Dictaat
SOPX2E1C1
PROGMI1T2

© 2006 Harry Broeders

Studiewijzer E: SOPX2E1C1 en SOPX2E1P1
Studiewijzer TI: PROGMI1T2 en PROGMI1P2

titel:	Object Georiënteerd Programmeren in C++
code onderwijsdeel:	E: SOPX2E1C1 en SOPX2E1P1 TI: PROGMI0T2 en PROGMI0P2
studiebelasting:	84 SBU
semester / kwartaal:	E: H2 / 1 TI: H1 / 1
contacturen:	2 uur/week college en 2 uur/week practicum
toetsing:	tentamen (cijfer) en practicumbeoordeling (O/V)
benodigde voorkennis:	Programmeren uit de P fase
verantwoordelijke docent:	Harry Broeders

Inleiding

In steeds meer producten en systemen wordt programmatuur gebruikt. Het is zeker dat een aanstaande bachalor of engineering elektrotechniek of technische informatica hiermee te maken krijgt. In de propedeuse heb je leren programmeren in de programmeertaal C met behulp van de functionele decompositie ontwerpmethode (structured programming and structured design). Deze methode van werken is prima geschikt voor kleine programmeerprojecten maar levert bij grote projecten problemen op, vooral op het gebied van onderhoudbaarheid, uitbreidbaarheid en hergebruik. Op dit moment is de zogenaamde object georiënteerde ontwerpmethode (OOD = Object Oriënted Design) erg populair.

Tijdens deze onderwijseenheid zal je de basisaspecten van object georiënteerd programmeren (OOP) leren aan de hand van de, op C gebaseerde, programmeertaal C++. Je zult leren hoe het met behulp van templates mogelijk is om algoritmen en datastructuren generiek te definiëren. Een generiek algoritme is een algoritme dat onafhankelijk is van de gebruikte datatypen. De programmeertaal C++ ondersteunt dus 3 verschillende programmeer paradigma's: structured programming, object oriënted programming en generic programming.

In deze onderwijseenheid maak je kennis met object oriënted programming en generic programming. Voor E studenten die kiezen voor de variant Computers & Datacommunicatie (C&D) wordt in semester H3 in de onderwijseenheid SOPX3 verder op deze onderwerpen ingegaan. Voor TI studenten wordt in semesters H1 t/m H3 in verschillende onderwijsdelen verder op deze onderwerpen ingegaan. In de C++ ISO/ANSI standaard is een verzameling generieke algoritmen en datastructuren opgenomen die in volgende onderwijsdelen behandeld zal worden.

Je zult na afloop van SOPX2E1C1/P1 of PROGMI1T2/P2 in staat zijn om herbruikbare software componenten te gebruiken, ontwerpen, implementeren en testen. In deze onderwijseenheid ligt de nadruk op OOP, in volgende onderwijsdelen zal dieper op OOD en OOA (object georiënteerde analyse) worden ingegaan.

Leerdoelen

Na ongeveer een week:

- ben je bekend met de volgende C++ taalaspecten:
  - plaats en vorm van definities en declaraties
  - constanten door middel van `const`.
  - standaard include files.
  - input en output met `iostream`.
  - de standaard typen `bool`, `string` en `vector`.
  - default parameters en function name overloading.

    - dynamic memory allocation met `new` en `delete`.
    - references.
- kun je gebruik maken van de standaard C++ class `string` en van de `iostream` library.

Na ongeveer vier weken:
- ben je bekend met de volgende C++ taalaspecten:
    - class en object (type en instance).
    - member functions en datamembers (behaviour en state).
    - public en private (interface en implementatie).
    - constructor en destructor.
    - speciale constructors (default, copy en conversion).
    - function en operator overloading.
    - conversion operator.
    - inline functies.
    - `this` pointer.
- snap je het nut van ADT's.
- kun je een eenvoudig ADT programmeren met behulp van C++.

