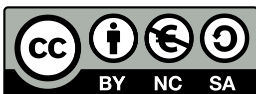
Algoritmen en Datastructuren

DE HAAGSE HOGESCHOOL

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



Algoritmen en Datastructuren van 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 het gebruik van algoritmen en datastructuren in C++, geschreven in WordPerfect en inmiddels uitgegroeid tot dit met LaTeX opgemaakte document. In de tussentijd hebben veel studenten en collega's mij feedback gegeven. De opbouwende kritiek van Harm Jongsma, Henk van den Bosch, John Visser en vele anderen hebben dit dictaat ontegenzeggelijk beter gemaakt. Desondanks heb ik niet de illusie dat dit dictaat foutvrij is. Op- en aanmerkingen zijn dus nog altijd welkom, mail me op J.Z.M.Broeders@hhs.nl.

Alle programmacode in dit dictaat is getest met Microsoft Visual Studio 2013[1] en GCC[2] versie 4.9.2. Op `http://bd.eduweb.hhs.nl/algods/` 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 het gebruik van algoritmen en datastructuren in C++ 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.

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

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

Inhoudsopgave

Inleiding 1

1 Analyse van algoritmen 5

2 Datastructuren 7

3 Gebruik van een stack 9

3.1 Haakjes balanceren 10
3.2 Eenvoudige rekenmachine 11
3.2.1 Postfix notatie 11
3.2.2 Een postfix calculator 12
3.2.3 Een infix calculator 14

4 Implementaties van een stack 19

4.1 Stack met behulp van een array 19
4.2 Stack met behulp van een gelinkte lijst 21
4.3 Stack met array versus stack met gelinkte lijst 23
4.4 Dynamisch kiezen voor een bepaald type stack 24

5 Advanced C++ 27

6 De ISO/ANSI standaard C++ library 29

6.1 Voorbeeldprogramma met `std::string` 29
6.2 Containers 31
6.2.1 Sequentiële containers 32
6.2.1.1 Voorbeeldprogramma met `std::vector` 34
6.2.2 Container adapters 38
6.2.2.1 Voorbeeldprogramma met `std::stack` 39
6.2.3 Associatieve containers 40
6.2.3.1 Voorbeeldprogramma met `std::set` 42
6.2.3.2 Voorbeeldprogramma met `std::multiset` (bag) 43
6.2.3.3 Voorbeeldprogramma met `std::unordered_set` 44
6.2.3.4 Voorbeeldprogramma met `std::map` 46
6.3 Iteratoren 47
6.3.1 Voorbeeldprogramma's met verschillende iterator soorten 50
6.3.2 Reverse-iteratoren 52
6.3.3 Insert-iteratoren 53

6.3.4 Stream-iteratoren 53
6.3.5 Voorbeeldprogramma's met iteratoren 54
6.4 Algoritmen 55
6.4.1 Zoeken, tellen, testen, bewerken en vergelijken 56
6.4.1.1 Voorbeeldprogramma met std::find 57
6.4.1.2 Voorbeeldprogramma met std::find_if 59
6.4.1.3 Voorbeeldprogramma met std::find_first_of 61
6.4.1.4 Voorbeeldprogramma met std::for_each 61
6.4.2 Lambda-functies 63
6.4.2.1 Closure 63
6.4.2.2 Returntype van een lambda-functie 64
6.4.2.3 Lambda-functieparametertype met auto 65
6.4.3 Kopiëren, bewerken, vervangen, roteren, schudden, verwisselen, verwijderen en vullen 65
6.4.3.1 Voorbeeldprogramma met std::transform 67
6.4.3.2 Voorbeeldprogramma met std::remove 69
6.4.3.3 Voorbeeldprogramma met std::generate_n 70
6.4.4 Sorteren en bewerkingen op gesorteerde ranges 70
6.4.4.1 Voorbeeldprogramma met std::sort 71
6.4.4.2 Voorbeeldprogramma met std::includes 73
6.4.4.3 Voorbeeldprogramma dat generiek en objectgeoriënteerd programmeren combineert 75
6.4.5 Automatisch de meest efficiënte implementatie kiezen afhankelijk van de beschikbare iterator soort 76
6.4.6 Standaard functie-objecten 78

7 Toepassingen van datastructuren **81**
7.1 Boter, Kaas en Eieren 81
7.1.1 Minimax algoritme 81
7.1.2 Alpha-beta pruning 92
7.1.3 Verschillende implementaties van Boter, Kaas en Eieren 99
7.2 Het zoeken van het kortste pad in een graph 100

Inleiding

Dit dictaat is bestemd voor studenten die de module ALGODS (Algoritmen en Datastructuren) volgen als onderdeel van de verdiepende minor Elektrotechniek die wordt aangeboden door de opleiding Elektrotechniek van De Haagse Hogeschool. Het kan worden gebruikt in combinatie met de boeken: *Thinking in C++, Volume 1: Introduction to Standard C++* [3] en *Thinking in C++, Volume 2: Standard Libraries & Advanced Topics* [4] van Eckel. Beide boeken zijn gratis te downloaden van `http://www.mindview.net/Books/TICPP/ThinkingInCPP2e.html`. Deze boeken zijn wel enigszins verouderd dus voor gedetailleerde informatie verwijs ik je naar `http://en.cppreference.com/w/cpp`. Dit dictaat is bedoeld als primair studiemateriaal. Het overige studiemateriaal kun je vinden op `http://bd.eduweb.hhs.nl/algods/`.

Dit dictaat is zoals alle mensenwerk niet foutloos, verbeteringen en suggesties zijn altijd welkom!

Bij GESPRG en MICPRG heb je kennis gemaakt met de *statische* datastructuren array en struct en bij OGOPRG heb je al kort kennis gemaakt met *dynamische* datastructuren. De eerste hogere programmeertalen, zoals FORTRAN (1957) en ALGOL (1958) hadden al 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 aantalDeelnemers;
    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 array's `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 `aan` `talDeelnemers` (is statisch).
- Het verwijderen van een `deelnemer` uit de `stand` is een heel bewerkelijke operatie.[3]

Bij OGOPRG heb je geleerd dat je 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. De array met 100 deelnemers genaamd `lijst` kan 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(const string& n, int p);
    int punten() const;
    const string& naam() const;
    void verhoogpunten(int p);
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 aantalDeelnemers() const;
private:
    vector<Deelnemer> lijst;
};
```

[3] 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`.

```
Stand s;
```

De lijst met deelnemers kan in C ook dynamisch gemaakt worden 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 [7, 8, 9]. Er zijn in de loop der jaren misschien wel duizend 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 [14].

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++ standaard opgenomen. Deze bibliotheek werd eerst de STL (Standard Template Library) genoemd maar sinds deze library in de C++ standaard is opgenomen spreken we over de standaard C++ library. Deze library is in de één na laatste versie van de C++ standaard[4] C++11 behoorlijk uitgebreid. Microsoft Visual Studio 2012 Express for Windows Desktop (de C++ compiler die wij ge-

[4] De laatste versie van de C++ standaard C++14 [6] [5] bevat slecht kleine aanpassingen en verbeteringen. Zie `http://en.wikipedia.org/wiki/C++14`.

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

bruiken) ondersteunt de meeste van deze uitbreidingen (maar niet alle). In dit dictaat gaan we uitgebreid op deze standaard C++ library in.

Deze module worden afgesloten met het behandelen van een aantal applicaties waarin gebruik gemaakt wordt van standaard datastructuren. Hierbij wordt gebruik gemaakt van UML [12] om de werking van de gegeven C++ programma's uit te leggen.

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. We zullen bijvoorbeeld spreken over een algoritme met een executietijd $O(n^2)$. De $O(\ldots)$ notatie wordt *Big-O-notatie* genoemd en uitgesproken als “van de orde ...”. $O(n^2)$ betekent dat de executietijd van het algoritme rechtevenredig toeneemt met het kwadraat van het aantal data-elementen (n). In tabel 1.1 kun je zien hoe een algoritme van een 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 1.1: 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 executietijd $O(n^2)$ niet bruikbaar is voor grote hoeveelheden data en dat een algoritme met een executietijd $O(n^3)$ of $O(10^n)$ sowieso niet bruikbaar is. De meest voor de hand liggende sorteermethoden[6] (insertion sort, bubble sort enz.) blijken allemaal een executietijd $O(n^2)$ te hebben. Deze algoritmen zijn dus voor grotere datasets absoluut onbruikbaar! Al in 1962 heeft Hoare het zogenaamde Quicksort algoritme [5] ontworpen dat gemiddeld een executietijd $O(n \cdot \log n)$ heeft. In elk boek over algoritmen en datastructuren kun je een implementatie van dit algoritme vinden. Maar op zich is

[6] In veel inleidende boeken over programmeertalen worden dit soort sorteeralgoritmen als voorbeeld gebruikt.

dat niet zo interessant want de standaard C++ library heeft een efficiënte sorteermethode met een executietijd $O(n \cdot \log n)$. 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* [9] van Knuth. Een meer toegankelijk boek is *Algorithms in C++, Parts 1-4: Fundamentals, Data Structure, Sorting, Searching* [11] van Sedgewick. Zeker als de data aan bepaalde voorwaarden voldoet (als 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.

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 2.1 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++ standaard 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 2.1: 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)

Zie vervolg op volgende pagina.

Tabel 2.1: 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[7] , 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[7]) + 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

[7] 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.

3

Gebruik van een stack

Als voorbeeld zullen we in dit hoofdstuk enkele toepassingen van de bekende datastructuur *stack* bespreken. 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.h[8]:

```
#ifndef _TISD_Bd_Stack_
#define _TISD_Bd_Stack_

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;
};

#endif
```

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++11 kun je automatisch gegenereerde memberfuncties verwijderen door er `= delete` achter te plaatsen. De `operator=` en de copy constructor van de class `Stack` zijn

[8] In de ISO/ANSI C++ standaard library is ook een type stack gedefinieerd, zie hoofdstuk 6 op pagina 29.

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. Sinds C++11 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 [1, paragraaf 5.4]), door de toevoeging = default genereert de compiler zelf de default implementatie van deze destructor.

3.1 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 StackWithList<T>, zie stacklist.h, die is afgeleid van de hierboven gedefinieerde Stack<T>.

```
// Gebruik een stack voor het controleren van de haakjes
// Invoer afsluiten met .

#include <iostream>
#include "stacklist.h"
using namespace std;

int main() {
    StackWithList<char> s;
    char c;
    cout << "Type een expressie met haakjes () [] of {} en sluit ↩
     ↪ af met ." << endl;
    cin.get(c);
    while (c != '.') {
        if (c == '(' || c == '{' || c == '[') {
            s.push(c);
        }
        else {
            if (c == ')' || c == '}' || c == ']') {
                if (s.empty()) {
                    cout << "Fout " << c << " bijbehorend haakje ↩
                     ↪ openen ontbreekt." << endl;
                }
                else {
                    char d = s.top();
                    s.pop();
                    if (d == '(' && c != ')' || d == '{' && c != ↩
                     ↪ '}' || d == '[' && c != ']') {
```

```
                        cout << "Fout " << c << " bijbehorend ↩
                         ↪ haakje openen ontbreekt." << endl;
                    }
                }
            }
        }
        cin.get(c);
    }
    while (!s.empty()) {
        char d = s.top();
        s.pop();
        cout << "Fout " << d << " bijbehorend haakje sluiten ↩
         ↪ ontbreekt." << endl;
    }
    cin.get();
    cin.get();
    return 0;
}
```

3.2 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.

3.2.1 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 t.o.v. 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[9]) de RPN notatie. Tot op de dag van vandaag bieden veel rekenmachines van HP nog steeds de mogelijkheid om formules in RPN notatie in te voeren.

3.2.2 Een postfix calculator

Een postfix expressie kan met het algoritme dat gegeven is in figuur 3.1 worden uitgerekend. Dit algoritme maakt gebruik van een stack.

We zullen nu eerst een postfix calculator maken. Later zul 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.

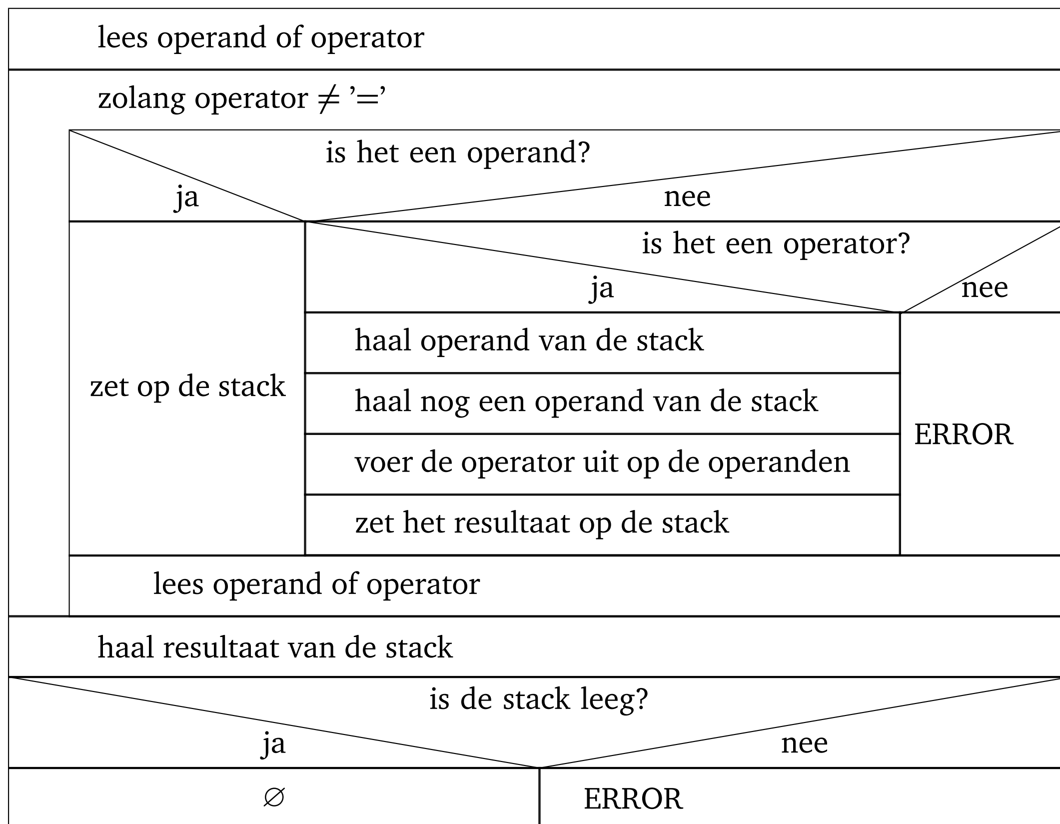
Vraag:

Implementeer nu zelf het algoritme uit figuur 3.1. Je mag jezelf beperken tot optellen en vermenigvuldigen.

Antwoord Postfix.cpp:

```
#include <cctype>
```

[9] Zie: http://en.wikipedia.org/wiki/HP-35.



Figuur 3.1: Algoritme voor het uitrekenen van een postfix expressie.

