Dictaat Algoritmen en Datastructuren
ALDAT

Literatuur

Bruce Eckel, Thinking in C++ 2nd Edition, Volume 1, ISBN 0139798099.
Bruce Eckel and Chuck Allison, Thinking in C++ 2nd Edition, Volume 2, ISBN 0131225529.
Deze boeken zijn ook gratis te downloaden van:
http://mindview.net/Books/TICPP/ThinkingInCPP2e.html.

Warmer en Kleppe, Praktisch UML, 3de editie, ISBN 9043008125.

Broeders, Dictaat: Object Georiënteerd Programmeren in C++ (PROS3).
Broeders, Dictaat: Algoritmen en Datastructuren (ALDAT).
De sheets en voorbeeldprogramma's zijn beschikbaar op het internet http://bd.eduweb.hhs.nl/aldat/.

De practicumopdrachten zijn beschikbaar op http://bd.eduweb.hhs.nl/aldat/. Daar kun je ook verwijzingen en achtergrondinformatie vinden.

Toetsing en beoordeling.

Er worden voor deze module twee deelresultaten vastgesteld waarbij het eerste resultaat (beoordeling practische eindopdracht) een cijfer (1..10) is, het tweede resultaat (practicum) een O(nvoldoende) of V(oldoende) is. Het eindresultaat wordt dan het cijfer van de eindopdracht als het tweede resultaat een V is en een 1 als het tweede resultaat een O is. Het bodemcijfer is 4.5.

Je kunt naar keuze alleen werken of in tweetallen. Als je ervoor kiest om samen te werken met een partner kies dan wel een partner van ongeveer hetzelfde niveau.

Het practicum wordt beoordeeld met Onvoldoende of Voldoende. Alle opdrachten worden afzonderlijk beoordeeld met een voldoende of onvoldoende aan de hand van:

- een demonstratie om de juiste werking aan te tonen.
- een inhoudelijk gesprek over opzet en uitvoering van de implementatie. Tijdens dit gesprek zal de docent enkele vragen stellen over de manier van aanpak en/of de werking van het programma. Als je deze vragen (over je eigen programma) niet kunt beantwoorden dan krijg je onvoldoende! Als bij jou een opdracht met onvoldoende wordt beoordeeld krijg je 1 keer de kans een vervangende opdracht te maken.

Om het practicum met een voldoende af te sluiten moeten **alle** opdrachten voldoende zijn.

Globale weekplanning .

week	studiemateriaal	dictaat	onderwerp
1	Dictaat inleiding + H1 + H2	5	Inleiding en overzicht datastructuren
2	Dictaat H3 en H4	8	Toepassingen en implementaties van een stack
3+4	Thinking in C++ en dictaat H6	18	Standard Template Library: containers, iteratoren, algoritmen en functieobjecten.
5	hand-outs		Toepassingen van dynamische datastructuren: game-tree, min-max algoritme en alfa-beta pruning.
6	hand-outs		Toepassing: galgje. Lambda expressies.

Een gedetailleerde planning kun je vinden op het internet:
http://bd.eduweb.hhs.nl/alda/studiew.htm#planning.

Inhoudsopgave.

Inleiding.

Dit is het dictaat: “Algoritmen en Datastructuren”. Dit dictaat kan worden gebruikt in combinatie met de boeken: “*Thinking in C++ 2nd Edition, Volume 1*” van Bruce Eckel (2000 Pearson Education) en “*Thinking in C++ 2nd Edition, Volume 2*” van Bruce Eckel en Chuck Allison (2004 Pearson Education). Beide boeken zijn gratis te downloaden vanaf http://www.mindview.net/Books/TICPP/ThinkingInCPP2e.html.

Dit dictaat is in de eerste vier weken bedoeld als primair studiemateriaal. Tijdens de laatste drie weken van deze module maken we ook gebruik van verschillende hand-outs die in de klas worden uitgereikt. Het studiemateriaal kun je vinden op http://bd.eduweb.hhs.nl/aldat/. Dit dictaat is zoals alle mensenwerk niet foutloos, verbeteringen en suggesties zijn altijd welkom!

