Dictaat SYSO1

Studiewijzer.

onderwijsdeel:	Datastructuren en Algoritmen
onderwijseenheid:	SYSO1
code onderwijsdelen:	SYSO1I0T1 en SYSO1I0P1
studiebelasting:	112 SBU
semester / kwartaal:	H3 SYSO / 1
contacturen:	3 uur/week college 2 uur/week practicum
toetsing:	tentamen (cijfer) en practicumbeoordeling (O/V)
benodigde voorkennis:	alle programmeervakken uit P, H1 en H2
verantwoordelijke docent:	Harry Broeders

Inleiding

In de vorige semesters heb je datastructuren zoals array en struct leren gebruiken. Deze datastructuren worden statisch genoemd omdat hun grootte tijdens het compileren wordt bepaald en vast is. In de praktijk heb je al snel behoefte aan zogenaamde dynamische datastructuren waarvan de grootte tijdens het uitvoeren van het programma kan wijzigen. In de vorige semesters heb je ook al kennisgemaakt met enkele dynamische datastructuren (string, vector, tree). Tijdens dit onderwijsdeel worden alle klassieke dynamische datastructuren behandeld. Daarbij ligt de nadruk op de eigenschappen en het gebruik van deze datastructen. Van enkele datastructuren wordt ook de implementatie behandeld. In de vorige semesters heb je al enkele bekende algoritmen leren kennen. In dit onderwijsdeel zul je leren hoe het met behulp van templates mogelijk is om algoritmen zoals zoeken en sorteren generiek te definiëren. Een generiek algoritme is een algoritme dat onafhankelijk is van de gebruikte datastructuur.

In H1 heb je de basisaspecten van object georiënteerd programmeren (OOP) geleerd aan de hand van de programmeertaal C++. In deze onderwijseenheid wordt dieper op deze programmeertaal ingegaan. Het opvangen van fouten door middel van exceptions wordt besproken en ook namespaces en run-time type information komen aan de orde.

In de C++ ISO/ANSI standaard is een verzameling generieke algoritmen en datastructuren opgenomen die in deze onderwijseenheid worden behandeld.

Je zult na afloop van het dit onderwijsdeel in staat zijn om standaard algoritmen en datastructuren te gebruiken. Je kunt programma's die gebruik maken van deze algoritmen en datastructuren ontwerpen, implementeren en testen. Je kunt uitgaande van de eisen die aan een programma gesteld worden de juiste standaard algoritmen en datastructuren selecteren.

Leerdoelen

Als je deze onderwijseenheid met een voldoende hebt afgesloten:

- snap je het nut van dynamische datastructuren (in plaats van statische datastructuren).
- snap je dat niet alleen code maar ook data recursief gedefinieerd kan worden.
- ken je het begrip "orde van een algoritme" en de big-O notatie.
- ken je de eigenschappen, voor- en nadelen van de meest gebruikte datastructuren.
- ken je een aantal toepassingen van de meest gebruikte datastructuren.
- kun je gebruik maken van als ADT's gedefinieerde datastructuren.
- kun je met behulp van templates eenvoudige datastructuren implementeren.
- weet je hoe in C++ onverwachte omstandigheden en fouten op een nette manier afgehandeld kunnen worden door het gebruik van exceptions.
- weet je hoe je software exception safe kunt maken.

- ben je bekend met het begrip namespace en weet je hoe namespaces gebruikt kunnen worden om meerdere class libraries te combineren.
- heb je een overzicht van de in de ISO/ANSI standaard C++ opgenomen containers, algoritmen en iteratoren.
- kun je de in de ISO/ANSI standaard C++ opgenomen eenvoudige containers, algoritmen en iteratoren met behulp van je boek gebruiken.
- herken je bij het ontwerpen van een programma de mogelijkheden voor het gebruik van standaard algoritmen en datastructuren.
- kun je bij het ontwerpen van een programma een goed afgewogen keuze maken tussen de verschillende standaard algoritmen en datastructuren.

Literatuur

Mark Allen Weiss, Data Structures and Problem Solving Using C++, 2/E, ISBN: 0-201-61250-X, Addison Wesley, Copyright: 2000, 879 pp

Broeders, Dictaat: SYSO1
De sheets en voorbeeldprogramma's zijn beschikbaar op het internet http://bd.thrijswijk.nl/syso1/.

De practicumopdrachten zijn beschikbaar op http://bd.thrijswijk.nl/syso1/. Daar kun je ook verwijzingen en achtergrondinformatie vinden.

Toetsing en beoordeling.

Er worden voor deze onderwijsdelen twee deelresultaten vastgesteld. SYSO1I0T1 heeft als resultaat (tentamen) een cijfer (1..10), het resultaat van SYSO1I0P1 (practicum) is een O(nvoldoende) of V(oldoende). Bij het tentamen mag je boeken en dit dictaat gebruiken. Het tentamen bestaat uit open vragen.

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 theorie.

"les"	studiemateriaal	dictaat	onderwerp
1	dictaat inleiding, boek H6	7	Inleiding
2	dictaat H2	9	Overzicht datastructuren
3 en 4	dictaat H3 en H4, boek H12 en H16	10	Toepassingen en implementaties van een stack
5 t/m 7	dictaat H5 en boek H1 t/m 4 en Appendix A	19	Advanced C++
8	dictaat H6 en boek H5		Design Patterns (Functor, Adaptor, Iterator)
9 t/m 12	dictaat H7 en boek H7 en H9	56	Standard Template Library
13			Uitloop / behandelen vraagstukken
14 t/m 17	boek H8, H11, H13 en H14		Toepassingen van datastructuren.
18 en 19	boek H15		Graphs en hun toepassingen.
20 en 21	boek H20 en H21		Hash table en priority queue implementaties.

Een gedetailleerde planning kun je vinden op het internet: http://bd.thrijswijk.nl/syso1.

Weekplanning practicum.

"week"	opdracht
1	Homemade gelinkte lijst (toepassen van dynamisch geheugen, recursieve data en recursieve algoritmen).
2	Vergelijken van bubble sort $O(n^2)$ en quick sort ($n \bullet \log n$). Bepalen van de orde van een zelf ontworpen functie om het meerderheidselement in een array te bepalen (toepassen van algoritme analyse).
3	Fouten afvangen bij een generieke stack (toepassen van exceptions).
4	Quick sort generiek maken (toepassen van templates, functiepointers en functors).
5	Opsporen van fraude (toepassen van map, iteratoren en algoritmen).
6	Boter Kaas en eieren (toepassen van minimax algoritme met alpha-beta pruning en transposition table).
7	Uitloop

Inhoudsopgave.

Inleiding.

Dit is het dictaat: “Datastructuren en Algoritmen”. Dit dictaat wordt gebruikt in combinatie met het boek: “*Data Structures and Problem Solving Using C++, 2/E*” van Mark Allen Weiss (2000 Addison Wesley).

In dit dictaat zijn enkele extra voorbeeldprogramma’s en enige extra theorie opgenomen. Dit dictaat is zoals alle mensenwerk niet foutloos, verbeteringen en suggesties zijn altijd welkom!

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.

In H1 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.

In H1 heb je ook geleerd om de ingebouwde datastructuur `struct` te combineren met pointers. Hierdoor kunnen ook de problemen met de lijst van deelnemers worden opgelost. Het fundamentele inzicht dat je moet hebben 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 versie 6 van Borland C++ Builder (de compiler die wij gebruiken) meegeleverd. In dit semester gaan we uitgebreid op deze STL library in.

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

[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 toegankelijker boek is "Algorithms in C++" van R. Sedgewick.

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;
    virtual bool empty() const =0;
    virtual bool full() const =0;
private:
```

[3] In de ISO/ANSI C++ standaard library is ook een type stack gedefinieerd, zie hoofdstuk 7.

```
    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 2 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<<"Fout haakje openen ontbreekt."<<endl;
                }
                else {
                    char d(s.top());
                    s.pop();
                    if (d=='('&&c!=')'||d=='{'&&c!='}'||
                                                d=='['&&c!=']') {
                        cout<<"Fout haakje openen ontbreekt."<<endl;
                    }
                }
            }
        }
        cin.get(c);
    }
    if (!s.empty()) {
        cout<<"Fout 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
```

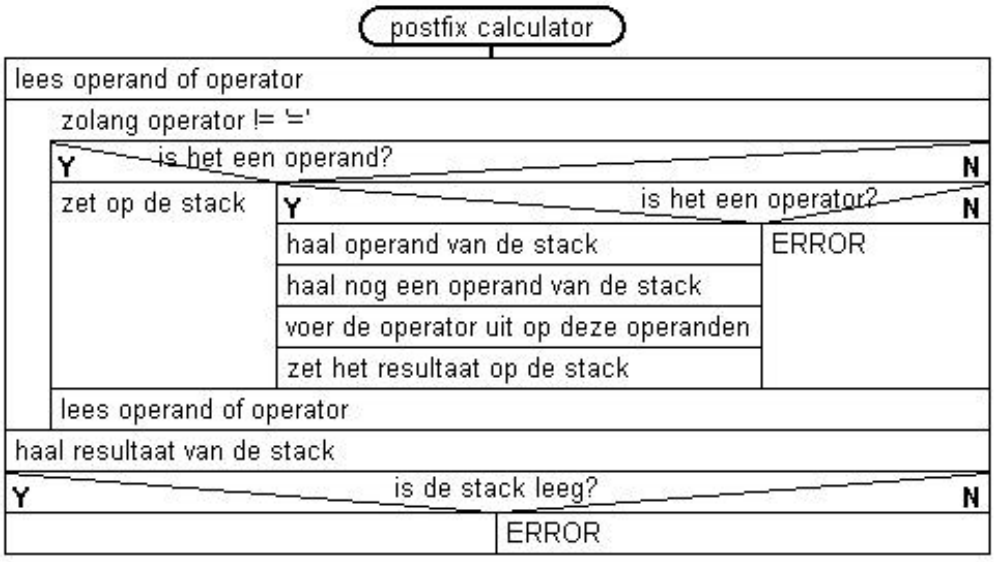
en

```
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 (of vector) 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 stackvector.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:

- `vector`. Zie boek 1.2.
- `static` class members. Zie boek 2.4.2.
- `enum` class members. Zie boek 2.4.3.
- Default template parameters. Zie boek 3.6.2.
- Overerving en overloading. Zie boek 4.4.1.
- Covariant return types for overridden methods. Zie boek 4.4.4
- `stringstream`. Zie boek A.4.3
- Namespaces. Zie boek A.5 dictaat 5.1
- Exceptions. Zie boek 2.5 dictaat 5.2
- casting en RTTI (Run Time Type Information). Zie dictaat 5.3

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.

5.1 `vector`.

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

- Een `vector` is in tegenstelling tot een built-in array een "echt" object.
- Je kunt een `vector` "gewoon" vergelijken en toekennen.

- Een `vector` kan groeien en krimpen.
- Een `vector` heeft memberfuncties:
  - `size()` geef het aantal elementen in de vector.
  - `at(...)` geef element op de opgegeven positie. Bijna gelijk aan `operator[]` maar `at` controleert of de index geldig is en gooit een exception als dit niet zo is.
  - `capacity()` geef het aantal elementen waarvoor geheugen gereserveerd is. Als de `vector` groter groeit worden automatisch meer elementen gereserveerd. De capaciteit wordt dan telkens verdubbeld. Hierdoor blijft de gemiddelde tijd voor het toevoegen van een element van de orde O(1).
  - `push_back(...)` voeg een element toe aan de `vector`. Na afloop is de `size` van de `vector` dus met 1 toegenomen.
  - `resize(...)` Verander de `size` van de `vector`.
  - ...

Voorbeeld van een programma met een `vector`.

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

int main() {
    // definieer vector van 10 integers
    vector<int> v(10);
    // vul met kwadraten
    for (vector<int>::size_type i(0); i<v.size(); ++i) {
        v[i]=i*i;
    }
    // druk af
    for (vector<int>::size_type i(0); i<v.size(); ++i) {
        cout<<v[i]<<" ";
    }
    cout<<endl;
    // maak een kopietje van een vector
    vector<int> w(v);
    for (vector<int>::size_type i(0); i<w.size(); ++i) {
        cout<<w[i]<<" ";
    }
    cout<<endl;
    // definieer vector van 0 integers
    vector<int> x;
    // toekennen van de een vector
    x=w;
    for (vector<int>::size_type i(0); i<x.size(); ++i) {
        cout<<x[i]<<" ";
    }
    cout<<endl;
    // vergelijken van vectoren
    if (x!=w)
        cout<<"DIT KAN NIET!"<<endl;

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

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

Uitvoer:

```
0 1 4 9 16 25 36 49 64 81
0 1 4 9 16 25 36 49 64 81
0 1 4 9 16 25 36 49 64 81
```

Voorbeeldprogramma om een onbekend aantal elementen in een `vector` in te lezen.