```
#include "stacklist.h"
using namespace std;

int main() {
    StackWithList<int> s;
    char c;
    cout << "Type een postfix expressie (met + en * operator) en ↩
    ↪ sluit af met =" << endl;
    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 {
            cout << "Syntax error" << endl;
        }
        cin >> c;
    }
    cout << "= " << s.top() << endl;
    s.pop();
    if (!s.empty()) {
        cout << "Fout operator ontbreekt." << endl;
    }
    cin.get();
    cin.get();
    return 0;
}
```

3.2.3 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[10] (shunting-yard algorithm), gepubliceerd [2] 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.

In tabel 3.1 kun je zien hoe de expressie:

[10] Zie http://en.wikipedia.org/wiki/Shunting-yard_algorithm

```
12 + (40 / (23 - 3)) * 2 =
```

omgezet wordt in de postfix expressie:

```
12 40 23 3 - / 2 * + =
```

Tabel 3.1: Voorbeeld van een conversie van in- naar postfix.

gelezen karakter(s)	stack	uitvoer
12		12
+	+	12
(	+ (	12
40	+ (	12 40
/	+ (/	12 40
(	+ (/ (	12 40
23	+ (/ (	12 40 23
-	+ (/ (-	12 40 23
3	+ (/ (-	12 40 23 3
)	+ (/	12 40 23 3 -
)	+	12 40 23 3 - /
*	+ *	12 40 23 3 - /
2	+ *	12 40 23 3 - / 2
=		12 40 23 3 - / 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 Infix2Postfix.cpp:

```
#include <iostream>
#include <cctype>
#include "stacklist.h"
using namespace std;

bool hasLowerPrio(char op1, char op2) {
    // geeft true als prio(op1) < prio(op2)
    // eenvoudig omdat alleen + en * toegestaan zijn
    return op1 == '+' && op2 == '*';
}

int main() {
```

```
    StackWithList<char> s;
    char c;
    cout << "Type een infix expressie (met + en * operator) en ↩
     ↪ sluit af met =" << endl;
    cin >> c;
    while (c != '=') {
        if (isdigit(c)) {
            cin.putback(c);
            int i;
            cin >> i;
            cout << i << " ";
        }
        else if (c == '(') {
            s.push(c);
        }
        else if (c == '+' || c == '*') {
            while (!s.empty() && s.top() != '(' && ↩
             ↪ !hasLowerPrio(s.top(), c)) {
                cout << s.top() << " ";
                s.pop();
            }
            s.push(c);
        }
        else if (c == ')') {
            while (s.top() != '(') {
                cout << s.top() << " ";
                s.pop();
            }
            s.pop();
        }
        else {
            cout << "Syntax error" << endl;
        }
        cin >> c;
    }
    while (!s.empty()) {
        cout << s.top() << " ";
        s.pop();
    }
    cin.get();
    cin.get();
    return 0;
}
```

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:

```
#include <iostream>
#include <cctype>
#include "stacklist.h"
using namespace std;

bool hasLowerPrio(char op1, char op2) {
    return op1 == '+' && op2 == '*';
}

void processOperator(StackWithList<char>& s1, StackWithList<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() {
    StackWithList<char> s1;
    StackWithList<int> s2;
    char c;
    cout << "Type een infix expressie (met + en * operator) en ↩
      ↪ sluit af met =" << endl;
    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() != '(' && ↩
              ↪ !hasLowerPrio(s1.top(), c)) {
                processOperator(s1, s2);
            }
            s1.push(c);
        }
        else if (c == ')') {
            while (s1.top() != '(') {
                processOperator(s1, s2);
```

```
            }
            s1.pop();
        }
        else {
            cout << "Syntax error" << endl;
        }
        cin >> c;
    }
    while (!s1.empty()) {
        processOperator(s1, s2);
    }
    cout << "= " << s2.top() << endl;
    s2.pop();
    if (!s2.empty()) {
        cout << "Fout operator ontbreekt." << endl;
        s2.pop();
    }
    cin.get();
    cin.get();
    return 0;
}
```

Het is natuurlijk netter om het geheel in een class `Calculator` in te kapselen zodat de calculator eenvoudig kan worden (her)gebruikt. Zie InfixCalc.cpp.

4

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 zal in de les beide implementaties bespreken.

4.1 Stack met behulp van een array

Inhoud van de file stackarray.h:

```
#ifndef _TISD_Bd_StackWithArray_
#define _TISD_Bd_StackWithArray_

#include <iostream>
#include <cstdlib>
#include "stack.h"

template <typename T> class StackWithArray: public Stack<T> {
public:
    explicit StackWithArray(size_t size);
    ~StackWithArray();
    virtual void push(const T& t) override;
    virtual void pop() override;
    virtual const T& top() const override;
    virtual bool empty() const override;
    virtual bool full() const override;
private:
    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> StackWithArray<T>::StackWithArray(size_t ↩
 ↪ size): a(0), s(size), i(0) {
```

```
    if (s == 0) {
        std::cerr << "Stack size should be >0" << std::endl;
        s = 0;
    }
    else {
        a = new T[s];
    }
}

template <typename T> StackWithArray<T>::~StackWithArray() {
    delete[] a;
}

template <typename T> void StackWithArray<T>::push(const T& t) {
    if (full()) {
        std::cerr << "Can't push on an full stack" << std::endl;
    }
    else {
        a[i++] = t;
    }
}

template <typename T> void StackWithArray<T>::pop() {
    if (empty()) {
        std::cerr << "Can't pop from an empty stack" << std::endl;
    }
    else {
        --i;
    }
}

template <typename T> const T& StackWithArray<T>::top() const {
    if (empty()) {
        std::cerr << "Can't top from an empty stack" << std::endl;
        std::exit(-1);
        // no valid return possible
    }
    return a[i - 1];
}

template <typename T> bool StackWithArray<T>::empty() const {
    return i == 0;
}

template <typename T> bool StackWithArray<T>::full() const {
    return i == s;
}

#endif
```

Het programma stackarray.cpp om deze implementatie te testen:

```
#include <iostream>
#include "stackarray.h"
using namespace std;

int main() {
    StackWithArray<char> s(32);
    char c;
    cout << "Type een tekst en sluit af met ." << endl;
    cin.get(c);
    while (c != '.') {
        s.push(c);
        cin.get(c);
    }
    while (!s.empty()) {
        cout << s.top();
        s.pop();
    }
    cin.get();
    cin.get();
    return 0;
}
```

Sinds C++11 kun je expliciet aangeven dat je een memberfunctie wilt overridden. Dit kun je doen door `override` achter de betreffende memberfunctie te plaatsen. Dit voorkomt het per ongeluk overloaden van memberfuncties (door bijvoorbeeld en typefout). In de bovenstaande class `StackWithArray` is dit toegepast. Zie eventueel [1, paragraaf 4.9].

4.2 Stack met behulp van een gelinkte lijst

Inhoud van de file stacklist.h:

```
#ifndef _TISD_Bd_StackWithList_
#define _TISD_Bd_StackWithList_
template <typename T> class StackWithList: public Stack<T> {
public:
    StackWithList();
    virtual ~StackWithList() throw();
    virtual void push(const T& t) override;
    virtual void pop() override;
    virtual const T& top() const override;
    virtual bool empty() const override;
    virtual 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> StackWithList<T>::StackWithList(): p(0) {
}

template <typename T> StackWithList<T>::~StackWithList() throw() {
    while (!empty())
        pop();
}

template <typename T> void StackWithList<T>::push(const T& t) {
    p = new Node(t, p);
}

template <typename T> void StackWithList<T>::pop() {
    if (empty())
        std::cerr << "Can't pop from an empty stack" << std::endl;
    else {
        Node* old(p);
        p = p->next;
        delete old;
    }
}

template <typename T> const T& StackWithList<T>::top() const {
    if (empty()) {
        std::cerr << "Can't top from an empty stack" << std::endl;
        std::cin.get();
        std::cin.get();
        std::exit(-1);
        // no valid return possible
    }
    return p->data;
}

template <typename T> bool StackWithList<T>::empty() const {
    return p == 0;
}

template <typename T> bool StackWithList<T>::full() const {
    return false;
}

template <typename T> StackWithList<T>::Node::Node(const T& t, ↩
 ↪ Node* n):
```

```
        data(t), next(n) {
}

#endif
```

Het programma stacklist.cpp om deze implementatie te testen:

```
#include <iostream>
#include "stacklist.h"
using namespace std;

int main() {
    StackWithList<char> s;
    char c;
    cout << "Type een tekst en sluit af met ." << endl;
    cin.get(c);
    while (c != '.') {
        s.push(c);
        cin.get(c);
    }
    while (!s.empty()) {
        cout << s.top();
        s.pop();
    }
    cin.get();
    cin.get();
    return 0;
}
```

4.3 Stack met array versus stack met gelinkte lijst

Beide implementaties `StackWithArray` en `StackWithList` hebben voor- en nadelen. Het geheugengebruik van `StackWithArray` is statisch (bij het aanmaken moet je de maximale grootte opgeven) terwijl het geheugengebruik van `StackWithList` dynamisch is. De `StackWithArray` 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 `StackWithList` 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 `StackWithArray` 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. Een `StackWithArray` heeft dus een overhead van maximaal 100% en minimaal 0%. De `StackWithList` gebruikt, als de stack niet leeg is, meer geheugen dan eigenlijk nodig is. Voor elk element dat op de stack staat moet ook een pointer in het geheugen worden opgeslagen. De overhead is afhankelijk van de grootte van het data-element en de grootte van een pointer. Op een 64 bits machine (waar een pointer acht bytes groot is) heeft een `StakWithList` van `char`'s (die één byte groot zijn) dus een overhead van maar liefst 800%. Een `StackWithList` met objecten die

1 Kbyte groot zijn heeft slechts een overhead van 0,78%. We kunnen dus concluderen dat een StackWithArray minder overhead heeft voor een stack met kleine objecten en dat een StackWithList minder overhead heeft voor een stack met grote objecten.

4.4 Dynamisch kiezen voor een bepaald type stack

We kunnen de keuze voor het type stack ook aan de gebruiker overlaten! Dit is een uitstekend voorbeeld van het gebruik van polymorfisme, zie stack.cpp.

```
#include <iostream>
#include <cassert>
#include "stacklist.h"
#include "stackarray.h"
using namespace std;

int main() {
    Stack<char>* s = 0;

    cout << "Welke stack wil je gebruiken (l = list, a = array): ";
    char c;
    do {
        cin.get(c);
        if (c == 'l' || c == 'L') {
            s = new StackWithList<char>;
        }
        else if (c == 'a' || c == 'A') {
            cout << "Hoeveel elementen wil je gebruiken: ";
            int i;
            cin >> i;
            s = new StackWithArray<char>(i);
        }
    } while (s == 0);

    cout << "Type een tekst en sluit af met ." << endl;
    cin.get(c);
    while (c != '.') {
        s->push(c);
        cin.get(c);
    }
    while (!s->empty()) {
        cout << s->top();
        s->pop();
    }
    delete s;

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

```
}
```

In de praktijk zal het niet vaak voorkomen 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.

5

Advanced C++

Er zijn nog enkele geavanceerde onderwerpen uit C++ die nog niet zijn behandeld die wel van belang zijn bij het gebruiken (en implementeren) van een library met herbruikbare algoritmen en datastructuren. Dit zijn de onderwerpen:

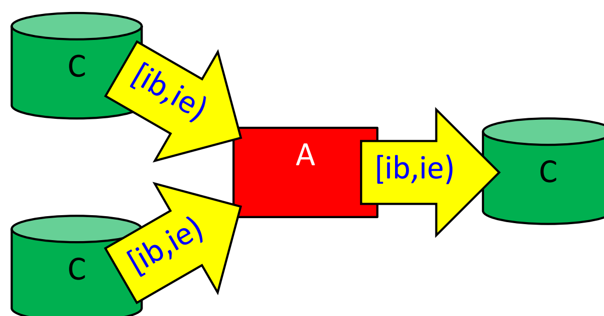
- namespaces;
- exceptions;
- RTTI (Run Time Type Information).

Namespaces maken het mogelijk om verschillende class libraries waarin dezelfde class namen voorkomen toch met elkaar te combineren. Exceptions maken het mogelijk om op een gestructureerde manier met onverwachte omstandigheden en fouten in programma's om te gaan. RTTI (Run Time Type Information) maakt het mogelijk om tijdens run time het type (de class) van een object te achterhalen. Deze onderwerpen staan beschreven in paragraaf 6.2, 6.8 en 6.9 van het dictaat *Objectgeoriënteerd Programmeren in C++* [1] dat je bij OGOPRG hebt gebruikt. Gezien de beperkte tijd worden deze onderwerpen niet behandeld bij deze module maar wel als huiswerk opgegeven.

6

De ISO/ANSI standaard C++ library

De standaard C++ library bevat: *containers*, *iteratoren* en *algoritmen*. De in hoofdstuk 2 op pagina 7 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 (*range*) aan waar het betreffende algoritme op wordt uitgevoerd. Een iterator wijst naar het eerste data-element van de range en de andere iterator wijst naar de locatie net voorbij het laatste element van de range. De iteratoren zorgen ervoor dat de algoritmen generiek gehouden kunnen worden en dus op verschillende soorten containers uitgevoerd kunnen worden. In figuur 6.1 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 6.1: De samenhang tussen containers, iteratoren en algoritmen. A = Algoritme, C = Container, i = iterator, b = begin, e = end, [ib,ie) = range van ib tot ie.

6.1 Voorbeeldprogramma met `std::string`

Het voorbeeldprogramma stdstring.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[] = "Henk";
for (const char* p = naam; *p != '\0'; ++p) {
    cout << *p << " ";
}
```

In C++11 kun je het keyword auto gebruiken om de compiler zelf het type van de pointer te laten bepalen.

```
char naam[] = "Henk";
for (const auto* p = naam; *p != '\0'; ++p) {
    cout << *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.begin(); i != naam.end(); ↩
 ↪ ++i) {
    cout << *i << " ";
}
```

Met behulp van een string::const_iterator kun je de karakters uit de string alleen lezen. 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.

In C++11 kun je 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) {
    cout << *i << " ";
}
```

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.

6.2 Containers

De standaard C++ library bevat verschillende containers. Een container waarmee je bij het vak OGOPRG al hebt kennisgemaakt is de std::vector [1, paragraaf 3.8]. 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 [1, paragraaf 4.12].

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:

- [] (een gewone C-array)
- string
- array (sinds C++11)
- vector
- forward_list (single linked list, sinds C++11)
- list (double linked list)
- deque (double ended queue, had eigenlijk double ended vector moeten heten)
- bitset (wordt in dit dictaat verder niet besproken)

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

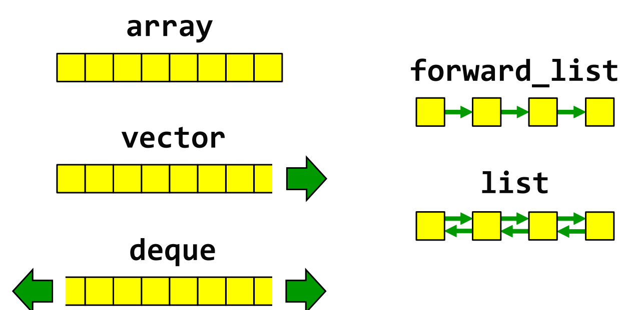
De standaard C++ library bevat de volgende associatieve containers:

- set
- multiset
- map
- multimap

- unordered_set (sinds C++11)
- unordered_multiset (sinds C++11)
- unordered_map (sinds C++11)
- unordered_multimap (sinds C++11)

6.2.1 Sequentiële containers

In figuur 6.2 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 [1, paragraaf 3.7]. 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 [1, paragraaf 3.8]. 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 6.2: Overzicht van de belangrijkste sequentiële containers.

In tabel 6.1 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

Tabel 6.1: 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	$O(1)$

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[11] 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[12] . 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[12].

Alle sequentiële containers uit tabel 6.1 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 worden

[11] 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.

[12] Dit is, volgens de C++ standaard, implementatie afhankelijk.

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.

6.2.1.1 Voorbeeldprogramma met `std::vector`

Dit voorbeeldprogramma stdvector.cpp laat zien:

- hoe een standaard vector gevuld kan worden;
- hoe door (een deel van) de vector heengelopen kan worden door middel van indexering[13];
- hoe door (een deel van) de vector heengelopen kan worden door middel van een *iterator*[14];
- hoe (in C++11) 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.

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

// Afdrukken van een vector door middel van indexering.
void print1(const vector<int>& vec) {
    cout << "De inhoud van de vector is:" << endl;
    for (vector<int>::size_type index = 0; index != vec.size(); ↩
     ↪ ++index) {
        cout << vec[index] << " ";
    }
    cout << endl;
}


// Afdrukken van een vector door middel van indexering met decltype.
void print2(const vector<int>& vec) {
```

[13] In C++11 kun je hierbij gebruik maken van een `decltype` om het type van de index variabele te definiëren.

[14] In C++11 kun je hierbij gebruik maken van een zogenoemde *auto-typed* variabele. Deze variabele wordt geïnitialiseerd met `cbegin()`. De memberfunctie geeft een `const_iterator` naar het eerste element in de container terug.

