in de class Boek met het woord **final** markeren. Het is dan niet meer toegestaan om deze functie (verder) te overridden. Zie final_override.cpp:

```
class Uitleenbaar_item {
public:
    virtual string id() const = 0;
};

class DVD: public Uitleenbaar_item {
public:
    string id() const override {
        return barcode;
    }
private:
    string barcode;
};

class Boek: public Uitleenbaar_item {
public:
    string id() const final {
        return ISBN;
    }
private:
    string ISBN;
};
```

Als je nu toch probeert om in een derived class van Boek de memberfunctie id te overridden, dan geeft de compiler een foutmelding:

```
class Reisgids: public Boek {
    string id() const override {
//  error: virtual function 'virtual std::string Reisgids::id() ↩
 ↪ const' overriding final function
    }
};
```

De C++ Core Guideline C.138 raad je aan om het keyword **final** spaarzaam te gebruiken omdat het de uitbreidbaarheid en aanpasbaarheid van je code belemmerd. Soms is dat zinvol en nodig, maar meestal niet.

18.5 `final` overerving

Soms is het niet de bedoeling dat een class als base class gebruikt wordt. Je kunt expliciet aangeven dat je niet kunt overerven van een bepaalde class door in de class declaratie meteen na de naam het woord **`final`** te gebruiken. Stel dat we in een bibliotheekapplicatie een basis class `Uitleenbaar_item` hebben gedefinieerd waar de class `DVD` van afgeleid is. Stel dat we niet willen dat deze class als basis class gebruikt wordt, dan kunnen we de class `DVD` met het woord **`final`** markeren. Het is dan niet meer toegestaan om van deze class te overerven.

Zie final_inheritance.cpp:

```
class Uitleenbaar_item {
public:
    virtual string id() const = 0;
};

class DVD final: public Uitleenbaar_item {
public:
    string id() const override {
        string s;
        // ...
        return s;
    }
};
```

Als je nu toch probeert om van de class `DVD` te overerven, dan geeft de compiler een foutmelding:

```
class DisneyDVD: public DVD {
//  error: cannot derive from 'final' base 'DVD' in derived type ↩
 ↪ 'DisneyDVD'
};
```

De C++ Core Guideline C.138 raad je aan om het keyword **`final`** spaarzaam te gebruiken omdat het de uitbreidbaarheid en aanpasbaarheid van je code belemmerd. Soms is dat zinvol en nodig, maar meestal niet.

18.6 Slicing problem

Een object `d` van een class `Derived`, die public overerft van class `Base`, mag worden toegekend aan een object `b` van de class `Base` met de toekenning `b = d;`. Er geldt immers: een `Derived`

is een Base. Evenzo mag een object b van de class Base worden geïnitialiseerd met een object d van de class Derived door het aanroepen van de copy constructor Base b(d);. Deze toekenning en initialisatie leveren problemen op als het object d van de class Derived meer geheugenruimte inneemt dan een object b van de class Base. Dit is het geval als in class Derived (extra) datavelden zijn opgenomen. Deze extra datavelden kunnen niet aan het object van class Base toegekend worden omdat het object van class Base hier geen ruimte voor heeft. Dit probleem wordt het *slicing problem* genoemd. Het is dus aan te raden om nooit een object van een derived class toe te kennen aan een object van de base class.

Voorbeeld van slicing (slice.cpp):

```
class Mens {
public:
    Mens(const string& n): name{n} {
    }
    string naam() const {
        return name;
    }
    virtual string soort() const {
        return "mens";
    }
    virtual unsigned int salaris() const {
        return 0;
    }
// ...
private:
    const string name;
};

class Docent: public Mens {
public:
    Docent(const string& n, unsigned short s): Mens{n}, sal{s} {
    }
    string soort() const override {
        return "docent";
    }
    unsigned int salaris() const override {
        return sal;
    }
    virtual void verhoog_salaris_met(unsigned short v) {
        sal += v;
```

```
    }
// ...
private:
    unsigned short sal;
};

int main() {
    Docent brojz {"Harry", 30000};
    println("{} {} verdient {}", brojz.soort(), brojz.naam(), ↩
 ↪ brojz.salaris());
    brojz.verhoog_salaris_met(10000);
    println("{} {} verdient {}", brojz.soort(), brojz.naam(), ↩
 ↪ brojz.salaris());

    Mens m {brojz}; // Waar blijft het salaris?
    println("{} {} verdient {}", m.soort(), m.naam(), m.salaris());

    Mens& mr {brojz};
    println("{} {} verdient {}", mr.soort(), mr.naam(), ↩
 ↪ mr.salaris());

    Mens* mp {&brojz};
    println("{} {} verdient {}", mp->soort(), mp->naam(), ↩
 ↪ mp->salaris());
}
```

Uitvoer:

```
docent Harry verdient 30000
docent Harry verdient 40000
mens Harry verdient 0
docent Harry verdient 40000
docent Harry verdient 40000
```

Merk op dat het slicing probleem *niet* kan voorkomen als je overerft van een abstracte base classes, zie paragraaf 4.4. Van een ABC kunnen namelijk geen objecten aangemaakt worden. Probeer dus zoveel mogelijk over te erven van abstracte base classes en maak classes die bedoeld zijn om van over te erven abstract.

18.7 Voorbeeld: opslaan van polymorfe objecten in een vector

In het voorgaande programma heb je gezien dat alleen pointers en references polymorf kunnen zijn. Als we dus polymorfe objecten willen opslaan, kan dit alleen door pointers of references naar deze objecten op te slaan. Het opslaan van references in een `vector` is echter niet mogelijk omdat een reference altijd moet verwijzen naar een object dat al bestaat en bij het aanmaken van de `vector` weten we nog niet welke objecten in de `vector` moeten worden opgeslagen. De enige mogelijkheid is dus het opslaan van polymorfe pointers naar objecten. Hier volgt een voorbeeld van het gebruik van een `vector` en polymorfisme (`Fruitmand.cpp`):

```
class Fruit {
public:
    virtual string soort() const = 0;
// ...
};

class Appel: public Fruit {
public:
    string soort() const override {
        return "Appel";
    }
// ...
};

class Peer: public Fruit {
public:
    string soort() const override {
        return "Peer";
    }
// ...
};

class Fruitmand {
public:
    void voeg_toe(Fruit& p) {
        fp.push_back(&p);
    }
    void print_inhoud() const {
        println("De fruitmand bevat:");
```

```
        for (const Fruit* e: fp)
            println("  {}", e->soort());
    }
private:
    vector<Fruit*> fp;
};

int main() {
    Fruitmand m;
    Appel a1, a2;
    Peer p1;
    m.voeg_toe(a1);
    m.voeg_toe(a2);
    m.voeg_toe(p1);
    m.print_inhoud();
}
```

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.

De abstract base class `Fruit` moet eigenlijk een **`virtual`** destructor hebben, zie paragraaf 5.5.

18.8 Casting en overerving

Als we een reference naar een `Base` class hebben dan kan die refereren naar een `Derived` (van `Base` afgeleide) class, maar niet vice versa. Bijvoorbeeld:

```
class Hond { /* ... */ };
class Sint_bernard: public Hond { /* ... */ };
//  ...
    Sint_bernard boris;
    Hond& h {boris}; // OK: een Sint_bernard is een Hond
    Hond fikkie;
```

```
    Sint_bernard& sb {fikkie}; // NOT OK: een Hond is geen ↩
 ↪ Sint_bernard
//  error: cannot bind non-const lvalue reference of type ↩
 ↪ 'Sint_bernard&' to an rvalue of type 'Sint_bernard'
```

Als we een pointer naar een Base class hebben, mogen we een pointer naar een Derived (van Base afgeleide) class toekennen aan deze Base class pointer, maar niet vice versa. Bijvoorbeeld:

```
class Hond { /* ... */ };
class Sint_bernard: public Hond { /* ... */ };
//  ...
    Sint_bernard boris;
    Hond* hp {&boris}; // OK: een Sint_bernard is een Hond
    Sint_bernard* sbp {&fikkie}; // NOT OK: een Hond is geen ↩
 ↪ Sint_bernard
//  error: invalid conversion from 'Hond*' to 'Sint_bernard*'
```

Het omzetten van een Hond& naar een Sint_bernard& of van een Hond* naar een Sint_bernard* kan soms toch nodig zijn. We noemen dit een *downcast* omdat we afdalen in de class hiërarchie. Voorbeeld (casting4.cpp):

```
class Hond {
public:
    virtual ~Hond() = default;
    virtual void blaf() const {
        println("Blaf.");
    }
//  ...
};

class Sint_bernard: public Hond {
public:
    explicit Sint_bernard(int w = 10): whisky{w} {
    }
    void blaf() const override {
        println("Woef!");
    }
    int geef_drank() {
        println("Geeft drank.");
        int w {whisky};
        whisky = 0;
```

```
        return w;
    };
//  ...
private:
    int whisky;
};

void geef_hulp(Hond& h) {
    h->blaf();
//  println("{}", h.geef_drank());
//  Error: 'class Hond' has no member named 'geef_drank'
//  error: 'class Hond' has no member named 'geef_drank'
//  We kunnen een cast gebruiken maar dat geeft foutieve uitvoer ↩
 ↪ als h niet naar een Sint_bernard refereert.
    println("{} liter.", static_cast<Sint_bernard&>(h).geef_drank());
}
```

In dit geval is een **static_cast** gebruikt om een downcast te maken. Als je de functie geef_hulp aanroept met een Sint_bernard, dan gaat alles goed.[235]

```
    Sint_bernard boris;
    geef_hulp(boris);
```

Uitvoer:

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

Als je de functie geef_hulp echter aanroept met een Hond als argument, dan geeft het programma onvoorspelbare resultaten (en/of loopt het vast):

```
    Hond fikkie;
    geef_hulp(fikkie);
```

Uitvoer:

```
Blaf.
Geeft drank.
4328928 liter.
```

[235] Als de functie geef_hulp alleen maar aangeroepen wordt met een Sint_bernard& als argument, is het natuurlijk veel slimmer om deze functie te definiëren als: **void** geef_hulp(Sint_bernard& sbp). De downcast is dan niet meer nodig!

Een **static_cast** is dus alleen maar geschikt als downcast als je zeker weet dat de cast geldig is. In het bovenstaande programma zou je de mogelijkheid willen hebben om te kijken of een downcast mogelijk is. Dit kan met een zogenoemde **dynamic_cast**. Zie casting5.cpp:

```
void geef_hulp(Hond& h) {
    h.blaf();
    try {
        println("{} liter.", ↩
 ↪ dynamic_cast<Sint_bernard&>(h).geef_drank());
    } catch (const bad_cast& e) {
    }
}
```

Je kunt de functie geef_hulp nu veilig aanroepen zowel met een Sint_bernard als met een Hond als argument.

```
int main() {
    Sint_bernard boris;
    geef_hulp(boris);

    Hond fikkie;
    geef_hulp(fikkie);
}
```

Uitvoer:

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

Een **dynamic_cast** is alleen mogelijk met polymorfe pointers en polymorfe references. Als een **dynamic_cast** van een pointer mislukt, geeft de cast een **nullptr** terug. Bij een **dynamic_cast** van een reference is dit niet mogelijk (omdat een nul reference niet bestaat). Als een **dynamic_cast** van een reference mislukt, wordt de standaard exception **bad_cast** gegooid.