Dynamische datastructuren.

Bij PROS1 en PROS2 heb je kennis gemaakt met *statische* datastructuren en bij PROS3 heb je al kort kennis gemaakt met *dynamische* datastructuren.

De eerste hogere programmeertalen (bijvoorbeeld Algol60) hadden al ingebouwde statische datastructuren zoals struct (record) en array. Ook de in het begin van de jaren '70 van Algol afgeleide talen zoals Pascal en C hebben deze datastructuren ingebouwd. Al heel snel in de geschiedenis van het programmeren (begin jaren '60) werd duidelijk dat de in hogere programmeertalen ingebouwde statische datastructuren in de praktijk vaak niet voldoen.

De stand van een spelletjes competitie kan bijvoorbeeld in de volgende datastructuur opgeslagen worden:

```
struct deelnemer {
    int  punten;
    char naam[80];
};
struct stand {
    int       aantalDeelnemers;
    deelnemer lijst[100];
};
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 (=statisch).
- elke `deelnemer` neemt hierdoor evenveel ruimte in onafhankelijk van de lengte van zijn `naam` (=statisch).
- elke `stand` neemt hierdoor evenveel ruimte in onafhankelijk van `aantalDeelnemers` (=statisch).
- het verwijderen van een `deelnemer` uit de `stand` is een heel bewerkelijke operatie.

Bij PROS3 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.

Door de ingebouwde datastructuur `struct` te combineren met pointers kunnen ook de problemen met de lijst van deelnemers worden opgelost. 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:

lijst van deelnemers = leeg of
deelnemer met een pointer naar een lijst van deelnemers.

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:

stamboom van een persoon = leeg of
persoon met een pointer naar de stamboom van zijn moeder én een pointer naar de stamboom van zijn vader.

Het standaardwerk waarin de (nu nog) meest gebruikte datastructuren helemaal worden "uitgekauwd" is Knuth [1973] en [1975]. 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 (Wirth[1976]) luidt dan ook niet voor niets: "*Algorithms + Data Structures = Programs*".

Het spreekt voor zich dat de klassieke datastructuren al snel door gebruik te maken van object georiënteerde 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 heet STL (Standard Template Library) en wordt gratis met Microsoft Visual C++ 2010 Express Edition (de compiler die wij gebruiken) meegeleverd. In deze module gaan we uitgebreid op deze STL library in.

Applicaties van datastructuren.

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

1 Algorithm Analysis

Bij het bespreken van algoritmen is het belangrijk om een een maat te hebben om de run-time van algoritmen met elkaar te kunnen vergelijken. We zullen bijvoorbeeld spreken over een $O(n^2)$ algoritme. De O(...) notatie wordt *big-O notatie* genoemd en uitgesproken als "is van de orde ...". $O(n^2)$ betekent dat de executietijd van het algoritme rechtevenredig toeneemt met het kwadraat van het aantal data-elementen. In de onderstaande tabel 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).

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•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 $O(n^2)$ algoritme niet bruikbaar is voor grote hoeveelheden data en dat een $O(n^3)$ en $O(10^n)$ algoritme sowieso niet bruikbaar zijn. De meest voor de hand liggende sorteermethoden[1] (insertion sort, bubble sort enz.) blijken allemaal $O(n^2)$ te zijn. Deze algoritmen zijn dus voor grotere datasets (n>1000) absoluut onbruikbaar! Al in 1962 heeft C.A.R. Hoare het zogenaamde Quicksort algoritme ontworpen dat “gemiddeld” O(n•log n) is. In elk boek over algoritmen en datastructuren kun je een implementatie van dit algoritme vinden. Maar op zich is dat niet zo interessant want elke class library waarin algoritmen en datastructuren zijn opgenomen heeft een efficiënte, dat is O(n•log n), sorteermethode. 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 STL library die we later in deze module 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[2]. 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 Data structures.

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. Hieronder vind je een poging de eigenschappen van de verschillende datastructuren samen te vatten.

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++ compiler is (of moet worden) opgenomen is het minder belangrijk om zelf te weten hoe z'n data-

[1] In veel inleidende boeken over programmeertalen worden dit soort sorteeralgoritmen als voorbeeld gebruikt. Zie bijvoorbeeld het C boek dat in de propedeuse gebruikt is.

[2] Het standaardwerk op het gebied van sorteer- en zoekalgoritmen is “The art of computer programming, volume 3: sorting and searching” van D.E. Knuth. Een meer toegankelijk boek is “Algorithms in C++” van R. Sedgewick.

structuur geïmplementeerd moet worden. Het belangrijkste is dat je weet wanneer je welke datastructuur kunt toepassen en hoe je dat dan moet doen.

naam	insert	remove	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 + meer overhead in space en time
queue	enqueue O(1)	dequeue O(1) FIFO	front O(1) FIFO	printer queue wachtrij	array, statisch + snel linked list, dynamisch + meer overhead in space en time
vector	O(1)	O(n) schuiven	O(n) op inhoud O(1) op index	vaste lijst code conversie	array, static 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, static + binary search algorithm
linked list	O(1)	O(n)	O(n)	dynamische lijst waarin je weinig zoekt en verwijdert	linked list, dynamic + more space overhead sequential access via iterator
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	more space overhead + minimal n+1 pointers with value 0
search tree	O(log n)	O(log n)	O(log n)	dynamische lijst waarin je veel muteert en zoekt	sorted binary tree, more space overhead than list
hash table	O(1)	O(1)	O(1)	symbol table (compiler) dictionary	semi-static, reduced performance if overfilled
priority queue	O(log n)	O(log n)	O(1)	event driven simulation	binary heap - array, static + little space overhead - binary tree, dynamic + space overh.

3 Voorbeelden van het gebruik van een datastructuur.

Als voorbeeld zullen we in dit hoofdstuk enkele toepassingen van de bekende datastructuur *stack* bespreken. Een stack is een datastructuur waarin dataelementen 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[3]:

```
#ifndef _THR_Bd_Stack_
#define _THR_Bd_Stack_

template <typename T> class Stack {
public:
    Stack();
    virtual ~Stack();
    virtual void push(const T& t) =0;
    virtual void pop() =0;
    virtual const T& top() const =0;
```

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

```
    virtual bool empty() const =0;
    virtual bool full() const =0;
private:
    void operator=(const Stack&); // Voorkom gebruik
    Stack(const Stack&);          // Voorkom gebruik
};

template <typename T> Stack<T>::Stack() {
}
template <typename T> Stack<T>::~Stack() {
}

#endif
```

Je ziet dat het ABC `Stack` bestaat uit 2 doe-functies en 3 vraag-functies. Het van de stack afhalen van een element gaat dus 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.

3.1 Balanced symbol checker.

Als voorbeeld bekijken we nu een C++ programma dat controleert of alle haakjes in een tekst gebalanceerd zijn. We maken daarbij gebruik van de class `StackWithList<T>` die is afgeleid van de hierboven gedefinieerde `Stack<T>`.

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

#include <iostream>
#include "stacklist.h"

using namespace std;

int main() {
    StackWithList<char> s;
    char c;
    cout<<"Type een expressie met haakjes 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<<"Haakje openen ontbreekt."<<endl;
                }
                else {
                    char d(s.top());
                    s.pop();
                    if (d=='('&&c!=')'||d=='{'&&c!='}'||
                                                    d=='['&&c!=']') {
                        cout<<"Haakje openen ontbreekt."<<endl;
                    }
                }
            }
        }
        cin.get(c);
    }
```

```
    if (!s.empty()) {
        cout<<"Haakje sluiten ontbreekt."<<endl;
    }
    cin.get();
    cin.get();
    return 0;
}
```