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

// vergelijk met figuur 1.3 uit boek van Weiss.

void getInts(vector<int>& vec) {
    vector<int>::size_type itemsRead(0);
    int inputValue;
    cout<<"Enter any number of integers ";
    cout<<"(end with non numeric character):"<<endl;
    while (cin>>inputValue) {
        if (itemsRead==vec.size())
            vec.resize(vec.size()*2+1);
        vec[itemsRead++]=inputValue;
    }
    vec.resize(itemsRead);
}

int main() {
    // definieer vector van 10 integers
    vector<int> v(10);
    // vul en pas indien nodig grootte aan.
    getInts(v);
    // druk af
    for (vector<int>::size_type i(0); i<v.size(); ++i) {
        cout<<v[i]<<" ";
    }
    cout<<endl;
//  ...
}
```

De functie `getInts` kan met behulp van `push_back` veel eenvoudiger geïmplementeerd worden.

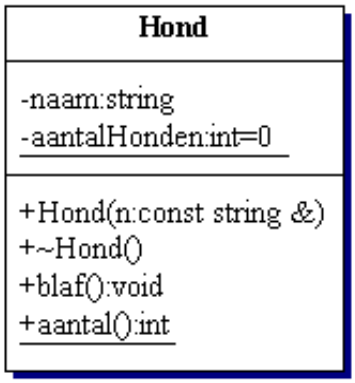
```
void getInts(vector<int>& vec) {
    vec.resize(0);
    int inputValue;
    cout<<"Enter any number of integers ";
    cout<<"(end with non numeric character):"<<endl;
    while (cin>>inputValue) {
        vec.push_back(inputValue);
    }
}
```

5.2 `static` class members.

In semester H1 heb je geleerd dat elk object zijn eigen datamembers heeft terwijl de memberfuncties door alle objecten van een bepaalde class "gedeeld" worden. Stel nu dat je wilt tellen hoeveel objecten van een bepaalde class “levend” zijn. Dit zou kunnen door een globale “teller” te definiëren die in de

constructor van de class met 1 wordt verhoogd en in de destructor weer met 1 wordt verlaagd. Het gebruik van een globale variabele maakt het programma echter slecht onderhoudbaar.

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




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

class Hond {
public:
     Hond(const string& n);
     ~Hond();
     void blaf() const;
     static int aantal();
private:
     string naam;
     static int aantalHonden;
};

int Hond::aantalHonden=0;

Hond::Hond(const string& n): naam(n) {
     ++aantalHonden;
}
Hond::~Hond() {
     --aantalHonden;
}
int Hond::aantal() {
     return aantalHonden;
}
void Hond::blaf() const {
     cout<<naam<<" zegt: WOEF"<<endl;
}

int main() {
     cout<<"Er zijn nu "<<Hond::aantal()<<" honden."<<endl;
     {
          Hond h1("Boris");
          h1.blaf();
          cout<<"Er zijn nu "<<Hond::aantal()<<" honden."<<endl;
          Hond h2("Fikkie");
          h2.blaf();
          cout<<"Er zijn nu "<<Hond::aantal()<<" honden."<<endl;
     }
     cout<<"Er zijn nu "<<Hond::aantal()<<" honden."<<endl;
     cin.get();
     return 0;
}
```

Uitvoer:

```
Er zijn nu 0 honden.
Boris zegt: WOEF
Er zijn nu 1 honden.
Fikkie zegt: WOEF
Er zijn nu 2 honden.
Er zijn nu 0 honden.
```

Je kunt een `static` memberfunctie op twee manieren aanroepen.

- via een object van de class: `object_naam.member_functie_naam(parameters)`
  Voorbeeld: `cout<<h1.aantal()<<endl;`
- direct via de classnaam: `class_naam::member_functie_naam(parameters)`
  Voorbeeld: `cout<<Honden::aantal()<<endl;`

De laatste methode heeft de voorkeur omdat die ook gebruikt kan worden als er nog geen objecten zijn.

Een `static` memberfunctie heeft ten opzichte van een “normale” memberfunctie de volgende beperkingen:

- Een `static` memberfunctie heeft geen receiver (ook niet als hij via een object aangeroepen wordt).
- Een `static` memberfunctie heeft dus geen `this` pointer.
- Een `static` memberfunctie kan dus geen "gewone" memberfuncties aanroepen en ook geen "gewone" datamembers gebruiken.

5.3 Constanten in een class.

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

Voorbeeld met `static` constanten:

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

Deze constanten kunnen als volgt gebruikt worden:

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

Voorbeeld met anonymous (naamloze) `enum`.

```
class Color {
public:
    Color();
    Color(int c);
    int getValue() const;
```

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

Deze constanten kunnen als volgt gebruikt worden:

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

5.4 Overloading en overerving.

In H1 heb je al gezien dat overloading en overerving niet goed samengaan. Een memberfunctie kan een andere memberfunctie met dezelfde naam *overloaden*. Bij een aanroep van de memberfunctienaam wordt de juiste memberfunctie door de compiler aangeroepen door naar de argumenten bij aanroep te kijken.

Voorbeeld van het gebruik van overloading van memberfuncties:

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

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

Uitvoer:

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

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

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

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

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

class Derived: public Base {
public:
    void f(int i) const[4] { // Verberg f() !! Geen goed idee !!!
        cout<<"Ik ben f(int)"<<endl;
    }
};

int main() {
    Base b;
    Derived d;
    b.f();
//  d.f();
//  [C++ Error]: Too few parameters in call to 'Derived::f(int)'[5]
    d.f(3);
    d.Base::f();[6]
// ...
```

Uitvoer:

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

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

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

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

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

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

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

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

Uitvoer:

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

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

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

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

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

Er is nog een reden dat overloading en overerving niet goed samengaan. Een functie met een basis class als argument kan overloaded worden met een functie met een afgeleide class als argument. Als deze functie wordt aangeroepen dan wordt door de compiler de aan te roepen functie gekozen. Dit gaat niet goed als de parameter een polymorphic pointer of reference is. De functie behorende bij het type van de pointer of de reference wordt aangeroepen en niet het type waarnaar de reference refereerd of waar de pointer naar wijst. In dit geval wordt dus **niet** het verwachte (polymorphic) type gekozen.

Voorbeeld waarin overerving en overloading op **onjuiste** wijze worden gecombineerd:

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

class Hond {
public:
    Hond(const string& n);
    const string& geefNaam() const;
    virtual ~Hond();
// ...
private:
    string naam;
};

class SintBernard: public Hond {
public:
    SintBernard(const string& n);
// ...
};

Hond::Hond(const string& n): naam(n) {
```

```
}

const string& Hond::geefNaam() const {
    return naam;
}

Hond::~Hond() {
}

SintBernard::SintBernard(const string& n): Hond(n) {
}

// Gebruik overloading om de operator<< te definieren:

ostream& operator<<(ostream& o, const Hond& h) {
    return o<<"Ik ben een Hond en heet: "<<h.geefNaam()<<endl;
}

ostream& operator<<(ostream& o, const SintBernard& b) {
    return o<<"Ik ben een Sint Bernard en heet: "<<b.geefNaam()<<endl;
}

int main() {
    Hond fikkie("Fikkie");
    SintBernard boris("Boris");

    cout<<fikkie;
    cout<<boris;

    Hond* hp(0);
    hp=&fikkie;  // een Hond* kan wijzen naar een Hond
    cout<<*hp;
    hp=&boris;   // een Hond* kan ook wijzen naar een SintBernard
    cout<<*hp;

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

Uitvoer:

```
Ik ben een Hond en heet: Fikkie
Ik ben een Sint Bernard en heet: Boris
Ik ben een Hond en heet: Fikkie
Ik ben een Hond en heet: Boris
```

Je ziet dat de laatste regel van de uitvoer niet goed is omdat bij het kiezen van een functie uit een aantal overloaded functies het statische type wordt gebruikt dus:

```
    cout<<*hp;
```

wordt altijd vertaald in een aanroep naar:

```
    ostream& operator<<(ostream& o, const Hond& h);
```

Dit kan ook niet anders want deze keuze wordt al tijdens het vertalen van het programma gemaakt (en de compiler kan nooit weten naar welk soort `Hond` de `Hond* hp` staat te wijzen tijdens het uitvoeren van het programma!

Als je overriding gebruikt in plaats van overloading is de uitvoer wel zoals verwacht.

```
#include <iostream>
```

```
using namespace std;

class Hond {
public:
    Hond(const string& n);
    const string& geefNaam() const;
    virtual ~Hond();
    virtual void print(ostream& o) const;
// ...
private:
    string naam;
};

class SintBernard: public Hond {
public:
    SintBernard(const string& n);
    virtual void print(ostream& o) const; // print is overridden.
// ...
};

Hond::Hond(const string& n): naam(n) {
}

const string& Hond::geefNaam() const {
    return naam;
}

void Hond::print(ostream& o) const {
    o<<"Ik ben een Hond en heet: "<<geefNaam()<<endl;
}

Hond::~Hond() {
}

SintBernard::SintBernard(const string& n): Hond(n) {
}

void SintBernard::print(ostream& o) const {
    o<<"Ik ben een Sint Bernard en heet: "<<geefNaam()<<endl;
}

ostream& operator<<(ostream& o, const Hond& h) {
    h.print(o);
    return o;
}

int main() {
    Hond fikkie("Fikkie");
    SintBernard boris("Boris");

    cout<<fikkie;
    cout<<boris;

    Hond* hp(0);
    hp=&fikkie;  // een Hond* kan wijzen naar een Hond
    cout<<*hp;
    hp=&boris;   // een Hond* kan ook wijzen naar een SintBernard
    cout<<*hp;

    cin.get();
```

```
    return 0;
}
```

Uitvoer:

```
Ik ben een Hond en heet: Fikkie
Ik ben een Sint Bernard en heet: Boris
Ik ben een Hond en heet: Fikkie
Ik ben een Sint Bernard en heet: Boris
```

Je ziet dat de laatste regel van de uitvoer nu wel goed is omdat vanuit de functie:
`ostream& operator<<(ostream& o, const Hond& h);`
de virtual memberfunctie `print(ostream& o)` wordt aangeroepen.

Er wordt dus (tijdens het uitvoeren van het programma) een boodschap gestuurd naar de door de `Hond* hp` aangewezen hond!

5.5 Covariant return type.

Tot nu toe heb je geleerd dat het prototype van een functie die een andere functie overridde **exact** gelijk moet zijn aan het prototype van de functie die overridden wordt. Dit kan echter tot vreemde problemen leiden. In het onderstaande voorbeeld worden honden “gekloond” met een “normale” virtuele functie.

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

class Hond {
public:
    Hond(const string& n);
    const string& geefNaam() const;
    virtual ~Hond();
    virtual void print(ostream& o) const;
    virtual Hond* kloon();
// ...
private:
    string naam;
};

class SintBernard: public Hond {
public:
    SintBernard(const string& n);
    virtual void print(ostream& o) const;
    virtual Hond* kloon();
// ...
};

Hond::Hond(const string& n): naam(n) {
}

const string& Hond::geefNaam() const {
    return naam;
}

void Hond::print(ostream& o) const {
    o<<"Ik ben een Hond en heet: "<<geefNaam()<<endl;
}

Hond* Hond::kloon() {
    return new Hond(*this);
```

```
}

Hond::~Hond() {
}

SintBernard::SintBernard(const string& n): Hond(n) {
}

void SintBernard::print(ostream& o) const {
    o<<"Ik ben een Sint Bernard en heet: "<<geefNaam()<<endl;
}

Hond* SintBernard::kloon() {
    return new SintBernard(*this);
}

ostream& operator<<(ostream& o, const Hond& h) {
    h.print(o);
    return o;
}

int main() {
    Hond fikkie("Fikkie");
    SintBernard boris("Boris");

    cout<<fikkie;
    cout<<boris;

    Hond* hp(0);
    hp=fikkie.kloon();
    cout<<*hp;
    delete hp;
    hp=boris.kloon();
    cout<<*hp;
    delete hp;

//  SintBernard* bp(0);
//  bp=boris.kloon();
//  [C++ Error]: Cannot convert 'Hond *' to 'SintBernard *'
//  cout<<*bp;
//  delete bp;

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

Uitvoer:

```
Ik ben een Hond en heet: Fikkie
Ik ben een Sint Bernard en heet: Boris
Ik ben een Hond en heet: Fikkie
Ik ben een Sint Bernard en heet: Boris
```

Je ziet dat het klonen van een `Hond` goed gaat. Een `SintBernard*` kan echter niet wijzen naar een gekloonde `SintBernard` (alleen naar een echte). Dat is **vreemd**!

Dit probleem kan opgelost worden door een zogenaamd *covariant* return type te gebruiken. De regel dat het prototype van een functie die een andere functie override exact gelijk moet zijn aan het prototype van de overridden functie wordt versoepeld. Het returntype van de functie die een andere functie over-

ridde mag een afgeleide class zijn van het return type van de overridden functie. Let op: dit geldt **alleen** voor het **returntype**. In het onderstaande voorbeeld worden honden “gekloond” met een virtual functie met een covariant return type.

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

class Hond {
public:
    Hond(const string& n);
    const string& geefNaam() const;
    virtual ~Hond();
    virtual void print(ostream& o) const;
    virtual Hond* kloon();
// ...
private:
    string naam;
};

class SintBernard: public Hond {
public:
    SintBernard(const string& n);
    virtual void print(ostream& o) const;
    virtual SintBernard* kloon();
// ...
};

Hond::Hond(const string& n): naam(n) {
}

const string& Hond::geefNaam() const {
    return naam;
}

void Hond::print(ostream& o) const {
    o<<"Ik ben een Hond en heet: "<<geefNaam()<<endl;
}

Hond* Hond::kloon() {
    return new Hond(*this);
}

Hond::~Hond() {
}

SintBernard::SintBernard(const string& n): Hond(n) {
}

void SintBernard::print(ostream& o) const {
    o<<"Ik ben een Sint Bernard en heet: "<<geefNaam()<<endl;
}

SintBernard* SintBernard::kloon() {
    return new SintBernard(*this);
}

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

```
int main() {
     Hond fikkie("Fikkie");
     SintBernard boris("Boris");

     cout<<fikkie;
     cout<<boris;

     Hond* hp(0);
     hp=fikkie.kloon();
     cout<<*hp;
     delete hp;
     hp=boris.kloon();
     cout<<*hp;
     delete hp;

     SintBernard* bp(0);
     bp=boris.kloon();
     cout<<*bp;
     delete bp;

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

Uitvoer:

```
Ik ben een Hond en heet: Fikkie
Ik ben een Sint Bernard en heet: Boris
Ik ben een Hond en heet: Fikkie
Ik ben een Sint Bernard en heet: Boris
Ik ben een Sint Bernard en heet: Boris
```

Probleem opgelost!

5.6 Namespaces.

Bij grote programma's kunnen verschillende classes "per ongeluk" dezelfde naam krijgen. In C++ kun je classes (en functies etc.) groeperen in zogenaamde *namespaces*.

```
namespace Bd {
   void f(int);
   double sin(double x);
}

// andere file zelfde namespace:
namespace Bd {
   class string {
   //...
   };
}

// andere namespace:
namespace Vi {
   class string {
   //...
   };
}
```

Je ziet dat in de namespace `Bd` een functie `sin` is opgenomen die ook in de standaard library is opgenomen. De in de namespace `Bd` gedefinieerde class `string` is ook in de namespace `Vi` en ook in de standaard library gedefinieerd. Je hebt in H1 gezien dat alle functies en classes uit de standaard library in de namespace `std` zijn opgenomen. Je kunt met de *scope-resolution* operator `::` aangeven uit welke namespace je een class of functie wilt gebruiken:

```
Bd::string s1("Harry");
Vi::string s2("John");
std::string s3("Standard");
```

In het bovenstaande codefragment worden 3 objecten gedefinieerd van 3 verschillende classes. Het object `s1` is van de class `string` die gedefinieerd is in de namespace `Bd`, het object `s2` is van de class `string` die gedefinieerd is in de namespace `Vi` en het object `s3` is van de class `string` die gedefinieerd is in de namespace `std` (de standaard library). Je ziet dat je met behulp van namespaces classes die toevallig dezelfde naam hebben toch in 1 programma kunt combineren. Namespaces maken dus het hergebruik van code eenvoudiger.