Na ongeveer zeven weken:
- ben je bekend met de volgende C++ taalaspecten:
    - templates.
    - inheritance.
    - protected members.
    - virtual member function overriding.
    - polymorphism.
- kun je een eenvoudige "vector" ADT ontwerpen, implementeren en testen.
- kun je een eenvoudige class hiërarchie ontwerpen en implementeren.

Als je deze onderwijseenheid met een voldoende hebt afgesloten ben je in staat om de basisaspecten van OOD en OOP toe te passen. Deze aspecten omvatten:
- responsibility driven design (ontwerpen uitgaande van verantwoordelijkheden).
- information hiding (het afschermen van informatie door middel van het scheiden van interface en implementatie).
- abstraction (het afschermen van complexiteit door middel van het scheiden van interface en implementatie).
- inheritance (het mogelijk maken van een nieuwe vorm van hergebruik, ... is een ... in plaats van ... heeft een ...).
- polymorphism (veelvormigheid mogelijk gemaakt door dynamic binding).

Literatuur

Bij dit onderwijsdeel heb je **geen** boek nodig. Alle leerstof kun je in dit dictaat vinden. Als je het dictaat onduidelijk vindt of als je meer voorbeelden zoekt zul je zelf op zoek moeten gaan naar een goed C++ boek. Een bruikbaar Nederlands boek is: Leen Ammeraal, C++ / druk 6, ISBN 9039519358. Als je meer achtergrondinformatie of diepgang zoekt kun je het volgende boek gebruiken: Bruce Eckel, Thinking in C++ 2nd Edition, Volume 1, ISBN 0-13-979809-9. Dit boek is ook gratis te downloaden van: http://mindview.net/Books/TICPP/ThinkingInCPP2e.html.

Broeders, Sheets en voorbeeldprogramma’s zijn beschikbaar op internet http://bd.thrijswijk.nl/sopx2/. Broeders, Dictaat Object Georiënteerd Programmeren in C++. Behalve deze studiewijzer bevat dit dictaat:
- alle theorie die behandeld wordt.
- een handleiding voor het practicum.

Op mijn homepage kun je verwijzingen en achtergrond informatie vinden: http://bd.thrijswijk.nl/.

Toetsing en beoordeling.

Er worden voor deze onderwijsdelen twee deelresultaten vastgesteld waarbij het eerste resultaat (tentamen SOPX2E1C1/PROGMI1T2) een cijfer (1..10) is en het tweede resultaat (practicum SOPX2E1P1/PROGMI1P2) een O(nvoldoende) of V(oldoende) is. Als het tweede resultaat een O is dan kan de onderwijseenheid waar deze onderwijsdelen deel van uitmaken niet worden behaald. Bij het tentamen mag je een boek en dit dictaat gebruiken. Het tentamen bestaat uit open vragen. Het practicum wordt beoordeeld met Onvoldoende of Voldoende. Het practicum bestaat uit 3 opdrachten. De opdrachtbeschrijving kun je vinden in de practicumhandleiding die is opgenomen in dit dictaat. 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 een 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	7	Inleiding
2,3	dictaat H1 en H2	10	C++ as a better C
4	dictaat H3.1	23	Inleiding OOD en OOP
5 t/m 9	dictaat H3	26	Objects en Classes
10	dictaat H4	55	Templates
11 t/m 13	dictaat H5	59	Inheritance en Polymorphisme
14			Uitloop

Een gedetailleerde planning voor de theorielessen kun je vinden op het internet:
http://bd.thrijswijk.nl/sopx2/studiew.htm#planning.

Globale weekplanning practicum.

"week"	**studiemateriaal**	**opdracht**
1,2	dictaat H1 en H2	Gebruik van `string` en `iostream`
3 t/m 5	dictaat H3	Abstract Data Type
6,7	dictaat H5	Inheritance en Polymorphisme

Een gedetailleerde planning voor de practicumlessen kun je vinden op het internet:
http://bd.thrijswijk.nl/sopx2/studiew.htm#planning_practicum.