Een versie van geef_hulp die werkt met een Hond* in plaats van met een Hond& kun je dus als volgt implementeren (casting6.cpp):

```
void geef_hulp(Hond* hp) {
    hp->blaf();
```

```
    Sint_bernard* sbp {dynamic_cast<Sint_bernard*>(hp)};
    if (sbp != nullptr) {
        println("{} liter.", sbp->geef_drank());
    }
}
```

Deze functie kun je als volgt aanroepen:

```
    Sint_bernard boris;
    geef_hulp(&boris);
    Hond fikkie;
    geef_hulp(&fikkie);
```

Uitvoer:

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

18.9 Dynamic casting en RTTI

Om tijdens run time te kunnen controleren of een **dynamic_cast** mogelijk is moet informatie over het type tijdens run time beschikbaar zijn. Dit wordt *RTTI* = *R*un *T*ime *T*ype *I*nformation genoemd. In C++ hebben alleen classes met één of meer virtuele functies RTTI. Dat is logisch omdat polymorfisme ontstaat door het gebruik van virtual memberfuncties. RTTI maakt dus het gebruik van **dynamic_cast** mogelijk. Je kunt ook de RTTI gegevens behorende bij een object rechtstreeks opvragen. Deze gegevens zijn opgeslagen in een object van de class **type_info**. Deze class heeft een vraagfunctie `name()` waarmee de naam van de class opgevraagd kan worden. Om het **type_info** object van een (ander) object op te vragen moet je het C++ keyword **typeid** gebruiken. Zie `rtti.cpp`:

```
void print_ras(Hond& h) {
    println("{}", typeid(h).name());
}
```

Deze functie kun je als volgt aanroepen:

```
    Sint_bernard boris;
    print_ras(boris);
    Hond fikkie;
```

```
    print_ras(fikkie);
```

De uitvoer is compiler afhankelijk. De GCC C++-compiler produceert de volgende uitvoer:

```
12Sint_bernard
4Hond
```

18.10 Maak geen misbruik van RTTI en dynamic_cast

Het verkeerd gebruik van **dynamic_cast** en RTTI kan leiden tot code die *niet* uitbreidbaar en aanpasbaar is! Je zou bijvoorbeeld op het idee kunnen komen om een functie `blaf()` als volgt te implementeren (casting7.cpp):

```
class Hond {
public:
    virtual ~Hond() = default;
//  ...
};

Hond::~Hond() {
}

class Sint_bernard: public Hond {
//  ...
};

class Tekkel: public Hond {
//  ...
};

//  Deze code is NIET uitbreidbaar!
//  ******* DON'T DO THIS IN YOUR CODE *******
//  blaf moet als virtual memberfunctie geimplementeerd worden!

void blaf(const Hond* hp) {
    if (dynamic_cast<const Sint_bernard*>(hp) != 0)
        println("Woef!");
    else if (dynamic_cast<const Tekkel*>(hp) != 0)
        println("Kef kef!");
    else
        println("Blaf.");
```

```
}
```

Bedenk zelf wat er moet gebeuren als je de class `Duitse_herder` wilt toevoegen. In plaats van een losse functie die expliciet gebruikt maakt van dynamic binding (**`dynamic_cast`**) met behulp van RTTI moet je een **`virtual`** memberfunctie gebruiken. Deze **`virtual`** memberfunctie maakt impliciet gebruik van dynamic binding. Alleen in uitzonderingsgevallen (er is maar één soort hond die whisky bij zich heeft) moet je **`dynamic_cast`** gebruiken. Alle honden kunnen blaffen (alleen niet allemaal op dezelfde manier) dus is het gebruik van **`dynamic_cast`** om blaf te implementeren niet juist. In dit geval moet je een **`virtual`** memberfunctie gebruiken.

18.11 Nog meer 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 en meer verwijs ik je naar [1, hoofdstuk 13 tot en met 17] en [23, hoofdstuk 20 en 21].

19

Dynamic memory allocation en destructors (de details)

In hoofdstuk 5 heb je geleerd hoe je dynamisch (tijdens run time) geheugen kan aanvragen en weer vrij kan geven. In C++ doe je dit als programmeur meestal niet zelf, maar maak je gebruik van classes uit de standaard library die dit voor je doen. In dit hoofdstuk wordt bespreken hoe je zelf een class kunt ontwikkelen die gebruikt maakt van dynamic memory allocation waarbij je dit kunt verbergen voor de programmeur die gebruik maakt van deze class. In dit hoofdstuk wordt dynamic memory allocation toegepast bij het maken van een UDT `Array`. Ook laat ik in dit hoofdstuk zien hoe je er voor kunt zorgen dat je deze UDT:

- met een range-based `for` kunt doorlopen;
- met alle algoritmen uit de `ranges` library kunt gebruiken;
- met `print` en `println` kunt afdrukken;
- en met een initialisatielijst kunt initialiseren.

19.1 Voorbeeld class `Array`

Het in C ingebouwde array type heeft een aantal nadelen en beperkingen. De belangrijkste daarvan zijn:

- De grootte van de array moet bij het compileren van het programma bekend zijn. In de praktijk komt het vaak voor dat pas bij het uitvoeren van het programma bepaald kan worden hoe groot de array moet zijn.

- Er vindt bij het gebruiken van de array geen controle plaats op de gebruikte index. Als je element N benadert uit een array die $N-1$ elementen heeft, krijg je geen foutmelding maar wordt de geheugenplaats achter het einde van de array benaderd. Dit is vaak de oorzaak van fouten die lang onopgemerkt kunnen blijven en dan (volgens de wet van bedrog[236] 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 ga je zien hoe ik zelf een eigen array type genaamd `Array` heb gedefinieerd[237] waarbij:

- de grootte pas tijdens het uitvoeren van het programma bepaald kan worden;
- bij het indexeren de index wordt gecontroleerd en een exception[238] `out_of_range` wordt gegooid als de index zich buiten de grenzen van de `Array` bevindt.

De door de compiler gegenereerde copy constructor, destructor en **`operator`**= blijken voor de class `Array` niet correct te werken. Ik heb daarom voor deze class zelf een copy constructor, destructor en **`operator`**= gedefinieerd. Na het voorbeeldprogramma `Array.cpp` worden de belangrijkste aspecten toegelicht en worden verwijzingen naar verdere literatuur gegeven.

```
import std;
using namespace std;

class Array {
public:
    using size_type = size_t;
    explicit Array(size_type s);
    Array(const Array& r);
    Array& operator=(const Array& r);
    ~Array();
    int& operator[](size_type index);
    const int& operator[](size_type index) const[239];
```

[236] https://nl.wikipedia.org/wiki/Wet_van_bedrog.

[237] In de standaard C++ library is ook een type `std::array` opgenomen, zie paragraaf 3.6. Dit type is enigszins vergelijkbaar met mijn zelfgemaakte type `Array`. Het grootste verschil is dat de size van de array bij de `std::array` als template argument wordt meegegeven en bij mijn `Array` als constructor argument wordt meegegeven. De grootte (Engels: size) van een `std::array` moet dus tijdens het compileren van het programma (compile time) bekend zijn, maar de size van een `Array` kan tijdens het uitvoeren van het programma (run time) bepaald worden. Wat dat betreft, lijkt mijn `Array` dus meer op een `std::vector` die in paragraaf 3.7 is behandeld. Het verschil tussen een `std::vector` en mijn `Array` is dat een `std::vector` nadat hij is aangemaakt nog kan groeien en krimpen. Mijn `Array` kan dat niet.

[238] Zie hoofdstuk 6.

```
    size_type length() const;
    bool operator==(const Array& r) const;
private:
    void check_index(size_type index) const;
    size_type size;
    int* data;
friend ostream& operator<<(ostream& out, const Array& r);
};

Array::Array(size_type s): size{s}, data{new int[s]} {
    for (size_type i {0}; i < size; ++i)
        data[i] = 0;
}

Array::Array(const Array& r): size{r.size}, data{new int[r.size]} {
    for (size_type i {0}; i < size; ++i)
        data[i] = r.data[i];
}

Array& Array::operator=(const Array& r) {
    Array t {r};
    swap(data, t.data);
    swap(size, t.size);
    return *this;
}

Array::~Array() {
    delete[] data;
}
```

[239] Het overloaden van de `operator[]` is noodzakelijk omdat ik vanuit deze operator een reference terug wil geven zodat met deze reference in het `Array` object geschreven kan worden bijvoorbeeld `v[12] = 144;`. Als ik deze operator als `const` zou hebben gedefinieerd (wat in eerste instantie logisch lijkt, de `operator[]` verandert immers niets in het `Array` object), dan zou deze operator ook gebruikt kunnen worden voor een `const Array` object. Met de teruggegeven reference kun je dan in een `const Array` object schrijven en dat is natuurlijk niet de bedoeling. Om deze reden heb ik de `operator[]` niet als `const` gedefinieerd. Dit heeft tot gevolg dat de `operator[]` niet meer gebruikt kan worden voor `const Array` objecten. Dit is weer teveel van het goede want nu kun je ook niet meer lezen met behulp van `operator[]` uit een `const Array` object bijvoorbeeld `i = v[12];`. Om het lezen uit een `const Array` object toch weer mogelijk te maken heb ik naast de non-const `operator[]` nog een `const operator[]` gedefinieerd. Deze `const operator[]` geeft een `const` reference terug en zoals je weet kan een `const` reference alleen gebruikt worden om te lezen. (Als je deze voetnoot na één keer lezen begrijpt, is er waarschijnlijk iets niet helemaal in orde). Zie `Array_operator[].cpp` voor een voorbeeldprogramma.

```
void Array::check_index(size_type index) const {
    if (index >= size) {
        throw std::out_of_range("Gebruikte index (" + ↩
 ↪ to_string(index) + ") ligt buiten de grenzen van de Array ↩
 ↪ (0.." + to_string(size - 1) + ")");
    }
}

int& Array::operator[](size_type index) {
    check_index(index);
    return data[index];
}

const int& Array::operator[](size_type index) const {
    check_index(index);
    return data[index];
}

Array::size_type Array::length() const {
    return size;
}

bool Array::operator==(const Array& r) const {
     if (size != r.size)
        return false;
    for (size_type i {0}; i < size; ++i)
        if (data[i] != r.data[i])
            return false;
    return true;
}

ostream& operator<<(ostream& out, const Array& r) {
    for (Array::size_type i {0}; i < r.size; ++i) {
        out << r.data[i];
        if (i != r.size - 1)
            out << ", ";
}
    return out;
}

template<>
```

```
struct std::formatter<Array>: public formatter<string> {
    auto format(const Array& a, auto& context) const {
        ostringstream ss;
        ss << a;
        return formatter<string>::format(ss.str(), context);
    }
};

int main() {
    print("Hoeveel elementen moet de Array bevatten? ");
    Array::size_type i;
    cin >> i;
    Array a {i};
    println("a = {}", a);
    for (Array::size_type j {0}; j < a.length(); ++j)
        a[j] = j * j; // vul a met kwadraten
    println("a = {}", a);
    Array b {a};
    println("b = {}", b);
    try {
        println("a[12] = {}", a[12]);
        println("b[12] = {}", b[12]);
    } catch (const std::out_of_range& e) {
        println("Exception: {}", e.what());
    }
    try {
        a[0] = 4;
        println("a[0] = {}", a[0]);
    } catch (const std::out_of_range& e) {
        println("Exception: {}", e.what());
    }
    println("a = {}", a);
    println("b = {}", b);

    if (a == b)
        println("a is nu gelijk aan b.");
    else
        println("a is nu ongelijk aan b.");

    b = a;
    println("b = a is uitgevoerd.");
```

```
    println("a = {}", a);
    println("b = {}", b);

    if (a != b)
        println("a is nu ongelijk aan b.");
    else
        println("a is nu gelijk aan b.");
}
```

Je ziet dat de typenaam `size_type` in de class is gelijk gesteld is aan het standaard type `size_t` met behulp van een **`using`**-declaratie. Het type `size_t` wordt in C en C++ gebruikt om de grootte van een object mee aan te geven en een variabele van dit type kan alleen positief zijn. Dit is een logische keuze omdat het aantal elementen in een `Array` ook alleen maar positief kan zijn. Het type `size_type` wordt gebruikt voor de private variabele waarin de `size` van de `Array` wordt opgeslagen. Ook de parameter `s` van de constructor en de parameter `index` van de **`operator`**`[]`-memberfuncties zijn ook van dit type. Deze `index` kan ook alleen maar positief zijn.