```
    cout << "De inhoud van de vector is:" << endl;
    for (decltype(vec.size()) index = 0; index != vec.size(); ↩
     ↪ ++index) {
        cout << vec[index] << " ";
    }
    cout << endl;
}

// Afdrukken van een vector door middel van iterator.
void print3(const vector<int>& vec) {
    cout << "De inhoud van de vector is:" << endl;
    for (vector<int>::const_iterator iter = vec.begin(); iter != ↩
     ↪ vec.end(); ++iter) {
        cout << *iter << " ";
    }
    cout << endl;
}

// Afdrukken van een vector door middel van iterator met auto.
void print4(const vector<int>& vec) {
    cout << "De inhoud van de vector is:" << endl;
    for (auto iter = vec.cbegin(); iter != vec.cend(); ++iter) {
        cout << *iter << " ";
    }
    cout << endl;
}

// Afdrukken van een vector door middel van range-based for.
void print5(const vector<int>& vec) {
    cout << "De inhoud van de vector is:" << endl;
    for (auto elm : vec) {
        cout << elm << " ";
    }
    cout << endl;
}

// 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;
    cout << "Geef een aantal getallen (afgesloten door een 0):" << ↩
    ↪ endl;
    cin >> i;
    while (i != 0) {
        v.push_back(i);
        cin >> i;
    }
    print1(v);
    print2(v);
    print3(v);
    print4(v);
    print5(v);
    cout << "Het gemiddelde is: " << gem1(v) << endl;
    cout << "Het gemiddelde is: " << gem2(v) << endl;
    cout << "Nu wordt een deel van de vector bewerkt." << endl;
    if (v.size() >= 4) {
        for (auto iter = v.begin() + 2; iter != v.begin() + 4; ↩
        ↪ ++iter) {
            *iter *= 2;
        }
    }
    print3(v);
    cout << "Nu wordt de vorige bewerking weer teruggedraaid." << ↩
    ↪ endl;
    if (v.size() >= 4) {
        for (decltype(v.size()) i = 2; i < 4; ++i) {
            v[i] /= 2;
        }
    }
    print5(v);
    cin.get();
    cin.get();
    return 0;
}
```

In het bovenstaande programma worden maar liefst 5 verschillende manieren gedemonstreerd om een `vector<int>` af te drukken. De eerste twee implementaties `print1` en `print2` 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 `print3` en `print4` gebruiken een iterator om de elementen in de vector te benaderen. De laatste implementatie `print5` gebruikt een range-based for om de elementen in de vector te benaderen en is het eenvoudigst.

Geïnspireerd door dit voorbeeld zou je een generieke `print` functie kunnen bedenken waarmee elk willekeurige container afgedrukt kan 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 generiekPrint-iterator.cpp:

```
template<typename C> void print(const C& c) {
    cout << "De inhoud van de container is:" << endl;
    for (auto iter = c.cbegin(); iter != c.cend(); ++iter) {
        cout << *iter << " ";
    }
    cout << endl;
}
```

Het is, ook in dit geval, eenvoudiger om gebruik te maken van een range-based for, zie generiekPrint-range-based-for.cpp:

```
template<typename C> void print(const C& c) {
    cout << "De inhoud van de container is:" << endl;
    for (const auto& e: c) {
        cout << e << " ";
    }
    cout << endl;
}
```

In de bovenstaande functie is een `const auto&` gebruikt om een onnodig kopietje te voorkomen. We weten immers niet hoe groot de elementen in de container zijn.

Bovenstaande twee generieke `print` functies drukken altijd de inhoud van de gehele container af. Als we ook een gedeelte van een container af willen kunnen drukken met de generieke `print` functie, dan moeten we de functie twee iteratoren meegeven, zie generiekPrint-range.cpp:

```
template <typename Iter> void print(Iter begin, Iter end) {
    cout << "De inhoud van de container is:" << endl;
    for (Iter iter = begin; iter != end; ++iter) {
        cout << *iter << " ";
    }
```

```
    cout << endl;
}
```

De eerste iterator `begin` wijst naar het eerste element dat moet worden afgedrukt en de tweede iterator `end` wijst net voorbij het laatste element dat moet worden afgedrukt. 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` afgedrukt. In de wiskunde zouden we dat noteren als het halfopen interval `[i1,i2)`. In C++ wordt zo'n halfopen interval van twee iteratoren een *range* genoemd. Alle algoritmen uit de standaard C++ library gebruiken ranges als parameters zodat ze ook op een deel van een container uitgevoerd kunnen worden.

De bovenstaande generieke `print` kan gebruikt worden om de vector `v` volledig af te drukken:

```
print(v.cbegin(), v.cend());
```

Dezelfde functie kan ook gebruikt worden om alle elementen van de vector `v` af te drukken *behalve* de eerste en de laatste:

```
print(v.cbegin() + 1, v.cend() - 1);
```

6.2.2 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 4.4 op pagina 24 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 6.2, 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[15] 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 6.1.

6.2.2.1 Voorbeeldprogramma met std::stack

Het onderstaande voorbeeldprogramma stdbalanced.cpp is identiek aan het programma uit paragraaf 3.1 op pagina 10 maar in plaats van de zelfgemaakte `StackWithList` wordt nu de standaard `stack` gebruikt.

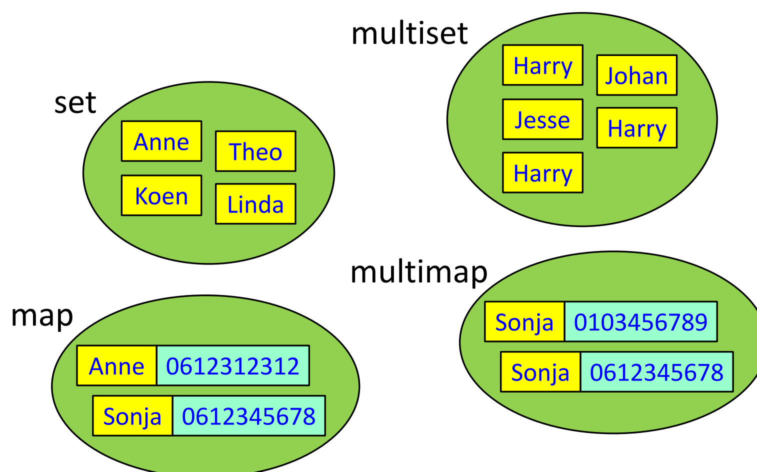
```
// Controleer op gebalanceerde haakjes. Algoritme wordt
// besproken in de les. Invoer afsluiten met een punt.

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

int main() {
    stack<char> s;
    char c;
    cout << "Type een expressie met haakjes () [] of {} en sluit ↩
     ↪ af met ." << endl;
    cin.get(c);
    while (c != '.') {
        if (c == '(' || c == '{' || c == '[') {
            s.push(c);
        }
        else {
            if (c == ')' || c == '}' || c == ']') {
                if (s.empty()) {
                    cout << "Fout " << c << " bijbehorend haakje ↩
                     ↪ openen ontbreekt." << endl;
                }
                else {
                    char d = s.top();
                    s.pop();
                    if (d == '(' && c != ')' || d == '{' && c != ↩
                     ↪ '}' || d == '[' && c != ']') {
                        cout << "Fout " << c << " bijbehorend ↩
                         ↪ haakje openen ontbreekt." << endl;
                    }
```

[15] 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`. In de les wordt dit verder uitgelegd.

```
                }
            }
        }
        cin.get(c);
    }
    while (!s.empty()) {
        char d = s.top();
        s.pop();
        cout << "Fout " << d << " bijbehorend haakje sluiten ↩
          ↪ ontbreekt." << endl;
    }
    cin.get();
    cin.get();
    return 0;
}
```

6.2.3 Associatieve containers

In een sequentiële container blijft de volgorde van de elementen ongewijzigd, zie paragraaf 6.2.1 op pagina 32. 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 6.3 zijn de belangrijkste associatieve containers conceptueel weergegeven.



Figuur 6.3: 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 6.3 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?[16]. Deze sleutels worden zodanig in de `set` opgeslagen dat je snel kunt opzoeken of een sleutel zich in de `set` bevindt.

[16] 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 6.3 bevat de roepnamen van enkele docenten van TIS Delft. 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 6.3 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:

```
std::map<std::string, unsigned int> telefoonNummer;
```

Met behulp van indexering kun je nu het telefoonnummer[17] 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.

```
std::cout << "Sonja : " << telefoonNummer["Sonja"] << std::endl;
```

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 array's worden in veel andere programmeertalen (waaronder JavaScript en PHP) associatieve array's genoemd.

In een multimap mag een sleutel wel meerdere keren voorkomen. In de multimap in figuur 6.3 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.

---

[17] 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.

Sinds C++11 bevat de standaard library twee verschillende implementaties van deze vier associatieve containers. De al sinds C++98 aanwezige containers `set`, `multiset`, `map` en `multimap` worden geïmplementeerd met behulp van een zoekboom (Engels: search tree). Zoals je in tabel 2.1 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 in sinds C++11 beschikbare containers `unordered_set`, `unordered_multiset`, `unordered_map` en `unordered_multimap` worden geïmplementeerd met behulp van een hashtabel (Engels: hash table). Zoals je in tabel 2.1 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 zullen automatisch groeien 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[18].

6.2.3.1 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 de volgende vier memberfuncties:

- `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.

[18] Zie `http://en.cppreference.com/w/cpp/utility/hash`

- 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.

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

void print(const set<string>& s) {
    cout << "De set bevat: ";
    for (const auto& e: s)
        cout << e << " ";
    cout << endl;
}

int main() {
    set<string> docenten {"Jesse", "Paul", "Ineke", "Harry"};
    docenten.insert("Paul");
    print(docenten);
    auto result = docenten.insert("Harry");
    if (!result.second)
        cout << "1 Harry is genoeg." << endl;
    cout << "Er is " << docenten.count("Jesse") << " Jesse." << endl;
    docenten.erase("Harry");
    print(docenten);
    cin.get();
    return 0;
}
```

De uitvoer van dit programma:

```
De set bevat: Harry Ineke Jesse Paul
1 Harry is genoeg.
Er is 1 Jesse.
De set bevat: Ineke Jesse Paul
```

6.2.3.2 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.

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

void print(const multiset<string>& bag) {
    cout << "De bag bevat: ";
    for (const auto& e: bag)
        cout << e << " ";
    cout << endl;
}

int main() {
    multiset<string> docenten {"Harrie", "Jesse", "Paul", "Ineke", ↩
     ↪ "Harry", };
    docenten.insert("Paul");
    print(docenten);
    docenten.insert("Harry");
    print(docenten);
    cout << "Er zijn " << docenten.count("Harry") << " Harry's." ↩
     ↪ << endl;
    docenten.erase("Harry");
    print(docenten);
    docenten.erase(docenten.find("Paul"));
    print(docenten);
    docenten.erase(docenten.find("Ineke"), docenten.end());
    print(docenten);
    cin.get();
    return 0;
}
```

De uitvoer van dit programma:

```
De bag bevat: Harrie Harry Ineke Jesse Paul Paul
De bag bevat: Harrie Harry Harry Ineke Jesse Paul Paul
Er zijn 2 Harry's.
De bag bevat: Harrie Ineke Jesse Paul Paul
De bag bevat: Harrie Ineke Jesse Paul
De bag bevat: Harrie
```

6.2.3.3 Voorbeeldprogramma met `std::unordered_set`

Vergelijk het onderstaande programma unordered_set.cpp met het programma uit paragraaf 6.2.3.1 op pagina 42 en let op de verschillen tussen een `set` en een `unordered_set`.

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

```
void print(const unordered_set<string>& s) {
    cout << "De set bevat: ";
    for (const auto& e: s)
        cout << e << " ";
    cout << endl;
}

int main() {
    unordered_set<string> mensen = {"Jesse", "Paul", "Ineke", ↩
     ↪ "Harry"};
    mensen.insert("Paul");
    print(mensen);
    auto result = mensen.insert("Harry");
    if (!result.second)
        cout << "1 Harry is genoeg." << endl;
    cout << "Er is " << mensen.count("Jesse") << " Jesse." << endl;
    mensen.erase("Harry");
    print(mensen);
    cout << "De hash table heeft " << mensen.bucket_count() << " ↩
     ↪ buckets.";
    cout << endl << "Elke bucket bevat gemiddeld " << ↩
     ↪ mensen.load_factor() << " elementen.";
    cout << endl << "De maximale load_factor is " << ↩
     ↪ mensen.max_load_factor() << ".";
    cout << endl << "We voegen nu nog 34 namen toe." << endl;
    mensen.insert({"Sebastiaan", "Patrick", "Bas", "Ivan", "Eric", ↩
     ↪ "Ralf", "Leo", "Rens", "Mark", "Jaap", "Pascal", "Arend", ↩
     ↪ "Patrick", "Daniël", "Tom", "Jelle", "Raymond", "Rick", ↩
     ↪ "Miguel", "Niels", "Chesron", "Bart", "David", "Remko", ↩
     ↪ "Thijs", "Richard", "Robin", "Remy", "Nakib", "René", ↩
     ↪ "Tako", "Maikel", "Jory", "Willem-Pieter"});
    print(mensen);
    cout << "De hash table heeft " << mensen.bucket_count() << " ↩
     ↪ buckets.";
    cout << endl << "Elke bucket bevat gemiddeld " << ↩
     ↪ mensen.load_factor() << " elementen.";
    cin.get();
    return 0;
}
```

Een mogelijke[19] uitvoer van dit programma:

```
De set bevat: Harry Ineke Paul Jesse
1 Harry is genoeg.
Er is 1 Jesse.
De set bevat: Ineke Paul Jesse
```

[19] De uitvoer kan variëren afhankelijk van de gebruikte compiler. In dit geval is GCC 4.9.2 gebruikt.

```
De hash table heeft 5 buckets.
Elke bucket bevat gemiddeld 0.6 elementen.
De maximale load_factor is 1.
We voegen nu nog 34 namen toe.
De set bevat: Jory René Nakib Remy Robin Richard Chesron Niels ↩
  ↪ Miguel Thijs Jelle Tom Daniël Tako Arend Pascal Rens Maikel ↩
  ↪ Raymond Leo Bart Ralf Rick Eric Ivan Willem-Pieter Mark Bas ↩
  ↪ Patrick Jaap Sebastiaan Jesse Remko Paul David Ineke
De hash table heeft 41 buckets.
Elke bucket bevat gemiddeld 0.878049 elementen.
```

6.2.3.4 Voorbeeldprogramma met `std::map`

Het onderstaande voorbeeldprogramma stdcntwords.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 paragraaf 6.2.3.1 op pagina 42. Je kunt daarnaast ook gebruik maken van indexering waarbij de sleutel als index wordt gebruikt.

```cpp
#include <iostream>
#include <fstream>
#include <string>
#include <map>
using namespace std;

int main() {
    string w;
    map<string, int> freq;
    cout << "Geef filenaam: ";
    cin >> w;
    ifstream fin(w);
    while (fin >> w) {
        ++freq[w];
    }
    for (const auto& wordcount: freq) {
        cout << wordcount.first << " " << wordcount.second << endl;
    }
    cout << "Enkele belangrijke keywords:" << endl;
    cout << "do: " << freq["do"] << endl;
    cout << "else: " << freq["else"] << endl;
    cout << "for: " << freq["for"] << endl;
    cout << "if: " << freq["if"] << endl;
    cout << "return: " << freq["return"] << endl;
    cout << "switch: " << freq["switch"] << endl;
    cout << "while: " << freq["while"] << endl;
    cin.get();
    cin.get();
    return 0;
```

```
}
```

6.3 Iteratoren

In figuur 6.1 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<int> l = {12, 18, 6};
sort(l.begin(), l.end()); // Error!
```

Om deze reden zijn er meerder soorten iteratoren in de C++ standaard 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++ standaard vijf iteratorsoorten gedefinieerd:

- Een *input iterator* kan gebruikt worden om uit een container te lezen. Deze iteratorsoort ondersteund drie 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.[20]

  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 twee operaties:
  - met de `*` operator kan het element waar de iterator naar refereert worden *overschreven*.

[20] 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.

   - 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 drie 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 geplaatst en de container opnieuw worden doorlopen.[21] 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.

[21] In feite kan een forward iterator naar elke onthouden iterator worden verplaatst en daarvandaan weer verder stappen in voorwaartse richting.

- 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.[22] [23]

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 6.2 is aangegeven welke bewerkingen door de verschillende iteratorsoorten ondersteund worden.