Als je in een stuk code steeds de `string` uit de namespace `Bd` wilt gebruiken dan kun je dat opgeven met behulp van een *using declaration*.

```
using Bd::string;
string s4("Hallo");
string s5("Dag");
```

De objecten `s4` en `s5` zijn nu beide van de class `string` die gedefinieerd is in de namespace `Bd`. De using declaratie blijft net zolang geldig als een gewone variabele declaratie. Tot de bijbehorende accolade sluiten dus.

Als je in een stuk code steeds classes en functies uit de namespace `Bd` wilt gebruiken dan kun je dat opgeven met behulp van een *using directive*.

```
using namespace Bd;
string s6("Hallo");
double d(sin(0,785398163397448));
```

Het object `s6` is nu van de class `string` die gedefinieerd is in de namespace `Bd`. De functie `sin` die wordt aangeroepen is nu de in de namespace `Bd` gedefinieerde functie. De using directive blijft net zolang geldig als een gewone variabele declaratie. Tot de bijbehorende accolade sluiten dus.

5.7 Exceptions.

Vaak zal in een functie of memberfunctie gecontroleerd worden op “uitzonderlijke” situaties (fouten). De volgende functie berekent de gemiddelde snelheid in km/uur als de afstand (in kilometers) en de reistijd (in uren en minuten) bekend zijn.

```
double gemSnelheid(double afstand, int uur, int min) {
    return afstand*60/(uur*60+min);
}
```

Deze functie kan als volgt aangeroepen worden om de gemiddelde snelheid te berekenen als we een afstand van 75,6 km in 40 minuten afleggen:

```
cout<<gemSnelheid(75.6, 0, 40)<<endl;
```

Als we bij het aanroepen van deze functie een reistijd van 0 uur en 0 minuten invullen loopt het programma volledig vast. Er verschijnt een window waarin (als op de knop details wordt gedrukt) de volgende foutmelding verschijnt:

```
GEMSNELHEID1 heeft een uitzondering 10H veroorzaakt in module
GEMSNELHEID1.EXE op 017f:004011ec.
Registers:
EAX=00000000 CS=017f EIP=004011ec EFLGS=00010203
...
```

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

5.7.1 Het gebruik van `assert`.

In H1 heb je al kennis gemaakt met de standaard functie `assert`. Deze functie doet niets als de, als parameter opgegeven, expressie true oplevert maar breekt het programma met een passende foutmelding af als dit niet zo is. Je kunt zogenaamde "assertions" gebruiken om tijdens de ontwikkeling van het programma te controleren of aan een bepaalde voorwaarden (waarvan je "zeker" weet dat ze geldig zijn) wordt voldaan. Je kunt de functie `gemSnelheid` voorzien van een assertion:

```
double gemSnelheid(double afstand, int uur, int min) {
    assert(!(uur==0 && min==0));
    return afstand*60/(uur*60+min);
}
```

Als we nu bij het aanroepen van deze functie een reistijd van 0 uur en 0 minuten invullen dan zal de `assert` functie het programma beëindigden. De volgende foutmelding verschijnt:

```
Assertion failed: !(uur==0 && min==0), file gemSnelheid2.cpp, line 6

Abnormal program termination
```

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

De standaard functie `assert` is bedoeld om tijdens het ontwikkelen van programma’s te controleren of iedereen zich aan bepaalde afspraken houdt. Als bijvoorbeeld is afgesproken dat de functie `gemSnelheid` alleen mag worden aangeroepen met een reistijd die niet gelijk aan 0 uur en 0 minuten is dan is dat prima met `assert` te controleren. Elk programmadeel waarin `gemSnelheid` wordt aangeroepen moet nu voor de aanroep zelf controleren of de parameters geldige waarden hebben. Bij het testen van het programma (gecompileerd met debug informatie) zorgt de `assert` ervoor dat het snel duidelijk wordt als de afspraak geschonden wordt. Als na het testen duidelijk is dat iedereen zich aan de afspraak houdt dan is het niet meer nodig om de `assert` uit te voeren. Het programma wordt gecompileerd zonder debug informatie en alle `assert` aanroepen worden verwijderd.

Het gebruik van `assert` heeft de volgende nadelen:

- Het programma wordt nog steeds abrupt afgebroken en krijgt niet de kans om tijdelijk opgeslagen data op te slaan.

- Op elke plek waar deze functie aangeroepen wordt moeten, voor aanroep, eerst de parameters gecontroleerd worden. Als de functie op veel plaatsen aangeroepen wordt dan is het logischer om de controle in de functie zelf uit te voeren. Dit maakt het programma ook beter onderhoudbaar (als de controle aangepast moet worden dan hoeft de code maar op 1 plaats gewijzigd te worden) en betrouwbaarder (je kunt de functie niet meer zonder controle aanroepen, de controle zit nu immers in de functie zelf).

We kunnen concluderen dat `assert` in dit geval niet geschikt is.

5.7.2 Het gebruik van een `bool` returnwaarde.

In C (en ook in C++) werd dit traditioneel opgelost door de functie een returnwaarde te geven die aangeeft of het uitvoeren van de functie gelukt is:

```
bool gemSnelheid(double& res, double afstand, int uur, int min) {
     if (uur==0 && min==0) {
          return false;
     }
     else {
          res=afstand*60/(uur*60+min);
          return true;
     }
}
```

Deze functie kan nu als volgt aangeroepen worden:

```
double gem;
if (gemSnelheid(gem, 75.6, 0, 40)) {
     cout<<gem<<endl;
}
else {
     cout<<"Kan gemiddelde snelheid niet berekenen."<<endl;
}
if (gemSnelheid(gem, 75.6, 0, 0)) {
     cout<<gem<<endl;
}
else {
     cout<<"Kan gemiddelde snelheid niet berekenen."<<endl;
}
```

Het programma wordt nu niet meer abrupt afgebroken. Het gebruik van een return waarde om aan te geven of een functie gelukt is heeft echter de volgende nadelen:

- Bij **elke** aanroep moet de returnwaarde getest worden.
- Op de plaats waar de fout ontdekt wordt kan hij meestal niet opgelost worden.
- De “echte” returnwaarde van de functie moet nu via een call by reference parameter worden teruggegeven. Dit betekent dat je om de functie aan te roepen altijd een variabele aan moet maken (om het resultaat in op te slaan) ook als je het resultaat alleen maar wilt doorgeven aan een andere functie of operator.

De C library stdio werkt bijvoorbeeld met returnwaarden van functies die aangeven of de functie gelukt is. Zo geeft de functie `printf` bijvoorbeeld een `int` returnwaarde. Als de functie gelukt is geeft `printf` het aantal geschreven bytes terug maar als er een error is opgetreden geeft `printf` de waarde `EOF` terug. Een goed geschreven programma moet dus bij elke aanroep naar `printf` de returnwaarde testen!

5.7.3 Het gebruik van standaard exceptions.

C++ heeft *exceptions* ingevoerd voor het afhandelen van “uitzonderlijke” fouten. Een exception is een **object** dat in de functie waar de fout ontstaat “gegooid” kan worden en dat door de aanroepende functie (of door zijn aanroepende functie enz...) “opgevangen” kan worden. In de standaard library zijn ook een aantal standaard exceptions opgenomen. De class van de exception die bedoeld is om te gooien als een parameter van een functie een ongeldige waarde heeft is de naam `domain_error`[7].

```
#include <stdexcept>
using namespace std;

double gemSnelheid(double afstand, int uur, int min) {
     if (uur==0 && min==0)
          throw domain_error("Reistijd is 0 uur en 0 minuten");
     return afstand*60/(uur*60+min);
}
```

Je kunt een exception object gooien door het C++ keyword `throw` te gebruiken. Bij het aanroepen van de constructor van de class `domain_error` die gedefinieerd is in de include file `<exception>` kun je een string meegeven die de oorzaak van de fout aangeeft. Als de `throw` wordt uitgevoerd dan wordt de functie meteen afgebroken. Lokale variabelen worden wel netjes opgeruimd (de destructors van deze lokale objecten wordt netjes aangeroepen). Ook de functie waarin de functie `gemSnelheid` is aangeroepen wordt meteen afgebroken. Dit proces van afbreken wordt gestopt zodra de exception wordt opgevangen. Als de exception nergens wordt opgevangen dan wordt het programma gestopt.

```
try {
     cout<<gemSnelheid(75.6, 0, 40)<<endl;
     cout<<gemSnelheid(75.6, 0, 0)<<endl;
     cout<<"Dit was het!"<<endl;
} catch (domain_error& e) {
     cout<<e.what()<<endl;
}
cout<<"The END."<<endl;
```

Exceptions kunnen worden opgevangen als ze optreden in een zogenaamd *try* blok. Dit blok begint met het keyword `try` en wordt afgesloten met het keyword `catch`. Na het `catch` keyword moet je tussen haakjes aangeven welke exceptions je wilt opvangen gevolgd door een codeblok dat uitgevoerd wordt als de gespecificeerde exception wordt opgevangen. Na het eerste catch blok kunnen er nog een willekeurig aantal catch blokken volgen om andere exceptions ook te kunnen vangen. Als je alle mogelijke exceptions wilt opvangen dan kun je dat doen met: `catch(...) { /*code*/ }`.

Een exception kun je het beste als reference opvangen. Dit heeft 2 voordelen:

- Het voorkomt een extra kopie.
- We kunnen op deze manier ook van `domain_error` afgeleide classes opvangen zonder dat het slicing probleem optreedt.

De class `domain_error` heeft een memberfunctie `what()` die de bij constructie van het object meegegeven string weer teruggeeft. De uitvoer van het bovenstaande programma is:

```
113.4
Reistijd is 0 uur en 0 minuten
The END.
```

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

Merk op dat het laatste statement in het try blok niet meer uitgevoerd wordt omdat bij het uitvoeren van het tweede statement een exception optrad. Het gebruik van exceptions in plaats van een `bool` return-waarde heeft de volgende voordelen.

- Het programma wordt niet meer abrupt afgebroken. Door de exception op te vangen op een plaats waar je er wat mee kunt heb je de mogelijkheid om het programma na een exception gewoon door te laten gaan of op z'n minst netjes af te sluiten.
- Je hoeft niet bij elke aanroep te testen. Je kunt meerder aanroepen opnemen in hetzelfde try blok. Als in het bovenstaande programma een exception zou optreden in het eerste statement dan wordt het tweede (en derde) statement niet meer uitgevoerd.
- De returnwaarde van de functie kan nu weer gewoon gebruikt worden om de berekende gemiddelde snelheid terug te geven.

5.7.4 Het gebruik van zelf gedefinieerde exceptions.

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

```
class ReistijdError {};

double gemSnelheid(double afstand, int uur, int min) {
     if (uur==0 && min==0)
          throw ReistijdError();
     return afstand*60/(uur*60+min);
}
```

Voorbeeld van gebruik:

```
try {
     cout<<gemSnelheid(75.6, 0, 40)<<endl;
     cout<<gemSnelheid(75.6, 0, 0)<<endl;
     cout<<"Dit was het!"<<endl;
} catch (ReistijdError& e) {
     cout<<"Ongeldige reistijd!"<<endl;
}
cout<<"The END."<<endl;
```

Uitvoer:

```
113.4
Ongeldige reistijd!
The END.
```

Zelf gedefinieerde classes kunnen we m.b.v. overerving volgens een generalisatie/specialisatie structuur indelen. Bij een `catch` kunnen we nu kiezen of we een specifieke of een generieke exception willen afvangen. De specifieke zijn polymorf met de generieke. Doordat exceptions gewoon objecten zijn kun je ze ook data en gedrag geven.

Je kunt b.v. een `virtual` memberfunctie definiëren die een bij de exception passende foutmelding geeft. Als je deze memberfunctie in specifiekere exceptions override dan kun je een generieke exception vangen en toch d.m.v. dynamic binding de juiste foutmelding krijgen!

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

```
SnelheidError::~SnelheidError() {
}

class ReistijdError: public SnelheidError {
public:
    virtual string getErrorMessage() const;
};

string ReistijdError::getErrorMessage() const {
    return "Ongeldige reistijd!";
}

class AfstandError: public SnelheidError {
public:
    virtual string getErrorMessage() const;
};

string AfstandError::getErrorMessage() const {
    return "Ongeldige afstand!";
}

double gemSnelheid(double afstand, int uur, int min) {
    if (afstand<0.0)
        throw AfstandError();
    if (uur==0 && min==0)
        throw ReistijdError();
    return afstand*60/(uur*60+min);
}
```

Voorbeeld van gebruik:

```
try {
    cout<<gemSnelheid(75.6, 0, 40)<<endl;
    cout<<gemSnelheid(-12.6, 1, 20)<<endl;
    cout<<"Dit was het!"<<endl;
} catch (SnelheidError& e) {
    cout<<e.getErrorMessage()<<endl;
}
cout<<"The END."<<endl;
```

Uitvoer:

```
113.4
Ongeldige afstand!
The END.
```

Als je nu bijvoorbeeld alleen de `ReistijdError` exception wilt opvangen maar de `AfstandError` exception niet dan kan dat eenvoudig door in de bovenstaande catch het type `SnelheidError` te vervangen door `ReistijdError`.

5.7.5 Exception details.

Over exceptions valt nog veel meer te vertellen:

- Exceptions in constructors en destructors. Gebruik nooit throw in een destructor!
- Function try-blok. Speciale syntax om exceptions die optreden bij het initialiseren van datamembers in een constructor op te vangen.
- Re-throw. Gebruik van throw zonder argument in een catch blok om de zojuist opgevangen exception weer door te gooien.

- Exception specification. Speciale syntax waarmee in het prototype van een functie aangegeven kan worden welke exceptions deze functie kan veroorzaken.
- Exceptions in de std. Een overzicht van alle exceptions die in de standaard library gedefinieerd zijn en een overzicht van alle exceptions die bij het gebruik van standaard C++ kunnen optreden.

Voor al deze details verwijs ik je naar een goed C++ boek.

In deze paragraaf wordt verder ingegaan op enkele details die bij het gebruik van exceptions heel belangrijk kunnen zijn.