19.2 `explicit` constructor

De constructor `Array(size_type)` is **`explicit`** gedeclareerd om te voorkomen dat de compiler deze constructor gebruikt om een variabele van het type `size_type` oftewel `size_t` automatisch om te zetten naar een `Array`. Zie paragraaf 2.6.

19.3 Copy constructor en default copy constructor

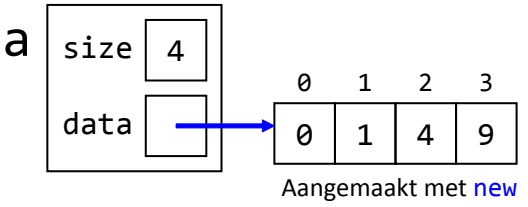
Een copy constructor wordt gebruikt als een object gekopieerd moet worden. Dit is het geval als:

- een object geïnitialiseerd wordt met een object van dezelfde class;
- een object als argument wordt doorgegeven aan een value parameter van een functie;
- een object als waarde wordt teruggegeven vanuit een functie.

De compiler genereert als de programmeur geen copy constructor definieert zelf een default copy constructor. 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 bespreek eerst wat het probleem is bij de door de compiler gedefinieerde default copy constructor en **`operator`**=. Daarna bespreek ik hoe we zelf een copy constructor en een operator kunnen declareren en implementeren. Een `Array` a met vier elementen gevuld met kwadraten is schematisch weergegeven in figuur 19.1.



Figuur 19.1: Object a van de class `Array`.

De door de compiler gegenereerde copy constructor voert een memberwise copy uit. De datavelden `size` en `data` worden dus gekopieerd. Als je de `Array` a naar de `Array` b kopieert door middel van het statement `Array b(a);`[240], ontstaat de in figuur 19.2 weergegeven situatie.

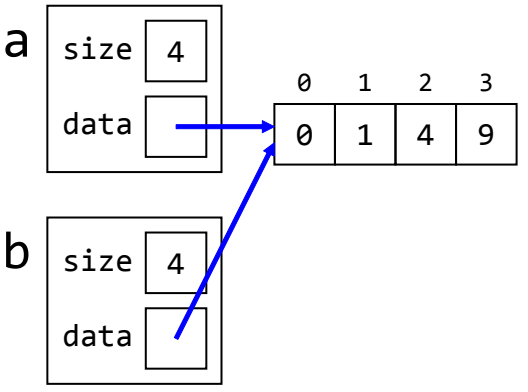
Dit is niet goed omdat als je nu de kopie wijzigt (bijvoorbeeld `b[2] = 8;`) dan is ook het origineel (`a[2]`) gewijzigd en dat is natuurlijk niet de bedoeling.

De gewenste situatie na het kopiëren van de `Array` a naar de `Array` b is gegeven in figuur 19.3. Om deze situatie te bereiken moeten je zelf de copy constructor van de class `Array` declareren: `Array::Array(const Array&);`

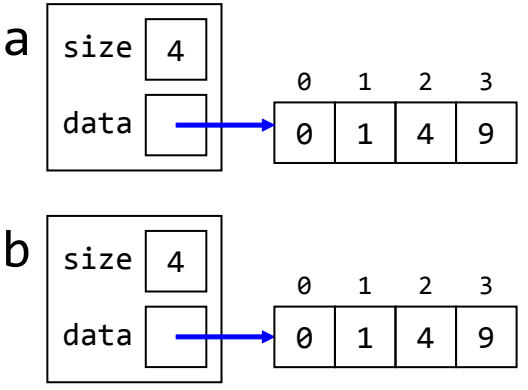
De copy constructor kan eenvoudig als volgt gedefinieerd worden:

```
Array::Array(const Array& r): size{r.size}, data{new int[r.size]} {
    for (size_type i {0}; i < size; ++i)
        data[i] = r.data[i];
```

[240] 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 assignment wordt gedaan terwijl in werkelijkheid de copy constructor wordt aangeroepen. Om deze reden heb ik je in paragraaf 1.4 al aangeraden om bij initialisatie altijd de notatie `Array w{v};` te gebruiken.



Figuur 19.2: Het effect van de default copy constructor.



Figuur 19.3: Het gewenste effect van de copy constructor.

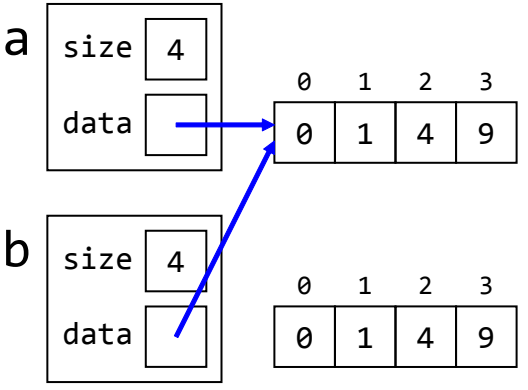
```
}
```

De parameter van de copy constructor moet een `Array&` zijn en kan geen `Array` zijn. Als je namelijk een `Array` als (call by value) parameter gebruikt, moet een kopietje worden gemaakt bij aanroep van de copy constructor, maar daarvoor moet de copy constructor worden aangeroepen, waarvoor een kopietje moet worden gemaakt, maar daarvoor moet de copy constructor . . . enz.

19.4 Overloading `operator=`

Voor de door de compiler gegenereerde assignment operator geldt ongeveer hetzelfde verhaal als voor de door de compiler gegenereerde copy constructor. Na het statement `b = a;`

ontstaat de situatie zoals gegeven in figuur 19.4. Je ziet dat de door de compiler gegenereerde assignment operator niet alleen onjuist werkt maar bovendien een memory leak veroorzaakt.



Figuur 19.4: Het effect van de default assignment operator.

Als je de assignment operator voor de class `Array` zelf wilt definiëren, moet je de memberfunctie **`operator`**= declareren.

Deze memberfunctie kun je dan als volgt implementeren:

```
Array& Array::operator=(const Array& r) {
    Array t {r};
    swap(data, t.data);
    swap(size, t.size);
    return *this;
}
```

Voor het return type van **`operator`**= heb ik `Array&` gebruikt. Dit zorgt ervoor dat assignment operatoren achter elkaar 'geregen' kunnen worden, bijvoorbeeld: `c = b = a` zie paragraaf 2.19. De implementatie van de **`operator`**= moet de `Array r` toekennen aan de receiver. In de implementatie van de **`operator`**= is gebruik gemaakt van de standaard functie `swap` die de inhoud van de twee meegegeven argumenten verwisseld[241].

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.

[241] Zie eventueel https://en.cppreference.com/w/cpp/algorithm/swap.

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) {
    swap(data, r.data);
    swap(size, r.size);
    return *this;
}
```

Antwoord:

Ja! De copy constructor wordt nu impliciet aangeroepen bij het doorgeven van het bij aanroep gebruikte argument aan de parameter `r` (call by value).

19.5 Wanneer moet je zelf een destructor, copy constructor en operator= definiëren?

Een class moet een *zelf* gedefinieerde copy constructor, **operator**= en destructor bevatten als:

- die class een pointer bevat en;
- als bij het kopiëren van een object van de class niet de pointer, maar de data waar de pointer naar wijst moet worden gekopieerd en;
- als bij een toekenning aan een object van de class niet de pointer, maar de data waar de pointer naar wijst moet worden toegekend en;
- als bij het vrijgeven van de geheugenruimte van een object van de class niet de pointer, maar de data waar de pointer naar wijst moet worden vrijgegeven (door middel van **delete**).

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

In de vorige paragraaf heb ik het zelfgedefinieerde type `Array` besproken. Dit type heeft een aantal voordelen ten opzichte van het ingebouwde array type. De belangrijkste zijn dat het aantal elementen pas tijdens het uitvoeren van het programma bepaald wordt en dat bij het gebruik van de **`operator[]`** de index wordt gecontroleerd. Het zelfgedefinieerde type `Array` heeft ten opzichte van het ingebouwde array type als nadeel dat de elementen alleen maar van het type **`int`** kunnen zijn. Als je een `Array` met elementen van het type **`double`** nodig hebt, kun je natuurlijk gaan kopiëren, plakken, zoeken en vervangen maar daar zitten weer de bekende nadelen aan (elke keer als je code kopieert wordt de onderhoudbaarheid van die code slechter). Als je verschillende versies van `Array` met de hand genereert, moet je bovendien elke versie een andere naam geven omdat een class naam uniek moet zijn. In plaats daarvan kun je ook het template mechanisme gebruiken om een `Array` met elementen van het type `T` te definiëren, waarbij het type `T` pas bij het gebruik van de class template `Array` wordt bepaald. Bij het gebruik van de class template `Array` kan de compiler niet zelf bepalen wat het type `T` moet zijn. Vandaar dat je dit bij het gebruik van de class template `Array` zelf moet specificeren.

Bijvoorbeeld:

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

19.6 Voorbeeld class template `Array`

Zie Array_template.cpp:

```
import std;
using namespace std;

template <typename T> class Array;
template <typename T> ostream& operator<<(ostream& out, const Array<T>& r);

template <typename T> class Array {
public:
    using size_type = size_t;
    explicit Array(size_type s);
    Array(const Array<T>& v);
    Array<T>& operator=(const Array<T>& r);
    ~Array();
    T& operator[](size_type index);
```

```
    const T& operator[](size_type index) const;
    size_type length() const;
    bool operator==(const Array<T>& r) const;
private:
    void check_index(size_type index) const;
    size_type size;
    T* data;
friend ostream& operator<< <>(ostream& out, const Array<T>& r);
};

template <typename T> Array<T>::Array(size_type s): size{s}, ↩
 ↪ data{new T[s]} {
    for (size_type i {0}; i < size; ++i)
        data[i] = T{};
}

template <typename T> Array<T>::Array(const Array<T>& r): ↩
 ↪ size{r.size}, data{new T[r.size]} {
// ... enz. ...
    print("Hoeveel elementen moet de Array bevatten? ");
    Array<double>::size_type i;
    cin >> i;
    Array<double> v {i};
    println("v = {}", v);
    for (Array<double>::size_type j {0}; j < v.length(); ++j)
        v[j] = sqrt(j); // Vul v met wortels
    println("v = {}", v);
    if (v != v)
        println("Dit kan niet waar zijn!");
    try {
        println("v[12] = {}", v[12]);
    } catch (const out_of_range& e) {
        println("Exception: {}", e.what());
    }
    Array<int> w {i};
    println("w = {}", w);
    for (Array<int>::size_type t {0}; t < w.length(); ++t)
        w[t] = t * t;   // Vul w met kwadraten
    println("w = {}", w);
    try {
        println("w[12] = {}", w[12]);
    } catch (const out_of_range& e) {
```

```
        println("Exception: {}", e.what());
        t[i].join();
```

Een template kan ook meerdere parameters hebben. Een template-parameter kan in plaats van een typename parameter ook een normale parameter zijn. Zo zou je ook de volgende class template kunnen definiëren:

```
template <typename T, size_t size> class Fixed_array[242] {
    // ...
private:
    T data[size];
}
```

Deze class template kan dan als volgt gebruikt worden:

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

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

- De `size` moet bij de laatste template tijdens het compileren bekend zijn. De compiler genereert (instantieert) namelijk de benodigde versie van de class template en vult daarbij de opgegeven template argumenten in.
- De compiler genereert een nieuwe versie van de class telkens als deze class gebruikt wordt met andere template argumenten. Dit betekent dat `Fixed_array<int, 3> a` en `Fixed_array<int, 4> b` verschillende types zijn. Dus expressies zoals `a != b` en `a = b` enz. zijn dan niet toegestaan. Telkens als je een `Fixed_array` met een andere `size` definieert, wordt er weer een nieuw type met bijbehorende machinecode voor alle memberfuncties gegenereerd[243]. Bij de class template `Array` wordt maar één versie aangemaakt als de variabelen `Array<int> a(3)` en `Array<int> b(4)` gedefinieerd worden. De expressies `a != b` en `a = b` enz. zijn dan wel toegestaan.

[242] Deze `Fixed_array` lijkt erg veel op de in de standaard opgenomen `std::array`, zie paragraaf 3.6.

[243] Dit is niet de waarheid. Een memberfunctie van een class template wordt niet gegenereerd als de template geïnstantieerd wordt, maar pas als de compiler daadwerkelijk een aanroep naar de betreffende memberfunctie moet vertalen. Dit is hier echter niet van belang.

19.7 Ondersteuning range-based for voor class template Array

De in paragraaf 19.6 gegeven class template Array kan niet met behulp van een range-based **for** (zie paragraaf 1.7) doorlopen worden. Als je dit toch probeert, dan krijg je de volgende foutmeldingen:

```
    for (auto e: v) {
    //  error:  'begin' was not declared in this scope; did you ↩
 ↪ mean 'std::begin'?
    //  error:  'end' was not declared in this scope; did you mean ↩
 ↪ 'std::end'?
```

Als we dit willen laten werken moeten we de functies `begin`, `cbegin`, `end` en `cend` voor de class template `Array` definiëren. De functies `begin` en `end` moet overloaded worden voor read-only (**const**) objecten van de class `Array` en voor non-**const** objecten van de class `Array`. De functies `cbegin` en `cend` moeten enkel gedefinieerd worden voor read-only (**const**) objecten van de class `Array`. De functies `begin` en `cbegin` moeten een pointer [244] teruggeven naar het eerste element van de `Array`. De functies `end` en `cend` moeten een pointer[244] teruggeven net na het laatste element van de `Array`.

Zie `Array_template_rbf.cpp`:

```
template <typename T> class Array {
public:
    using size_type = size_t;
    explicit Array(size_type s);
// ... enz. ...
    T* begin();
    const T* begin() const;
    const T* cbegin() const;
    T* end();
    const T* end() const;
    const T* cend() const;
// ... enz. ...
};

int main() {
    Array v {14, 12, 13, 11};
    println("v = {}", v);
```

[244] In het algemeen moeten deze functies een zogenoemde iterator teruggeven, zie hoofdstuk 12.

```
    ranges::sort(v);
    println("na sorteren: v = {}", v);
    Array r {11, 12, 13, 14};
    if (v != r)
        println("Sorteren is niet gelukt!");
    println("v (hexadecimal) = {::#X}", v);
    Array<double> w(10);
    int i {1};
    for (auto& e: w) {
        e = 1.0/i++;
    }
    println("w = {::.3f}", w);
}
```

19.8 Ondersteuning initialisatielijst voor class template `Array`

De in paragraaf 19.6 gegeven class template Array kan niet met behulp van een initialisatielijst geïnitialiseerd worden. Als je dit toch probeert, dan krijg je een foutmelding:

```
     Array v {1, 2, 3};
    //  error: no matching function for call to ↩
 ↪ 'Array<int>::Array(<brace-enclosed initializer list>)'
```

Als we dit willen laten werken moeten we een extra constructor toevoegen aan de class `Array`.

Zie `Array_template_rbf.cpp`:

```
template <typename T> class Array {
public:
    using size_type = size_t;
    explicit Array(size_type s);
    explicit Array(initializer_list<T> list);
// ... enz. ...
};

template <typename T> Array<T>::Array(initializer_list<T> list): ↩
 ↪ size{list.size()}, data{new T[size]} {
    auto list_iter{list.begin()};
    for (size_type i {0}; i < size; ++i) {
        data[i] = *list_iter++;
    }
```

```
}
```

Of nog eenvoudiger:

```
template <typename T> Array<T>::Array(initializer_list<T> list): ↩
 ↪ size{list.size()}, data{new T[size]} {
    ranges::copy(list, data);
}
```

In dit geval heb ik gebruik gemaakt van de functie copy uit de standaard library.

19.9 Return value optimization

De C++-compiler doet zijn best om het onnodig kopiëren van data te voorkomen. Deze compiler optimalisatie techniek wordt 'copy elision'[245] genoemd. Een speciaal geval van copy elision is de zogenoemde 'return value optimization (RVO)'. Deze optimalisatie voorkomt dat de returnwaarde van een functie eerst in een tijdelijke variabele wordt gekopieerd maar kopieert deze waarde meteen naar zijn uiteindelijke bestemming.

Je kunt kijken of deze optimalisatie wel of niet wordt toegepast door een class met een copy constructor, zie paragraaf 19.3, te definiëren en te kijken wat er gebeurt als je een object van deze class teruggeeft vanuit een functie.

Bijvoorbeeld, zie RVO.cpp:

```
class Int {
public:
    Int(int i): value{i} {
        println("Int met waarde {} wordt aangemaakt.", value);
    }
    ~Int() {
        println("Int met waarde {} wordt verwijderd.", value);
    }
    Int(const Int& r): value{r.value} {
        println("Int met waarde {} wordt gekopieerd.", value);
    }
    int get_value() const {
        return value;
    }
private:
```

[245] Het Engelse woord 'elision' betekent 'weglating'.

```
    int value;
};

Int maak_Int(int i) {
    Int result {i};
    return result;
}

int main() {
    Int local {maak_Int(42)};
    println("De waarde van local is {}.", local.get_value());
}
```

Het enige doel van dit programma is om te kijken of de compiler de return value optimization toepast. Als dit programma gecompileerd wordt met GCC 15.2[246], dan geeft dit programma de volgende uitvoer:

```
Int met waarde 42 wordt aangemaakt.
De waarde van local is 42.
Int met waarde 42 wordt verwijderd.
```

Blijkbaar is de GCC C++-compiler zo slim om alle benodigde kopieeracties weg te optimaliseren[247].

Als we bovenstaand programma compileren[249] en uitvoeren op een CC3220 Launchpad met de compiler optimalisatie op 'off', dan geeft het de volgende uitvoer:

```
[Cortex_M4_0] Int met waarde 42 wordt aangemaakt.
Int met waarde 42 wordt gekopieerd.
Int met waarde 42 wordt verwijderd.
Int met waarde 42 wordt gekopieerd.
Int met waarde 42 wordt verwijderd.
De waarde van local is 42.
Int met waarde 42 wordt verwijderd.
```

[246] `g++ -std=c++23 -fmodules -Wall -Wextra -Wpedantic -g3 -O0 RVO.cpp`

[247] Ondanks het feit dat we de optie `-O0` hebben meegegeven. Volgens de documentatie van GCC[248] disabled de optie `-O0` de meeste optimalisaties (maar de RVO blijkbaar niet.)

[248] Zie https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html.

[249] De gebruikte versie van de TI C++-compiler is 18.12.4.

We zien dat deze compiler in dit geval geen RVO toepast. Ook als de optimalisatie op het maximale level (4) wordt gezet, veranderd de uitvoer niet.

Door het programma zonder RVO te compileren en vervolgens stap voor stap uit te voeren in de debugger, kun je zien waar deze kopietjes aangemaakt en weer verwijderd worden[250]:

- Vanuit `main` wordt als eerste de functie `maak_Int` aangeroepen met de waarde `42` als argument.
- In de functie `maak_Int` krijgt de parameter `i` de waarde `42`.
- De variabele `result` wordt aangemaakt door de constructor van `Int` aan te roepen met de waarde van `i` als argument.
- De constructor slaat de waarde van `i` op in de private datamember `value`.
- Bij het uitvoeren van het **`return`**-statement wordt de waarde van `result` gekopieerd naar een naamloze tijdelijke variabele in `main` door de copy constructor van `Int` aan te roepen met `result` als argument.
- Bij het beëindigen van de functie `maak_Int` wordt de destructor van `Int` aangeroepen om de lokale variabele `result` te verwijderen.
- Vervolgens wordt de copy constructor van `Int` nogmaals aangeroepen om de variabele `local` in `main` aan te maken. De naamloze tijdelijke variabele wordt als argument meegegeven.
- Nadat de variabele `local` is aangemaakt wordt de destructor van `Int` aangeroepen, om de naamloze tijdelijke variabele te verwijderen.
- Dan wordt het `println`-statement uitgevoerd.
- Tot slot wordt de destructor van `Int` aangeroepen om de variabele `local` te verwijderen.

Als dit zelfde programma met RVO gecompileerd wordt en wordt uitgevoerd, dan gebeurt het volgende:

- Vanuit `main` wordt als eerste de functie `maak_Int` aangeroepen met de waarde `42` als argument.
- In de functie `maak_Int` krijgt de parameter `i` de waarde `42`.
- De variabele `result` wordt aangemaakt door de constructor van `Int` aan te roepen met de waarde van `i` als argument.
- De constructor slaat de waarde van `i` op in de private datamember `value`.

[250] Ik raad je sterk aan om dit zelf te doen!

- Bij het uitvoeren van het `return`-statement wordt de variabele `result`, als het ware hernoemd, tot `local`. Er wordt niets gekopieerd en ook niets verwijderd.
- Dan wordt het `println`-statement uitgevoerd.
- Tot slot wordt de destructor van `Int` aangeroepen om de variabele `local` (die bij het aanmaken nog `result` heette) te verwijderen.

Je ziet dat de RVO het maken van twee kopietjes heeft voorkomen.

19.10 Rvalue reference

Soms, zie voor een voorbeeld de volgende paragraaf, is het handig om te weten of een functie met een reference parameter is aangeroepen met een argument dat later nog gebruikt kan worden of met een argument dat na afloop van de functie wordt verwijderd.

Om dit mogelijk te maken is in C++ naast de gewone variabele, de pointer en de reference variabele nog een vierde soort variabele de zogenoemde *rvalue reference* gedefinieerd. Een rvalue reference is net zoals een gewone reference, zie paragraaf 2.14, niets anders dan een alias (andere naam) voor het gene waarnaar hij verwijst (of anders gezegd: refereert). Een gewone reference wordt, om verwarring met een rvalue reference te voorkomen, ook wel een lvalue reference genoemd.

Een lvalue reference kun je alleen laten refereren naar een expressie die ook aan de linkerkant van een toekenning gebruikt zou kunnen worden (vandaar de naam).

```
    int a {1};
    int b {2};
    int& r1 {a}; // r1 is een andere naam voor a
    int& r2 {a + b}; // error: cannot bind non-const lvalue ↩
↪ reference of type 'int&' to an rvalue of type 'int'
```

Een `const` lvalue reference kun je wel laten refereren naar een expressie die alleen aan de rechterkant van een toekenning gebruikt zou kunnen worden. Maar deze `const` lvalue reference kun je vanzelfsprekend, niet wijzigen.

```
int a {1};
int b {2};
const int& r3 {a + b};
r3 = 5; // error: assignment of read-only reference 'r3'
```

Een rvalue reference kun je wel laten refereren naar een expressie die alleen aan de rechterkant van een toekenning gebruikt zou kunnen worden (vandaar de naam). Deze rvalue reference kun je daarna ook nog wijzigen.