Tabel 6.2: Standaard iteratorsoorten en de bewerkingen die ze leveren.

iteratorsoort	* (lezen)	* (schrijven)	++	== en !=	--	+=, −=, +, −, >, >=, <, <= en []
input (single pass)	✓	✗	✓	✓	✗	✗
output (single pass)	✗	✓	✓	✗	✗	✗
forward	✓	✓	✓	✓	✗	✗
bidirectional	✓	✓	✓	✓	✓	✗
random access	✓	✓	✓	✓	✓	✓

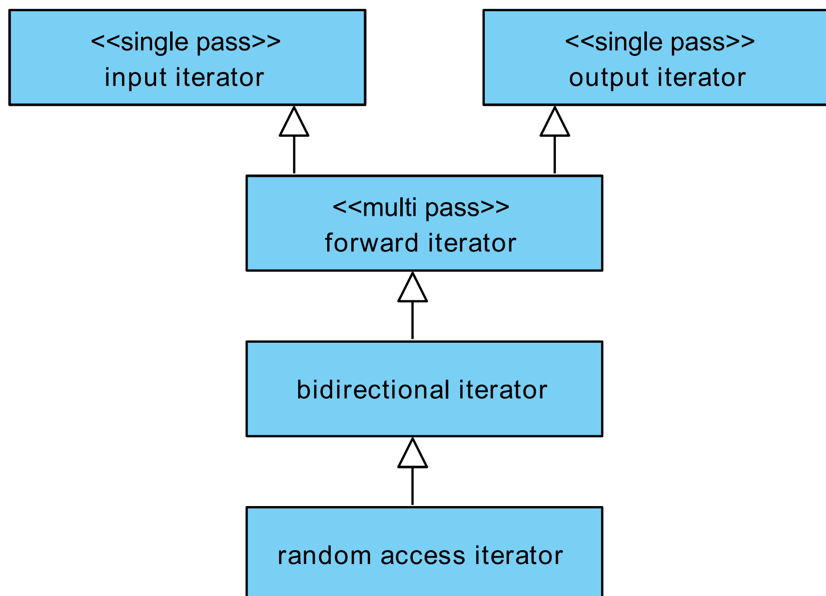
Je ziet in tabel 6.2 dat een forward iterator ook alle operaties levert die door een input en een output operator worden geleverd. Een bidirectional iterator levert ook alle operaties die door een forward iterator worden geleverd. Een random access iterator levert ook alle operaties die door een bidirectional iterator worden geleverd. Deze relatie tussen de verschillende iterator concepten is weergegeven in figuur 6.4.

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++ standaard aangegeven welke soort iterator ze leveren:

- `forward_list`, `unordered_set`, `unordered_multiset`, `unordered_map` en `unordered_multimap` leveren een forward iterator;

[22] 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.

[23] De expressie `i[n]` is gelijk aan de expressie `*(i + n)`.



Figuur 6.4: De relaties tussen de verschillende iterator concepten.

- `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++ standaard aangegeven welke soort iterator ze nodig hebben. Bijvoorbeeld:

- `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.

6.3.1 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 range [begin, end). Als de range een even aantal elementen bevat dan is er niet echt een middelste element (er zijn dan twee elementen die samen het midden van de range 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 <typename I>
I find_upper_middle_random_access(I begin, I end) {
```

```
    return begin + (end - begin)/2;
}
```

Door een template 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<int> l = {12, 18, 6};
    cout << *find_upper_middle_random_access(l.begin(), l.end()) ↩
     ↪ << endl; // Error: no match for 'operator-'
```

Als er bidirectional iteratoren beschikbaar zijn, dan kan de functie als volgt geïmplementeerd worden, zie find_upper_middle_bidirectional.cpp:

```
template <typename I>
I find_upper_middle_bidirectional(I begin, I end) {
    while (begin != end) {
        --end;
        if (begin != end) {
            ++begin;
        }
    }
    return begin;
}
```

Door een template te gebruiken kan deze functie aangeroepen worden voor elke container die bidirectional iteratoren levert[24], 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:

```
    forward_list<int> l = {12, 18, 6};
    cout << *find_upper_middle_bidirectional(l.begin(), l.end()) ↩
     ↪ << endl; // Error: no match for 'operator--'
```

Als er forward iteratoren beschikbaar zijn, dan kan de functie als volgt geïmplementeerd worden, zie find_upper_middle_forward.cpp:

```
template <typename I>
I find_upper_middle_forward(I begin, I end) {
    I i = begin;
    while (begin != end) {
        ++begin;
        if (begin != end) {
            ++begin;
            ++i;
        }
```

[24] Deze functie kan ook aanroepen worden voor elke container die random access iteratoren levert maar in dat geval kan beter de functie `find_upper_middle_random_access` aangeroepen worden want die is sneller.

```
    }
    return i;
}
```

Door een template te gebruiken kan deze functie aangeroepen worden voor elke container die forward iteratoren levert[25], bijvoorbeeld een `forward_list`. De executietijd van deze functie is $O(n)$.

Het zou natuurlijk mooi zijn als we de compiler zelf de meeste efficiënte versie van `find_upper_middle` zouden kunnen laten kiezen afhankelijk van het gebruikte iterator soort. Dit blijkt inderdaad mogelijk door gebruik te maken van zogenoemde iterator tags. Dit wordt later behandeld in paragraaf 6.4.5 op pagina 76.

6.3.2 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 6.1 op pagina 29. 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()) {
```

[25] 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` respectievelijk `find_upper_middle_random_access` aangeroepen worden want die zijn sneller.

```
        *laatste_vijf = 7;
    }
```

Het volledige voorbeeldprogramma reverse_iterator.cpp kun je online vinden.

6.3.3 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 6.1. 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 6.1. 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 `*`.[26] 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".

6.3.4 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:

[26] 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 6.1 en voor de associatieve containers paragraaf 6.2.3 op pagina 40. Elke schrijfoperatie via een `insert_iterator` in bijvoorbeeld een `vector` heeft dus een executietijd van de $O(n)$.

- `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.

6.3.5 Voorbeeldprogramma's met iteratoren

In het onderstaande voorbeeldprogramma streamitr.cpp wordt het gebruik van stream- en insert-iteratoren gedemonstreerd:

```
#include <vector>
#include <iostream>
#include <fstream>
#include <iterator>
#include <algorithm>
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));
    sort(rij.begin(), rij.end());
    ofstream fout("getallen_gesorteerd.txt");
    if (!fout) return 2; // kan fout niet openen
    ostream_iterator<int> iout(fout, " ");
    copy(rij.begin(), rij.end(), iout);
    cin.get();
    return 0;
}
```

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:

```
#include <iostream>
```

```
#include <fstream>
#include <string>
#include <set>
#include <iterator>
#include <algorithm>
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"));
    }
    cin.get();
    return 0;
}
```

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).

6.4 Algoritmen

De C++-standaard bevat vele generiek algoritmen. Een generiek algoritme werkt op één of meer ranges, zie figuur 6.1. Een range wordt aangegeven door twee iteratoren i1 en i2. De range loopt van i1 *tot* (dus niet tot en met) i2. In de wiskunde zouden we dit noteren als het halfopen interval [i1, i2).

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 paragraaf 6.3 op pagina 47 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<int> 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 6.2.3 op pagina 40 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<int> 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 = find(priem.begin(), priem.end(), 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. Alle algoritmen uit de standaard C++ library gebruiken ranges als parameters zodat ze ook op een deel van een container uitgevoerd kunnen worden.

6.4.1 Zoeken, tellen, testen, bewerken en vergelijken

De volgende algoritmen kun je gebruiken om een bepaald element te zoeken in een range:

- find; Zoek het eerste element met een bepaalde waarde.
- find_if; Zoek het eerste element dat voldoet aan een bepaalde voorwaarde.[27]
- find_if_not; Zoek het eerste element dat niet voldoet aan een bepaalde voorwaarde (sinds C++11).
- find_first_of[28]; 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:

- 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.

[27] Bijvoorbeeld: zoek het eerste element > 0.

[28] 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 6.3.2 op pagina 52.

- 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 zal altijd sneller zijn.

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.[29]

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[30]; 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 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 $O(m \times (n - m))$.

6.4.1.1 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. 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, 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

[29] Bijvoorbeeld: vermenigvuldig elk element met 2.

[30] De naam van dit algoritme is inconsequent gekozen. Een betere naam zou zijn search_end.

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. 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 60, wordt besproken hoe we dit met behulp van `any_of` kunnen doen.

```
#include <iostream>
#include <string>
#include <vector>
#include <algorithm>
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) {
            cout << (gevonden[i] ? w[i] : '.');
        }
        cout << endl << "Raad een letter: ";
        char c = cin.get(); cin.get();
        auto itr = w.cbegin();
        while ((itr = find(itr, w.cend(), c)) != w.cend()) {
            gevonden[(itr - w.cbegin())] = true;
            ++itr;
        }
    }
    while (count(gevonden.begin(), gevonden.end(), false) != 0);
    cout << "Je hebt het woord \"" << w << "\" geraden." << endl;
    cin.get();
    return 0;
}
```

Invoer [31] en uitvoer van dit programma:

```
......
Geef een letter: h
......
Geef een letter: a
.a....
Geef een letter: r
.a....
Geef een letter: r
```

[31] De invoer is groen en onderstreept weergegeven.

```
.a....
Geef een letter: y
.a....
Geef een letter: g
ga.g..
Geef een letter: l
galg..
Geef een letter: j
galgj.
Geef een letter: e
galgje
Je hebt het woord "galgje" geraden!
```

6.4.1.2 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*[32] met een overloaded `operator()` die een `bool` teruggeeft die aangeeft of aan de voorwaarde wordt voldaan;
- door middel van een *lambda-functie*[33] die een `bool` teruggeeft die aangeeft of aan de voorwaarde wordt voldaan.

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

bool isPos(int i) {
    return i >= 0;
}

template<typename T>
class IsGreaterEqual {
public:
    IsGreaterEqual(int r): right(r) {
    }
    bool operator()(int left) const {
        return left >= right;
    }
private:
```

[32] Een functie-object is een object dat zich voordoet als een functie door middel van operator overloading.

[33] 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 6.4.2 op pagina 63, nog uitgebreid besproken.

```
    T right;
};

int main() {
    list<int> 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 = find_if(l.cbegin(), l.cend(), isPos);
    if (r != l.cend()) {
        cout << "Het eerste positieve element is: " << *r << endl;
    }
//  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 = find_if(l.cbegin(), l.cend(), IsGreaterEqual<int>(0));
    if (r != l.cend()) {
        cout << "Het eerste positieve element is: " << *r << endl;
    }
//  Zoeken met behulp van een lambda functie als zoekvoorwaarde.
//  Voordeel: handig als zoekvoorwaarde uniek is.
//  Nadeel: geen hergebruik mogelijk.
    r = find_if(l.cbegin(), l.cend(), [](int i) {
        return i >= 0;
    });
    if (r != l.end()) {
        cout << "Het eerste positieve element is: " << *r << endl;
    }
    cin.get();
    return 0;
}
```

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 6.4.1.1 op pagina 57 verbeteren.

De regel:

```
    while (count(gevonden.begin(), gevonden.end(), false) != 0);
```

kan vervangen worden door (zie galgje_find.cpp):

```
    while (any_of(gevonden.cbegin(), gevonden.cend(), [](bool 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 zal altijd sneller zijn.

6.4.1.3 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.

```
#include <iostream>
#include <fstream>
#include <list>
#include <iterator>
#include <algorithm>
using namespace std;

int main() {
    list<int> 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.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>()) {
        cout << "Het eerste priemgetal kleiner dan 100 is: " << ↩
         ↪ *eerste_priem << endl;
    }
    cin.get();
    return 0;
}
```

6.4.1.4 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 2x 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.

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

void printDubbel(int i) {
    cout << i << " " << i << " ";
}

int main() {
    vector<int> v = {-3, -4, 3, 4};
    ostream_iterator<int> iout(cout, " ");
    copy(v.cbegin(), v.cend(), iout);
    cout << endl;

//  Bewerking opgeven met een functie.
//  Nadeel: voor elke bewerking moet een aparte functie worden ↩
 ↪ geschreven.
    for_each(v.cbegin(), v.cend(), printDubbel);
    cout << endl;

//  Bewerking opgeven met een lambda functie.
//  Voordeel: handig als bewerking uniek is.
//  Nadeel: geen hergebruik mogelijk.
    for_each(v.cbegin(), v.cend(), [](int i) {
        cout << i << " " << i << " ";
    });
    cout << endl;

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

De uitvoer 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) {
        cout << i << " " << i << " ";
    };
    cout << endl;
```

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.

6.4.2 Lambda-functies

In paragraaf 6.4.1.2 op pagina 59 heb je gezien dat een lambda-functie gebruikt kan worden om de voorwaarde van een `find_if` te specificeren. In paragraaf paragraaf 6.4.1.4 op pagina 61 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.

6.4.2.1 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. Een lambda functie waarbij dit mogelijk is 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 gebruikte variabele wordt “by reference” ingesloten.
- `[=]` elke gebruikte variabele wordt “by value” ingesloten.
- `[=, &c]` c wordt “by reference” ingesloten en alle andere 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.[34]

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

int main() {
    vector<int> v = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
    int somEven = 0;
    for_each(v.cbegin(), v.cend(), [&somEven](int i) {
        if (i % 2 == 0) {
            somEven += i;
```

[34] Als alternatief zou je ook een `accumulate` algoritme met een “gewone” lambda-functie kunnen gebruiken. Zie online: accumulate.cpp.

```
        }
    });
    cout << "Som van alle even getallen in v = " << somEven << endl;
    cin.get();
    return 0;
}
```

In plaats van `[&somEven]` kan ook `[&]` gebruikt worden. Het expliciet specificeren dat de variabele `somEven` ingesloten moet worden in de closure heeft als voordeel dat de compiler een foutmelding geeft als je per ongeluk een andere variabele in de code van de closure gebruikt.

6.4.2.2 Returntype van een lambda-functie

Bij alle tot nu toe gebruikte lambda-functies kan de compiler zelf het returntype van de functie bepalen. Als de code van een lambda-functie slechts uit een `return`-statement bestaat dan is dit altijd mogelijk. Als de lambda-functie uit meer instructies bestaat dan neemt een C++11 compiler aan dat het returntype `void` is. In C++14 probeert de compiler in dit geval ook het correct type te bepalen maar 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 res2geheel.cpp wordt een `list` met toetsresultaten met één decimaal cijfer getransformeerd[35] 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.

```
#include <iostream>
#include <list>
#include <vector>
#include <algorithm>
#include <iterator>
#include <cmath>
using namespace std;

int main() {
    list<double> resultaten = {0.4, 1.4, 1.5, 4.9, 5.0, 8.9, 9.1, ↩
     ↪ 9.5, 10.0 };
    vector<int> geheleResultaten;
    // Alle resultaten < 1.5 moeten worden afgerond tot 1
    // Alle overige resultaten moeten worden afgerond
    transform(resultaten.cbegin(), resultaten.cend(), ↩
     ↪ back_inserter(geheleResultaten), [](double r) -> int {
        if (r < 1.5) {
            return 1;
```

[35] Zie voor informatie over het `transform` algoritme `http://en.cppreference.com/w/cpp/algorithm/transform` of verderop in dit dictaat: paragraaf 6.4.3.1 op pagina 67.

```
        }
        return round(r);
    });
    for (auto r : geheleResultaten) {
        cout << r << " ";
    }
    cout << endl;
    cin.get();
    return 0;
}
```

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 (ook niet in C++14) omdat de `round` functie een `double` teruggeeft. Het returntype van de lambda-functie zou dus `int` of `double` kunnen zijn en alleen de programmeur weet wat de bedoeling is.

6.4.2.3 Lambda-functieparametertype met auto

In C++14 is het mogelijk om het parametertype van een lambda-function als `auto` te definiëren.[36] De compiler bepaalt dan zelf het juiste type. Zo kan in de `find_if` uit paragraaf 6.4.1.2 op pagina 59 de zoekvoorwaarde in C++14 als volgt gedefinieerd worden:

```
    list<int> l = {-3, -4, 3, 4};
//  Zoeken met behulp van een lambda functie als zoekvoorwaarde.
//  Gebruik auto als parametertype (werkt niet voor C++11, wel ↩
 ↪ voor C++14).
    auto r = find_if(l.cbegin(), l.cend(), [](auto i) {
        return i >= 0;
    });
    if (r != l.end()) {
        cout << "Het eerste positieve element is: " << *r << endl;
    }
```

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.

Je ziet dat de code door het gebruik van `auto` om het parametertype van de lambda-functie te definiëren beter aanpasbaar wordt.

6.4.3 Kopiëren, bewerken, vervangen, roteren, schudden, verwisselen, verwijderen en vullen

In paragraaf 6.4.1 op pagina 56 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`.

[36] De code in deze paragraaf is getest met GCC 4.9.2 en gecompileerd met de optie `-std=c++14`.

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 6.4.1.4 op pagina 61.
- `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 6.4.2.2 op pagina 64.

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 6.3.5 op pagina 54.
- `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`.[37]
- `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`.

De volgende algoritmen kun je gebruiken om elementen uit een range te verwijderen[38]:

- `remove`; Verwijder alle elementen met een bepaalde waarde.

---

[37] 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.

[38] 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 6.4.3.2 op pagina 69).

- 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[39] 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 $O(n)$.

6.4.3.1 Voorbeeldprogramma met std::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[40] verschillende manieren opgegeven:

- door middel van een *functie* die de transformatie uitvoert;
- door middel van een *lambda-functie* die de transformatie uitvoert.

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

int telop(int i, int j) {
    return i + j;
}

int main() {
    vector<int> v = {-3, -4, 3, 4};
    vector<int> w = {1, 2, 3, 4};
    ostream_iterator<int> iout(cout, " ");
    copy(v.cbegin(), v.cend(), iout);
    cout << endl;
    copy(w.cbegin(), w.cend(), iout);
    cout << endl;
```

[39] Door deze algoritmen te combineren met een insert-iterator, zie paragraaf 6.3.3 op pagina 53, kunnen ze ook gebruikt worden om containers te vullen (of aan te vullen).