5.7.5.1 De volgorde van `catch` blokken

Als je meerdere catch blokken gebruikt om exceptions af te vangen dan moet je er op letten dat de meest specifieke exceptions vóór de generieke exceptions komen. Omdat de catch blokken van boven naar beneden worden “geprobeerd” als er een exception gegooid wordt. Dus:

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

Vraag:
Waarom verschijnt bij het uitvoeren van de onderstaande code de foutmelding “AfstandError exception” nooit op het scherm.

```
try {
//  ...
} catch (SnelheidError& e) {
    cout<<"Exception derived from SnelheidError"<<endl;
} catch (AfstandError& e) {
    cout<<"AfstandError exception"<<endl;
}
```

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

5.7.5.2 Exception-safe code.

Een programma dat ook bij het optreden van exceptions correct blijft werken wordt *exception-safe* genoemd. De problemen die kunnen optreden bij exceptions en de mogelijke oplossingen daarvoor worden pas sinds kort goed begrepen[8]. Voorbeeld:

```
void overboeking(Rekening& van, Rekening& naar, Euro bedrag) {
    if (van.beschikbaar(bedrag)) {
        van.neemOp(bedrag);
        naar.stort(bedrag);
    }
}
```

[8] Zie het boek: Exceptional C++ van Herb Sutter ©2000 Addison Wesley.

Deze functie is niet exception-safe. Stel dat er bij het opnemen van een bedrag en ook bij het storten van een bedrag een exception kan optreden. Bijvoorbeeld doordat deze handelingen opgeslagen moeten worden in een database, die via het netwerk bereikbaar is, en dat deze database op dit moment onbereikbaar is (door netwerk of server problemen).

Als de `neemop` memberfunctie mislukt en een exception gooit is er niets aan de hand. De functie overboeking wordt meteen afgebroken en de memberfunctie `stort` wordt nooit aangeroepen. Als de `stort` memberfunctie echter mislukt en een exception gooit dan werkt de functie overboeking niet meer correct omdat er al wel een bedrag is opgenomen. De functie `overboeking` moet om correct te werken:

- helemaal zijn werk doen en geen exception gooien of
- helemaal *niets* doen en een exception gooien.

Zo'n alles of niets handeling wordt een *transaction* genoemd.

Poging om het probleem op te lossen:

```
void overboeking(Rekening& van, Rekening& naar, Euro bedrag) {
    if (van.beschikbaar(bedrag)) {
        van.neemOp(bedrag);
        try {
            naar.stort(bedrag);
        } catch (stortException& e) {
            van.stort(bedrag);
            throw; // gooi de opgevangen exceptie opnieuw
        }
    }
}
```

Deze oplossing is natuurlijk ook niet exception-safe omdat het terugstorten van het opgenomen bedrag op dezelfde rekening ook een exception kan opleveren.

De correcte oplossing maakt gebruik van een `swap` functie waarmee de gegevens van twee rekeningen verwisseld kunnen worden. Deze `swap` functie moet zijn werk kunnen doen zonder dat er een exception kan optreden. In een volgend voorbeeld zul je zien dat het maken van z'n `swap` niet zo moeilijk is. De `swap` functie kan als volgt gedeclareerd worden:

```
void swap(Rekening& r1, Rekening& r2) throw()[9];
```

Je kunt de functie `overboeking` nu als volgt met behulp van `swap` implementeren:

```
void overboeking(Rekening& van, Rekening& naar, Euro bedrag) {
    if (van.beschikbaar(bedrag)) {
        Rekening vanKopie(van);
        Rekening naarKopie(naar);
        vanKopie.neemOp(bedrag);
        naarKopie.stort(bedrag);
    //  het opnemen en storten is zonder problemen verlopen:
        swap(vanKopie, van);
        swap(naarKopie, naar);
    }
}
```

Er wordt eerst een kopietje van beide rekeningen gemaakt en vervolgens wordt de overboeking uitgevoerd bij de gekopieerde rekeningen. Als er bij het maken van de kopietjes of bij het maken van de

[9] Met het toevoegen van `throw()` na een functiedeclaratie belooft de programmeur van de functie dat deze functie geen exception zal gooien.

overboeking iets fout gaat en er een exception optreedt dan is de overboeking mislukt en de beide rekeningen ongewijzigd (exception-safe). Als het overboeken bij de kopietjes gelukt is dan worden de kopietjes verwisseld met de orginele rekeningen. Als we er dan voor kunnen zorgen dat bij het verwisselen (`swap`) geen exceptions kunnen optreden hebben we het probleem opgelost. Dit lijkt op het verschuiven van het probleem want: "Hoe maken we nu een exception-safe `swap`?". Het blijkt dat het voor types die "onder water" een verwijzing (pointer) gebruiken naar de "echte" data eenvoudig een exception-safe `swap` gemaakt kan worden door de pointers te verwisselen (bij het verwisselen van 2 pointers kan geen exception optreden). Kijk maar eens hoe dat bij een zelfgemaakte `Vector` gedaan wordt in de volgende paragraaf.

5.7.5.3 Vector opnieuw bekeken.

In H1 hebben we zelf een class `Vector` gedefinieerd. Voor deze class hebben we zelf een copy constructor en `operator=` gedefinieerd:

```
Vector::Vector(const Vector& v): size(v.size), data(new int[v.size]) {
    *this=v;
}
Vector& Vector::operator=(const Vector& r) {
    if (size!=r.size) {
        delete[] data;
        data=new int[r.size];
        size=r.size;
    }
    for (int i(0);i<size;++i)
        data[i]=r.data[i];
    return *this;
}
```

De `operator=` is echter niet exception-safe. De operator `new` kan namelijk de standaard exception `bad_alloc` gooien als er onvoldoende geheugenruimte beschikbaar is. De array waar de pointer `data` naar wijst wordt vrijgegeven met `delete[]`. Maar als er in `new[]` een `bad_alloc` exception optreedt wijst de pointer `data` na afloop van de `operator=` nog steeds naar de (al vrijgegeven) geheugenruimte.

```
{
    Vector v(10);
    for (int j(0); j<v.length(); ++j)
        v[j]=j*j;    // vul v met kwadraten
    cout<<"v = "<<v<<endl;
    Vector u(1);
    u[0]=13;
    cout<<"u = "<<u<<endl;
    try {
        u=v;
    } catch (bad_alloc& e) {
        cout<<"Toekennen aan u mislukt!"<<endl;
        cout<<"u = "<<u<<endl;
        cout<<"v = "<<v<<endl;
    }
}
```

Het bovenstaande programmadeel geeft als er een `bad_alloc` exception in de `operator=` optreedt[10] de volgende uitvoer:

```
v = 0,1,4,9,16,25,36,49,64,81
u = 13
Toekennen aan u mislukt!
u = 0
v = 0,1,4,9,16,25,36,49,64,81
```

Gevolgd door het volledig vastlopen van het programma met de volgende foutmelding:

```
VECTOR2 heeft een fout met betrekking tot een ongeldige pagina veroor-
zaakt in module CC3250.DLL op0177:3250325e.
Registers:
EAX=0065401c ...
```

Tijdens het uitvoeren van `u=v` wordt in `Vector::operator=` eerst `u.data` vrijgegeven met behulp van `delete[]`. Als daarna in de volgende regel bij het uitvoeren van `new[]` een `bad_alloc` optreedt wordt de `operator=` meteen verlaten en behouden `u.data` en `u.size` hun waarden. Je kunt duidelijk in de uitvoer zien dat `u` nog steeds een `size` van 1 heeft. De vrijgegeven geheugenruimte is blijkbaar meteen weer ergens anders voor gebruikt. Als nu aan het einde van het programma de destructor van het object `u` wordt aangeroepen dan wordt de geheugenruimte waar `u.data` naar wijst voor de *tweede* maal vrijgegeven door de `delete[]` in `Vector::~Vector`. Het programma loopt hierdoor vast!

De class `Vector` kan als volgt exception-safe worden geïmplementeerd:

```
// file: vector3.cpp
#include <iostream>
#include <algorithm>
#include <exception>

class Vector {
public:
    explicit Vector(int);
    Vector(const Vector&);
    Vector& operator=(const Vector&);
    ~Vector();
    int& operator[](int);
    const int& operator[](int) const;
    int length() const;
private:
    int size;
    int* data;
    void swap(Vector&) throw();
friend ostream& operator<<(ostream&, const Vector&);
};

Vector::Vector(int s): size(s), data(new int[s]) {
}

Vector::Vector(const Vector& v): size(v.size), data(new int[v.size]) {
    for (int i(0); i<size; ++i)
        data[i]=v.data[i];
}
```

[10] Natuurlijk zal er in de praktijk als `v` 10 elementen bevat geen exception optreden in `operator=`. In het testprogramma vector2.cpp (te vinden op http://bd.thrijswijk.nl/syso1) wordt “kunstmatig”een exception opgewekt. Als `v` heel veel elementen bevat kan een echte exception optreden.

```
Vector& Vector::operator=(const Vector& r) {
    Vector t(r);
    swap(t);
    return *this;
}

Vector::~Vector() {
    delete[] data;
}

void Vector::swap(Vector& v) throw() {
    std::swap(size, v.size);
    std::swap(data, v.data);
}

int& Vector::operator[](int index) {
    if (index<0 || index>=size)
        throw out_of_range("illegal index for Vector used");
    return data[index];
}

const int& Vector::operator[](int index) const {
    if (index<0 || index>=size)
        throw out_of_range("illegal index for Vector used");
    return data[index];
}

int Vector::length() const {
    return size;
}

ostream& operator<<(ostream& o, const Vector& v) {
    for (int i(0); i<v.size; ++i) {
        o<<v.data[i];
        if (i!=v.size-1)
            o<<',';
    }
    return o;
}
```

Zoals je ziet is er een private memberfunctie bijgekomen:

```
void Vector::swap(Vector& v) throw() {
    std::swap(size, v.size);
    std::swap(data, v.data);
}
```

Deze memberfunctie verwisselt de receiver met de als argument meegegeven `Vector v`. Daarbij is gebruik gemaakt van de standaard functie `swap` die als volgt gedefinieerd is in `<algorithm>`:

```
template <class T>
inline void swap (T& a, T& b) {
    T tmp(a);
    a=b;
    b=tmp;
}
```

De `throw()` achter de functie declaratie van `Vector::swap` zegt dat deze functie geen exception kan gooien. Uit de implementatie zal duidelijk zijn dat dit inderdaad zo is (er worden alleen twee `int`'s en twee `int*`'s verwisseld).

De `swap` memberfunctie wordt gebruikt bij het implementeren van de `operator=`.

```
Vector& Vector::operator=(const Vector& r) {
    Vector t(r);
    swap(t);
    return *this;
}
```

De `operator=` begint met het maken van een kopie van de als argument meegegeven `Vector r`. Tijdens het uitvoeren van de copy constructor kan uiteraard een `bad_alloc` exception optreden. Als dit gebeurt wordt de `operator=` meteen afgebroken (omdat de constructie van `t` niet gelukt is wordt de destructor van `t` niet aangeroepen). Als bij het maken van `t` geen exception optreedt wordt `t` (de kopie van `r`) verwisseld met de receiver. De nieuwe waarde van de receiver wordt dus gelijk aan `r`. Zoals we al hebben gezien kan tijdens `Vector::swap` geen exception optreden. Na het uitvoeren van het `return` statement wordt het object `t` verwijderd. Hierdoor wordt de oude waarde van de receiver opgeruimd. Bedenk dat `t` door de verwisseling de oude waarde van de receiver bevat! Waarschijnlijk moet je deze paragraaf een paar keer lezen om het helemaal te vatten. Het heeft niet voor niets 15 jaar geduurd voordat er iemand op dit geniale idee kwam.

```
{
    Vector v(10);
    for (int j(0); j<v.length(); ++j)
        v[j]=j*j;    // vul v met kwadraten
    cout<<"v = "<<v<<endl;
    Vector u(1);
    u[0]=13;
    cout<<"u = "<<u<<endl;
    try {
        u=v;
    } catch (bad_alloc& e) {
        cout<<"Toekennen aan u mislukt!"<<endl;
        cout<<"u = "<<u<<endl;
        cout<<"v = "<<v<<endl;
    }
}
```

Het bovenstaande programmadeel geeft als er een `bad_alloc` exception in de `operator=` optreedt de volgende uitvoer:

```
v = 0,1,4,9,16,25,36,49,64,81
u = 13
Toekennen aan u mislukt!
u = 13
v = 0,1,4,9,16,25,36,49,64,81
```

Het programma loopt niet meer vast. Zoals je ziet houden de objecten `u` en `v` gewoon hun waarden als de toekenning `u=v` mislukt. De `operator=` van `Vector` is nu volledig exception-safe.

Voor de liefhebbers:

Vraag:
Is de onderstaande implementatie van `operator=` correct en exception-safe? Verklaar je antwoord!

```
Vector& Vector::operator=(Vector r) {
```

```
    swap(r);
    return *this;
}
```

Antwoord:
Ja! De copy constructor wordt nu impliciet aangeroepen bij het doorgeven van de parameter `r`. (call by value).

Vraag:
Is de onderstaande implementatie van `operator=` correct en exception-safe? Verklaar je antwoord!

```
Vector& Vector::operator=(Vector r) {
    std::swap(*this, r);
    return *this;
}
```

Antwoord:
Nee, deze versie is **niet** exception-safe! Er kan nu in de `swap` een exception optreden omdat de copy constructor daar wordt aangeroepen!

5.8 Casting en runtime type information.

5.8.1 Casting.

In C kun je met een eenvoudige vorm van “casting” typeconversies doen. Stel dat we het adres van een C string in een integer willen opslaan[11] dan kun je dit als volgt proberen:

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

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

```
[C++ Error] Cannot convert 'char *' to 'int'
```

Het is namelijk helemaal niet zeker dat een `char*` in een `int` variabele past. Stel dat je zeker weet dat het past (omdat je het programma alleen maar onder Win32 wilt gebruiken) dan kun je de compiler “dwingen” met behulp van een zogenaamde *cast*:

```
i=(int)"Hallo";
```

In C++ mag je deze cast ook als volgt coderen:

```
i=int("Hallo");
```

Het vervelende van beide vormen van casting is dat een cast erg moeilijk te vinden is. Omdat een cast in bijna alle gevallen code oplevert die niet portable is, is het echter wel belangrijk om alle casts in een programma op te kunnen sporen. Een cast wordt door C programmeurs ook vaak gebruikt terwijl dat helemaal niet nodig is (zoals in het bovenstaande geval).