3.2 A simple calculator.

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 naam 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

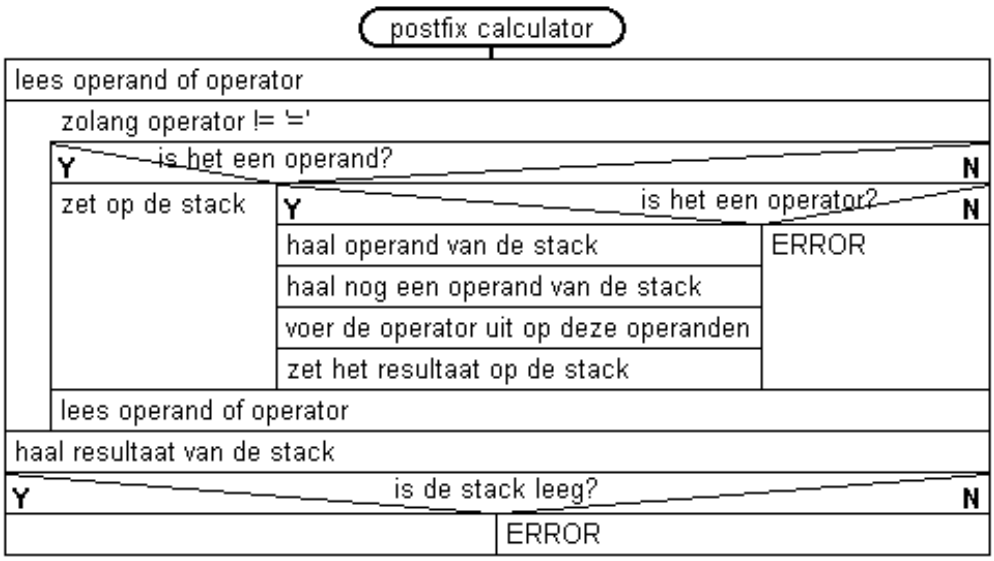
```
2 3 + 5 * = 25
```

Postfix heeft de volgende voordelen ten opzichte van infix:

- Geen prioriteitsregel nodig.
- Geen haakjes nodig.
- Eenvoudiger te berekenen m.b.v. Stack

3.2.2 Een postfix calculator.

Een postfix expressie kan met het volgende algoritme worden opgelost. Dit algoritme maakt gebruik van een stack.



We zullen nu eerst een postfix calculator maken. Later zullen we 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 bovenstaande algoritme. Je mag jezelf beperken tot optellen en vermenigvuldigen.

Antwoord:

```
// Gebruik een stack voor een post-fix calculator
// Invoer afsluiten met =

#include <iostream>
#include <cctype>
#include "stacklist.h"

using namespace std;

int main() {
    StackWithList<int> s;
    char c;
    int i;
    cout<<"Type een postfix expressie en sluit af met ="<<endl;
    cin>>c;
    while (c!='=') {
        if (isdigit(c)) {
            cin.putback(c);
            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 1963 heeft Floyd het volgende algoritme gepubliceerd 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 openenen 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.

Laten we als voorbeeld eens bekijken hoe de expressie:

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

omgezet wordt in de postfix expressie:

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

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.

Antwoord:
Deze mag je echt zelf doen :-)

We kunnen nu dit programma combineren met de al eerder gemaakte postfix calculator zodat een infix calculator ontstaat. Het is dan netjes om het geheel in een class `Calculator` in te kapselen zodat de calculator eenvoudig kan worden (her)gebruikt.

4 Voorbeelden van de implementatie van een datastructuur.

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 _THR_Bd_StackWithArray_
#define _THR_Bd_StackWithArray_

#include "stack.h"

template <typename T> class StackWithArray: public Stack<T> {
public:
    explicit StackWithArray(int size);
    ~StackWithArray();
    virtual void push(const T& t);
    virtual void pop();
    virtual const T& top() const;
    virtual bool empty() const;
    virtual bool full() const;
```

```
private:
    T* a;  // pointer naar de array
    int s; // size van a (max aantal elementen op de stack)
    int i; // index in a van de top van de stack
};

template <typename T> StackWithArray<T>::StackWithArray(int size):
        a(0), s(size), i(-1) {
    if (s<=0) {
        cerr<<"Stack size should be >0"<<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())
        cerr<<"Can't push on a full stack"<<endl;
    else
        a[++i]=t;
}
template <typename T> void StackWithArray<T>::pop() {
    if (empty())
        cerr<<"Can't pop from an empty stack"<<endl;
    else
        --i;
}
template <typename T> const T& StackWithArray<T>::top() const {
    if (empty()) {
        cerr<<"Can't top from an empty stack"<<endl;
        exit(-1);
        // no valid return possible
    }
    return a[i];
}
template <typename T> bool StackWithArray<T>::empty() const {
    return i==-1;
}
template <typename T> bool StackWithArray<T>::full() const {
    return i==s-1;
}

#endif
```