```
    int a {1};
    int b {2};
    int&& r5 {a + b}; // r5 is een andere naam voor het resultaat ↩
↪ van a + b
    r5 = 5; // OK: de tijdelijke variabele waar r5 naar refereert ↩
↪ wordt 5
```

Als je nu wilt weten of de functie `f` met een lvalue of met een rvalue wordt aangeroepen, dan kun je deze functie als volgt overloaden, zie paragraaf 1.12:

```
void f(int& ilr) {
    println("f(int& ilr) is aangeroepen met ilr = {}", ilr);
}
void f(int&& irr) {
    println("f(int&& irr) is aangeroepen met irr = {}", irr);
}
```

De code gegeven in `rvalue_reference.cpp`:

```
    int a {1};
    int b {2};
    f(0);
    f(a);
    f(a + b);
```

Geeft nu de volgende uitvoer:

```
f(int&& irr) is aangeroepen met irr = 0
f(int& ilr) is aangeroepen met ilr = 1
f(int&& irr) is aangeroepen met irr = 3
```

Je ziet dat de functie `f` met de lvalue reference `int&` wordt aangeroepen als het argument een lvalue is (een waarde die ook aan de linkerkant van een toekenning gebruikt zou kunnen worden). Je ziet dat de functie `f` met de rvalue reference `int&&` wordt aangeroepen als het argument een rvalue is (een waarde die ook aan de rechterkant van een toekenning gebruikt zou kunnen worden).

Het nuttige toepassing van een rvalue reference wordt in de volgende paragraaf behandeld.

19.11 Move constructor en move assignment operator

Als een class data bevat die dynamisch is aangemaakt, dan is het belangrijk dat het onnodig kopiëren van deze data wordt voorkomen. Om dit te bereiken moet een *move constructor* en *move assignment operator* gedefinieerd worden. De parameters van deze move functies zijn een zogenoemde rvalue reference, zie paragraaf 19.10. Een rvalue reference refereert naar een *tijdelijke* variabele die aangepast mag worden omdat deze variabele niet meer nodig is en de bijbehorende geheugenruimte binnenkort vrijgegeven wordt.

De waarde waar zo'n rvalue reference naar refereert wordt na afloop van de move constructor of move assignment operator niet meer gebruikt. We kunnen de dynamische data van zo'n rvalue dus stelen oftewel verplaatsen zonder dat we deze data hoeven te kopiëren. In een aantal gevallen kan de compiler zelf bepalen dat een move constructor in plaats van een copy constructor gebruikt kan worden. Ook kan de compiler in een aantal gevallen zelf bepalen dat een move assignment operator in plaats van een copy assignment operator gebruikt kan worden. Als voorbeeld bekijken we een eenvoudige class genaamd `Dozijn` die 12 integers beheert die opgeslagen zijn op de heap[251].

```
class Dozijn {
public:
    Dozijn();
    Dozijn(const Dozijn& r);
    Dozijn& operator=(const Dozijn& r);
    ~Dozijn();
    int& operator[](int index);
    const int& operator[](int index) const;
    int* begin();
    const int* begin() const;
    const int* cbegin() const;
    int* end();
    const int* end() const;
    const int* cend() const;
private:
    int* data;
};
```

De implementatie van de memberfuncties kun je vinden in `Dozijn_zonder_move.cpp`.

[251] Zie eventueel pagina 135.

Voor deze class kan een **operator**+ worden gedefinieerd waarmee twee dozijnen opgeteld kunnen worden:

```
Dozijn operator+(const Dozijn& links, const Dozijn& rechts) {
    Dozijn res;
    ranges::transform(links, rechts, res.begin(), [](int l, int r) ↩
 ↪ { return l + r; });
    return res;
}
```

Deze operator kun je als volgt gebruiken om twee dozijnen b en c bij elkaar op te tellen:

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

De werking is[252], stap voor stap, als volgt[253]:

- De **operator**+ wordt aangeroepen waarbij de parameter `left` een referentie is naar b en `right` een referentie is naar c[254].
- De constructor `Dozijn()` wordt aangeroepen om de variabele `res` aan te maken.
- De inhoud van elke element in `res` wordt gelijk gemaakt aan de som van de corresponderende elementen uit `left` (een andere naam voor b) en `right` (een andere naam voor c). Omdat de iteratoren `begin()` en `end()` voor de class `Dozijn` gedefinieerd zijn, kunnen we gebruik maken van het standaard `ranges::transform` algoritme om de `Dozijnen` op te tellen.
- Door het toepassen van return value optimization, zie paragraaf 19.9, wordt de variabele `res` niet gekopieerd of verwijderd maar meteen gebruikt als *rvalue* van de **operator**=. Met andere woorden: de instructie `a = b + c` wordt als het ware vervangen door `a = res`.
- De **operator**= van `Dozijn` wordt aangeroepen. De parameter `r` refereert naar de variabele `res` het resultaat van de optelling. Deze operator is als volgt gedefinieerd:

```
Dozijn& Dozijn::operator=(const Dozijn& r) {
    println("Copy assignment operator aangeroepen");
    ranges::copy(r, begin());
    return *this;
}
```

[252] Ervan uitgaande dat het programma gecompileerd is met GCC 15.2.

[253] Gebruik een debugger met een single step functie om dit zelf na te gaan.

[254] Door hier const reference parameters te gebruiken zorgen we ervoor dat b en c niet gekopieerd hoeven te worden en niet veranderd kunnen worden. Zie paragraaf 2.16.

- De inhoud van elke element in de receiver van de `operator=`, dat is `a`, wordt gelijk gemaakt aan het corresponderende element uit `r` (een andere naam voor `res`). De variabele `a` is na afloop van de `operator=` dus een kopietje van `res` en bevat de som van `b` en `c`.
- Nu `a` de juiste waarde heeft gekregen wordt de destructor `~Dozijn()` aangeroepen voor de variabele `res`.
- De destructor `~Dozijn()` geeft de bij `res` behorende data (die inmiddels is gekopieerd naar `a`) weer vrij, door `delete[] data` aan te roepen.

Het kopiëren van het resultaat van de optelling (in de variabele `result`) naar de variabele `a` kan voorkomen worden door een move assignment operator te definiëren.

```
class Dozijn {
public:
    Dozijn();
    Dozijn(const Dozijn& r);
    Dozijn& operator=(const Dozijn& r);
    Dozijn& operator=(Dozijn&& r); // move assignment
    ~Dozijn();
    int& operator[](int index);
    const int& operator[](int index) const;
    int* begin();
    const int* begin() const;
    const int* cbegin() const;
    int* end();
    const int* end() const;
    const int* cend() const;
private:
    int* data;
};
```

De parameter `r` van deze `operator=` is van het type `Dozijn&&`. Dit is een zogenoemde rvalue reference. Een rvalue reference refereert naar een *tijdelijke* variabele die aangepast mag worden omdat deze variabele niet meer nodig is en de bijbehorende geheugenruimte binnenkort vrijgegeven wordt. De werking van dit programma is[255], stap voor stap, nu als volgt[256]:

[255] Ervan uitgaande dat het programma gecompileerd is met GCC 15.2.

[256] Gebruik een debugger met een single step functie om dit zelf na te gaan.

- De `operator+` wordt aangeroepen waarbij de parameter `left` een referentie is naar `b` en `right` een referentie is naar `c`.
- De constructor `Dozijn()` wordt aangeroepen om de variabele `res` aan te maken.
- De inhoud van elke element in `res` wordt gelijk gemaakt aan de som van de corresponderende elementen uit `left` (een andere naam voor `b`) en `right` (een andere naam voor `c`).
- Door het toepassen van return value optimization, zie paragraaf 19.9, wordt de variabele `res` niet gekopieerd of verwijderd maar meteen gebruikt als *rvalue* van de `operator=`. Met andere woorden: de instructie `a = b + c` wordt als het ware vervangen door `a = res`.
- De move assignment operator van `Dozijn` wordt nu aangeroepen. Deze operator is als volgt gedefinieerd:

```
Dozijn& Dozijn::operator=(Dozijn&& r) {
    println("Move assignment operator aangeroepen");
    swap(data, r.data);
    return *this;
}
```

- De rvalue reference parameter `r` refereert naar de variabele `res` het resultaat van de optelling.
- De `data` van de receiver wordt verwisseld met de `data` van de parameter `r` (een andere naam voor `res`). Let er op dat niet de data zelf maar de pointer naar de data wordt verwisseld. Om deze pointers te verwisselen gebruiken we de standaard `swap`-functie.
- Nu `a` de juiste waarde heeft gekregen wordt de destructor `~Dozijn()` aangeroepen voor de variabele `res`. De variabele `res` bevat de oude waarden van `a` die nu worden vrijgegeven.

Doordat nu de move assignment operator gebruikt is in plaats van de copy assignment operator is voorkomen dat de gehele inhoud van `res` (twaalf integers) naar `a` gekopieerd is. In plaats daarvan zijn slechts twee pointers verwisseld. De besparing is in dit geval nog gering, maar je begrijpt dat het gebruik van een move assignment bij classes die veel data beheren een behoorlijke besparing kan opleveren.

Als we een `Dozijn` aanmaken en initialiseren met een ander `Dozijn` dan wordt, uiteraard, de copy constructor aangeroepen, zie paragraaf 19.3. De code:

```
Dozijn d {a};
```

Roept de copy constructor aan.

Als je deze copy constructor aanroept met een rvalue als argument, bijvoorbeeld:

```
Dozijn e {a + b};
```

Dan kun je het maken van een extra kopie voorkomen door een move constructor te definiëren:

```
class Dozijn {
public:
    Dozijn();
    Dozijn(const Dozijn& r);
    Dozijn(Dozijn&& r); // move constructor
    Dozijn& operator=(const Dozijn& r);
    Dozijn& operator=(Dozijn&& r); // move assignment
    ~Dozijn();
    int& operator[](int index);
    const int& operator[](int index) const;
    int* begin();
    const int* begin() const;
    const int* cbegin() const;
    int* end();
    const int* end() const;
    const int* cend() const;
private:
    int* data;
};
```

De parameter `r` van deze move constructor is van het type `Dozijn&&`. Deze move constructor kan als volgt gedefinieerd worden:

```
Dozijn::Dozijn(Dozijn&& r): data(r.data) {
    println("Move constructor aangeroepen");
    r.data = nullptr;
}
```

Je ziet dat deze constructor simpelweg de data van de parameter overneemt. Niet door de data zelf te kopiëren maar door de pointer naar de data te kopiëren. De pointer naar de data van de parameter wordt gelijk gemaakt aan de **`nullptr`** zodat de variabele waar deze parameter naar refereert later veilig opgeruimd kan worden. Volgens de C++-standaard mag **`delete`**`[]` worden aangeroepen op de **`nullptr`**, zie [10, paragraaf 8.3.5].

Als je zeker weet dat een lvalue die je als argument van een constructor of assignment operator gebruikt na afloop niet meer nodig is, dan kun je de standaard `move`-functie gebruiken om de lvalue om te zetten in een rvalue reference. Hierdoor wordt dan de move constructor of de move assignment operator aangeroepen. Bijvoorbeeld:

```
Dozijn d {move(a)};
```

De standaard functie `move` verplaatst zelf niets[257], maar geeft een rvalue reference die refereert naar a terug, waardoor de move constructor aangeroepen wordt. Pas wel goed op. Na deze instructie kun je a niet meer uitlezen![258]

Een meer realistische toepassing van de standaard functie `move` is als volgt. Stel dat je regelmatig twee objecten van de hierboven gedefinieerde class `Dozijn` moet verwisselen. Je schrijft vervolgens een functie genaamd `swap` die twee als argumenten meegegeven dozijnen als volgt verwisselt:

```
void swap(Dozijn& d1, Dozijn& d2) {
    println("swap aangeroepen:");
    Dozijn hulp {d1};
    d1 = d2;
    d2 = hulp;
}
```

Deze functie roep je vervolgens als volgt aan:

```
swap(a, b);
```

Het uitvoeren van deze regel blijkt echter behoorlijk wat tijd in beslag te nemen. In de eerste regel wordt de copy constructor van `Dozijn` aangeroepen om de lokale variabele `hulp` aan te maken. Deze copy constructor reserveert geheugenruimte voor twaalf integers (door het aanroepen van `new int[12]`) en kopieert vervolgens de gehele inhoud van `d1` (twaalf integers) naar deze geheugenruimte. Vervolgens wordt de assignment operator van `Dozijn` aangeroepen die de gehele inhoud (twaalf integers) van `d2` kopieert naar `d1`. Tot slot wordt de assignment operator van `Dozijn` nogmaals aangeroepen die gehele inhoud (twaalf integers) van `hulp` kopieert naar `d2`. Tot slot wordt de destructor `~Dozijn` aangeroepen om de lokale variabele `hulp` te verwijderen. In deze destructor wordt de geheugenruimte die in

[257] De functie levert een rvalue reference die refereert naar zijn argument. Het is dus eigenlijk een soort cast, zie paragraaf 15.15.

[258] Het gebruik van de functie `move` is in dit geval dus erg discutabel. Waarom zou je een kopie van a maken, genaamd `d`, als je a daarna niet meer kan uitlezen? Je kunt dan net zo goed geen kopie maken en a gebruiken in plaats van `d`.

de copy constructor is gereserveerd weer vrijgegeven. Dus er worden bij het uitvoeren van de functie swap 12 integers gereserveerd en ook weer vrijgegeven en er worden $3 \times 12 = 36$ integers gekopieerd.

Je kunt echter bedenken dat het helemaal niet nodig is om geheugenruimte te reserveren en weer vrij te geven. Ook is het niet nodig om integers te kopiëren. Het enige dat moet gebeuren is het verwisselen van de twee private pointers `data`. Je kunt dit eenvoudig voor elkaar krijgen door gebruik te maken van de move constructor en de move assignment van `Dozijn`. De waarde van `d1` hoeft namelijk niet meer uitgelezen te worden nadat `hulp` is aangemaakt. De waarde van `d2` hoeft niet meer uitgelezen te worden nadat deze is toegekend aan `d1` en hetzelfde geldt voor de waarde van `hulp` nadat die is toegekend aan `d2`. Je kunt de `swap_move` dus als volgt definiëren:

```
void swap_move(Dozijn& d1, Dozijn& d2) {
    println("swap_move aangeroepen:");
    Dozijn hulp {move(d1)};
    d1 = move(d2);
    d2 = move(hulp);
}
```

Deze functie roep je vervolgens als volgt aan:

```
    swap_move(a, b);
```

De code is nu sneller[259] omdat er slechts 8 pointers gekopieerd hoeven te worden. Ook wordt de destructor `~Dozijn` één maal aangeroepen om de lokale variabele `hulp` op te ruimen. Er hoeft echter geen geheugen vrijgegeven te worden omdat de pointer `data` van `hulp` op dat moment gelijk is aan de **`nullptr`**.

In plaats van zelf een swap functie te definiëren, hadden we natuurlijk ook de standaard template functie swap aan kunnen roepen:

```
    std::swap(a, b);
```

Het zal je waarschijnlijk niet verbazen dat deze standaard template functie ook gebruikt maakt van de move constructor en de move assignment operator. Dat kan natuurlijk alleen maar als deze gedefinieerd zijn voor de betreffende class, in dit geval `Dozijn`.

[259] Op een CC3220S Launchpad gecompileerd met optimization off duurt de swap 12306 clockcycles en de `swap_move` en `std::swap` duren elk 10965 clockcycles. Dit is slechts 11 % sneller. De besparing is in dit geval nog gering, maar je begrijpt dat het gebruik van een move assignment en move assignment operator bij classes die veel data beheren een behoorlijke besparing kan opleveren.

Het is dus belangrijk om voor een class die een zelfgedefinieerde copy constructor en assingment operator nodig heeft, zie paragraaf 19.5, ook een move constructor en een move assignment operator te definiëren.

20

De standaard C++ library (de details)

Dit hoofdstuk bevat een aantal handige wetenswaardigheden die van pas kunnen komen bij het gebruik van de C++ standaard library, zie paragraaf 10.2.

20.1 Het meten van de executietijd van een stukje code

In C++ kun je een probleem vaak op vele verschillende manieren oplossen. Soms wil je bepalen welk van deze oplossingen sneller is op een bepaald platform. De enige goede manier om dit vast te stellen is om te *meten* hoe lang het uitvoeren van elke oplossing duurt.

Met behulp van de standaard library opgenomen `high_resolution_clock` kun je eenvoudig de executietijd van een stuk code meten.

```
    auto start {chrono::high_resolution_clock::now()};
    // code die je wilt timen
    auto stop {chrono::high_resolution_clock::now()};
    auto duration ↩
↪ {chrono::duration_cast<chrono::microseconds>(stop - ↩
↪ start).count()};
    println("Executietijd = {} µs", duration);
```

20.2 Random getallen

Soms is het handig om random getallen te gebruiken in een programma. De C++ standaard library bevat hier uitgebreide faciliteiten voor. In Dobbelsteen.cpp is de class Dobbelsteen gedefinieerd waarmee eenvoudig een random getal tussen 1 en 6 gegenereerd kan worden:

```
Dobbelsteen d;
int i {d.rol()};
```

20.3 vector::reserve

Als een vector groeit kunnen de elementen van plaats veranderen. Pointers en iteratoren naar deze elementen staan dan naar verkeerde adressen te wijzen. Dit kun je voorkomen door met behulp van de functie reserve, van tevoren, voldoende elementen te reserveren. Zie vector_reserve.cpp.

20.4 emplace

De standaardcontainer hebben zogenoemde *emplace*-operaties. Zo heeft bijvoorbeeld een vector naast de bekende push_back-operatie ook een emplace_back-operatie. De emplace_back combineert het toevoegen van een element in de vector met het aanroepen van de constructor van dit element. Dit voorkomt het maken van een onnodig kopietje. In het programmafragment:

```
vector<Breuk> v;
v.push_back(Breuk {1, 2});
```

Kan de laatste regel efficiënter gecodeerd worden als:

```
v.emplace_back(1, 2);
```

Zie vector_emplace.cpp.

20.5 optional<T>

Een object van de standaard class optional<T> kan een waarde van het type T of *niets* bevatten. In paragraaf 6.2 heb je een functie gezien die de impedantie van een condensator berekent. Omdat de impedantie van een condensator niet berekend kan worden (oneindig groot is) als de capaciteit of de frequentie nul is, werd er een **bool** returnwaarde gebruikt

om aan te geven of de berekening mogelijk is of niet. Dit was niet zo handig omdat de returnwaarde van de functie niet de berekende impedantie bevatte, maar via een output parameter werd teruggegeven.

Door deze functie te herschrijven zodat deze een `optional` returnwaarde heeft, kan de returnwaarde van de functie de berekende impedantie bevatten of *leeg* zijn als de berekening niet mogelijk is, zie `impedance_C3_optional.cpp`:

```
optional<complex<double>> impedance_C(double c, double f) {
    if (c != 0.0 && f != 0.0) {
        return complex<double> {0, -1 / (2 * numbers::pi * f * c)};
    }
    else
        return {};
}
```

Je kunt controleren of een `optional` een waarde bevat met behulp van de `has_value()` functie of door de `optional` te gebruiken in een boolean context. De waarde van een `optional` kan worden opgevraagd met behulp van de `value()` functie of door de `optional` te gebruiken met behulp van de `*` operator.

Het programmadeel:

```
    auto result = impedance_C(1e-6, 1e3);
    if (result) {
        println("{:.3f}", *result);
    }
    else {
        println("Kan impedantie niet berekenen.");
    }
    result = impedance_C(1e-6, 0);
    if (result.has_value()) {
        println("{:.3f}", result.value());
    }
    else {
        println("Kan impedantie niet berekenen.");
    }
```

Geeft als uitvoer:

```
0.000-159.155j
Kan impedantie niet berekenen.
```

Ook bij de **operator**<< die gebruikt kan worden om een Breuk in te lezen kan een optional returnwaarde handig zijn, zie optional.cpp.

20.6 Van numeriek naar string en van string naar numeriek

Om een numerieke waarde om te zetten in een string, kun je de format functie gebruiken. Daarbij kun je zelf bepalen hoe de numerieke waarde geformatteerd moet worden, bijvoorbeeld met een bepaald aantal decimalen, in wetenschappelijke notatie, enzovoort[260].

Bijvoorbeeld, als je een **double** waarde wilt omzetten in een string met twee decimalen, dan kun je de volgende code gebruiken:

```
string s {format("{:.2f}", numbers::pi)};
```

Na het uitvoeren van deze code bevat de string s de waarde "3.14".

Als het niet nodig is om de numerieke waarde op een specifieke manier te formatteren, dan kun je ook de to_string functie gebruiken.

```
string to_string(T value);
```

Waarbij het type **T** (**unsigned**) **int**, (**unsigned**) **long**, (**unsigned**) **long long**, **float**, **double** of **long double** kan zijn.

Bijvoorbeeld:

```
string s {to_string(numbers::pi)};
```

Na het uitvoeren van deze code bevat de string s de waarde "3.141593". Zie to_string.cpp.

De volgende functies zijn beschikbaar om een string om te zetten in een numerieke waarde.

```
T stoX(const string& str, size_t *pos = nullptr, int base = 10);
```

Waarbij **X** i, ui, l, ul, ll, ull, f, d of ld kan zijn. **T** is dan respectievelijk **int**, **unsigned int**, **long**, **unsigned long**, **long long**, **unsigned long long**, **float**, **double** of **long double**. De functies die een floating point waarde teruggeven, hebben geen parameter base. pos is een output parameter. Na afloop bevat *pos de index van het eerste niet-omgezette teken. base geeft het grondtal aan dat tijdens de omzetting moet worden gebruikt. Als base 0 is, wordt het grondtal automatisch bepaald op basis van de inhoud van de string:

[260] Zie https://en.cppreference.com/w/cpp/utility/format/spec.html.

- als de string begint met `0`, wordt het grondtal 8;
- als de string begint met `0x`, wordt het grondtal 16;
- anders wordt het grondtal 10[261].

Deze functies slaan voorafgaande spaties over. Er wordt een genereren een `invalid_argument` exception gegooid als er geen conversie kan worden uitgevoerd. Er wordt een `out_of_range` exception gegooid als de geconverteerde waarde buiten het bereik van het return type valt. Zie `sto_numeriek.cpp`.

Er zijn low level conversie functies beschikbaar (`from_chars`[262] en `to_chars`[263]) om numerieke gegevens te serialiseren naar en te deserialiseren vanuit formaten zoals JSON, XML, enzovoort.

20.7 Unordered container met UDT's

Als je een user-defined datatype, bijvoorbeeld `Breuk` als key in een ongeordende container, bijvoorbeeld `unordered_set`, wilt gebruiken, dan moet

- bepaald kunnen worden of twee objecten van dit type gelijk zijn;
- een object van dit type omgezet kunnen worden naar een hashwaarde.

Aan de eerste voorwaarde kun je simpel voldoen door de **`operator`**= voor het user-defined datatype te overloaden, zie paragraaf 19.4.

Om aan de tweede voorwaarde te voldoen, heb je twee opties. Als eerste kun je een type definiëren met een overloaded **`operator`**`()`, waarmee een zogenoemd *functie-object*, aangemaakt kan worden. Dit functie-object moet dan de hash waarde voor het betreffende object produceren.

Zie `unordered_set1.cpp`. Het type dat het functie-object moet aanmaken is `Breuk_hash`:

```
struct Breuk_hash {
    size_t operator()(Breuk const& b) const noexcept;
};
```

[261] Je zou misschien verwachten dat als de string begint met `0b`, het grondtal 2 zou worden, maar dat is (helaas) niet het geval in C++23. Zie https://en.cppreference.com/w/cpp/string/basic_string/stol.

[262] Zie https://en.cppreference.com/cpp/utility/from_chars.

[263] Zie https://en.cppreference.com/cpp/utility/to_chars.

De overloaded **operator()** moet de als argument meegegeven Breuk omzetten in een hashwaarde van het type size_t. Het keyword **noexcept** geeft aan dat deze functie geen exception kan gooien. Daardoor kan de compiler code verder optimaliseren. De implementatie van deze overloaded **operator()** is als volgt:

```
size_t Breuk_hash::operator()(Breuk const& b) const noexcept {
    return hash<int>{}(b.boven) ^ hash<int>{}(b.onder);
}
```

De hashwaarde wordt bepaald door de hashwaarde van de twee datamembers van Breuk, boven en onder, met elkaar te exoren (met de operator ^). Omdat de datamembers van Breuk**private** zijn, moet deze **operator()** als **friend** in Breuk gedeclareerd worden:

```
friend size_t Breuk_hash::operator()(Breuk const& b) const noexcept;
```

Het type Breuk_hash moet nu als tweede argument aan de template unordered_set worden meegegeven:

```
    unordered_set<Breuk, Breuk_hash> breuken;
```

Als alternatief kun je ook een template specialisation voor hash<T> definiëren in namespace std.

Zie unordered_set2.cpp:

```
namespace std
{
    template<> struct hash<Breuk> {
        size_t operator()(Breuk const& h) const noexcept;
    };
}
```

De overloaded **operator()** moet de als argument meegegeven Breuk omzetten in een hashwaarde van het type size_t. De implementatie van deze overloaded **operator()** is als volgt:

```
namespace std
{
    size_t hash<Breuk>::operator()(Breuk const& b) const noexcept {
        return hash<int>{}(b.boven) ^ hash<int>{}(b.onder);
    }
}
```

De hashwaarde wordt bepaald door de hashwaarde van de twee datamembers van Breuk, boven en onder, met elkaar te exoren (met de operator ^). Omdat de datamembers van Breuk **private** zijn, moet deze **operator**() als **friend** in Breuk gedeclareerd worden:

```
friend size_t hash<Breuk>::operator()(Breuk const& h) const noexcept;
```

Er hoeft nu geen tweede argument aan de template unordered_set te worden meegegeven:

```
    unordered_set<Breuk> breuken;
```

De volgende code:

```
    breuken.insert(Breuk {1,2});
    breuken.insert(Breuk {1,3});
    breuken.insert(Breuk {2,3});
    breuken.insert(Breuk {1,4});
    breuken.insert(Breuk {2,4});
    breuken.insert(Breuk {3,4});
    println("breuken = {::s}", breuken);
```

Levert nu, in beide gevallen, de volgende uitvoer op:

```
breuken = 3/4, 1/4, 2/3, 1/3, 1/2
```

Omdat breuken genormaliseerd worden, wordt de Breuk {2,4} niet in de set opgenomen omdat de Breuk {1,2} er al inzit.

20.8 Voorbeeldprogramma dat generiek en objectgeoriënteerd programmeren combineert

De standaard functie mem_fn vormt de koppeling tussen generiek programmeren en objectgeoriënteerd programmeren omdat hiermee een memberfunctie kan worden omgezet in een functie-object, zie mem_fn.cpp.

```
import std;
using namespace std;

class Hond {
public:
    virtual ~Hond() = default;
    virtual void blaf() const = 0;
};
```

```
class Tekkel: public Hond {
public:
    void blaf() const override {
        print("Kef kef ");
    }
};

class St_bernard: public Hond {
public:
    void blaf() const override {
        print("Woef woef ");
    }
};
    list<shared_ptr<Hond>> kennel {make_shared<Tekkel>(), ↩
 ↪ make_shared<St_bernard>(), make_shared<Tekkel>()};
    ranges::for_each(kennel, mem_fn(&Hond::blaf));
    println();
}
```

De uitvoer van dit programma:

```
Kef kef Woef woef Kef kef
```

Je kunt alle honden in de kennel ook laten blaffen met behulp van lambda-functie:

```
    ranges::for_each(kennel, [](const auto p) {
        p->blaf();
    });
    println();
```

Je kunt alle honden in de kennel ook laten blaffen met behulp van een range-based for:

```
    for (const auto& p: kennel) {
        p->blaf();
    }
    println();
```

Het gebruik van een range-based for is eenvoudiger dan het gebruik van een `for_each`. De range-based for kan echter alleen gebruikt worden als de hele container moet worden doorlopen. Met een `for_each` kan ook een deel van een container doorlopen worden.

20.9 Standaard functie-objecten

Je hebt gezien dat je een bepaalde voorwaarde of een bepaalde bewerking kunt doorgeven aan een algoritme door middel van een functie-pointer, een zelfgemaakt functie-object of een lambda-functie, zie paragraaf 13.1.2 en paragraaf 13.3.1. Je kunt echter ook gebruik maken van een zogenoemd standaard functie-object. Voor alle rekenkundige en logische operaties zijn template classes beschikbaar. Zo wordt, bijvoorbeeld, door een object van de class `plus<T>` een **`operator`**`+` uitgevoerd op twee objecten van het type `T` en wordt door een object van de class `greater<T>` een **`operator`**`>` uitgevoerd op twee objecten van het type `T`. Zie voor een compleet overzicht `http://en.cppreference.com/w/cpp/utility/functional`.

Het gebruik van standaard functie-objecten is vooral handig als de operatie die door het algoritme uitgevoerd moet worden uit één enkele operator bestaat.

Zo kun je bijvoorbeeld de `vector w` bij de `vector v` optellen door middel van een `transfer` algoritme met een standaard functie-object `plus`, zie `transform_plus`.

```
import std;
using namespace std;

int main() {
    vector v {-3, -4, 3, 4};
    vector w {1, 2, 3, 4};
    println("v: {}", v);
    println("w: {}", w);

//  Bewerking opgeven met een standaard functie-object.
//  Voordeel: hergebruik van standaardcomponenten.
    ranges::transform(v, w, v.begin(), plus());
    println("v na transform: {}", v);
}
```

Als je dit vergelijkt met het gebruik van een functie of lambda-functie zoals in paragraaf 13.3.1, dan zie je dat het gebruik van een standaard functie-object in dit geval eenvoudiger is.

De code die in paragraaf 13.4.1 gegeven is om een `vector` met integers van hoog naar laag te sorteren (zie `sort_lambda.cpp`):

```
    ranges::sort(rij, [](auto i, auto j) {
        return i > j;
    });
```

kan sterk vereenvoudigd worden door het gebruik van een standaard functie-object (zie sort_greater.cpp):

```
ranges::sort(rij, greater());
```

Het gebruik van een standaard functie-object om een voorwaarde door te geven aan één van de _if algoritmen is minder eenvoudig. In paragraaf 13.1.2 heb je gezien dat het eerste getal ≥ 0 in een list met integers opgezocht kan worden door deze voorwaarde aan find_if door te geven met behulp van een functie of een lambda-functie. Er is in dit geval een functie-object nodig dat één **int** als parameter heeft en een **bool** teruggeeft die aangeeft of het meegegeven argument ≥ 0 is. Het is dus *niet* mogelijk om rechtstreeks gebruik te maken van de standaard class greater_equal<T> omdat een functie-object van die class twee parameters heeft. Met een omweg is dit echter wel mogelijk. Er moet dan gebruik gemaakt worden van de standaard functie bind waarmee een functie-object omgezet kan worden in een ander functie-object waarbij bepaalde parameters uit het oorspronkelijke functie-object gebonden kunnen worden aan bepaalde waarden. Het standaard functie-object greater_equal<**int**>() heeft twee parameters en bepaalt bij aanroep of het eerste argument $\geq$ aan het tweede argument is. Met behulp van bind kan hiervan een functie-object gemaakt worden met slechts één parameter die bij aanroep bepaald of het argument ≥ 0 is, door de tweede parameter van greater_equal<**int**>() te binden aan de waarde 0. De aanroep van bind ziet er dan als volgt uit:

```
bind(greater_equal<int>(), _1, 0)
```

De _1 is een zogenoemde placeholder[264] en geeft aan dat de eerste (en enige) parameter van het door bind teruggegeven functie-object ingevuld moet worden als eerste argument van greater_equal<**int**>(). De 0 geeft aan dat de constante waarde 0 ingevuld moet worden als tweede argument van greater_equal<**int**>(). Het door bind teruggegeven functie-object kan vervolgens aan het find_if algoritme worden doorgegeven om de eerste integer ≥ 0 in de vector rij te vinden, zie find_if_bind.cpp:

```
import std;
using namespace std;
using namespace std::placeholders;

int main() {
    list l {-3, -4, 3, 4};
    auto r {ranges::find_if(l, bind(greater_equal(), _1, 0))};
```

[264] De placeholders zijn gedefinieerd in een aparte namespace genaamd std::placeholders.

```
    if (r != l.cend()) {
        println("Het eerste positieve element is: {}", *r);
    }
}
```

Persoonlijk vind ik het gebruik van een lambda-functie in dit geval eenvoudiger, zie paragraaf 13.1.2.

In paragraaf 13.3.2 heb je gezien hoe de even getallen uit een `vector` met integers verwijderd kunnen worden met behulp van een lambda-functie, zie `remove.cpp`:

```
    w.erase(ranges::remove_if(w, [](auto i) {
        return i % 2 == 0;
    }).cbegin(), w.cend());
```

Dit zou ook met behulp van standaard functie-objecten kunnen, zie `remove_bind.cpp`:

```
    w.erase(ranges::remove_if(w, bind(equal_to(), bind(modulus(), ↩
↪ _1, 2), 0)).begin(), w.end());
```

Ook in dit geval vind ik het gebruik van een lambda-functie eenvoudiger.

20.10 Views

TODO: nog toevoegen!

20.11 Dynamische format specifiers

Sinds Cpp23 moet de format string in de `format`, `print` en `println` functies een compile time constant zijn. De code:

```
    string s {"Hello World!"};
    println(s); // error:: call to consteval function ↩
↪ 'std::basic_format_string<char>(s)' is not a constant expression
```

Geef dus een foutmelding omdat de format sting geen compile time constante is, maar een variabele.

Op pagina 26 heb je het programma `lees_en_print_double.cpp` gezien waarbij de het aantal decimalen dat wordt afgedrukt van een floating point getal `d` afhankelijk is van de waarde van de variabele `aantal_decimalen`.

```
    println("d = {:.{}f}", d, aantal_decimalen);
```

Als je echter probeert om tijdens runtime te bepalen of een floating point getal in wetenschappelijke notatie[265] moet worden afgedrukt of niet, dan kun je dat niet als volgt doen:

```
    println("{:{}}", wetenschappelijke_notatie ? "e" : "f", ↩
↪ 12345.67890);
```

Deze code compileert wel, maar geef bij het uitvoeren een exception:

```
terminate called after throwing an instance of 'std::format_error'
 what():  format error: argument used for width or precision must ↩
 ↪ be a non-negative integer
Aborted
```

Je kunt dit natuurlijk oplossen door twee aparte `println` aanroepen te maken, maar dat is niet erg elegant.

```
    if (wetenschappelijke_notatie) {
        println("{:e}", 12345.67890);
    } else {
        println("{:f}", 12345.67890);
    }
```

Als je de format string tijdens runtime wilt samenstellen, dan moet je gebruik maken van de `vformat`, `vprint_unicode` of `vprint_nonunicode` functies.

Op de volgende wijze kun je dan bepalen of een floating point getal in wetenschappelijke notatie moet worden afgedrukt of niet:

```
    println("{}", vformat(wetenschappelijke_notatie ? "{:e}" : ↩
↪ "{:f}", make_format_args(d)));
```

De `vformat` functie neemt een format string en een object van het type `format_args` als argumenten en retourneert een `string` die is opgemaakt volgens de format string en de meegegeven argumenten. De functie `make_format_args` neemt een willekeurig aantal argumenten en retourneert een object van het type `format_args` dat deze argumenten bevat. De functie `println` wordt gebruikt om de door `vformat` geretourneerde string af te drukken.

[265] Zie eventueel https://nl.wikipedia.org/wiki/Wetenschappelijke_notatie.

Als alternatief kun je ook de functie vprint_unicode of vprint_nonunicode gebruiken om de string direct af te drukken zonder dat er een tijdelijk object van het type string hoeft te worden gemaakt:

```
    vprint_unicode(wetenschappelijke_notatie ? "{:e}\n" : ↩
↪ "{:f}\n", make_format_args(d));
```

Omdat er geen println-versie van deze functies bestaat moet je zelf een newline karakter toevoegen aan de format string. Zie vformat.cpp.

In C++26 zijn vformat en de vprint-varianten niet meer nodig en kan de bovenstaande code vervangen worden door:

```
    println(dynamic_format(wetenschappelijke_notatie ? "{:e}" : ↩
↪ "{:f}"), d);
```

Als je een programma schrijft dat meerdere talen moet ondersteunen, dan komt het gebruik van vformat (of een van de vprint-varianten) vaak goed van pas. Bijvoorbeeld, zie getal_number.cpp:

```
import std;
using namespace std;

int main() {
    char taal;
    do {
        print("Nederlands (N) / English (E): ");
        cin >> taal;
    } while (taal != 'N' && taal != 'E');
    int index {taal == 'N'? 0 : 1};
    array vraag {"Geef een getal: ", "Enter a number: "};
    array antwoord {"Je hebt het getal {} ingevoerd.", "You have ↩
↪ entered the number {}."};
    print("{}", vraag[index]);
    int getal;
    cin >> getal;
    println("{}", vformat(antwoord[index], make_format_args(getal)));
}
```

In dit voorbeeld zijn de Nederlandse en Engelse teksten opgenomen in een array maar in de praktijk zou je deze teksten waarschijnlijk in aparte bestanden opslaan en deze bestanden inlezen op het moment dat het programma wordt uitgevoerd. Ook dan is het noodzakelijk

om `vformat` (of een van de `vprint`-varianten) te gebruiken omdat de format string tijdens runtime wordt bepaald.

In C++26 kan de laatste regel van het bovenstaande programma vervangen worden door:

```
    println(dynamic_format(antwoord[index]), getal);
```

20.12 Lokalisatie

De manier waarop een datum weergegeven wordt varieert per regio. Hetzelfde geldt voor het weergeven van floating point getallen, in sommige regio's wordt een punt gebruikt als scheidingstekens en in andere regio's een komma. Het aanpassen van een programma aan een bepaalde regio wordt lokalisatie (Engels: *localization*) genoemd, ook wel afgekort tot *L10n*.

Als in het programma `getal_number.cpp`, zie pagina 412 een groot getal wordt ingetypt, dan zou het handig zijn om dit getal met scheidingstekens voor duizendtallen af te drukken, bijvoorbeeld (in het Nederlands) `1.000.000` in plaats van `1000000`. Het enige probleem hierbij is dat dit afhankelijk is van de gebruikte taal. In het Engels is het gebruikelijk om een komma als scheidingsteken voor duizendtallen te gebruiken, dus `1,000,000` in plaats van `1000000`. C++ biedt ondersteuning voor lokale conventies zoals datum en tijd-notaties, valuta, getalnotaties enzovoort, via de class `locale`. Bijvoorbeeld:

```
import std;
using namespace std;
int main() {
    char taal;
    do {
        print("Nederlands (N) / English (E): ");
        cin >> taal;
    } while (taal != 'N' && taal != 'E');
    int index {taal == 'N'? 0 : 1};
    auto loc {taal == 'N'? locale{"nl_NL.utf8"} : ↩
 ↪ locale{"en_US.utf8"}};
    array vraag {"Geef een groot getal: ", "Enter a big number: "};
    array antwoord {"Je hebt het getal {} ingevoerd.", "You have ↩
 ↪ entered the number {}."};
    print("{}", vraag[index]);
    int getal;
    cin >> getal;
```

```
    string getal_als_string {format(loc, "{:L}", getal)};
    println("{}", vformat(antwoord[index], ↩
 ↪ make_format_args(getal_als_string)));
}
```

In dit voorbeeld wordt ervan uitgegaan dat de locales `nl_NL.utf8` en `en_US.utf8` beschikbaar zijn op het systeem waarop het programma wordt uitgevoerd[266]. Aan de functie `format` kun je als eerste argument een locale meegeven en als je dan in de format specifier `{:L}` gebruikt, dan zal de betreffende locale worden toegepast bij het formateren van de betreffende data. Helaas kan aan de functie `vformat` geen locale worden meegegeven, dus als je een format string tijdens runtime wilt samenstellen en daarbij gebruik wilt maken van een locale, dan moet je zelf de locale toepassen op de argumenten voordat je deze aan `vformat` meegeeft.

In C++26 kunnen de laatste zes regels van het bovenstaande programma vervangen worden door:

```
    array antwoord {"Je hebt het getal {:L} ingevoerd.", "You have ↩
 ↪ entered the number {:L}."};
    print("{}", vraag[index]);
    int getal;
    cin >> getal;
    println("{}", format(loc, dynamic_format(antwoord[index]), ↩
 ↪ getal);
```

De uitvoer van het programma `getal_nummer_locale.cpp` is bijvoorbeeld:

```
Nederlands (N) / English (E): N
Geef een groot getal: 1234567890
Je hebt het getal 1.234.567.890 ingevoerd.
```

Een andere mogelijke uitvoer is:

```
Nederlands (N) / English (E): E
Enter a big number: 1234567890
You have entered the number 1,234,567,890.
```

[266] Op een Linux- of WSL-systeem kun je de beschikbare locales afdrukken met het commando `locale -a` in een terminal. Als `en_US.utf8` en `nl_NL.utf8` niet in de lijst staan, kunt u ze inschakelen met het volgende comando in de terminal:
`sudo nano /etc/locale.gen`
Verwijder het commentaar bij de regels voor `en_US.UTF-8` en `nl_NL.UTF-8` en voer vervolgens het volgende commando uit om de landinstellingen te genereren: `sudo locale-gen`.

Bibliografie

[1] Frank Brokken. *C++ Annotations*. University of Groningen, 2026. ISBN: 978-90-367-0470-0. URL: http://www.icce.rug.nl/documents/cplusplus/ (geciteerd op pp. 14, 26, 34, 156, 186, 347, 371).

[2] Spencer Collyer. "User-Defined Formatting in std::format". In: *Overload* 180 (2024), p. 20–26. ISSN: ISSN: 1354-3172. URL: https://accu.org/journals/overload/32/180/overload180.pdf#page=22 (geciteerd op p. 347).

[3] Edsger W. Dijkstra. *Algol 60 translation : An Algol 60 translator for the x1 and Making a translator for Algol 60*. Tech. rap. 35. Mathematisch Centrum, Amsterdam, 1961. URL: http://www.cs.utexas.edu/users/EWD/MCReps/MR35.PDF (geciteerd op p. 171).

[4] Marc Gregoire. *Professional C++*. 6th Edition. Hoboken, New Jersey: John Wiley & Sons, Inc., 2024. ISBN: 978-1-39419-317-2 (geciteerd op p. 14).

[5] Jens Gustedt. *Modern C*. 2nd Edition. Manning Publications, 2019. ISBN: 978-1-61729-581-2. URL: https://hal.inria.fr/hal-02383654/document (geciteerd op pp. 16, 321).

[6] C. A. R. Hoare. "Quicksort". In: *The Computer Journal* 5.1 (jan 1962), p. 10–16 (geciteerd op p. 161).

[7] ISO. *ISO/IEC 14882:2011 Information technology — Programming languages — C++*. Final draft: https://wg21.link/n3337. Geneva, Switzerland: International Organization for Standardization, 2012. URL: https://www.iso.org/standard/57853.html (geciteerd op p. 160).

[8] ISO. *ISO/IEC 14882:2014 Information technology — Programming languages — C++*. Final draft: https://timsong-cpp.github.io/cppwp/n4140/draft.pdf. Geneva, Switzerland: International Organization for Standardization, 2014. URL: https://www.iso.org/standard/64029.html (geciteerd op pp. 90, 160).

[9] ISO. *ISO/IEC 14882:2017 Information technology — Programming languages — C++*. Final draft: https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2017/n4659.pdf. Geneva, Switzerland: International Organization for Standardization, 2017. URL: https://www.iso.org/standard/68564.html (geciteerd op pp. 160, 186).

[10] ISO. *ISO/IEC 14882:2024 Information technology — Programming languages — C++*. Final draft: https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2023/n4950.pdf. Geneva, Switzerland: International Organization for Standardization, 2024. URL: https://www.iso.org/standard/83626.html (geciteerd op pp. 13, 22, 184, 396).

[11] ISO. *ISO/IEC 9899:2023 Information technology — Programming languages — C*. Working draft: https://open-std.org/JTC1/SC22/WG14/www/docs/n3467.pdf. Geneva, Switzerland: International Organization for Standardization, 2023. URL: https://www.iso.org/standard/83627.html (geciteerd op pp. 14, 270, 272).

[12] Donald E. Knuth. *The Art of Computer Programming, Volume 1: Fundamental Algorithms*. 3rd Edition. Redwood City, CA, USA: Addison Wesley, 1997. ISBN: 978-0-201-89683-1 (geciteerd op p. 160).

[13] Donald E. Knuth. *The Art of Computer Programming, Volume 2: Seminumerical Algorithms*. 3rd Edition. Boston, MA, USA: Addison-Wesley, 1997. ISBN: 978-0-201-89684-8 (geciteerd op p. 160).

[14] Donald E. Knuth. *The Art of Computer Programming, Volume 3: Sorting and Searching*. 2nd Edition. Redwood City, CA, USA: Addison Wesley, 1998. ISBN: 978-0-201-89685-5 (geciteerd op pp. 160, 161).

[15] Donald E. Knuth en Ronald W. Moore. “An Analysis of Alpha-Beta Pruning.” In: *Artif. Intell.* 6.4 (1975), p. 293–326 (geciteerd op pp. 257, 258).

[16] Barbara H. Liskov en Jeannette M. Wing. “A Behavioral Notion of Subtyping”. In: *ACM Trans. Program. Lang. Syst.* 16.6 (nov 1994), p. 1811–1841. ISSN: 0164-0925 (geciteerd op p. 111).

[17] Scott Meyers. *Effective C++: 55 Specific Ways to Improve Your Programs and Designs*. 3rd Edition. Addison-Wesley, 2005. ISBN: 978-0-321-33487-9 (geciteerd op p. 295).

[18] Scott Meyers. *More Effective C++: 35 New Ways to Improve Your Programs and Designs*. Boston, MA, USA: Addison-Wesley, 1995. ISBN: 978-0-201-63371-9 (geciteerd op p. 295).

[19] Robert Sedgewick. *Algorithms in C++, Parts 1-4: Fundamentals, Data Structure, Sorting, Searching*. 3rd Edition. Addison-Wesley, 1998. ISBN: 978-0-7686-8533-6 (geciteerd op p. 161).

[20] Bjarne Stroustrup. *A Tour of C++*. 3rd Edition. Boston, MA, USA: Addison-Wesley Professional, 2022. ISBN: 978-0-13-681648-5. URL: https://www.stroustrup.com/tour3.html (geciteerd op p. 14).

[21] Bjarne Stroustrup. *Principles and Practice Using C++*. 3rd Edition. Boston, MA, USA: Addison-Wesley, 2024. ISBN: 978-0-13-830868-1. URL: https://www.stroustrup.com/tour3.html (geciteerd op p. 14).

[22] Bjarne Stroustrup. *The C++ Programming Language*. Special Edition. Addison-Wesley, 2000. ISBN: 978-0-201-70073-2. URL: http://my.safaribooksonline.com/book/programming/cplusplus/0201700735/a-tour-of-cplusplus/ch02lev1sec1 (geciteerd op p. 19).

[23] Bjarne Stroustrup. *The C++ Programming Language*. 4th Edition. Boston, MA, USA: Addison-Wesley, 2013. ISBN: 978-0-321-56384-2 (geciteerd op pp. 156, 295, 347, 371).

[24] David Vandevoorde, Nicolai M. Josuttis en Douglas Gregor. *C++ Templates – The Complete Guide*. 2nd Edition. Boston, MA, USA: Addison-Wesley, 2017. ISBN: 978-0-321-71412-1. URL: http://www.tmplbook.com (geciteerd op p. 347).

[25] Mark A. Weiss. *Data Structures and Problem Solving Using C++*. 2nd Edition. Addison Wesley, 1999. ISBN: 978-0-201-61250-9 (geciteerd op pp. 265, 266).

[26] Niklaus Wirth. *Algorithms + Data Structures = Programs*. Upper Saddle River, NJ, USA: Prentice Hall, 1978. ISBN: 978-0-13-022418-7 (geciteerd op p. 160).

[27] Ed Yourdon. *Just Enough Structured Analysis*. 2019. URL: http://zimmer.csufresno.edu/~sasanr/Teaching-Material/SAD/JESA.pdf (geciteerd op p. 48).