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

```
i=reinterpret_cast<int>("Hallo");
```

In dit geval moeten we een `reinterpret_cast` gebruiken omdat de cast niet portable is.

[11] Er is geen enkele reden te bedenken waarom je dit zou willen. Het adres van een C string moet je natuurlijk in een `char*` opslaan!

Stel dat je een C++ programma schrijft dat moet draaien op een 68HC11 microcontroller. Als je de output poort B van deze controller wilt aansturen dan kan dat via adres $1004. Dit kun je in C++ als volgt doen:

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

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

Vraag:
Wat is de uitvoer van het volgende programma:

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

Antwoord:
`d = 0`
Dit is niet wat de meeste mensen verwachten. De computer gebruikt bij het berekenen van `i1/i2` echter een integer deling omdat `i1` en `i2` beiden van het type `int` zijn. Het antwoord van deze integer deling is de integer waarde 0. Vervolgens wordt deze waarde omgezet naar het type `double`.

Als we willen dat het bovenstaande programma `0.5` als antwoord geeft dan moeten we ervoor zorgen dat in plaats van een integer deling een floating point deling wordt gebruikt. De computer gebruikt een floating point deling als 1 van de 2 argumenten een floating point getal is. De juiste deling is dus:

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

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

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

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

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

Uitvoer:

```
s = Dag
s = Hallo
```

Het zal duidelijk zijn dat je het gebruik van `const_cast` zoveel mogelijk moet beperken. Als de reference `a` in het bovenstaande programma naar een `const string` refereert dan is het resultaat onbepaald omdat een compiler `const` objecten in het ROM geheugen kan plaatsen (dit gebeurt veel bij embedded systems).

5.8.2 Casting en overerving.

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

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

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

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

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

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

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

In dit geval is een `static_cast` gebruikt om een down-cast te maken. Zolang je de functie `geefHulp` alleen maar aanroept met een `SintBernard*` als argument gaat alles goed.[12]

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

Uitvoer:

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

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

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

Uitvoer:

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

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

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

Je kunt de functie `geefHulp` nu veilig aanroepen zowel met een `SintBernard*` als met een `Hond*` als argument.

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

Uitvoer:

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

[12] Als de functie `geefHulp` alleen maar aangeroepen wordt met een `SintBernard*` als argument is het natuurlijk veel slimmer om deze functie te definiëren als:
`void geefHulp(SintBernard* sbp)`
De down-cast is dan niet meer nodig!

```
Blaf.
```

Een `dynamic_cast` is alleen mogelijk met polymorphic pointers en polymorphic references. Als een `dynamic_cast` van een pointer mislukt dan geeft de cast een nul pointer terug. Bij een `dynamic_cast` van een reference is dit niet mogelijk (omdat een nul reference niet bestaat). Als een `dynamic_cast` van een reference mislukt dan wordt de standaard exception `bad_cast` gegooid.

Een versie van `geefHulp` die werkt met een `Hond&` in plaats van met een `Hond*` kun je dus als volgt implementeren:

```
#include <typeinfo>

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

Deze functie kun je als volgt aanroepen:

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

Uitvoer:

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

5.8.3 Dynamic casting en RTTI.

Om tijdens run time te kunnen controleren of een `dynamic_cast` mogelijk is moet informatie over het type tijdens run time beschikbaar zijn. Dit wordt **RTTI** = **R**un **T**ime **T**ype **I**nformation genoemd. In C++ hebben alleen classes met één of meer virtual functions RTTI. Dat is logisch omdat polymorphism ontstaat door het gebruik van virtual memberfuncties. RTTI maakt dus het gebruik van `dynamic_cast` mogelijk. Je kunt ook de RTTI gegevens behorende bij een object rechtstreeks opvragen. Deze gegevens zijn opgeslagen in een object van de class `type_info`. Deze class heeft een vraagfunctie `name()` waarmee de naam van de class opgevraagd kan worden. Om het `type_info` object van een (ander) object op te vragen moet je het C++ keyword `typeid` gebruiken.

```
#include <typeinfo>

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

Deze functie kun je als volgt aanroepen:

```
SintBernard boris;
printRas(boris);
```

```
Hond h;
printRas(h);
```

Uitvoer:

```
SintBernard
Hond
```

5.8.4 Maak geen misbruik van RTTI en `dynamic_cast`.

Het verkeerd gebruik van `dynamic_cast` en RTTI kan leiden tot code die *niet* uitbreidbaar en aanpasbaar is! Je zou bijvoorbeeld op het idee kunnen komen om een functie `blaf()` als volgt te implementeren:

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

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

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

Bedenk zelf wat er moet gebeuren als je de class `DuitseHerder` wilt toevoegen. In plaats van een losse functie die expliciet gebruikt maakt van dynamic binding (`dynamic_cast`) met behulp van RTTI moet je een `virtual` memberfunctie gebruiken. Deze `virtual` memberfunctie maakt impliciet gebruik van dynamic binding. Alleen in uitzonderingsgevallen (er is maar 1 soort hond die whisky bij zich heeft) moet je `dynamic_cast` gebruiken. Alle honden kunnen blaffen (alleen niet allemaal op dezelfde manier) dus is het gebruik van `dynamic_cast` om blaf te implementeren niet juist. In dit geval moet je een `virtual` memberfunctie gebruiken.

6 Design patterns in de std C++ library.

In de C++ standaard wordt ook gebruik gemaakt van een aantal design patterns. De belangrijkste zijn:
- *Functor*. Gebruik objecten als een functie.
- *Adapter* (Wrapper). Gebruikt om de interface van een class aan te passen.
- *Iterator*. Gebruikt om een datastructuur te doorlopen.

6.1 Functor.

De onderstaande templatefunctie `findMax` geeft het maximale element uit een `vector` terug. Deze functie is gedefinieerd als `template` zodat vectoren van elk gewenst type gebruikt kunnen worden.

```
template <typename T>
const T& findMax(const vector<T>& v) {
    assert(v.size()>0);  // precondition
    vector<T>::size_type maxIndex(0);
    for (vector<T>::size_type i(1); i<v.size(); ++i)
        if (v[maxIndex]<v[i])
```

```
            maxIndex=i;
    return v[maxIndex];
}
```

Deze functietemplate maakt, zoals je ziet, gebruik van de `operator<` om elementen met elkaar te vergelijken. Niet elk type heeft echter een logische definitie voor `operator<`. Kijk maar eens naar de volgende class:

```
class Rectangle {
public:
    Rectangle(int l, int w);
    int getLength() const;
    int getWidth() const;
    void print(ostream& o) const;
private:
    int length;
    int width;
};
```

Wanneer is de ene `Rectangle` kleiner dan de andere `Rectangle`? Als de `length` kleiner is? Als de `width` kleiner is? Als de oppervlakte (`length*width`) kleiner is? Als de omtrek (`2*(length+width)`) kleiner is? Ik zou het niet weten. De ene keer wil je de functie `findMax` gebruiken om de `Rectangle` met de maximale `length` te vinden en even later wil je de functie `findMax` gebruiken om de `Rectangle` met de maximale oppervlakte te vinden. Maar je kunt natuurlijk maar 1 `operator<` definiëren. De hierboven gegeven templatefunctie `findMax` is dus niet generiek genoeg.

Je kunt een tweede parameter aan de functie `findMax` toevoegen waarmee je de te gebruiken vergelijkingsfunctie kunt meegeven.

```
template <typename T>
const T& findMax(const vector<T>& v, bool (*f)(const T&, const T&)) {
    assert(v.size()>0);
    vector<T>::size_type maxIndex(0);
    for (vector<T>::size_type i(1); i<v.size(); ++i)
        if (f(v[maxIndex],v[i]))
            maxIndex=i;
    return v[maxIndex];
}
```

De syntax is erg ingewikkeld. De tweede parameter is `bool (*f)(const T&, const T&)`. De naam van de parameter is `f` met als type `bool (*)(const T&, const T&)`. Dat wil zeggen een pointer naar een functie die een `bool` teruggeeft en twee `const T&` parameters heeft.

De volgende functie kan gebruikt worden om `int`'s te vergelijken.

```
bool lessInt(const int& i1, const int& i2) {
    return i1<i2;
}
```

De grootste `int` in een `vector` met `int`'s kan nu als volgt gevonden worden:

```
vector<int> vi;
// vul de vector vi
cout<<"Het maximum in vi is: "<<findMax(vi, lessInt)<<endl;
```

Hieronder zijn enkele functies gedefinieerd die gebruikt kunnen worden om `Rectangle`'s te vergelijken.

```
bool lessLength(const Rectangle& r1, const Rectangle& r2) {
    return r1.getLength()<r2.getLength();
}

bool lessWidth(const Rectangle& r1, const Rectangle& r2) {
    return r1.getWidth()<r2.getWidth();
}

bool lessArea(const Rectangle& r1, const Rectangle& r2) {
    return r1.getLength()*r1.getWidth()<r2.getLength()*r2.getWidth();
}
```

In een `vector` met `Rectangle`'s kun je nu de `Rectangle` met de grootste `length` als volgt vinden:

```
vector<Rectangle> vr;
// vul de vector vr
cout<<"De maximum length in vr is: "<<findMax(vr, lessLength)<<endl;
```

Maar je kunt de `findMax` functie ook gebruiken om de `Rectangle` met de grootste `width` of de grootste oppervlakte te vinden:

```
cout<<"De maximum width in vr is: "<<findMax(vr, lessWidth)<<endl;
cout<<"De maximum area in vr is: "<<findMax(vr, lessArea)<<endl;
```

De syntax van de templatefunctie `findMax` kan sterk vereenvoudigd worden door het type van de tweede functie parameter als (tweede) template parameter op te geven. De compiler bepaalt dan zelf bij aanroep van de templatefunctie het juiste type!

```
template <typename T, typename F>
const T& findMax(const vector<T>& v, F f) {
    assert(v.size()>0);
    vector<T>::size_type maxIndex(0);
    for (vector<T>::size_type i(1); i<v.size(); ++i)
        if (f(v[maxIndex],v[i]))
            maxIndex=i;
    return v[maxIndex];
}
```

Je kunt deze nieuwe versie van de templatefunctie `findMax` op exact dezelfde wijze gebruiken als de eerste versie:

```
vector<int> vi;
// vul vector vi
cout<<"Het maximum in vi is: "<<findMax(vi, lessInt)<<endl;
vector<Rectangle> vr;
// vul vector vr
cout<<"De maximum length in vr is: "<<findMax(vr, lessLength)<<endl;
cout<<"De maximum width in vr is: "<<findMax(vr, lessWidth)<<endl;
cout<<"De maximum area in vr is: "<<findMax(vr, lessArea)<<endl;
```

Het gebruik van een tweede template parameter waardoor de syntax sterk is vereenvoudigd heeft nog meer voordelen. Er kunnen nu vergelijkingsfuncties gebruikt worden met een iets ander prototype. Bijvoorbeeld:

```
bool lessInt(int i1, int i2) {
    return i1<i2;
}
```

De aanroep:

```
cout<<"Het maximum in vi is: "<<findMax(vi, lessInt)<<endl;
```

geeft bij `findMax` met 1 template parameter de volgende compilatiefout:

```
Could not find a match for 'findMax<T>(vector<int>,bool (*)(int,int))'
```

Omdat bij die versie de vergelijkingsfunctie twee `const int&` parameters moet hebben! Bij de laatste versie van `findMax` (met 2 template parameters) is dit geen enkel probleem omdat het type van de vergelijkingsfunctie door de compiler wordt ingevuld.

Nog een voordeel van het gebruik van de tweede template parameter is dat je in plaats van een functie ook een object kunt meegeven! Bijvoorbeeld:

```
class RectangeLessArea {
public:
    bool operator()(const Rectangle& r1, const Rectangle& r2) const;
};

bool RectangeLessArea::operator()(const Rectangle& r1,
                                  const Rectangle& r2) const {
    return r1.getLength()*r1.getWidth()<r2.getLength()*r2.getWidth();
}
```

Je kunt een object van deze class gebruiken om de `Rectangle` te vinden met de grootste oppervlakte:

```
cout<<"De maximum area in vr is: "<<findMax(vr,
                                    RectangeLessArea())<<endl;
```

Zo'n object dat gebruikt wordt alsof het een functie is wordt een *Functor* genoemd. In dit geval is het echter handiger om een gewone functie te gebruiken. In veel gevallen is een functor echter handiger omdat een functor een object is en dus ook data kan bevatten.

Als voorbeeld nemen we een generieke functie `findPred` die de index teruggeeft van het eerste element dat aan het predicate `P` voldoet of `v.size()` als geen enkel element aan het predicate `P` voldoet.

```
template <typename T, typename P>
vector<T>::size_type findPred(const vector<T>& v, P p) {
    vector<T>::size_type i(0);
    while (i<v.size() && !p(v[i]))
        ++i;
    return i;
}
```

Als predicate kunnen we een functiepointer meegeven:

```
bool groterDan_10(int i) {
    return i>10;
}

vector<int> vi;
// vul vector vi
```

```
cout<<"Index "<<findPred(vi, groterDan_10)<<" is > 10"<<endl;
```

Maar dit is erg onhandig. Als je de index wilt vinden van het eerste element dat groter dan 11 is, moet je weer een nieuwe functie schrijven:

```
bool groterDan_11(int i) {
    return i>11;
}

cout<<"Index "<<findPred(vi, groterDan_11)<<" is > 11"<<endl;
```

Het is in dit geval eenvoudiger om een funtor te gebruiken:

```
template <typename T>
class GroterDan {
public:
    GroterDan(const T& t);
    bool operator()(const T& t) const;
private:
    const T& t_;
};

template <typename T>
GroterDan<T>::GroterDan(const T& t): t_(t) {
}

template <typename T>
bool GroterDan<T>::operator()(const T& t) const {
    return t>t_;
}

cout<<"Index "<<findPred(vi, GroterDan<int>(10))<<" is > 10"<<endl;
cout<<"Index "<<findPred(vi, GroterDan<int>(11))<<" is > 11"<<endl;
cout<<"Index "<<findPred(vi, GroterDan<int>(12))<<" is > 12"<<endl;
```

Het meegeven van een object werkt ook als je het eerste element groter dan een variabele `x` zoekt:

```
int x;
cout<<"Geef x: ";
cin>>x;
cout<<"Index "<<findPred(vi, GroterDan<int>(x))<<" is > x"<<endl;
```

6.2 Adapter.

De volgende code:

```
void showAbout() {
    TForm* fp(new AboutForm());
    fp->showModal();
    delete fp;
}
```

levert een probleem op als de memberfunctie `showModal()` een exception gooit. De met `new` aangevraagde geheugenruimte wordt dan niet meer teruggeven. We noemen dit een memorylek.