Een programma 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;
}
```

4.2 Stack met behulp van gelinkte lijst.

Inhoud van de file stacklist.h:

```
#ifndef _THR_Bd_StackWithList_
#define _THR_Bd_StackWithList_

#include "stack.h"

template <typename T> class StackWithList: public Stack<T> {
public:
    StackWithList();
    virtual ~StackWithList();
    virtual void push(const T& t);
    virtual void pop();
    virtual const T& top() const;
    virtual bool empty() const;
    virtual bool full() const;
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() {
    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())
        cerr<<"Can't pop from an empty stack"<<endl;
    else {
        Node* old(p);
        p=p->next;
        delete old;
    }
}
template <typename T> const T& StackWithList<T>::top() const {
    if (empty()) {
        cerr<<"Can't top from an empty stack"<<endl;
```

```
        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
```

Een programma 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 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.

```
#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;
}
```

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. (Zie eventueel TICPPV1 Chapter10.html#Heading303 [4].)
- Exceptions. (Zie eventueel TICPPV2 Chapter 1.)
- RTTI (Run Time Type Information). (Zie eventueel TICPPV2 Chapter 8.)

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 6.5 tot en met 6.7 van het dictaat "Object Georiënteerd Programmeren in C++" dat je bij PROS3 hebt gebruikt. Gezien de beperkte tijd worden deze onderwerpen niet behandeld bij deze module maar wel als huiswerk opgegeven. Zie: http://bd.eduweb.hhs.nl/ogoprg/pdf/Dictaat_OGPiCpp.pdf.

[4] Deze "link" verwijst naar de HTML versie van "Thinking in C++" van Bruce Eckel die je gratis kunt ophalen van: http://mindview.net/Books/TICPP/ThinkingInCPP2e.html.

6 De ISO/ANSI standard C++ library.

De C++ standard library zal aan de hand van het boek “Thinking in C++, Volume 2” (hoofdstuk 3 t/m 7) besproken worden. In dit hoofdstuk van het dictaat zijn alleen enkele voorbeelden opgenomen.

6.1 Voorbeeldprogramma met een standaard `vector`.

Dit voorbeeld laat zien:
- hoe een standaard vector gevuld kan worden.
- hoe door (een deel van) de vector heengelopen kan worden door middel van indexering.
- hoe door (een deel van) de vector heengelopen kan worden door middel van een iterator.
- hoe het gemiddelde van een rij getallen (opgeslagen in een vector) bepaald kan worden.

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

//  Adrukken van een vector door middel van een iterator.
void print2(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;
}

//  Berekenen van het gemiddelde door middel van een iterator.
double gem(const vector<int>& vec) {
    double som(0.0);
    for (vector<int>::const_iterator iter(vec.begin());
         iter!=vec.end(); ++iter) {
        som+=*iter;
    }
    return som/vec.size();
}

int main() {
//  Vullen van een vector.
    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);
    cout<<"Het gemiddelde is: "<<gem(v)<<endl;
//  Deel van een vector bewerken door middel van een iterator.
```

```
    cout<<"Nu wordt een deel van de vector bewerkt."<<endl;
    if (v.size()>=4) {
        for (vector<int>::iterator iter(v.begin()+2);
             iter!=v.begin()+4; ++iter) {
            *iter*=2;
        }
    }
    print2(v);
//  Deel van een vector bewerken door middel van indexering.
    cout<<"Nu wordt de vorige bewerking weer teruggedraaid."<<endl;
    if (v.size()>=4) {
        for (vector<int>::size_type i(2); i<4; ++i) {
            v[i]/=2;
        }
    }
    print1(v);
    cin.get();
    return 0;
}
```