Inhoudsopgave.

Inleiding.

Dit is het dictaat: "Object Georiënteerd Programmeren in C++". Dit dictaat kan zonder boek gebruikt worden. Als je meer achtergrondinformatie of diepgang zoekt kun je gebruik maken van het boek: *"Thinking in C++ 2nd Edition, Volume 1"* van Bruce Eckel (2000 Pearson Education). De practicum-handleiding is in dit dictaat opgenomen. Dit dictaat is zoals alle mensenwerk niet foutloos, verbeteringen en suggesties zijn altijd welkom!

Halverwege de jaren '70 werd steeds duidelijker dat de veel gebruikte software ontwikkelmethode structured design (ook wel functionele decompositie genoemd) niet geschikt is om grote uitbreidbare en onderhoudbare software systemen te ontwikkelen. Ook bleken de "onderdelen" van een applicatie die met deze methode is ontwikkeld, meestal niet herbruikbaar in een andere applicatie. Het heeft tot het begin van de jaren '90 geduurd voordat een alternatief voor structured design het zogenaamde, object oriented design (OOD), echt doorbrak. Object georiënteerde programmeertalen bestaan al sinds het begin van de jaren '70. Deze manier van ontwerpen (OOD) en programmeren (OOP) is echter pas in het begin van de jaren '90 populair geworden nadat in het midden van de jaren '80 de programmeertaal C++ door Bjarne Stroustrup was ontwikkeld. Deze taal voegt taalconstructies toe aan de op dat moment in de praktijk meest gebruikte programmeertaal C. Deze object georiënteerde versie van C heeft de naam C++ gekregen en heeft zich in korte tijd (de eerste release van Borland C++ was in 1990 en de eerste release van Microsoft C++ was in 1992) ontwikkeld tot één van de meest gebruikte programmeertalen van dit moment. C++ is echter geen pure OO taal (zoals bijvoorbeeld smalltalk) en kan ook gebruikt worden als procedurele programmeertaal. Dit heeft als voordeel dat de overstap van C naar C++ eenvoudig te maken is maar heeft als nadeel dat C++ gebruikt kan worden als een soort geavanceerd C zonder gebruik te maken van OOP.

Een terugblik op C.

We starten deze onderwijseenheid met met de overgang van C naar C++. **Ik ga er van uit dat je de taal C zoals behandeld in de propedeuse beheerst.** Misschien is het nodig om deze kennis op te frissen. Vandaar dat dit dictaat begint met een terugblik op C (wordt verder in de les niet behandeld). We zullen dit doen aan de hand van een voorbeeldprogramma dat een lijst met gewerkte tijden (in uren en minuten) inleest vanuit de file `lijst.txt` en de totaal gewerkte tijd (in uren en minuten) bepaalt en afdrukt.

```
#include <stdio.h>

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

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

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

/* Deze functie berekent de totaal gewerkte tijd
   uit een rij met n gewerkte tijden */
```