[40] Er is nog een derde manier, zie paragraaf 6.4.6 op pagina 78

```
//  Bewerking opgeven met een functie.
//  Nadeel: voor elke bewerking moet een aparte functie worden ↩
 ↪ geschreven.
    transform(v.cbegin(), v.cend(), w.cbegin(), v.begin(), telop);
    copy(v.cbegin(), v.cend(), iout);
    cout << endl;

//  Bewerking opgeven met een lambda functie.
//  Voordeel: handig als bewerking uniek is.
//  Nadeel: geen hergebruik mogelijk.
    transform(v.cbegin(), v.cend(), w.cbegin(), v.begin(), [](int ↩
     ↪ i1, int i2) {
        return i1 + i2;
    });
    copy(v.cbegin(), v.cend(), iout);
    cout << endl;

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

De uitvoer van dit programma:

```
-3 -4 3 4
1 2 3 4
-2 -2 6 8
-1 0 9 12
```

In C++14 kun je `auto` gebruiken om de parametertypes van de lambda-functie te definiëren, zoals besproken in paragraaf 6.4.2.3 op pagina 65:

```
    vector<int> v = {-3, -4, 3, 4};
    vector<int> w = {1, 2, 3, 4};
    transform(v.cbegin(), v.cend(), w.cbegin(), 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.

6.4.3.2 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:

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

int main() {
    vector<int> v;
    for (int i = 0; i < 10; ++i) {
        v.push_back(i * i);
    }
    vector<int> w(v);

//  code om te laten zien hoe remove werkt:
    ostream_iterator<int> out(cout, " ");
    cout << "Na initialisatie:" << endl;
    copy(v.cbegin(), v.cend(), out);
    auto end = remove_if(v.begin(), v.end(), [](int i) {
        return i % 2 == 0;
    });
    cout << endl << "Na remove (tot returned iterator):" << endl;
    copy(v.begin(), end, out);
    cout << endl << "Na remove (hele vector):" << endl;
    copy(v.cbegin(), v.cend(), out);
    v.erase(end, v.end());
    cout << endl << "Na erase (hele vector):" << endl;
    copy(v.cbegin(), v.cend(), out);

//  in de praktijk gebruiken we een remove altijd binnen een erase:
    w.erase(remove_if(w.begin(), w.end(), [](int i) {
        return i % 2 == 0;
    }), w.end());
    cout << endl << "Na remove binnen erase:" << endl;
    copy(w.cbegin(), w.cend(), out);

    cin.get();
```

```
    return 0;
}
```

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
```

6.4.3.3 Voorbeeldprogramma met `std::generate_n`

In paragraaf 6.4.3.2 op pagina 69 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 6.3.3 op pagina 53) en een closure (zie paragraaf 6.4.2.1 op pagina 63). Al kun je jezelf in dit geval natuurlijk wel afvragen of de `for`-lus niet eenvoudiger en duidelijker is.

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

int main() {
    vector<int> kwadraten;
    int n = 0;
    generate_n(back_inserter(kwadraten), 10, [&n]() {
        ++n; return n * n;
    });
    copy(kwadraten.begin(), kwadraten.end(), ↩
     ↪ ostream_iterator<int>(cout, " "));
    cout << endl;
    cin.get();
}
```

6.4.4 Sorteren en bewerkingen op gesorteerde ranges

In paragraaf 6.4.1 op pagina 56 en paragraaf 6.4.3 op pagina 65 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[41] 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 $S2$

6.4.4.1 Voorbeeldprogramma met std::sort

In paragraaf 6.3.5 op pagina 54 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.

Als je de getallen in het programma streamitr.cpp uit paragraaf 6.3.5 op pagina 54 van hoog naar laag wilt sorteren dan kun je de regel:

```
sort(rij.begin(), rij.end());
```

vervangen door (zie sort_lambda.cpp)[42]:

[41] Kijk, als je niet bekend bent met verzamelingenleer of als je een opfrisser nodig hebt, op: http://nl.wikipedia.org/wiki/Verzameling_(wiskunde).

[42] Er is een nog eenvoudigere manier, zie paragraaf 6.4.6 op pagina 78.

```
    sort(rij.begin(), rij.end(), [](int i, int 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.

```
#include <vector>
#include <iostream>
#include <iomanip>
#include <iterator>
#include <algorithm>
using namespace std;

class Score {
public:
    Score(const string& n, int p);
    int punten() const;
    const string& naam() const;
private:
    string nm;
    int pnt;
};

Score::Score(const string& n, int p): nm(n), pnt(p) {
}
int Score::punten() const {
    return pnt;
}
const string& Score::naam() const {
    return nm;
}

ostream& operator<<(ostream& left, const Score& d) {
    return left << setw(5) << d.punten() << " " << d.naam();
}

int main() {
    vector<Score> 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)
    };
    sort(scores.begin(), scores.end(), [](const Score& d1, const ↩
     ↪ Score& d2) {
        return d1.naam() < d2.naam();
    });
    stable_sort(scores.begin(), scores.end(), [](const Score& d1, ↩
     ↪ const Score& d2) {
        return d1.punten() > d2.punten();
    });
    ostream_iterator<Score> iout(cout, "\n");
    copy(scores.begin(), scores.end(), iout);
    cin.get();
    return 0;
}
```

De uitvoer van dit programma:

```
400 Marie-Louise
300 Anne
300 Koen
300 Linda
300 Marie-Louise
300 Theo
 50 Marie-Louise
```

6.4.4.2 Voorbeeldprogramma met `std::includes`

In paragraaf 6.4.1.1 op pagina 57 is het `find` algoritme gebruikt om het bekende spelletje galgje te implementeren. Dit spel kan ook geïmplementeerd worden door gebruik te maken van verzamelingen, 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 6.3.3 op pagina 53). 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[43], doordat de letter al aanwezig is in deze verzameling dan wordt net zo lang om een nieuwe letter gevraagd tot het toevoegen wel

[43] In paragraaf 6.2.3.1 op pagina 42 is de `insert` memberfunctie van `set` behandeld.

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.

```
#include <iostream>
#include <string>
#include <set>
#include <algorithm>
#include <iterator>
using namespace std;

int main() {
    string w = "galgje";
    set<char> te_raden, geraden;
    copy(w.begin(), w.end(), inserter(te_raden, te_raden.begin()));
    do {
        for (auto c: w) {
            cout << (geraden.count(c) ? c : '.');
        }
        cout << endl << "Raad een letter: ";
        char c = cin.get(); cin.get();
        while (!geraden.insert(c).second) {
            cout << "De letter " << c << " had je al geraden...";
            cout << endl << "Raad een andere letter: ";
            c = cin.get(); cin.get();
        }
    } while (!includes(geraden.begin(), geraden.end(), ↩
     ↪ te_raden.begin(), te_raden.end()));
    cout << "Je hebt het woord " << w << " geraden." << endl;
    cin.get();
    return 0;
}
```

Invoer en uitvoer van dit programma:

```
......
Raad een letter: h
......
Raad een letter: a
.a....
Raad een letter: g
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
ga.g.e
Raad een letter: i
```

```
ga.g.e
Raad een letter: j
ga.gje
Raad een letter: l
Je hebt het woord galgje geraden.
```

6.4.4.3 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.

```
#include <iostream>
#include <list>
#include <algorithm>
#include <functional>
using namespace std;

class Hond {
public:
    virtual ~Hond() = default;
    virtual void blaf() const = 0;
};

class Tekkel: public Hond {
public:
    virtual void blaf() const override {
        cout << "Kef kef ";
    }
};

class StBernard: public Hond {
public:
    virtual void blaf() const override {
        cout << "Woef woef ";
    }
};

int main() {
    list<Hond*> kennel = {new Tekkel, new StBernard, new Tekkel};
    for_each(kennel.cbegin(), kennel.cend(), mem_fn(&Hond::blaf));
    cout << endl;

//  alternatief met lambda functie
    for_each(kennel.cbegin(), kennel.cend(), [](const Hond* p) {
        p->blaf();
    });
```

```
    cout << endl;

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

De uitvoer van dit programma:

```
Kef kef Woef woef Kef kef
Kef kef Woef woef Kef kef
```

Je kunt alle honden in de kennel ook laten blaffen met behulp van een range-based for:

```
    for (const auto p: kennel) {
        p->blaf();
    }
    cout << endl;
```

Het gebruik van een range-based for is eenvoudiger dan het gebruik van een `for_each` met een lambda-functie. 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.

6.4.5 Automatisch de meest efficiënte implementatie kiezen afhankelijk van de beschikbare iterator soort

In paragraaf 6.3.1 op pagina 50 heb je gezien dat de implementatie van een algoritme afhankelijk kan zijn van de soort iterator die beschikbaar is. Er zijn 3 functies gedefinieerd waarmee het upper middle element van een range bepaald kan worden. Als er random access iteratoren beschikbaar zijn, kan de functie `find_upper_middle_random_access` gebruikt worden. Als bidirectional iteratoren beschikbaar zijn, kan de functie `find_upper_middle_bidirectional` gebruikt worden. Als slechts forward iteratoren beschikbaar zijn, moet de functie `find_upper_middle_foward` gebruikt worden.

Het zou natuurlijk mooi zijn als je de compiler zelf de meeste efficiënte versie van `find_upper_middle` zou kunnen laten kiezen afhankelijk van het gebruikte iterator soort. Dit blijkt inderdaad mogelijk door gebruik te maken van zogenoemde iterator tags. In de standaard C++ library is voor elke iteratorsoort een zogenoemd `iterator_tag` type gedefinieerd. Er is dus een `random_access_iterator_tag` type, een `bidirectional_iterator_tag` en een `forward_iterator_tag`. We definiëren nu drie overloaded[44] template functies die allemaal `find_upper_middle` heten en behalve de twee parameters `begin` en `end` nog een derde parameter hebben genaamd `dummy`. Deze `dummy` parameter is van het type van de betreffende `iterator_tag`. Deze parameter wordt verder in de implementatie van de functie niet gebruikt. Vervolgens wordt er nog een vierde (overloaded) `find_upper_middle` template functie gedefinieerd die slechts twee parameters heeft: `begin` en `end`. Deze functie roept vervolgens één van de andere drie `find_upper_middle` template functies aan met als derde argument een bij de iteratoren `begin` en `end` behorende `iterator_tag` object.

[44] Zie [1, paragraaf 1.10].

We kunnen de bij een iterator horende iterator_tag opvragen met behulp van de standaard iterator_traits[45] template class. Het iterator type moet als template argument aan deze template class worden doorgegeven. In deze class is een typedef gedefinieerd genaamd iterator_category die overeenkomt met het betreffende iterator_tag type. Deze typedef kunnen we gebruiken om een (dummy) object aan te maken. Het type van dit object (het iterator_tag type) wordt vervolgens door de compiler gebruikt om de juiste find_upper_middle te selecteren. Zie find_upper_middle.cpp:

```
#include <iostream>
#include <vector>
#include <list>
#include <forward_list>
#include <iterator>
using namespace std;

template <typename I>
I find_upper_middle(I begin, I end, forward_iterator_tag dummy) {
    I i = begin;
    while (begin != end) {
        ++begin;
        if (begin != end) {
            ++begin;
            ++i;
        }
    }
    return i;
}

template <typename I>
I find_upper_middle(I begin, I end, bidirectional_iterator_tag ↩
 ↪ dummy) {
    while (begin != end) {
        --end;
        if (begin != end) {
            ++begin;
        }
    }
    return begin;
}

template <typename I>
I find_upper_middle(I begin, I end, random_access_iterator_tag ↩
 ↪ dummy) {
    return begin + (end - begin)/2;
}

template <typename I>
```

[45] Zie eventueel http://en.cppreference.com/w/cpp/iterator/iterator_traits.

```
I find_upper_middle(I begin, I end) {
    return find_upper_middle(begin, end, typename ↩
     ↪ iterator_traits<I>::iterator_category());
}

int main() {
    forward_list<int> fl = {1, 2};
    if (*find_upper_middle(fl.begin(), fl.end()) != 2) {
        cerr << "Test 1 failed!" << endl;
        return 1;
    }
    list<int> l = {1, 2, 3};
    if (*find_upper_middle(l.begin(), l.end()) != 2) {
        cerr << "Test 2 failed!" << endl;
        return 2;
    }
    vector<int> v = {1, 2, 3, 4};
    if (*find_upper_middle(v.begin(), v.end()) != 3) {
        cerr << "Test 3 failed!" << endl;
        return 3;
    }
    return 0;
}
```

Het keyword `typename` is nodig om de compiler duidelijk te maken dat `iterator_traits<I>::iterator_category` een typenaam is. Zie eventueel `http://en.cppreference.com/w/cpp/keyword/typename`.

6.4.6 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 6.4.1.2 op pagina 59 en paragraaf 6.4.3.1 op pagina 67. Je kunt echter ook gebruik maken van een zogenoemd standaard functie-object. De classes waarmee deze objecten gemaakt kunnen worden zijn gedefiniëerd in de headerfile `functional`. 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 transfer_plus.

```
#include <iostream>
#include <vector>
```

```
#include <iterator>
#include <functional>
#include <algorithm>
using namespace std;

int main() {
    vector<int> v = {-3, -4, 3, 4};
    vector<int> w = {1, 2, 3, 4};
    ostream_iterator<int> iout(cout, " ");
    copy(v.cbegin(), v.cend(), iout);
    cout << endl;
    copy(w.cbegin(), w.cend(), iout);
    cout << endl;

//  Bewerking opgeven met een standaard functie-object.
//  Voordeel: hergebruik van standaardcomponenten.
    transform(v.cbegin(), v.cend(), w.cbegin(), v.begin(), ↩
     ↪ plus<int>());
    copy(v.cbegin(), v.cend(), iout);
    cout << endl;

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

Als je dit vergelijkt met het gebruik van een functie of lambda-functie zoals in paragraaf 6.4.3.1 op pagina 67, dan zie je dat het gebruik van een standaard functie-object in dit geval eenvoudiger is.

De code die in paragraaf 6.4.4.1 op pagina 71 gegeven is om een `vector` met integers van hoog naar laag te sorteren (zie sort_lambda.cpp):

```
    sort(rij.begin(), rij.end(), [](int i, int j) {
        return i > j;
    });
```

kan sterk vereenvoudigd worden door het gebruik van een standaard functie-object (zie sort_greater.cpp):

```
    sort(rij.begin(), rij.end(), greater<int>());
```

Het gebruik van een standaard functie-object om een voorwaarde door te geven aan één van de `_if` algoritmen is minder eenvoudig. In paragraaf 6.4.1.2 op pagina 59 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[46] 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:

```
#include <iostream>
#include <list>
#include <algorithm>
#include <functional>
using namespace std;
using namespace std::placeholders;

int main() {
    list<int> l = {-3, -4, 3, 4};
    list<int>::const_iterator r = find_if(l.cbegin(), l.cend(), ↩
     ↪ bind(greater_equal<int>(), _1, 0));
    if (r != l.cend()) {
        cout << "Het eerste positieve element is: " << *r << endl;
    }
    cin.get();
    return 0;
}
```

Persoonlijk vind ik het gebruik van een lambda-functie in dit geval eenvoudiger, zie paragraaf 6.4.1.2 op pagina 59.

In paragraaf 6.4.3.2 op pagina 69 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(remove_if(w.begin(), w.end(), [](int i) {
    return i % 2 == 0;
}), w.end());
```

Dit zou ook met behulp van standaard functie-objecten kunnen, zie remove_bind.cpp:

[46] De placeholders zijn gedefinieerd in een aparte namespace genaamd std::placeholders.