6.2 Voorbeeldprogramma met een standaard `stack`.

Dit programma is identiek aan het programma uit paragraaf 3.1 maar in plaats van de zelfgemaakte stack wordt nu de standaard stack gebruikt.

```
// Controleer op gebalanceerde haakjes. 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 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<<"Haakje openen ontbreekt."<<endl;
                }
                else {
                    char d(s.top());
                    s.pop();
                    if (d=='('&&c!=')' || d=='{'&&c!='}' ||
                                                d=='['&&c!=']') {
                        cout<<"Haakje openen ontbreekt."<<endl;
                    }
                }
            }
        }
        cin.get(c);
    }
    if (!s.empty()) {
        cout<<"Haakje sluiten ontbreekt."<<endl;
    }
```

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

6.3 Voorbeeldprogramma met een standaard `set`.

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

void print(const set<string>& s) {
    cout<<"De set bevat: ";
    for (set<string>::const_iterator i(s.begin()); i!=s.end(); ++i)
        cout<<*i<<" ";
    cout<<endl;
}

int main() {
    set<string> docenten;
    docenten.insert("Ineke");
    docenten.insert("Harm");
    docenten.insert("Jan");
    docenten.insert("Harry");
    docenten.insert("Theo");
    print(docenten);
    pair<set<string>::iterator, bool> result(docenten.insert("Harry"));
    if (result.second==false)
        cout<<"1 Harry is genoeg."<<endl;
    cout<<"Er is "<<docenten.count("Jan")<<" Jan."<<endl;
    docenten.erase("Harry");
    print(docenten);
    cin.get();
    return 0;
}
```

De uitvoer van dit programma:

```
De set bevat: Harm Harry Ineke Jan Theo
1 Harry is genoeg.
Er is 1 Jan.
De set bevat: Harm Ineke Jan Theo
```

6.4 Voorbeeldprogramma met een standaard `multiset` (bag).

Vergelijk dit programma met het vorige en let op de verschillen tussen een `set` en een `multiset`.

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

void print(const multiset<string>& s) {
    cout<<"De bag bevat: ";
    for (multiset<string>::const_iterator i(s.begin()); i!=s.end(); ++i)
        cout<<*i<<" ";
    cout<<endl;
}
```

```
int main() {
    multiset<string> docenten;
    docenten.insert("Ineke");
    docenten.insert("Harm");
    docenten.insert("Jan");
    docenten.insert("Harry");
    print(docenten);
    docenten.insert("Harry");
    print(docenten);
    cout<<"Er zijn "<<docenten.count("Harry")<<" Harry's."<<endl;
    docenten.erase("Harry");
    print(docenten);
    multiset<string>::iterator itr(docenten.find("Ineke"));
    docenten.erase(itr, docenten.end());
    print(docenten);
    cin.get();
    return 0;
}
```

De uitvoer van dit programma:

```
De bag bevat: Harm Harry Ineke Jan
De bag bevat: Harm Harry Harry Ineke Jan
Er zijn 2 Harry's.
De bag bevat: Harm Ineke Jan
De bag bevat: Harm
```