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

#define MAX  100

int main () {
     FILE* fp=fopen("lijst.txt", "r");
     if (fp!=NULL) {
          struct Tijdsduur rij[MAX];
          int aantal=0;
          while (aantal<MAX &&
           fscanf(fp, "%d%d", &rij[aantal].uur, &rij[aantal].min)!=EOF)
              ++aantal;
          if (feof(fp)==0)
              printf("De file is niet volledig uitgelezen!\n");
          drukafRij(rij, aantal);
          printf("De totaal gewerkte tijd is:\n");
          drukaf(som(rij, aantal));
          fclose(fp);
     }
     else
          printf("De file lijst.txt kan niet worden geopend!\n");
   getchar();
   return 0;
}
```

Verklaring:

- In de eerste regel wordt de file `stdio.h` “included”. Dit is nodig om gebruik te kunnen maken van functies en typen die in de standaard C I/O library zijn opgenomen. In dit programma maak ik gebruik van `printf` (om te schrijven naar het scherm), van `getchar` (om te lezen vanaf het toetsenbord) en van `FILE`, `fopen`, `fclose`, `feof` en `fscanf` (om te lezen uit een file).

- Vervolgens is het samengestelde type `struct Tijdsduur` gedeclareerd. Variabelen van dit type bevatten twee datavelden (Engels: datamembers) van het type `int`. Deze datavelden heten `uur` en `min` en zijn bedoeld voor de opslag van de uren en de minuten van de betreffende tijdsduur.

- Vervolgens worden er drie functies gedefinieerd:
  - `drukaf`. Deze functie drukt de als parameter `td` meegegeven `struct Tijdsduur` af op het scherm door gebruik te maken van de standaard schrijffunctie `printf`. Het return type van deze functie is `void`. Dit betekent dat de functie geen waarde teruggeeft.
  - `drukafRij`. Deze functie drukt een rij met gewerkte tijden af. Deze functie heeft twee parameters. De eerste parameter genaamd `trij` is een array met elementen van het type `struct Tijdsduur`. De tweede parameter (een integer genaamd `n`) geeft aan hoeveel elementen uit de array afgedrukt moeten worden. Tijdens het uitvoeren van de `for` lus krijgt de lokale integer variabele `teller` achtereenvolgens de waarden `0` t/m `n-1`. Deze `teller` wordt gebruikt om de elementen uit `trij` één voor één te selecteren. Elk element

(een variabele van het type `struct Tijdsduur`) wordt met de functie `drukaf` afgedrukt.

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

Het return type van de functie `som` is van het type `struct Tijdsduur`. Aan het einde van de functie `som` wordt de waarde van de lokale variabele `s` teruggegeven (`return s`).

- Vervolgens wordt met de preprocessor directive `#define` de constante `MAX` gedefinieerd met als waarde `100`.

- Tot slot wordt de hoofdfunctie `main` gedefinieerd. Deze functie zal bij het starten van het programma aangeroepen worden. In de functie `main` wordt de lokale variabele `fp` gedefinieerd van het type `FILE*`. Het type `FILE` is gedeclareerd in de headerfile `stdio.h` en kan gebruikt worden om files te bewerken. Deze lokale variabele wordt meteen geïnitialiseerd met de return waarde van de functie `fopen`. Deze functie die gebruikt kan worden om een file te openen bevindt zich in de standaard C library en het prototype van de functie is gedeclareerd in de headerfile `stdio.h`. Aan de functie `fopen` moeten twee parameters worden doorgegeven. Als eerste de naam van de te openen file en als tweede de zogenaamde “mode”. In dit geval heb ik als filenaam `"lijst.txt"` en als mode `"r"` (dat wil zeggen open voor lezen (Engels: read)) gebruikt. De functie `fopen` geeft een `FILE*` naar de geopende file terug of als het openen van de file niet mogelijk is een speciale pointer genaamd `NULL`. Door de lokale variabele `fp` te vergelijken met `NULL` wordt in het eerste `if` statement getest of het openen van de file `lijst.txt` gelukt is. Als dit niet het geval is wordt een foutmelding afgedrukt met de standaard schrijffunctie `printf`.