Een onhandige oplossing:

```
void showAbout() {
```

```
    TForm* fp(new AboutForm());
    try {
        fp->showModal();
    catch(...) {
        delete fp;
        throw;
    }
    delete fp;
}
```

Door de “ruwe” pointer in te pakken (wrappen) in een object kan een handigere oplossing worden gebruikt.

```
template <typename T>
class Pointer {
public:
    Pointer(T* p);
    ~Pointer();
    T* operator->() const;
private:
    T* p_;
};

template <typename T>
Pointer<T>::Pointer(T* p): p_(p) {
}

template <typename T>
Pointer::~Pointer() {
    delete p_;
}

template <typename T>
T* Pointer::operator->() const {
    return p_;
}
```

Deze wrapper class kun je als volgt gebruiken:

```
void showAbout() {
    Pointer<TForm> fp(new AboutForm());
    fp->showModal();
}
```

Als nu een exception optreedt dan wordt de lokale variabele `fp` opgeruimd voordat de functie verlaten wordt. Ook als de functie op de normale wijze (zonder exception) wordt verlaten zal de lokale variabele worden opgeruimd. De met `new` gereserveerde geheugenruimte wordt in de destructor van de pointer-wrapper altijd weer vrijgegeven met `delete`. Er kan dus geen memorylek optreden. Deze methode wordt “resource allocation is initialisation” genoemd.

In de standaard C++ library is ook een pointerwrapper genaamd `auto_pointer` opgenomen. De bovenstaande code kun je met behulp van deze standaard wrapper als volgt implementeren:

```
void showAbout() {
    auto_ptr<TForm> fp(new AboutForm());
    fp->showModal();
}
```

7 De ISO/ANSI standard C++ library.

De C++ standard library zal aan de hand van het boek besproken worden. In dit hoofdstuk van het dictaat zijn alleen enkele voorbeelden opgenomen.

7.1 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(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(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(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;
}
```

7.2 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;
}
```

7.3 Voorbeeldprogramma met een standaard `set`.

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

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

int main() {
    set<string> docenten;
    docenten.insert("John");
    docenten.insert("Paul");
    docenten.insert("Paul");
    docenten.insert("Harry");
    docenten.insert("Bart");
    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("Paul")<<" Paul."<<endl;
    docenten.erase("Harry");
    print(docenten);
    cin.get();
    return 0;
}
```

De uitvoer van dit programma:

```
De set bevat: Bart Harry John Paul
1 Harry is genoeg.
Er is 1 Paul.
De set bevat: Bart John Paul
```

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

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

De uitvoer van dit programma:

```
De bag bevat: Bart Harry John Paul Paul
Er zijn 2 Paul's.
De bag bevat: Bart Harry John
De bag bevat: Bart
```

7.5 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;
```

```
}
```

7.6 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;
}
```

7.7 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
```

7.8 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) {
        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
```

7.9 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
```

7.10 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
```

7.11 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
```

7.12 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
```

Practicumhandleiding.

Het practicum bestaat uit 6 practicumopdrachten. De HTML versie van deze opgaven kun je vinden op: http://bd.thrijswijk.nl/syso1.

1 Opdracht 1: Een "homemade" gelinkte lijst.

In de propedeuse heb je datastructuren zoals array en struct leren gebruiken. Deze datastructuren worden statisch genoemd omdat hun grootte tijdens het compileren wordt bepaald en vast is. In de praktijk heb je al snel behoefte aan zogenaamde dynamische datastructuren waarvan de grootte tijdens het uitvoeren van het programma kan wijzigen. Later in dit kwartaal zul je leren dat in de standaard C++ library de class `list` is opgenomen die het gebruik van dynamische (gelinkte) lijsten erg eenvoudig maakt. Om een goed begrip te krijgen van de werking van dynamische datastructuren is het toch leerzaam om zelf eens een eenvoudige gelinkte lijst te maken.

1.1 Dynamische geheugentoewijzing (herhaling uit H1).

Een pointer is een variabele die het adres kan bevatten van een eerder gedeclareerde variabele. In de propedeuse heb je pointers als volgt leren gebruiken.

```
int i(3);    // i=3
int* pi(&i); // pi wijst naar i
*pi=*pi+5;   // i=8
```

Het is nu dus mogelijk de waarde van de variabele `i` te benaderen met de dereferentie van `pi`: `*pi`

In H1 heb je geleerd dat het ook mogelijk is voor een pointer een stuk geheugen te reserveren (met de `new` operator) dat niet direct gekoppeld is aan een eerder gedeclareerde variabele. De dereferentie van de pointer kan nu gebruikt worden als een gewone variabele. Dit is dan wel de enige manier om de waarde van deze “onbenoemde variabele” te benaderen. Voorbeeld van het gebruik van een pointer naar met `new` gereserveerd geheugen:

```
int* pi(new int);     // pi wijst naar een onbenoemde variabele
*pi=3;
*pi=*pi+5;
```

Als het gereserveerde geheugen dat toegewezen is aan een bepaalde pointer niet meer nodig is moet het weer vrijgegeven worden met de operator `delete`. Voorbeeld van het met `delete` vrijgeven van met `new` gereserveerd geheugen:

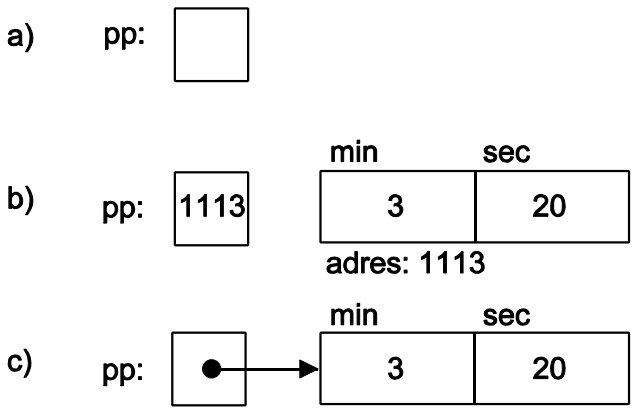
```
delete pi;
```

Stel bijvoorbeeld dat de volgende declaratie van een `class` gegeven is:

```
class PlayTime {
public:
    PlayTime(int m=0, int s=0);
    void add(const PlayTime& t);
private:
    void normalize();
    int min;
    int sec;
friend ostream& operator<<(ostream& out, const PlayTime& t);
};
// ...
PlayTime* pp;
```

```
pp=new PlayTime(3,20);
```

Na de declaratie `PlayTime* pp;` is de variabele `pp` gecreëerd, zie figuur 1a. Deze variabele bevat echter nog een niet gedefinieerde waarde en er is nog geen geheugen gereserveerd voor een object.
Na de toekenning `pp=new PlayTime;` is een stuk geheugen gereserveerd dat groot genoeg is om 1 `Playtime` object te bevatten en dit object is geïnitialiseerd in de constructor. De variabele `pp` bevat nu het adres van dit gereserveerde stuk geheugen. Zie figuur 1b.
In figuur 1c is dit abstracter weergegeven omdat de waarde van het adres onbelangrijk is. De door `pp` aangewezen tijdsduur kan door middel van de notatie `(*pp).add(*pp)` bij zichzelf worden opgeteld. Dit kan ook als `pp->add(*pp)` genoteerd worden.



Figuur1 Dynamische geheugentoewijzing

Na de instructie: `delete pp;` is het gereserveerde stuk geheugen weer vrijgegeven voor hergebruik, de variabele `pp` bestaat dan nog wel maar de inhoud hiervan is onbepaald. Zie weer figuur 1a.

1.2 Recursieve datastructuren.

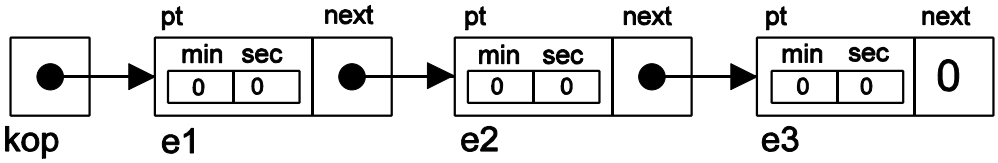
Een recursieve class is een class die naar zichzelf verwijst. Bijvoorbeeld:

```
class PlayListElement {
public[13]:                        // Een PlayListElement bestaat uit:
    PlayTime pt;                   // - een PlayTime
    PlayListElement* next;         // - een pointer naar het volgende
};                                 //   PlayListElement
```

Door het gebruik van recursieve classes is het mogelijk recursieve datastructuren (bijvoorbeeld lijsten) te creëren van naar elkaar verwijzende objecten. Bijvoorbeeld:

```
PlayListElement e1,e2,e3;
PlayListElement* kop(&e1);
e1.next=&e2;
e2.next=&e3;
e3.next=0;
```

Grafisch is direct duidelijk hoe zoiets eruit ziet:



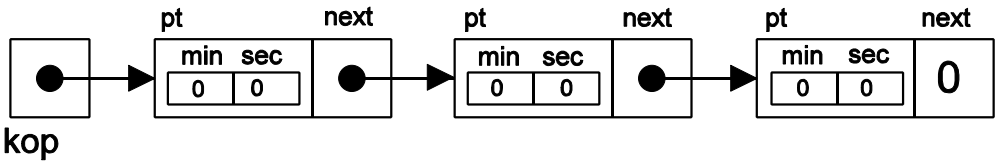
Figuur2 Een lijst van naar elkaar verwijzende objecten (gelinkte lijst).

[13] Deze class bevat `public` datamembers en er is dus *geen* sprake van “encapsulation” of datahiding. Dit was tegen de regels van goed object georiënteerd programmeren! We zullen later zien dat deze class alleen als `private` class binnen een andere class wordt gebuikt waardoor toch een goed object oriënted programma ontstaat.

Je kunt de lijst als volgt ook dynamisch opbouwen:

```
PlayListElement* kop(new PlayListElement);
PlayListElement* staart(kop);
staart->next=new PlayListElement;
staart=staart->next;
staart->next=new PlayListElement;
staart=staart->next;
staart->next=0;
```

Grafisch is dit als volgt weer te geven:



Figuur3 Lijst van naar elkaar verwijzende dynamisch aangemaakte objecten.

Merk op dat het enige verschil tussen figuur 2 en figuur 3 is dat in figuur 3 de variabelenamen `e1`, `e2` en `e3` niet voorkomen.

1.3 Voorbeeldprogramma.

Tot slot van deze inleiding over dynamische gelinkte lijsten volgt nog een voorbeeld waarin een aantal manipulaties met lijsten is opgenomen. Dit programma kun je op het practicum kopiëren vanaf http://bd.thrijswijk.nl/syso1/pract/lijst.cpp.

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

class PlayTime {
public:
    PlayTime(int m=0, int s=0);
    void add(const PlayTime& t);
private:
    void normalize();
    int min;
    int sec;
friend ostream& operator<<(ostream& out, const PlayTime& t);
};

istream& operator>>(istream& in, PlayTime& t);

PlayTime::PlayTime(int m, int s): min(m), sec(s) {
    normalize();
}

void PlayTime::add(const PlayTime& t) {
    min+=t.min;
    sec+=t.sec;
    normalize();
}

void PlayTime::normalize() {
```

```
    long tsec(min*60+sec);
    min=tsec/60;
    sec=tsec%60;
}

ostream& operator<<(ostream& out, const PlayTime& t) {
    return out<<setfill('0')<<setw(2)<<t.min<<":"<<setw(2)<<t.sec;
}

istream& operator>>(istream& in, PlayTime& t) {
    int i;
    if (in>>i) {
        if (in.peek()==':') {
            in.get();
            int j;
            if (in>>j)
                t=PlayTime(i, j);
        }
        else
            t=PlayTime(0, i);
    }
    return in;
}

// ========================================================================

class PlayList {
public:
    PlayList();
    ~PlayList();
    void insert(const PlayTime& t);
    PlayTime total() const;
private:
    class PlayListElement {
    public:
        PlayTime pt;
        PlayListElement* next;
    };
    PlayListElement* first;
friend ostream& operator<<(ostream& out, const PlayList& l);
};

istream& operator>>(istream& in, PlayList& l);

PlayList::PlayList(): first(0) {
}

PlayList::~PlayList() {
    while (first!=0) {
        PlayListElement* listElement(first);
        first=first->next;
        delete listElement;
    }
}

PlayTime PlayList::total() const {
    PlayTime t;
    PlayListElement* pList(first);
    while (pList) {
        t.add(pList->pt);
```

```
        pList=pList->next;
    }
    return t;
}

void PlayList::insert(const PlayTime& t) {
    PlayListElement* newlistElement(new PlayListElement);
    newlistElement->pt=t;
    newlistElement->next=first;
    first=newlistElement;
}

ostream& operator<<(ostream& out, const PlayList& l) {
    PlayList::PlayListElement* pList(l.first);
    while (pList!=0) {
        out<<pList->pt<<endl;
        pList=pList->next;
    }
    return out;
}

istream& operator>>(istream& in, PlayList& l) {
    PlayTime pt;
    while (in>>pt) {
        l.insert(pt);
    }
    return in;
}

// ========================================================================

int main () {
    ifstream fin("playlist.txt");
    if (fin!=0) {
        PlayList lijst;
        fin>>lijst;
        cout<<"De lijst:"<<endl<<lijst;
        cout<<"Totale speelduur: "<<lijst.total()<<endl;
    }
    else
        cout<<"De file playlist.txt kan niet worden geopend!"<<endl;
    cout<<"Druk op de return-toets."<<endl;
    cin.get();
    return 0;
}
```

Weet je de antwoorden op de volgende vragen nog?

Q: Noem twee voordelen van het gebruik van een ADT `PlayTime` in plaats van een struct `PlayTime`.

A: Doordat de datamembers in een ADT private zijn en dus alleen bereikbaar via de memberfuncties van de ADT (we noemen dit datahiding) kun je de implementatie van een ADT wijzigen zonder dat de rest van het programma daar iets van merkt. Het gebruik van een ADT maakt het programma beter *aanpasbaar* en *uitbreidbaar*. Doordat in een ADT niet alleen de structuur van de data is vastgelegd maar ook de bewerkingen die op of met deze data kunnen worden uitgevoerd (de memberfuncties) is een ADT eenvoudiger *herbruikbaar*.