6.5 Voorbeeldprogramma met een standaard `map`.

Dit voorbeeldprogramma gebruikt een map om de woordfrequentie te tellen. Van de belangrijkste C/C++ keywords wordt het aantal maal dat ze voorkomen afgedrukt.

```
#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.c_str());
    while (fin>>w) {
        ++freq[w];
    }
    for (map<string, int>::const_iterator i(freq.begin());
                                              i!=freq.end(); ++i) {
        cout<<i->first<<" "<<i->second<<endl;
    }
    cout<<"Belangrijkste 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();
    return 0;
}
```

6.6 Voorbeeldprogramma met standaard streamiteratoren.

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

int main() {
    vector<int> rij;
    ifstream fin("getallen.txt");
    istream_iterator<int> iin(fin);
    istream_iterator<int> einde;
    copy(iin, einde, back_inserter(rij));
    sort(rij.begin(), rij.end());
    ostream_iterator<int> iout(cout, " ");
    copy(rij.begin(), rij.end(), iout);
    cin.get();
    return 0;
}
```

6.7 Voorbeeldprogramma met het standaard algoritme `find`.

In dit programma wordt het find algoritme gebruikt om het bekende spelletje galgje te implementeren.

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

int main() {
    string s("galgje");
    set<int> v;
    do {
        for (string::size_type i(0); i<s.size(); ++i)
            if (find(v.begin(), v.end(), i)==v.end())
                cout<<'.';
            else
                cout<<s[i];
        cout<<endl<<"Geef een letter: ";
        char c;
        cin>>c;
        string::iterator r(s.begin());
        while ((r=find(r, s.end(), c))!=s.end()) {
            v.insert(r-s.begin());
            ++r;
        }
    }
    while (v.size()!=s.size());
    cout<<s;
    cin.get();
    cin.get();
    return 0;
```

```
}
```

Invoer en uitvoer van dit programma:

```
......
Geef een letter: h
......
Geef een letter: a
.a....
Geef een letter: r
.a....
Geef een letter: r
.a....
Geef een letter: y
.a....
Geef een letter: g
ga.g..
Geef een letter: e
ga.g.e
Geef een letter: l
galg.e
Geef een letter: j
galgje
```

6.8 Voorbeeldprogramma met het standaard algoritme `find_if`.

Dit programma laat zien hoe je het standaard algoritme `find_if` kan 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* met een overloaded `operator()` die een `bool` teruggeeft die aangeeft of aan de voorwaarde wordt voldaan.
- door middel van een *standaard functie-object*. In dit geval gebruiken we een standaard *comparisor* die wordt omgezet in een *predicate* met behulp van een *binder*.

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

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

class IsPos {
public:
    bool operator()(int i) const {
        return i>=0;
    }
};

int main() {
    list<int> l;
    l.push_back(-3);
    l.push_back(-4);
    l.push_back(3);
    l.push_back(4);
    list<int>::iterator r;
```

```
//  Zoeken met een functie als zoekvoorwaarde.
    r=find_if(l.begin(), l.end(), ispos);
    if (r!=l.end())
        cout<<"Het eerste positieve element is: "<<*r<<endl;
//  Zoeken met een functie-object als zoekvoorwaarde.
    r=find_if(l.begin(), l.end(), IsPos());
    if (r!=l.end())
        cout<<"Het eerste positieve element is: "<<*r<<endl;
//  Zoeken met een standaard functie-object als zoekvoorwaarde.
    r=find_if(l.begin(), l.end(), bind2nd(greater_equal<int>(),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
```

6.9 Voorbeeldprogramma met het standaard algoritme `for_each`.

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

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

int main() {
    vector<int> v;
    v.push_back(-3);
    v.push_back(-4);
    v.push_back(3);
    v.push_back(4);
    ostream_iterator<int> iout(cout, " ");
    copy(v.begin(), v.end(), iout);
    cout<<endl;
    for_each(v.begin(), v.end(), printDubbel);
    cout<<endl;
    cin.get();
    return 0;
}
```