```
w.erase(remove_if(w.begin(), w.end(), bind(equal_to<int>(), ↩
 ↪ bind(modulus<int>(), _1, 2), 0)), w.end());
```

Ook in dit geval vind ik het gebruik van een lambda-functie eenvoudiger.

7

Toepassingen van datastructuren

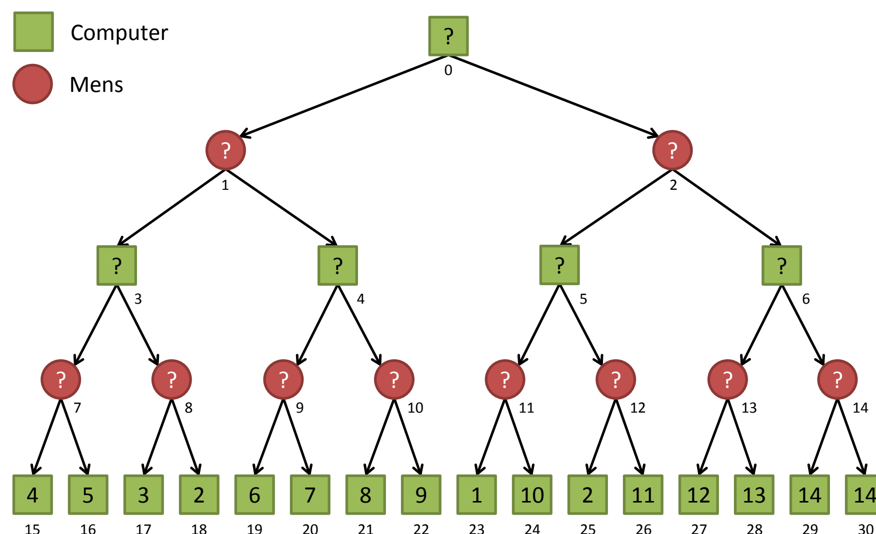
In dit hoofdstuk worden algoritmen en datastructuren toegepast om de computer het spelletje Boter, Kaas en Eieren te laten spelen. Een andere toepassing die, in de les maar niet in dit dictaat, wordt besproken is het berekenen van het kortste pad in een graph.

7.1 Boter, Kaas en Eieren

In deze paragraaf wordt besproken hoe je een computer het spelletje Boter, Kaas en Eieren kunt laten spelen. In de les is uitgelegd hoe je de beste zet kunt vinden in een spelboom (Engels: game tree) door het minimax algoritme toe te passen.

7.1.1 Minimax algoritme

Als je de beste zet in een spel (met 2 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 7.1 is gegeven.



Figuur 7.1: Voorbeeld van een spelboom.

In figuur 7.1 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

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 7.1 lijkt dit een van de twee posities met de waarde 14 maar deze zijn onbereikbaar denk maar even na. Bij zijn eerste beurt gaat de computer naar rechts (richting 14). De tegenstander (mens) zal dan echter naar links gaan. De computer zal nu bij zijn volgende beurt naar rechts gaan (richting 11) maar zijn tegenstander zal bij de volgende beurt weer voor links kiezen 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 MinMax0.cpp wordt de boom uit figuur 7.1 gebruikt en wordt de hoogst haalbare waarde berekend met het minimax algoritme. Er wordt daarbij gebruik gemaakt van 2 functies die elkaar (recursief) aanroepen.

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

int value(int pos);
int chooseComputerMove(int pos);
int chooseHumanMove(int pos);

const 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 chooseComputerMove(int pos) {
    int bestValue = value(pos);
    if (bestValue == UNDECIDED) {
        bestValue = 0;
        for (int i = 1; i < 3; ++i) {
            int value = chooseHumanMove(2 * pos + i);
            if (value > bestValue) {
                bestValue = value;
            }
        }
    }
    return bestValue;
}

int chooseHumanMove(int pos) {
    int bestValue = value(pos);
    if (bestValue == UNDECIDED) {
        bestValue = 15;
        for (int i = 1; i < 3; ++i) {
            int value = chooseComputerMove(2 * pos + i);
            if (value < bestValue) {
                bestValue = value;
            }
        }
    }
    return bestValue;
}

int main() {
    int value = chooseComputerMove(0);
    cout << "Minimaal te behalen Maximale waarde = " << value << ↩
     ↪ endl;
    cin.get();
}
```

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 `chooseComputerMove` 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 `chooseComputerMove` wordt eerst de functie `value` aangeroepen. De returnwaarde van deze functie wordt opgeslagen in de lokale variabele `bestValue`. Als de waarde van `bestValue` ongelijk is aan `UNDECIDED`, kan de functie `chooseComputerMove` 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 `chooseComputerMove` is aange-

roepen 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 `bestValue` 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 `chooseHumanMove` met als argument `2 * pos + i` waarbij `i` achtereenvolgens de waarde `1` en `2` heeft. We moeten hier de functie `chooseHumanMove` 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 zal zijn. 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 `bestValue`), dan wordt het tot nu toe gevonden maximum gelijk gemaakt aan `value`. Aan het einde van de `for`-lus zal de variabele `bestValue` dus het maximum bevatten van de (in dit geval twee) posities die vanuit de positie `pos` bereikt kunnen worden. De waarde van `bestValue` wordt teruggegeven aan de aanroepende functie.

De functie `chooseHumanMove` wordt aangeroepen als de mens aan de beurt is en lijkt erg veel op de functie `chooseComputerMove`, in plaats van het maximum wordt echter het minimum bepaald.

De functie `chooseHumanMove` 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 `chooseHumanMove` wordt eerst de functie `value` aangeroepen. De returnwaarde van deze functie wordt opgeslagen in de lokale variabele `bestValue`. Als de waarde van `bestValue` ongelijk is aan `UNDECIDED`, kan de functie `chooseHumanMove` 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 `chooseHumanMove` 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 `bestValue` 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 `chooseComputerMove` met als argument `2 * pos + i` waarbij `i` achtereenvolgens de waarde `1` en `2` heeft. We moeten hier de functie `chooseComputerMove` 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 zal zijn. 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 `bestValue`), dan wordt het tot nu toe gevonden minimum gelijk gemaakt aan `value`. Aan het einde van de `for`-lus zal de variabele `bestValue` dus het minimum bevatten van de (in dit geval twee) posities die vanuit de positie `pos` bereikt kunnen worden. De waarde van `bestValue` 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 chooseComputerMove(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 bestValue van de functie chooseComputerMove. De voorwaarde van de if is dus true en de code binnen de if wordt uitgevoerd. De lokale variabele bestValue wordt gelijk gemaakt aan 0 en de for-lus wordt uitgevoerd. De variabele i krijgt de waarde 1 en de functie chooseHumanMove wordt aangeroepen met 2 * 0 + 1 is 1 als argument.

1.1. Vanuit de functie chooseComputerMove(0) wordt dus de functie chooseHumanMove-(1) aangeroepen. Vervolgens wordt de functie value(1) aangeroepen en deze functie geeft UNDECIDED terug. Deze waarde wordt toegekend aan de lokale variabel bestValue van de functie chooseHumanMove. De voorwaarde van de if is dus true en de code binnen de if wordt uitgevoerd. De lokale variabele bestValue wordt gelijk gemaakt aan 15 en de for-lus wordt uitgevoerd. De variabele i krijgt de waarde 1 en de functie chooseComputerMove wordt aangeroepen met 2 * 1 + 1 is 3 als argument.

1.1.1. Vanuit de functie chooseHumanMove(1)wordt dus de functie chooseComputerMove (3) aangeroepen. Vervolgens wordt de functie value(3) aangeroepen en deze functie geeft UNDECIDED terug. Deze waarde wordt toegekend aan de lokale variabel bestValue van de functie chooseComputerMove. De code binnen de if wordt uitgevoerd. De lokale variabele bestValue wordt gelijk gemaakt aan 0 en de for-lus wordt uitgevoerd. De variabele i krijgt de waarde 1 en de functie chooseHumanMove wordt aangeroepen met 2 * 3 + 1 is 7 als argument.

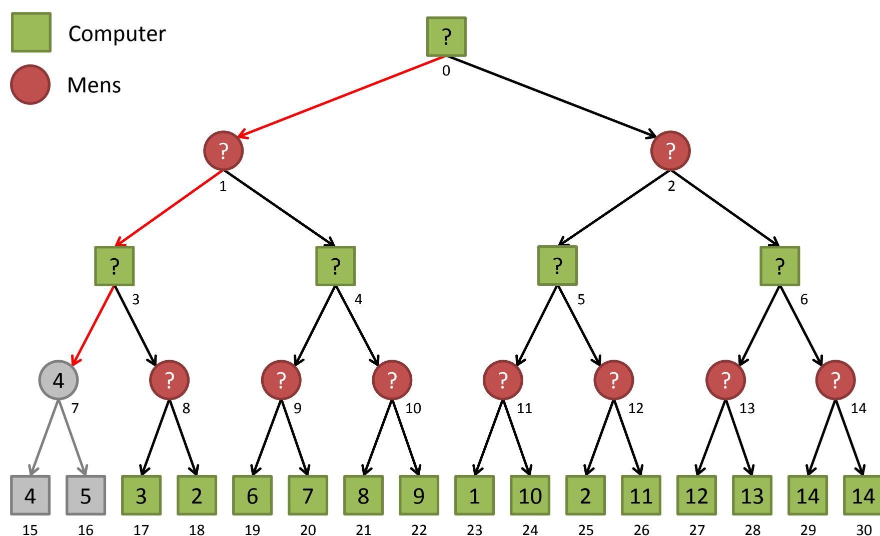
1.1.1.1. Vanuit de functie chooseComputerMove(3) wordt dus de functie chooseHuman Move(7) aangeroepen. Vervolgens wordt de functie value(7) aangeroepen en deze functie geeft UNDECIDED terug. Deze waarde wordt toegekend aan de lokale variabel bestValue van de functie chooseHumanMove. De code binnen de if wordt uitgevoerd. De lokale variabele bestValue wordt gelijk gemaakt aan 15 en de for-lus wordt uitgevoerd. De variabele i krijgt de waarde 1 en de functie chooseComputerMove wordt aangeroepen met 2 * 7 + 1 is 15 als argument.

1.1.1.1.1. Vanuit de functie chooseHumanMove(7) wordt dus de functie chooseComputerMove(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 bestValue van de functie chooseComputerMove. De voorwaarde van de if is nu false en de waarde 4 wordt teruggegeven.

1.1.1.2. We keren terug naar de functie chooseHumanMove(7). De lokale variabele value krijgt de waarde 4 en omdat de waarde van value kleiner is dan de waarde van bestValue (4 is kleiner dan 15) krijgt bestValue de waarde 4. De for-lus wordt vervolgd. De variabele i krijgt de waarde 2 en de functie chooseComputerMove wordt aangeroepen met 2 * 7 + 2 is 16 als argument.

1.1.1.2.1. Vanuit de functie chooseHumanMove(7) wordt dus de functie chooseComputerMove(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 bestValue van de functie chooseComputerMove. De voorwaarde van de if is nu false en de waarde 5 wordt teruggegeven.

1.1.1.3. We keren terug naar de functie chooseHumanMove(7). De lokale variabele value krijgt de waarde 5 en omdat de waarde van value niet kleiner is dan de waarde van bestValue (5 is niet kleiner dan 4) behoudt bestValue de waarde 4. De for-lus is nu voltooid. Deze situatie is weergegeven in figuur 7.2. De functie chooseHumanMove(7) geeft de waarde 4 terug.



Figuur 7.2: Spelboom net voor het einde van het uitvoeren van chooseHumanMove(7).

1.1.2. We keren terug naar de functie chooseComputerMove(3). De lokale variabele value krijgt de waarde 4 en omdat de waarde van value groter is dan de waarde van bestValue (4 is groter dan 0) krijgt bestValue de waarde 4. De for-lus wordt vervolgd. De variabele i krijgt de waarde 2 en de functie chooseHumanMove wordt aangeroepen met 2 * 3 + 2 is 8 als argument.

1.1.2.1. Vanuit de functie chooseComputerMove(3) wordt dus de functie chooseHumanMove(8) aangeroepen. Vervolgens geeft de functie value(8) UNDECIDED terug. Deze waarde wordt toegekend aan de lokale variabel bestValue van de functie chooseHumanMove en de code binnen de if wordt uitgevoerd. De lokale variabele bestValue wordt gelijk gemaakt aan 15 en de for-lus wordt uitgevoerd. De variabele i krijgt de waarde 1 en de functie chooseComputerMove wordt aangeroepen met 2 * 8 + 1 is 17 als argument.

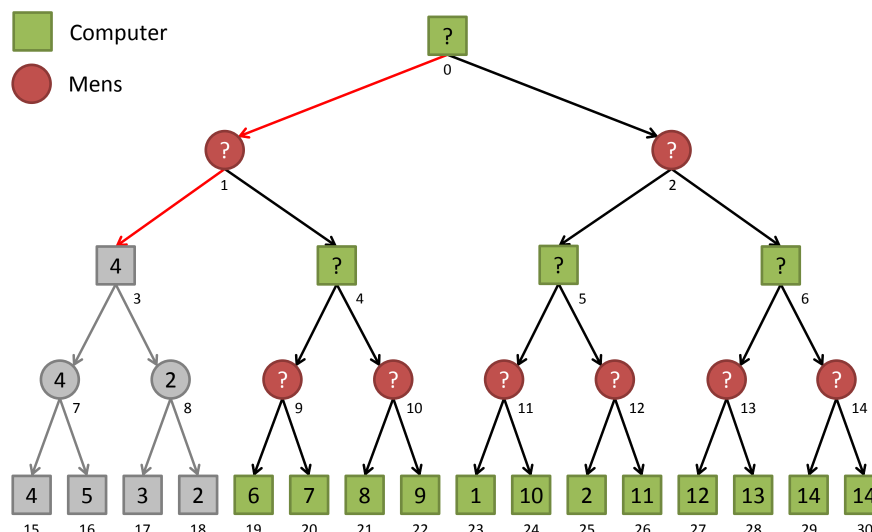
1.1.2.1.1. Vanuit de functie `chooseHumanMove(8)` wordt dus de functie `chooseComputerMove(17)` aangeroepen. Vervolgens wordt de functie `value(17)` aangeroepen en die geeft array-element `value[17 - 15]` is `value[2]` is 3 terug. Deze waarde wordt teruggegeven vanuit de functie `chooseComputerMove`.

1.1.2.2. We keren terug naar de functie `chooseHumanMove(8)`. De lokale variabele `value` krijgt de waarde `3` en omdat de waarde van `value` kleiner is dan de waarde van `bestValue` (`3` is kleiner dan `15`) krijgt `bestValue` de waarde `4`. De `for`-lus wordt vervolgd. De variabele `i` krijgt de waarde `2` en de functie `chooseComputerMove` wordt aangeroepen met `2 * 8 + 2` is 18 als argument.

1.1.2.2.1. Vanuit de functie `chooseHumanMove(8)` wordt dus de functie `chooseComputerMove(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 `chooseComputerMove`.

1.1.2.3. We keren terug naar de functie `chooseHumanMove(8)`. De lokale variabele `value` krijgt de waarde `2` en omdat de waarde van `value` kleiner is dan de waarde van `bestValue` (`2` is kleiner dan `3`) krijgt `bestValue` de waarde `2`. De `for`-lus is nu voltooid. De functie `chooseHumanMove(8)` geeft de waarde `2` terug.

1.1.3. We keren terug naar de functie `chooseComputerMove(3)`. De lokale variabele `value` krijgt de waarde `2` en omdat de waarde van `value` niet groter is dan de waarde van `bestValue` (`2` is niet groter dan `2`) behoudt `bestValue` de waarde `4`. De `for`-lus is nu voltooid. Deze situatie is weergegeven in figuur 7.3. De functie `chooseComputerMove(3)` geeft de waarde 4 terug.



Figuur 7.3: Spelboom net voor het einde van het uitvoeren van `chooseComputerMove(3)`.

1.2. We keren terug naar de functie `chooseHumanMove(1)`. Omdat de geretourneerde waarde (4) kleiner is dan de waarde van `bestValue` (`15`) krijgt `bestValue` de waarde `4`. De `for`-lus wordt vervolgd. De functie `chooseComputerMove` wordt aangeroepen met `2 * 1 + 2` is 4 als argument.