Q: De parameter `t` van de memberfunctie `add` is van het type `const PlayTime&`. Kan hier ook het type `PlayTime` gebruikt worden? Wat is dan het voordeel/nadeel?

A: Ja dat kan. Bij het gebruik van het type `PlayTime` wordt dan wel een kopie van de parameter die bij aanroep meegegeven wordt gebruikt. Het aanmaken en weer verwijderen van deze kopie kost tijd en heeft dus als *nadeel* dat het trager is. Bij het gebruik van een `PlayTime&` wordt geen kopie gemaakt van de parameter die bij aanroep wordt meegegeven maar wordt een reference naar (andere naam voor) deze parameter gebruikt. Omdat we deze reference alleen willen gebruiken om de waarde van de bij aanroep meegegeven parameter te lezen gebruiken we een `const PlayTime &`.

Q: Kun je een object van de class `PlayTime` als volgt definiëren?
```
PlayTime t;
```
Zo nee, waarom niet? Zo ja, wat is dan de waarde?

A: Ja dat kan. De class `PlayTime` moet dan wel een constructor hebben zonder argumenten of een constructor waarbij alle parameters een default waarde hebben. `PlayTime` heeft de volgende constructor `PlayTime(int m=0, int s=0);`. Als je deze constructor aanroept zonder parameters mee te geven dan wordt `m` gelijk aan 0 en `s` ook gelijk aan 0. In de implementatie van deze constructor kun je zien dat de private variabele `min` geïnitialiseerd wordt met `m` en dat de private variabele `sec` geïnitialiseerd wordt met `s`. De waarde van `t` is dus `00:00`.

Q: De `operator<<` voor het afdrukken van objecten van de class `PlayTime` heeft als tweede parameter een `const PlayTime&`. De `operator>>` voor het inlezen van objecten van de class `PlayTime` heeft als tweede parameter een `PlayTime&`. Verklaar dit verschil.

A: De parameter van `operator>>` moet veranderd (ingelezen) kunnen worden en kan dus geen `const` zijn. Bij het afdrukken mag de `operator<<` het af te drukken object niet wijzigen de parameter moet dus `const` zijn.

Q: De `operator<<` voor het afdrukken van objecten van de class `PlayTime` is als friend van `PlayTime` gedeclareerd. De `operator>>` voor het inlezen van objecten van de class `PlayTime` is niet als friend gedeclareerd. Verklaar dit verschil.

A: In de implementatie van de `operator<<` kun je zien dat in deze code de private variabelen van de class `PlayTime` gebruikt worden. Dit kan alleen als deze `operator<<` een memberfunctie van `PlayTime` is (maar dat is niet zo) of als deze `operator<<` een friend is van de class `PlayTime`. In de implementatie van de `operator>>` kun je zien dat deze operator de private variabelen van `PlayTime` helemaal niet nodig heeft.

Q: De class `PlayListElement` heeft public member variabelen. Dit mag toch niet als je het programma goed uitbreidbaar en aanpasbaar wil maken? Zie ook het antwoord op de eerste vraag.

A: In dit geval mag het wel omdat de class `PlayListElement` als private class binnen `PlayList` gedefinieerd is. De class `PlayListElement` is dus alleen in memberfuncties (of fiends) van de class `PlayList` te gebruiken. In dit geval zou je dus net zo goed een `struct` kunnen gebruiken.

Q: In de vierde regel van main staat:
```
fin>>list;
```
De `operator>>` die wordt aangeroepen is als volgt gedeclareerd:
```
istream& operator>>(istream& in, PlayList& l);
```
De eerste parameter is echter van het type `ifstream` in plaats van `istream`. Waarom geeft de compiler hier geen foutmelding?

A: Er wordt gebruik gemaakt van polymorphism. De class `ifstream` is afgeleid van de class `istream`. Er geldt een `ifstream` *is een* `istream`.

1.4 Opdrachtomschrijving.

Deze opdracht bestaat uit de deelopdrachten 1a t/m 1g. Opgave 1a t/m 1f moet je laten nakijken door de practicumdocent. Opgave 1g is **niet** verplicht maar wel leerzaam.

In het voobeeldprogramma worden de speeltijden die zijn opgeslagen in de file `playlist.txt` in de functie `main` door middel van een zelf gedefinieerde `operator>>` ingelezen in een dynamisch gelinkte lijst. Een datafile waarmee het programma getest kan worden is beschikbaar op http://bd.thrijswijk.nl/syso3/pract/playlist.txt. De gelinkte lijst wordt in de `operator>>` als volgt gevuld:

```
istream& operator>>(istream& in, PlayList& l) {
     PlayTime pt;
     while (in>>pt) {
          l.insert(pt);
     }
     return in;
}
```

De memberfunctie `insert` is als volgt gedefinieerd:

```
void PlayList::insert(const PlayTime& t) {
     PlayListElement* newlistElement(new PlayListElement);
     newlistElement->pt=t;
     newlistElement->next=first;
     first=newlistElement;
}
```

Opdracht 1a.

Loop stap voor stap het inlezen van de file door en teken daarbij de gelinkte lijst die ontstaat. Je kunt met C++ Builder het programma stap voor stap volgen door een breakpoint te zetten op de eerste regel (met de sneltoets F5) en daarna met de sneltoetsen F7 en F8 het programma regel voor regel uit te voeren:

F7 = Stap naar de volgende C++ regel die wordt uitgevoerd. Als de huidige regel een of meer (member)functies aanroept dan wordt gesprongen naar de eerste regel van de eerste (member)functie die wordt aangeroepen.

F8 = Stap naar de volgende C++ regel in de code. Als de huidige regel een of meer (member)functies aanroept dan worden deze (member)functies zonder te stoppen uitgevoerd.

Je kunt de objectstructuren mooi zichbaar maken door op een objectnaam (variabelenaam) te gaan staan met de muis en dan rechts te klikken en de popup menu optie Debug, Inspect... te kiezen (sneltoets Alt+F5). Als een datamember van het object een pointer is dan kun je het object waar die pointer naar wijst bekijken door deze datamember te selecteren en dan via de rechtermuis optie Inspect (sneltoets Ctrl+I) te kiezen.

Vervolgens wordt de lijst vanuit de functie `main` afgedrukt:

```
cout<<"De lijst:"<<endl<<lijst;
```

Deze zelf gedefinieerde `operator<<` drukt de inhoud van de gelinkte lijst op het beeldscherm af:

```
ostream& operator<<(ostream& out, const PlayList& l) {
     PlayList::PlayListElement* pList(l.first);
     while (pList!=0) {
          out<<pList->pt<<endl;
          pList=pList->next;
```

```
    }
    return out;
}
```

Vanuit deze `operator<<` wordt een andere zelf gedefinieerde `operator<<` aangeroepen die er voor zorgt dat een `PlayTime` wordt afgedrukt. Het gebruik van twee operatoren (of functies) met dezelfde naam (maar met verschillende parametertypes) wordt *operator overloading (of function overloading)* genoemd.

Opdracht 1b.

Speel stap voor stap het uitvoeren van de `operator<<` na. Maak daarbij gebruik van de in opgave 1a gemaakte tekening.

Opdracht 1c.

Ontwerp nu zelf een memberfunctie van `PlayList` die het aantal elementen in een lijst telt. Deze functie moet vanuit de functie `main` als volgt aangeroepen worden:

```
cout<<"Aantal elementen: "<<lijst.numberOfElements()<<endl;
```

De declaratie van de memberfunctie is dus:

```
int numberOfElements() const;
```

Aan het eind van de `if` wordt de destructor van `PlayList` automatisch aangeroepen. Deze destructor geeft het met `new` dynamisch gereserveerde geheugen, door middel van `delete`, weer vrij voor hergebruik. Deze destructor kan **niet** als volgt gecodeerd worden:

```
PlayList::~PlayList() {
    while (first!=0) {
        delete first;
        first=first->next;
    }
}
```

Opdracht 1d.

Bedenk waarom de bovenstaande destructor niet correct is. Als je de bovenstaande destructor probeert uit te voeren zul je merken dat deze destructor toch goed lijkt te werken. Bedenk waarom dit erg *gevaarlijk* is!

Door de pointer naar het volgende `PlayListElement` in een hulpvariabele op te slaan is de destructor in het voorbeeldprogramma wel correct gecodeerd:

```
PlayList::~PlayList() {
    while (first!=0) {
        PlayListElement* listElement(first);
        first=first->next;
        delete listElement;
    }
}
```

De destructor kan met behulp van een hulpfunctie `dump` ook als volgt gecodeerd worden:

```
void PlayList::dump(PlayListElement* p) {
    if (p!=0) {
        dump(p->next);
        delete p;
    }
}

PlayList::~PlayList() {
    dump(first);
}
```

De functie `dump` is een *recursieve* functie. Dat wil zeggen dat deze functie zichzelf aanroept.

Opdracht 1e.

Speel stap voor stap het uitvoeren van de destructor na. Maak daarbij gebruik van de in opgave 1a gemaakte tekening. In welke volgorde worden de elementen vrijgegeven?

Opdracht 1f.

Ontwerp en implementeer een memberfunctie die de inhoud van de gelinkte lijst van achter naar voor afdrukt met behulp van een recursieve hulpfunctie.

Als je een element uit een gelinkte lijst moet verwijderen dan is dit best wel lastig. Als je bijvoorbeeld het tweede element uit een gelinkte lijst moet verwijderen dan moet je er eerst voor zorgen dat het eerste element naar het derde element verwijst. Pas daarna kun je het tweede element met behulp van `delete` opruimen. Het verwijderen van het eerste element heeft tot gevolg dat de pointer naar het begin van de lijst wijzigt.

Opdracht 1g (niet verplicht).

Ontwerp en implementeer een functie waarmee element `n` uit een gelinkte lijst kan worden verwijderd.

```
void remove(int n)
```

Deze memberfunctie moet element `n` verwijderen uit de lijst. Het element dat verwijderd moet worden, moet natuurlijk netjes met `delete` worden vrijgegeven. Als de gelinkte lijst minder dan `n` elementen bevat mag de functie de lijst niet wijzigen. De functie mag dan natuurlijk ook niet "vastlopen".

Test deze functie door in het hoofdprogramma toe te voegen:

```
lijst.verwijder(3);
lijst.verwijder(1);
lijst.verwijder(55);
cout<<"Na verwijderen: "<<endl<<lijst;
```

De juiste uitvoer is:

```
Na verwijderen:
10:09
```

2 Opdacht 2: Algoritme analyse.

In de file http://bd.thrijswijk.nl/syso1/pract/timedsort.cpp is de class `StopWatch` opgenomen. Deze class kan gebruikt worden om tijd te meten. In het voorbeeldprogramma wordt de tijd gemeten die het `bubble_sort` algoritme nodig heeft om een array met `N` elementen te sorteren.

Opdracht 2a.

Start het voorbeeldprogramma en zoek een waarde van `N` waarbij het sorteren ongeveer 1 seconde duurt. Voer deze waarde 10 maal in en vergelijk de resultaten. Maak een grafiekje met behulp van Excel.

De tijd die nodig is om `N` getallen te sorteren is **niet** constant! Kun je dat verklaren?

Opdracht 2b.

Probeer nu de orde van het algoritme vast te stellen door de benodigde tijd te meten voor verschillende waarden van `N`. Maak een grafiek met behulp van Excel.

Een meerderheidselement in een array van `N` elementen is een element dat **meer dan** `N`/2 keer in de array voorkomt.
De array: `3, 3, 4, 2, 4, 4, 5, 4, 4` heeft als meerderheidselement de waarde `4`.
De array: `3, 3, 4, 2, 4, 4, 5, 4, 3` heeft **geen** meerderheidselement.

Er zijn verschillende algoritmen om het meerderheidselement van een array te vinden:

- **Methode 1:** Tel hoe vaak het eerste element voorkomt. Als dit element `N`/2+1 keer voorkomt is dit het meerderheidselement. Zo niet, tel dan hoe vaak het tweede element voorkomt. Als dit element `N`/2+1 keer voorkomt is dit het meerderheidselement. Enzovoort totdat je geteld hebt hoe vaak element `N`/2+1 voorkomt. Als je dan nog steeds geen meerderheidselement hebt gevonden dan is het er ook niet.
- **Methode 2:** Sorteer de array en tel steeds het aantal elementen met dezelfde waarde (die dan achter elkaar staan). Zodra je `N`/2+1 elementen met dezelfde waarde vindt heb je het meerderheidselement gevonden.
- **Methode 3:** Bij deze methode gaan we er (in eerste instantie) vanuit dat er een meerderheidselement is. Als er een meerderheidselement is dan kun je dat vinden door steeds twee *verschillende* waarden uit de array te verwijderen. Totdat de array nog maar één waarde bevat, deze waarde is dan het meerderheidselement. Dit "verwijderen" kan op een slimme manier gedaan worden door element voor element te "bekijken".
  Het element wat eventueel het meerderheidselement zou kunnen zijn noemen we de kandidaat. We houden ook in een teller bij hoe vaak we de kandidaat hebben gezien. We gaan nu 1 voor 1 de elementen van de array af. Bij het "bekijken" van een element zijn er twee mogelijkheden.
  - De teller is 0
    We hebben nog geen kandidaat dus we kiezen dit element als kandidaat en zetten de teller op 1.
  - De teller is >0
    Als het element gelijk is aan de kandidaat verhogen we de teller met 1 en anders verlagen we de teller met 1.

  Als we alle elementen hebben "bekeken" blijft er een kandidaat over (teller > 0) of niet (teller == 0). Omdat we er (in eerste instantie) vanuit gegaan zijn dat de array een meerderheidselement heeft moet je nu, als er een kandidaat is, nog wel even controlleren of dit het meerderheidselement is door te tellen hoe vaak de kandidaat in de array voorkomt.
  Uitleg:
  Als de teller 0 wordt hebben we net zoveel elementen gehad die gelijk waren aan de kandidaat (telkens +1) als elementen die ongelijk waren aan de kandidaat (-1). Deze elementen vormen dus paartjes van *ongelijke* elementen en die kunnen we dus allemaal uit de array verwijderen. Als er een meerderheidselement was in de oorspronkelijke array dan heeft de resterende rij hetzelfde meerderheidselement! (Dit zou wel of niet de vorige kandidaat kunnen zijn maar dat maakt niets uit!)
  Deze methode is bedacht door Moore en Boyer. Zie http://www.cs.utexas.edu/users/moore/best-ideas/mjrty/index.html voor een eenvoudige uitleg en een stap voor stap demo. Zie http://www.cs.utexas.edu/users/boyer/mjrty.pdf voor een uitgebreide uitleg.