De uitvoer van dit programma:

```
-3 -4 3 4
-3 -3 -4 -4 3 3 4 4
```

6.10 Voorbeeldprogramma met het standaard algoritme `transform`.

Dit programma laat zien hoe je het standaard algoritme `transform` kan gebruiken om een vector bij een andere vector op te tellen. De transformatie (bewerking) wordt op twee verschillende manieren opgegeven:

- door middel van een *functie* die de transformatie uitvoert.
- door middel van een *standaard functie-object*. In dit geval gebruiken we een standaard *arithmetic* functie-object.

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

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

int main() {
    vector<int> v;
    v.push_back(-3);
    v.push_back(-4);
    v.push_back(3);
    v.push_back(4);
    vector<int> w;
    w.push_back(1);
    w.push_back(2);
    w.push_back(3);
    w.push_back(4);
    ostream_iterator<int> iout(cout, " ");
    copy(v.begin(), v.end(), iout);
    cout<<endl;
//  Bewerking opgeven met een functie.
    transform(v.begin(), v.end(), w.begin(), v.begin(), telop);
    copy(v.begin(), v.end(), iout);
    cout<<endl;
//  Bewerking opgeven met standaard functie-objecten.
    transform(v.begin(), v.end(), w.begin(), v.begin(), plus<int>());
    copy(v.begin(), v.end(), iout);
    cout<<endl;
    cin.get();
    return 0;
}
```

De uitvoer van dit programma:

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

6.11 Voorbeeldprogramma met het standaard algoritme `remove`.

Na `remove` is nog een `erase` nodig om de elementen echt te verwijderen

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

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

int main() {
    vector<int> v;
    for (vector<int>::size_type i(0); i<10; ++i) {
        v.push_back(i*i);
    }
    ostream_iterator<int> out(cout, " ");
    cout<<"Na initialisatie:"<<endl;
    copy(v.begin(), v.end(), out);
    vector<int>::iterator end(remove_if(v.begin(), v.end(),
                                not1(bind2nd(modulus<int>(), 2))));
    cout<<endl<<"Na remove (tot returned iterator):"<<endl;
    copy(v.begin(), end, out);
    cout<<endl<<"Na remove (hele vector):"<<endl;
    copy(v.begin(), v.end(), out);
    v.erase(end, v.end());
    cout<<endl<<"Na erase (hele vector):"<<endl;
    copy(v.begin(), v.end(), out);
// ...
```

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
```

6.12 Voorbeeldprogramma waarin generiek en object georiënteerd programmeren zijn gecombineerd.

De standaard functie `mem_fun` vormt de koppeling tussen generiek programmeren en object georienteerd programmeren omdat hiermee een memberfunctie kan worden omgezet in een functie-object.

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

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

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

```
};

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

int main() {
    list<Hond*> kennel;
    kennel.push_back(new Tekkel);
    kennel.push_back(new StBernard);
    kennel.push_back(new Tekkel);
    for_each(kennel.begin(), kennel.end(), mem_fun(&Hond::blaf));
// ...
```

De uitvoer van dit programma:

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

7 Toepassingen van standaard datastructuren.

Het onderwijsmatriaal voor dit hoofdstuk wordt uitgedeeld in de les. Een van de toepassingen die wordt besproken is het spelletje Boter Kaas en Eieren. Je kunt (een deel hiervan) ook ophalen vanaf http://bd.eduweb.hhs.nl/aldat/bke.htm. Een andere toepassing die wordt besproken is het spel “galgje”.

8 Lambda expressies.

Het onderwijsmatriaal voor dit hoofdstuk wordt uitgedeeld in de les. Je kunt het ook vinden op http://bd.eduweb.hhs.nl/aldat/pdf/Handout1_ALDAT.pdf.