1.2.1. Vanuit de functie `chooseHumanMove(1)`wordt dus de functie `chooseComputerMove` `(4)` aangeroepen. De lokale variabele `bestValue` wordt gelijk gemaakt aan `0` en de `for`-lus wordt uitgevoerd. De functie `chooseHumanMove` wordt aangeroepen met `2 * 4 + 1` is 9 als argument.

1.2.1.1. Vanuit de functie `chooseComputerMove(4)` wordt dus de functie `chooseHuman` `Move(9)` aangeroepen. De lokale variabele `bestValue` wordt gelijk gemaakt aan 15 en de `for`-lus wordt uitgevoerd. De functie `chooseComputerMove` wordt aangeroepen met `2 * 9 + 1` is 19 als argument.

1.2.1.1.1. Vanuit de functie `chooseHumanMove(9)` wordt dus de functie `chooseComputerMove(19)` aangeroepen die de waarde 6 teruggeeft.

1.2.1.2. We keren terug naar de functie `chooseHumanMove(9)`. Omdat de geretourneerde waarde (6) kleiner is dan de waarde van `bestValue` `(15)` krijgt `bestValue` de waarde `6`. De `for`-lus wordt vervolgd. De functie `chooseComputerMove` wordt aangeroepen met `2 * 9 + 2` is `20` als argument.

1.2.1.2.1. Vanuit de functie `chooseHumanMove(9)` wordt dus de functie `chooseComputerMove(20)` aangeroepen die de waarde 7 teruggeeft.

1.2.1.3. We keren terug naar de functie `chooseHumanMove(9)`. Omdat de geretourneerde waarde (7) niet kleiner is dan de waarde van `bestValue` `(6)` geeft de functie `chooseHumanMove(9)` de waarde `6` terug.

1.2.2. We keren terug naar de functie `chooseComputerMove(4)`. Omdat de geretourneerde waarde (6) groter is dan de waarde van `bestValue` `(0)` krijgt `bestValue` de waarde `6`. De `for`-lus wordt vervolgd. De functie `chooseHumanMove` wordt aangeroepen met `2 * 4` + 2 is `10` als argument.

1.2.2.1. Vanuit de functie `chooseComputerMove(4)` wordt dus de functie `chooseHuman` `Move(10)` aangeroepen. De lokale variabele `bestValue` wordt gelijk gemaakt aan 15 en de `for`-lus wordt uitgevoerd. De functie `chooseComputerMove` wordt aangeroepen met `2 * 10 + 1` is 21 als argument.

1.2.2.1.1. De functie `chooseComputerMove(21)` geeft de waarde 8 terug.

1.2.2.2. Omdat de geretourneerde waarde (8) kleiner is dan de waarde van `bestValue` `(15)` krijgt `bestValue` de waarde `8`. De `for`-lus wordt vervolgd. De functie `chooseComputerMove` wordt aangeroepen met `2 * 10 + 2` is 22 als argument.

1.2.2.2.1. De functie `chooseComputerMove(22)` geeft de waarde 9 terug.

1.2.2.3. Omdat de geretourneerde waarde (9) niet kleiner is dan de waarde van `bestValue` (8) geeft de functie `chooseHumanMove(9)` de waarde `8` terug.

1.2.3. We keren terug naar de functie `chooseComputerMove(4)`. Omdat de geretourneerde waarde (8) groter is dan de waarde van `bestValue` `(6)` geeft de functie `chooseComputerMove(3)` geeft de waarde `8` terug.

1.3. We keren terug naar de functie `chooseHumanMove(1)`. Omdat de geretourneerde waarde (8) niet kleiner is dan de waarde van `bestValue` (4) geeft de functie `chooseHumanMove(1)` de waarde 4 terug.

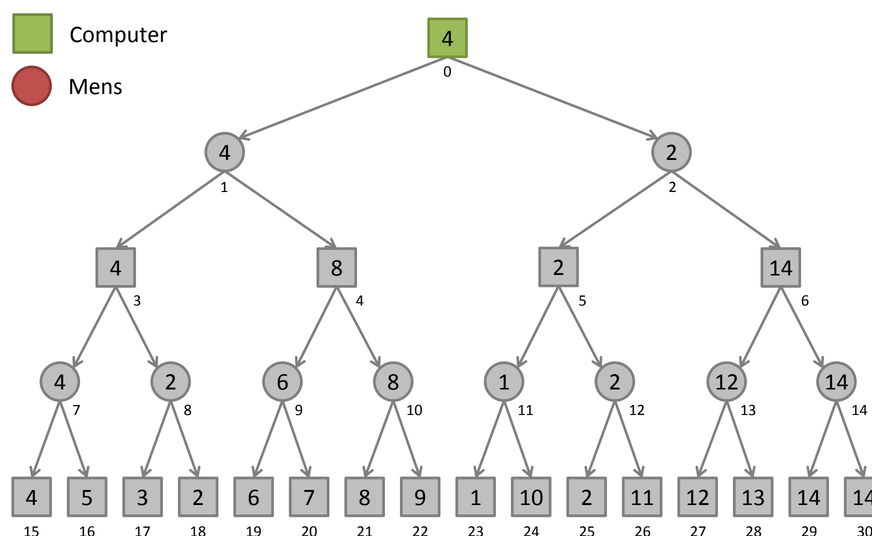
2. We keren terug naar de functie `chooseComputerMove(0)`. Omdat de geretourneerde waarde (4) groter is dan de waarde van `bestValue` (`0`) krijgt `bestValue` de waarde 4. De `for`-lus wordt vervolgd. De functie `chooseHumanMove` wordt aangeroepen met `2 * 0 + 2` is 2 als argument.

2.1 ···

2.2 ···

2.3 ··· De functie `chooseHumanMove(2)` geeft de waarde 2 terug.

3. We keren terug naar de functie `chooseComputerMove(0)`. Omdat de geretourneerde waarde (2) niet groter is dan de waarde van `bestValue` (4) geeft de functie `chooseComputerMove(0)` de waarde 4 terug. Deze situatie is weergegeven in figuur 7.4.



Figuur 7.4: Spelboom net voor het einde van het uitvoeren van `chooseComputerMove(0)`.

In de figuur 7.4 kun je zien dat met behulp van het minimax algoritme de best bereikbare positie voor de computer wordt gevonden.

In het programma MinMax0PrintTree.cpp wordt de spelboom weergegeven in figuur 7.1 geprint met behulp van de recursieve functie `printTree`. De berekende waarde voor elke positie weergegeven in figuur 7.4 wordt geprint met behulp van de recursieve functie `printCalculatedTree`. Beide functies zijn hieronder weergegeven:

```
void printTree(int pos, int level) {
    if (level != 5) {
        printTree(2 * pos + 2, level + 1);
        cout << setw(level * 5) << pos << ":" << value(pos) << endl;
        printTree(2 * pos + 1, level + 1);
    }
}

void printCalculatedTree(int pos, int level) {
```

```
    if (level != 5) {
        printCalculatedTree(2 * pos + 2, level + 1);
        cout << setw(level * 5) << pos << ":" << (level % 2 == 0 ? ↩
         ↪ chooseComputerMove(pos) : chooseHumanMove(pos)) << endl;
        printCalculatedTree(2 * pos + 1, level + 1);
    }
}
```

Het is ook mogelijk om de functies `valueMoveComputer` en `valueMoveHuman` te combineren tot 1 functie, zie MinMax1.cpp:

```
enum Side {HUMAN, COMPUTER};

int chooseMove(Side s, int pos) {
    int bestValue = value(pos);
    if (bestValue == UNDECIDED) {
        bestValue = s == COMPUTER ? 0 : 15;
        for (int i = 1; i < 3; ++i) {
            int value = chooseMove(s == COMPUTER ? HUMAN : ↩
             ↪ COMPUTER, 2 * pos + i);
            if (s == COMPUTER && value > bestValue || s == HUMAN ↩
             ↪ && value < bestValue) {
                bestValue = value;
            }
        }
    }
    return bestValue;
}
```

Er zijn echter tweee 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 `bestNextPos`. De functies `chooseComputerMove` en `chooseHumanMove` moeten dan als volgt worden aangepast:

```
int chooseComputerMove(int pos, int& bestNextPos) {
    int bestValue = value(pos);
    if (bestValue == UNDECIDED) {
        bestValue = 0;
        for (int i = 1; i < 3; ++i) {
```

```
            int dummyPos;
            int value = chooseHumanMove(2 * pos + i, dummyPos);
            if (value > bestValue) {
                bestValue = value;
                bestNextPos = 2 * pos + i;
            }
        }
    }
    return bestValue;
}

int chooseHumanMove(int pos, int& bestNextPos) {
    int bestValue = value(pos);
    if (bestValue == UNDECIDED) {
        bestValue = 15;
        for (int i = 1; i < 3; ++i) {
            int dummyPos;
            int value = chooseComputerMove(2 * pos + i, dummyPos);
            if (value < bestValue) {
                bestValue = value;
                bestNextPos = 2 * pos + i;
            }
        }
    }
    return bestValue;
}
```

In de functie main kunnen we deze functies als volgt gebruiken om beste zet voor de computer respectievelijk de gebruiker te bepalen:

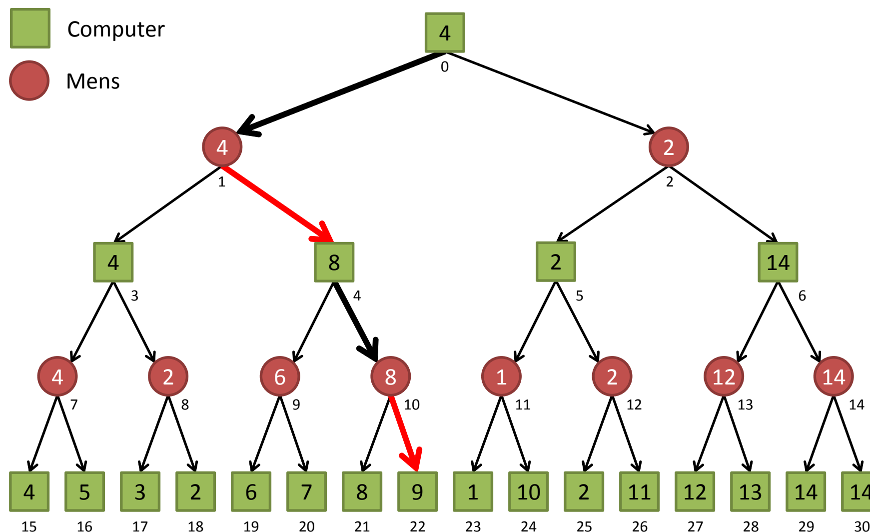
```
int main() {
    int pos = 0, bestNextPos, bestValue;
    while (pos < 15) {
        bestValue = chooseComputerMove(pos, bestNextPos);
        cout << "Minimaal te behalen Maximale waarde = " << ↩
         ↪ bestValue << endl;
        pos = bestNextPos;
        cout << "Computer kiest positie: " << pos << endl;
        int posL = 2 * pos + 1;
        int posR = 2 * pos + 2;
        if (pos < 15) {
            cout << "Je kunt kiezen voor positie " << posL << " of ↩
             ↪ positie " << posR << endl;
            chooseHumanMove(pos, bestNextPos);
            cout << "Pssst, " << bestNextPos << " is de beste ↩
             ↪ keuze." << endl;
            do {
                cout << "Maak je keuze: ";
                cin >> pos;
```

```
            } while (pos != posL && pos != posR);
        }
    }
    cout << "Behaalde waarde = " << value(pos) << endl;
    cin.get();
    cin.get();
}
```

Een mogelijke uitvoer van dit programma MinMax2.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
```

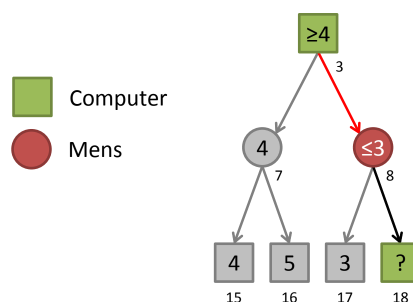
De bijbehorende weg door de spelboom is weergegeven in figuur 7.5. De computer behaald in dit geval de waarde 9. Deze waarde is hoger dan de minimaal te behalen maximale waarde van 4. Dit komt omdat de mens, ondanks de tips die hij/zij krijgt, steeds de "zwakste" keuze maakt.



Figuur 7.5: Mogelijke weg door de spelboom bij het uitvoeren van MinMax2.cpp.

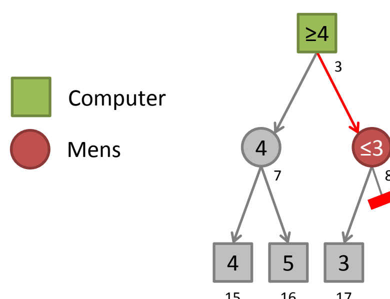
7.1.2 Alpha-beta pruning

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.



Figuur 7.6: In deze spelboom is het niet nodig om de waarde van positie 18 te bepalen.

In figuur 7.6 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 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 7.7

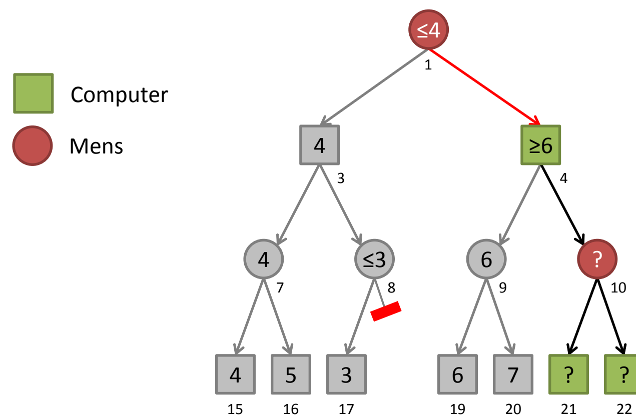


Figuur 7.7: De tak naar positie 18 kan worden gesnoeid uit de boom.

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 AlphaBeta0.cpp door middel van de parameter alpha. Deze naam is toevallig zo gekozen door de bedenkers van dit algoritme en heeft verder geen betekenis.

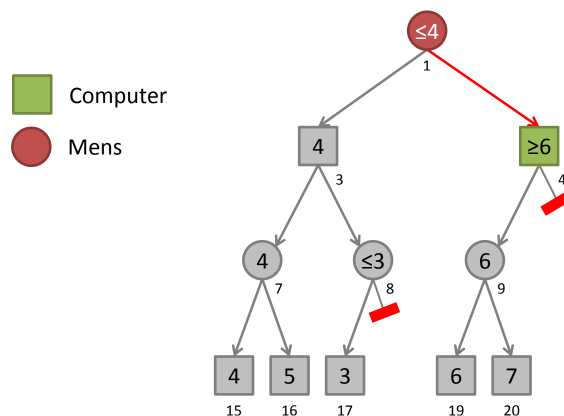
Bij het zoeken naar een maximale waarde op een bepaalde positie kun je stoppen zodra je een waarde hebt gevonden die *groter of gelijk* is aan de tot dan toe gevonden *minimale* waarde in de *bovenliggende* positie.

In figuur 7.8 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



Figuur 7.8: In deze spelboom is het niet nodig om de waarde van posities 10, 21 en 22 te bepalen.

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 7.7



Figuur 7.9: De tak naar positie 10 kan worden gesnoeid uit de boom.

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 AlphaBeta0.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 [10].