Opdracht 2d.

Schrijf een functie die kan bepalen of een array een meerderheidselement heeft (met één van de drie hierboven beschreven methoden). Als de array een meerderheidselement heeft dan moet de functie dit element ook teruggeven. Het prototype van de functie is als volgt:

```
bool zoekMeerderheidsElement(int& resultaat, int* start, int* end);
```

De parameter `start` moet wijzen naar het eerste element van de array en de parameter `end` moet wijzen 1 positie voorbij het laatste element van de array. De functie geeft `false` terug als geen meerderheidselement gevonden is. De functie geeft `true` terug als wel een meerderheidselement gevonden is. Dit meerderheidselement wordt dan opgeslagen in de variabele waar de parameter `resultaat` naar refereert.

Gebruik het programma http://bd.thrijswijk.nl/syso1/pract/timedzme.cpp om je programma te testen en om de orde van het door jou geïmplementeerde algoritme te bepalen.

3 Opdracht 3: Exceptions.

De in dit dictaat behandelde implementatie van een stack door middel van een array (zie paragraaf 4.1) breekt het programma af als een stackoverflow of stackunderflow optreedt.

Tip:
Als je een programma test met Borland Builder dan zal het uitvoeren van het programma, telkens als er een exception optreedt, onderbroken worden. Je moet dan na elke exception een "run" commando geven om het programma verder te laten gaan. Dit is bij het testen van exceptions (bij deze opdracht dus) erg irritant. Je kunt dit veranderen via het menu Tools, Debugger Options. Je moet in het tabblad "Language Exceptions" het vinkje "Stop on C++ Exceptions" uitzetten.

Opdracht 3.

De template class `StackWithArray<T>` (http://bd.thrijswijk.nl/syso1/pract/stackarray.h) is afgeleid van de abstract template base class `Stack<T>` (http://bd.thrijswijk.nl/syso1/pract/stack.h).Pas de file stackarray.h zodanig aan dat de stack een exception veroorzaakt bij stackoverflow, stackunderflow en andere errors. Ontwerp een hiërarchie van exceptionclasses die het mogelijk maken om een specifieke of generieke stackfout af te vangen. Geef de exceptionclasses een memberfunctie `print()` die ervoor zorgt dat een passende foutmelding wordt afgedrukt. Zorg ervoor dat in het volgende (http://bd.thrijswijk.nl/syso1/pract/stacktest.cpp) geval een juiste foutmelding verschijnt:

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

#include "stackarray.h"

int main() {
      StackWithArray<int> s(10);
      try {
            s.push(2);
            while (true)
                  s.pop();
      } catch (StackError& e) {
            // hier worden alle generieke StackError's opgevangen
            e.print();
      }
      try {
            while(true)
                  s.push(3);
      } catch (StackPushError& e) {
            // hier wordt de specifieke StackPushError opgevangen
            e.print();
      }
      cin.get();
      cin.get();
      return 0;
}
```

4 Opdracht 4: Functors.

In hoofdstuk 5 van het boek (behandeld in les 8) heb je kennis gemaakt met het Functor design pattern.

De file http://bd.thrijswijk.nl/syso1/pract/functor.cpp bevat de functie:

```
void quick_sort(int* begin, int* end);
```

Deze functie kunnen je gebuiken om een array met integers te sorteren. Dit wordt gedemonstreerd in de functie `main`:

```
    const int aantal(20);
    int a[aantal];
    for (int i(0); i<aantal; ++i) {
        a[i]=std::rand();
    }
    quick_sort(a, a+aantal);
```

Je kunt de functie `quick_sort` echter niet gebruiken om een array met objecten van de class `Student` te sorteren. De class `Student` is als volgt gedefinieerd:

```
class Student {
public:
    Student(const char* vn, const char* an, int n);
    const std::string& geefVoornaam() const;
    const std::string& geefAchternaam() const;
    int geefNummer() const;
private:
    std::string voornaam;
    std::string achternaam;
    int nummer;
};
```

Opdracht 4.

Maak van de functie `quick_sort` een template functie waardoor je deze functie kunt gebruiken om array's van elk willekeurig type te sorteren. Houd er rekening mee dat er verschillende types zijn (zoals bijvoorbeeld `Student` of `Rechthoek`) waarvoor geen eenduidige definitie van de `operator<` kan worden gegeven. De functie die wordt gebruikt om elementen te vergelijken moet dus ook aan `quick_sort` kunnen worden doorgegeven. Zorg ervoor dat niet alleen functies maar ook functors gebruikt kunnen worden.

Test de `quick_sort` template door de array met `Student`'en te sorteren:

- op nummer (van hoog naar laag) door een vergelijkings**functie** mee te geven:
  `quick_sort(array, array+n, /* vul hier een functie in! */);`
- op voornaam (van laag naar hoog) door een vergelijkings**functor** mee te geven:. Studenten met dezelfde voornaam moeten op achternaam gesorteerd worden.
  `quick_sort(array, array+n, /* vul hier een Functor in! */);`

5 Opdracht 5. Opsporen van fraude.

Bij het C++ practicum wordt wel eens een uitwerking van een practicumopdracht gekopieerd. Vaak is een student die een programma kopieert wel zo slim om de namen van variabelen te wijzigen en de opmaak van het programma te veranderen. Bij SYSO komt dit gelukkig niet meer voor ;-). Een docent wil daarom een programma maken dat een aantal kenmerken van een (ander) C++ programma vaststelt zodat verschillende door studenten ingeleverde programma's op deze kenmerken met elkaar kunnen worden vergeleken. De docent is van mening dat het aantal maal dat elk C++ keyword in een C++ programma voorkomt een belangrijk kenmerk van een programma is om de originaliteit van een programma vast te stellen.

Opdracht 5a.

Schrijf een C++ programma dat in een ander C++ programma telt hoeveel maal elk C++ keyword daarin voorkomt. Maak gebruik van de file http://bd.thrijswijk.nl/syso1/pract/keywords.txt waarin alle C++ keywords staan. (Deze file moet aangepast of uitgebreid kunnen worden.) Maak zoveel mogelijk gebruik van **datastructuren** uit de standaard C++ library. Na afloop moet het programma een lijst produceren met per regel een C++ keyword met daarbij het aantal maal dat dit keyword voorkomt. Lees de hierna volgende tips als je zelf geen plan van aanpak kunt bedenken.

Tip 1:
Als datastructuur kan een `map` met per keyword een teller gebruikt worden. Het programma uit paragraaf ? kan prima als uitgangspunt gebruikt worden. Natuurlijk moet je er dan wel voor zorgen dat alleen de keywords geteld worden. Dit kan op twee manieren:

- Maak een `set` aan met daarin alle keywords. Met de memberfunctie `count` van `set` kan eenvoudig bepaald worden of een ingelezen woord een keyword is (en geteld moet worden).
- Vul de `map` voordat je gaat beginnen met tellen eerst met alle keywords (alle tellers op `0`). Je moet dan alleen woorden tellen die al in de `map` voorkomen. Met de functie `count` van `map` kan eenvoudig bepaald worden of een ingelezen woord al in de `map` voorkomt.

Tip 2:
Als je er voor hebt gekozen om een `set` te gebruiken dan kun je de inhoud van de file keywords.txt naar de `set` kopiëren met het `copy` algoritme door gebruik te maken van `istream_iterator`'s

Tip 3:
De inhoud van de map kun je op het scherm afdrukken door gebruik te maken van het `copy` algoritme en `ostream_iterators`'s. Het is dan wel nodig om een operator<< te definieren voor de standaard class template pair (de elementen in de map zijn in ons geval van het type pair<const string, int>). Deze operator kan als volgt gedefinieerd worden:

```
template <typename Key, typename Value>
ostream& operator<<(ostream& o, pair<Key, Value> p) {
     return o<<p.first<<" "<<p.second<<endl;
}
```

Het programma uit de vorige opdracht is niet zo gebruiksvriendelijk. Als de docent twee programma's wil vergelijken dan moet hij het programma 2 keer uitvoeren en de uitvoer handmatig met elkaar vergelijken. De docent wil dit vergelijken automatiseren.

Opdracht 5b.

Schrijf een C++ programma dat in twee andere C++ programma telt hoeveel elk C++ keyword daarin voorkomt. Maak gebruik van de file `keywords.txt` waarin alle C++ keywords staan. (Deze file moet aangepast of uitgebreid kunnen worden.) Na afloop van dit programma moet één van de volgende meldingen worden gegeven:

1. Deze programma's zijn hoogstwaarschijnlijk niet origineel.
2. Deze programma's zijn misschien niet origineel.
3. Deze programma's zijn hoogstwaarschijnlijk wel origineel.

Melding 1 moet worden gegeven als alle C++ keywords in beide programma's exact evenvaak voorkomen. Melding 2 moet gegeven worden als het aantal maal dat een keyword gebruikt wordt slechts bij hoogstens 3 keywords afwijkt. In alle andere gevallen moet de derde melding gegeven worden. Maak zoveel mogelijk gebruik van datastructuren en **algoritmen** uit de standaard C++ library. Lees de hierna volgende tip als je zelf geen plan van aanpak kunt bedenken.

Tip:
Je kan natuurlijk voor elke file een aparte `map` gebruiken maar het is eenvoudiger om 1 `map` te gebruiken en bij het inlezen van de ene file de bij de keyword's behorende tellers te verhogen en bij het inlezen van de andere file de bij de keyword's behorende tellers te verlagen. Voor alle keyword's die even vaak in beide files gebruikt worden zullen de tellers na afloop op 0 staan. Met een `count_if` kun je vervolgens tellen hoeveel tellers ongelijk aan 0 zijn.

6 Opdracht 6: Boter Kaas en Eieren.

Bij deze opdracht mag je kiezen uit 6a t/m 6d. **Je hoeft dus maar 1 deelopgave te maken!** Alle opdrachten gaan over het uitbreiden of aanpassen van het in les 15 besproken Tic-Tac-Toe programma uit hoofdstuk 8.7 en 11.2 van het boek van Weiss. Je kunt de door mij aangepaste versies als uitgangspunt gebruiken:

- Tic-Tac-Toe: http://bd.thrijswijk.nl/syso1/progs/TicTacSlow.cpp
- Tic-Tac-Toe met alpha-beta pruning en transposition table: http://bd.thrijswijk.nl/syso1/progs/TicTac.cpp

Opdracht 6a: Verbeteren van het TTT algoritme.

Bij deze opgave kun je jezelf verder verdiepen in het toegepaste algoritme en de gebruikte datastructuren. Het toegepaste minimax algoritme met alpha-beta pruning en transposition table kan nog verder worden verbeterd:

- Als de computer gewonnen staat neemt hij niet altijd de kortste weg naar de winst (uiteindelijk wint hij natuurlijk wel). Zie ook opgave 11.8 pagina 407 van Weiss.
- Als de computer mag beginnen dan kiest de computer een willekeurige beginzet. Als de gebruiker mag beginnen dan volgt op een bepaalde beginzet van de gebruiker altijd dezelfde zet van de computer. Bij het uitvoeren van het minimax algoritme kiest de computer altijd hetzelfde maximum of minimum. Als je deze keuze random maakt speelt de computer met meer variatie.
- Het speelbord waarop Tic-Tac-Toe gespeeld wordt is symmetrisch. Door het bord te draaien of te spiegelen verandert de stelling niet. Als je stellingen die niets anders zijn dan een verdraaiing of spiegeling van een al geëvalueerde stelling herkent dan kan het aantal recursieve aanroepen van `chooseMove` nog verder verminderd worden. Bij de beginzet hoeft de computer nog maar 3 mogelijke eerste zetten te evalueren (in plaats van 9). Natuurlijk moet je de transpositietabel nu al vanaf level 1 gebruiken.

Pas het algoritme zo aan dat de computer de kortste weg naar de winst neemt, met meer variatie speelt en verdraaiingen en spiegelingen herkent.

Opdracht 6b: 5x5 TTT.

Bij het Tic-Tac-Toe spel is het mogelijk om een stelling helemaal door te rekenen. Bij veel andere spellen is dat niet mogelijk. Lees paragraaf 11.2.3 over schaakprogramma's.

Schrijf een uitgebreide versie van Tic-Tac-Toe dat op een bord van 5x5 wordt gespeeld en waarbij de eerste speler met 4 aansluitende kruisjes of rondjes wint. Beantwoord voor je gaat beginnen de volgende vragen:

- Kun je het spel helemaal doorrekenen?
- Hoe kun je de waarde bepalen van een stelling die geen eindstelling is?
- Hoe kun je voorkomen dat de computer een eenmaal gekozen pad helemaal tot het eind probeert door te rekenen?
- Hoe kun je de "intelligentie" van de computerspeler instelbaar maken?

Opdracht 6c: TTT GUI.

Bij deze opgave kun je jezelf verder bekwamen in het schrijven van windows applicaties. Er wordt bij deze opgave vanuit gegaan dat je de inleidende opdracht over het programmeren van windows applicaties met Borland 6 hebt gemaakt. Zie http://bd.thrijswijk.nl/pso2/winapps/pso2windows.html.

Maak een gebruiksvriendelijke (grafische) user interface voor het Tic-Tac-Toe spel.

Opdracht 6d: Mega TTT.

Ontwerp een mega versie van het spel boter, kaas en eieren. In deze mega versie bestaat het speelbord uit 9 spelletjes. Elk spelletje bestaat uit 9 vakjes waarin boter, kaas en eieren gespeeld kan worden. Als in een spelletje de kruisjes winnen wordt dit hele spelletje een kruis. Als in een spelletje de rondjes winnen wordt dit hele spelletje een rondje. Als een spelletje helemaal gevuld is zonder dat er drie kruisjes of rondjes op een rij staan wint degene die de meeste symbolen in dit spelletje heeft staan. Het speelveld bestaande uit de 9 spelletjes wordt op deze manier zelf ook weer een boter kaas en eieren spelletje. Zodra één van de twee spelers drie spelletjes op een rij wint, wint hij of zij het totale spel. Ontwerp het spel zodanig dat het door één persoon tegen de computer gespeeld kan worden.