```
#include <iostream>
#include <iomanip>
```

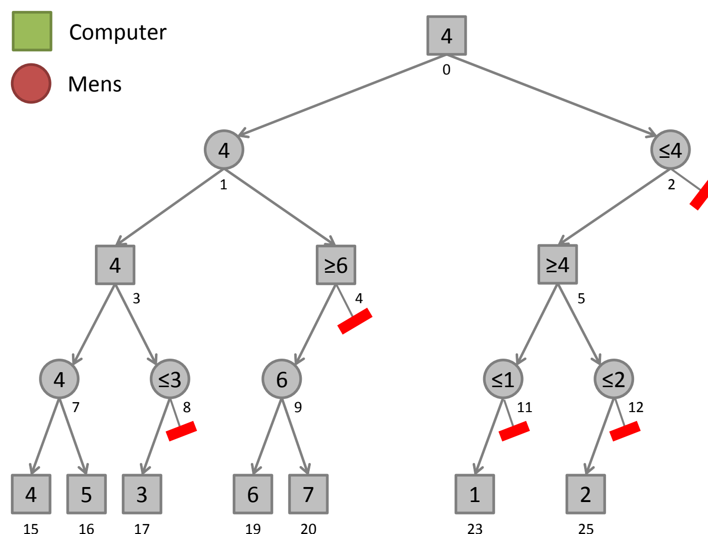
```
using namespace std;

int value(int pos);
int chooseComputerMove(int pos, int alpha = 0, int beta = 15);
int chooseHumanMove(int pos, int alpha = 0, int beta = 15);

const 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 chooseComputerMove(int pos, int alpha, int beta) {
    int bestValue = value(pos);
    if (bestValue == UNDECIDED) {
        bestValue = alpha;
        for (int i = 1; bestValue < beta && i < 3; ++i) {
            int value = chooseHumanMove(2 * pos + i, alpha, beta);
            if (value > bestValue) {
                bestValue = value;
                alpha = bestValue;
            }
        }
    }
    return bestValue;
}

int chooseHumanMove(int pos, int alpha, int beta) {
    int bestValue = value(pos);
    if (bestValue == UNDECIDED) {
        bestValue = beta;
        for (int i = 1; bestValue > alpha && i < 3; ++i) {
            int value = chooseComputerMove(2 * pos + i, alpha, beta);
            if (value < bestValue) {
                bestValue = value;
                beta = bestValue;
            }
        }
    }
    return bestValue;
}

int main() {
    int value = chooseComputerMove(0);
```

```
    cout << "Minimaal te behalen Maximale waarde = " << value << ↩
     ↪ endl;
    cin.get();
}
```

In een uitgebreidere versie van het bovenstaande programma, genaamd AlphaBeta0Verbose.cpp, wordt output naar cout 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 7.10 is te zien welke takken het alpha-beta algoritme de spelboom uit figuur 7.1 heeft gesnoeid.



Figuur 7.10: De gesnoeide boom.

Bij de eerste aanroep van de functie `valueMoveComputer` 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 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).

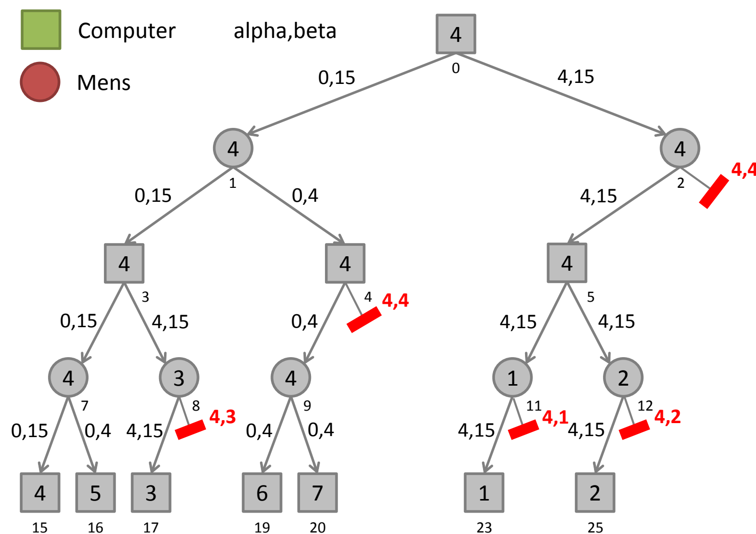
Je ziet dat bij het bepalen van de maximale waarde in een positie de waarde van `bestValue` wordt geïnitialiseerd met `alpha` en niet met het absolute minimum (`0`). Een, in een bovenliggende positie gevonden, tijdelijk maximum mag namelijk *niet* in een (2 lagen dieper) onderliggende positie op een *lagere* waarde worden gezet! Als je `bestValue` 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 bestValue 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 chooseComputerMove de expressie bestValue < beta vervangen kan worden door alpha < beta omdat bestValue en alpha in de for-lus altijd dezelfde waarde hebben. In de for-lus is het gebruik van bestValue dus overbodig. Merk op dat in de functie chooseHumanMove de expressie bestValue > alpha vervangen kan worden door beta > alpha omdat bestValue en beta in de for-lus altijd dezelfde waarde hebben. In de for-lus is het gebruik van bestValue dus overbodig. De expressie beta > alpha kan vervangen worden door alpha < beta die exact hetzelfde betekent. In het onderstaande programma AlphaBeta1.cpp zijn deze wijzigingen doorgevoerd:

```
int chooseComputerMove(int pos, int alpha, int beta) {
    int bestValue = value(pos);
    if (bestValue == UNDECIDED) {
        for (int i = 1; alpha < beta && i < 3; ++i) {
            int value = chooseHumanMove(2 * pos + i, alpha, beta);
            if (value > alpha) {
                alpha = value;
            }
        }
        bestValue = alpha;
    }
    return bestValue;
}

int chooseHumanMove(int pos, int alpha, int beta) {
    int bestValue = value(pos);
    if (bestValue == UNDECIDED) {
        for (int i = 1; alpha < beta && i < 3; ++i) {
            int value = chooseComputerMove(2 * pos + i, alpha, beta);
            if (value < beta) {
                beta = value;
            }
        }
        bestValue = beta;
    }
    return bestValue;
}
```

De voorwaarde om te snoeien is dus in de functies chooseComputerMove en chooseHumanMove exact hetzelfde, namelijk alpha $\geq$ beta. In figuur 7.11 zijn de waarden van alpha en beta bij elke aanroep van chooseComputerMove en chooseHumanMove gegeven. Om de werking van dit alpha-beta algoritme helemaal te doorgronden is het heel zinvol om het uitvoeren van het programma AlphaBeta1Verbose.cpp stap voor stap te volgen, eventueel met behulp van een debugger.



Figuur 7.11: De waarde van alpha en beta bij elke aanroep van `chooseComputerMove` en `chooseHumanMove`.

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 weer worden bereikt door bij het zoeken naar de maximale of minimale waarde de beste positie op te slaan in de uitvoerparameter `bestNextPos`. De functies `chooseComputerMove` en `chooseHumanMove` moeten dan als volgt worden aangepast:

```
int chooseComputerMove(int pos, int& bestNextPos, int alpha, int ↩
 ↪ beta) {
    int bestValue = value(pos);
    if (bestValue == UNDECIDED) {
        for (int i = 1; alpha < beta && i < 3; ++i) {
            int dummyPos;
            int value = chooseHumanMove(2 * pos + i, dummyPos, ↩
             ↪ alpha, beta);
            if (value > alpha) {
                alpha = value;
                bestNextPos = 2 * pos + i;
            }
        }
        bestValue = alpha;
    }
    return bestValue;
}

int chooseHumanMove(int pos, int& bestNextPos, int alpha, int ↩
 ↪ beta) {
    int bestValue = value(pos);
    if (bestValue == UNDECIDED) {
        for (int i = 1; alpha < beta && i < 3; ++i) {
            int dummyPos;
            int value = chooseComputerMove(2 * pos + i, dummyPos, ↩
             ↪ alpha, beta);
```

```
            if (value < beta) {
                beta = value;
                bestNextPos = 2 * pos + i;
            }
        }
        bestValue = beta;
    }
    return bestValue;
}
```

De volgorde waarin de mogelijke volgende posities worden onderzocht heeft invloed op het gedrag van het alpha-beta algoritme. In het programma AlphaBeta1Verbose.cpp wordt deze volgorde bepaald door de regel:

```
        for (i = 1; alpha < beta && i < 3; ++i) {
```

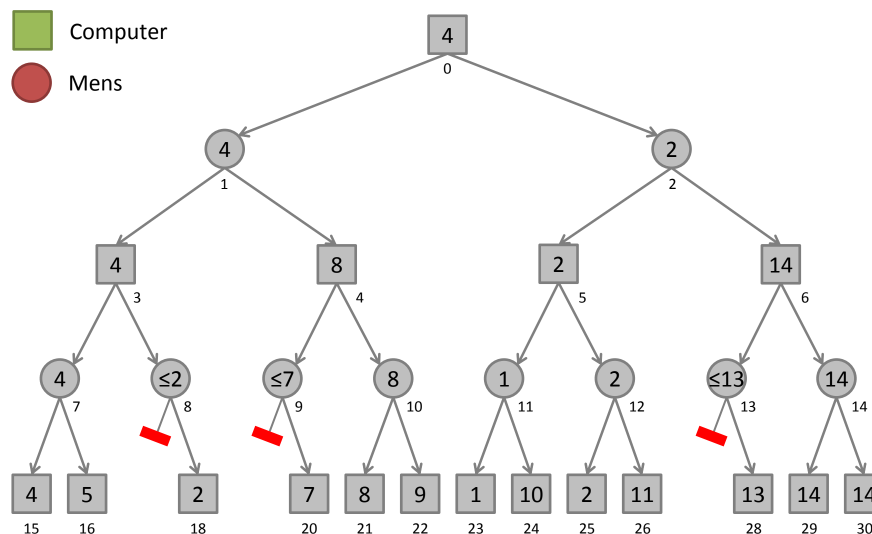
De mogelijke posities worden dus van links naar rechts doorzocht. We kunnen de mogelijke posities ook van links naar rechts doorzoeken door de bovenstaande regel te vervangen door de regel:

```
        for (i = 2; alpha < beta && i > 0; --i) {
```

In het resulterende programma AlphaBeta3Verbose.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 7.12 is te zien welke takken het alpha-beta algoritme de spelboom uit figuur 7.1 heeft gesnoeid als de boom van links naar rechts wordt doorlopen.



Figuur 7.12: De gesnoeide boom als de boom van links naar rechts wordt doorlopen.

7.1.3 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[13]. Alle deze programma's maken gebruik van een matrix class (zie Matrix.h) om de rondjes en kruisjes in op te slaan.

Online zijn de volgende programma's beschikbaar:

- Tic-Tac-Toe programma met minimax algoritme: TicTacSlow.cpp.
- Tic-Tac-Toe programma met alpha-beta pruning: TicTacAB.cpp.
- Tic-Tac-Toe programma met alpha-beta pruning en transposition table met behulp van `std::map`: TicTacMap.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 TicTacABLoser.cpp (met dank aan Marc Cornet).

Bij het gebruik van een transpositietabel kun je de diepte waarop de tabel gebruik wordt limiteren, zie TicTacMapRestrictedDepth.cpp. Daarbij moet de optimale waarde van `MAX_TABLE_DEPTH` worden bepaald. Dit kun je doen met het programma TicTacDetermineMaxTableDepth.cpp.

De programma's waarnaar in deze paragraaf wordt verwezen, worden in de les verder besproken. Zie eventueel de PowerPoint presentatie van week 5: `http://bd.eduweb.hhs.nl/algods/ppsx/ALGODS.pptm`.

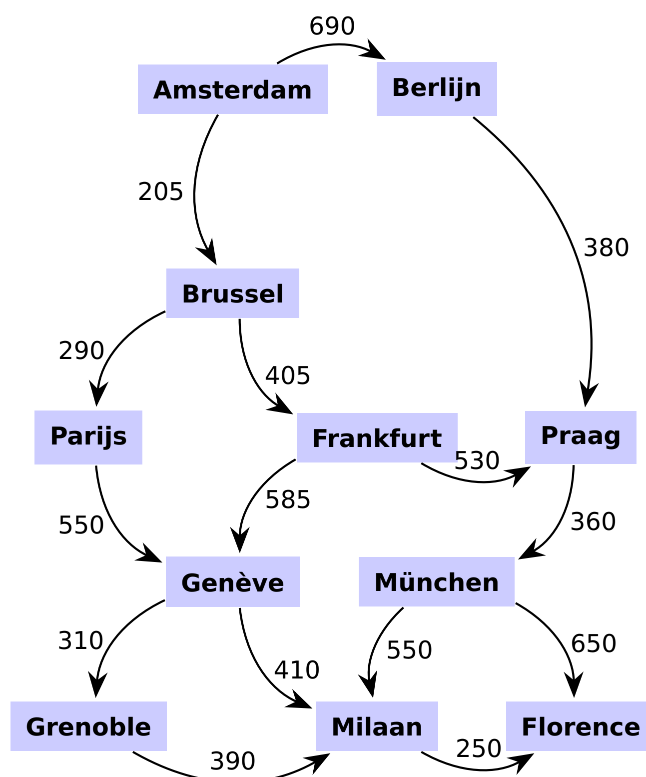
7.2 Het zoeken van het kortste pad in een graph

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[13].

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 7.13 is een graph weergegeven waarin de vertices Europese steden zijn. De verzameling van alle vertices wordt V genoemd en in figuur 7.13 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 7.13 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 7.13 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 7.13 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 7.13 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 7.13 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 zul je zien dat in figuur 7.13 geen enkele cycle voorkomt. Een directed graph zonder cycles wordt een directed acyclic graph genoemd, vaak afgekort tot DAG.



Figuur 7.13: 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[47]. 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[48]. Voor een complete digraph geldt: $|E| = |V| \cdot (|V|-1)$, er is dan namelijk tussen elk 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 7.13 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 7.13 te berekenen dan ziet de uitvoer er als volgt uit:

```
p = positive (http://bd.eduweb.hhs.nl/algods/graph_positive.png),
n = negative (http://bd.eduweb.hhs.nl/algods/graph_negative.png),
```

[47] graphs kunnen in verrassend veel vakgebieden worden toegepast, zie http://en.wikipedia.org/wiki/graph_theory#Applications.

[48] Dit is eenvoudig te bewijzen, zie http://nl.wikipedia.org/wiki/Grafentheorie#De_volledige_graaf.

```
a = acyclic  (http://bd.eduweb.hhs.nl/algods/graph_acyclic.png),
s = steden   (http://bd.eduweb.hhs.nl/algods/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
```

De algoritmen waarnaar in deze paragraaf wordt verwezen, worden in de les verder besproken. Zie eventueel de PowerPoint presentatie van week 6: `http://bd.eduweb.hhs.nl/algods/ppsx/ALGODS.pptm`.

Bibliografie

[1] Harry Broeders. *Objectgeoriënteerd Programmeren in C++*. 2014. URL: http://bd.eduweb.hhs.nl/ogoprg/pdf/Dictaat_OGPiCpp.pdf.

[2] 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.

[3] Bruce Eckel. *Thinking in C++, Volume 1: Introduction to Standard C++*. 2nd Edition. Prentice Hall, 2000. ISBN: 978-0-13-979809-2. URL: http://www.tigernt.com/onlineDoc/tij/TIC2Vone.pdf.

[4] Bruce Eckel. *Thinking in C++, Volume 2: Standard Libraries & Advanced Topics*. 2nd Edition. Prentice Hall, 1999. ISBN: 978-0-13-035313-9. URL: http://www.tigernt.com/onlineDoc/tij/TIC2Vtwo.pdf.

[5] C. A. R. Hoare. “Quicksort”. In: *The Computer Journal* 5.1 (jan 1962), p. 10–16.

[6] ISO. *ISO/IEC 14882:2014 Information technology — Programming languages — C++*. Geneva, Switzerland: International Organization for Standardization, 2014. URL: http://www.iso.org/iso/home/store/catalogue_ics/catalogue_detail_ics.htm?csnumber=64029.

[7] Donald E. Knuth. *The Art of Computer Programming, Volume 1: Fundamental Algorithms*. 3rd Edition. Redwood City, CA, USA: Addison Wesley Longman Publishing Co., Inc., 1997. ISBN: 978-0-201-89683-1.

[8] Donald E. Knuth. *The Art of Computer Programming, Volume 2: Seminumerical Algorithms*. 3rd Edition. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1997. ISBN: 978-0-201-89684-8.

[9] Donald E. Knuth. *The Art of Computer Programming, Volume 3: Sorting and Searching*. 2nd Edition. Redwood City, CA, USA: Addison Wesley Longman Publishing Co., Inc., 1998. ISBN: 978-0-201-89685-5.

[10] Donald E. Knuth en Ronald W. Moore. “An Analysis of Alpha-Beta Pruning.” In: *Artif. Intell.* 6.4 (1975), p. 293–326.

[11] Robert Sedgewick. *Algorithms in C++, Parts 1-4: Fundamentals, Data Structure, Sorting, Searching*. 3rd Edition. Addison-Wesley Professional, 1998. ISBN: 978-0-7686-8533-6.

[12] Jos Warmer en Anneke Kleppe. *Praktisch UML*. 5de Editie. Pearson Benelux B.V., 2011. ISBN: 978-90-430-2055-8.

[13] Mark A. Weiss. *Data Structures and Problem Solving Using C++*. 2nd Edition. Addison Wesley, 1999. ISBN: 978-0-201-61250-9.

[14] Niklaus Wirth. *Algorithms + Data Structures = Programs*. Upper Saddle River, NJ, USA: Prentice Hall PTR, 1978. ISBN: 978-0-13-022418-7.