  Als het openen wel gelukt is wordt een array `rij` aangemaakt met `MAX` elementen van het type `struct Tijdsduur`. Tevens wordt de integer variabele `aantal` gebruikt om het aantal ingelezen gewerkte tijden te tellen. Elke gewerkte tijd bestaat uit twee integers: een aantal uren en een aantal minuten. In de `while` lus worden gewerkte tijden uit de file aangewezen door `fp` ingelezen in de array `rij`. Telkens als een tijdsduur gelezen is wordt de variabele `aantal` met `1` verhoogd. De getallen worden uit de file gelezen met de standaard file leesfunctie `fscanf`. Deze functie bevindt zich in de standaard C library en het prototype van de functie is gedeclareerd in de headerfile `stdio.h`. De eerste parameter van `fscanf` genaamd `fp` wijst naar de geopende file waaruit gelezen kan worden. De tweede parameter specificeert wat er ingelezen moet worden. In dit geval twee integer getallen. De volgende parameters specificeren de adressen van de variabelen waar de ingelezen integer getallen moeten worden opgeslagen. Met de operator `&` wordt het adres van een variabele opgevraagd. De functie `fscanf` geeft een integer terug die aangeeft hoeveel velden ingelezen zijn. Als tijdens het lezen het einde van de file bereikt is geeft de functie `fscanf` de speciale waarde `EOF` terug. De conditie van de `while` lus is zodanig ontworpen dat de `while` lus uitgevoerd wordt zo lang `aantal<MAX` én de return waarde van `fscanf` ongelijk is aan `EOF`. De `while` lus wordt dus beëindigd als de array vol is of als het einde van de file bereikt is. Na afloop van de `while` lus wordt met de standaard functie `feof` getest of het einde van de file bereikt is (als dit niet zo is geeft `feof` de waarde `0` terug). Als het einde van de file nog niet bereikt is wordt een melding afgedrukt. Vervolgens wordt de lijst afgedrukt met de

functie `drukafRij`. Tot slot wordt de totaal gewerkte tijd (die berekend wordt met de functie `som`) afgedrukt met de functie `drukaf`, waarna de file met de functie `fclose` wordt afgesloten.

Net voor het einde van de functie `main` wordt de stdio functie `getchar()` aangeroepen. Deze functie wacht totdat de gebruiker een return intoetst. Dit is nodig omdat de debugger van C++ Builder het output window meteen sluit als het programma eindigt. Tot slot geeft de functie `main` de waarde `0` aan het operating system terug. De waarde 0 betekent dat het programma op normale wijze is geëindigd.

Merk op dat dit programma geen files met meer dan `MAX` gewerkte tijden kan verwerken. In het onderwijsdeel SOPX3E1C1 (in EH3C&D) of in PROGMI1T3 (in IH1) kun je leren hoe dit probleem is op te lossen.

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

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

1 Inleiding van C naar C++.

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

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

In dit dictaat bespreken we eerst de manieren waarop C++ zijn voorganger C heeft verbeterd. Data abstractie, object georiënteerd programmeren en generiek programmeren komen in de hoofdstukken 3 t/m 5 uitgebreid aan de orde. C++ is een zeer uitgebreide taal en deze onderwijseenheid moet dan ook zeker niet gezien worden als een cursus C++. Wij behandelen slechts de meest belangrijke delen van C++.

1.1 Commentaar met `//`.

De eerste uitbreiding die we bespreken is niet erg ingewikkeld maar wel erg handig. Je bent gewend om in C programma's commentaar te beginnen met `/*` en te eindigen met `*/`. In C++ kun je naast de oude methode ook commentaar beginnen met `//` dit commentaar eindigt dan aan het einde van de regel. Dit is handig (minder typewerk) als je commentaar wilt toevoegen aan een programmaregel.

1.2 Vorm van variabelen initialisaties.

Je bent gewend om in C programma's, variabelen (indien mogelijk) meteen bij definitie te initialiseren door middel van de syntax: `typenaam varnaam=initvalue;`. In C++ is, om redenen die in hoofdstuk 2 zullen blijken (zie blz. 51 voetnoot 52), de volgende syntax toegevoegd:

```
typenaam varnaam(initvalue);.
```

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

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

Je bent gewend om in C programma's, variabelen te definiëren aan het begin van een blok (meteen na `{`). In C++ kun je een variabele overal in een blok definiëren. De scope van de variabele loopt van het punt van definitie tot het einde van het blok waarin hij gedefinieerd is. Het is zelf mogelijk om de besturingsvariabele van een `for` lus in het `for` statement zelf te definiëren[3]. In C++ is het gebruikelijk om een variabele pas te definiëren als de variabele nodig is en deze variabele dan meteen te initialiseren. Dit maakt het programma beter te lezen (je hoeft niet als een jojo op en neer te springen) en de kans dat je een variabele vergeet te initialiseren wordt kleiner.

1.4 Constante waarden met const. (Zie eventueel TICPP Chapter03.html#Heading134, Chapter08.html#Heading248 en Chapter08.html#Heading253.)

Je bent gewend om in C programma's symbolische constanten[4] te definiëren met de preprocessor directive `#define`. Het is in C niet mogelijk om constanten te definiëren van zelfgemaakte types. In C++ kun je beter de nieuwe `const` declarator gebruiken.

Voorbeeld met `#define`:

```
#define aantalRegels 80
```

Hetzelfde voorbeeld met `const`:

```
const int aantalRegels(80);
```

Omdat je bij een `const` declarator het type moet opgeven kan de compiler meteen controleren of de initialisatie klopt met dit opgegeven type. Bij het gebruik van de preprocessor directive `#define` blijkt dit pas bij het gebruik van de constante en niet bij de definitie. De fout is dan vaak moeilijk te vinden. In hoofdstuk 3 (blz. 35) zul je zien dat het met een `const` declarator ook mogelijk wordt om constanten te definiëren van zelfgemaakte types.

Een constante moet je initialiseren:

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

Een constante mag je (vanzelfsprekend) niet veranderen:

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

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

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

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

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

1.4.1 `const *`

```
int i(3);
int j(4);
const int* p(&i);
```
[5]

Dit betekent: `p` wijst naar `i` en je kan `i` niet via `p` wijzigen[6]. Let op: je kan `i` zelf wel wijzigen!

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

1.4.2 `* const`

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

Dit betekent: `q` wijst naar `i` en je kan `q` nergens anders meer naar laten wijzen[7]. Let op: je kan `i` wel via `q` (of rechtstreeks) wijzigen.

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

1.4.3 `const * const`

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

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

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

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

C++ kent in tegenstelling tot C een apart type voor booleanse variabelen het type `bool`. Een variabele van dit type heeft slechts twee mogelijke waarden: `true` en `false`. In C (en ook in oudere versies van C++) wordt voor een booleanse variabele een `int` gebruikt met daarbij de afspraak dat een waarde `0` (nul) false en een waarde ongelijk aan nul true betekent. Om C++ zoveel mogelijk compatibel te houden met C wordt een `bool` indien nodig omgezet naar een `int` en vice versa. Het gebruik van het type `bool` maakt meteen duidelijk dat het om een booleanse variabele gaat.

---

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

[6] Dit is zinvol als je een pointer als parameter aan een functie wilt meegeven en als je niet wilt dat de variabele waar deze pointer naar wijst in de functie gewijzigd wordt. Je gebruikt dan als parameter een `const T*` in plaats van een `T*`. Dat de functie de variabele waar de pointer naar wijst niet mag wijzigen zouden we natuurlijk ook gewoon kunnen afspreken (en bijvoorbeeld in het commentaar van de functie vermelden) maar door deze afspraak expliciet als een `const` declaratie vast te leggen kan deze afspraak door de compiler gecontroleerd worden. Dit vermindert de kans op het maken van fouten bij het implementeren of wijzigen van de functie.

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

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

De ISO/ANSI standaard bevat ook een uitgebreide library. Als we de functies, types enz. uit deze standaard library willen gebruiken dan moeten we de definities van de betreffende functies, types, enz. in ons programma opnemen door de juiste header file te includen[8]. De C++ standaard header files eindigen niet zoals bij C op `.h` maar hebben helemaal geen extensie. Dus `#include <complex>` in plaatst van `#include <complex.h>`[9] [10]. Dit heeft als voordeel dat oude (deel)programma's met oude headerfiles nog steeds met de nieuwe ANSI/ISO C++ compiler vertaald kunnen worden. Bij het schrijven van nieuwe programma's gebruiken we uiteraard de nieuwe include files.

Om het mogelijk te maken dat de standaard library met andere (oude en nieuwe) libraries in één programma kan worden gebruikt zijn in ANSI/ISO C++ namespaces opgenomen. Dit wordt behandeld bij SOPX3 in EH3C&D of een volgend onderwijsdeel bij TI. Voor nu is het voldoende om te weten dat je in elk programma dat gebruik maakt van de ANSI/ISO C++ libray de volgende regel moet opnemen:

```
using namespace std;
```
[11]

De standaard C++ library bevat ook alle functies, types enz. uit de standaard C library. De headerfiles die afkomstig zijn uit de “oude” C library beginnen in de C++ library allemaal met de letter `c`. Dus `#include <cmath>` in plaats van `#include <math.h>`.

In een volgend onderwijsdeel komen we uitgebreid op het gebruik van de C++ standaard library terug.

1.7 I/O met << en >>. (Zie eventueel TICPP Chapter02.html#Heading90)

Je bent gewend om in C programma's de functies uit de `stdio` bibliotheek te gebruiken voor input en output. De meest gebruikte functies zijn `printf` en `scanf`. Deze functies zijn echter niet "type veilig" omdat de inhoud van de als eerste argument meegegeven format string pas tijdens het uitvoeren van het programma verwerkt wordt. De compiler merkt het dus niet als de "type aanduidingen" zoals `%d` die in de format string gebruikt zijn niet overeenkomen met de typen van de volgende argumenten. Tevens is het niet mogelijk om een eigen format type aanduiding aan de bestaande toe te voegen. Om deze redenen is in de ISO/ANSI C++ standaard naast de oude `stdio` bibliotheek (om compatibel te blijven) ook een nieuwe I/O library `iostream` opgenomen. Bij het ontwerpen van nieuwe software kun je het best van deze nieuwe library gebruik maken. De belangrijkste output faciliteiten van deze library zijn de standaard output stream `cout` (vergelijkbaar met `stdout`) en de bijbehorende `<<` operator. De belangrijkste input faciliteiten van deze library zijn de standaard input stream `cin` (vergelijkbaar met `stdin`) en de bijbehorende `>>` operator.

---

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

[9] Omdat de C++ taal pas in september 1998 door ANSI/ISO is gestandaardiseerd worden bij de meeste C++ compilers ook “standaard” headerfiles meegeleverd die nog gewoon de `.h` extensie gebruiken.

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

[11] Het gebruik van `using namespace std;` heeft ook een nadeel als je meerdere libraries in één programma wilt gebruiken. We komen daar bij een volgend onderwijsdeel nog op terug.

Voorbeeld met `stdio`:

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

Dit programmadeel bevat 2 fouten die niet door de compiler gesignaleerd worden en pas bij executie blijken.

Hetzelfde voorbeeld met `iostream`:

```
#include <iostream>
using namespace std;
// ...
double d;
cin>>d;               // lees d in vanaf het toetsenbord
cout<<"d="<<d<<endl;  // druk d af op het scherm en ga naar het
                      // begin van de volgende regel
```

De `iostream` library is zeer uitgebreid. In dit dictaat zullen we daar niet verder op ingaan. Zie voor verdere informatie hoofdstuk 10 en 14 van het TICPP boek of de bij de `iostream` library behorende help files.

1.8 Het type `string`. (Zie eventueel TICPP Chapter02.html#Heading94.)

In C (en ook in pre ANSI/ISO C++) werden character strings opgeslagen in variabelen van het type `char[]` (character array). De afspraak is dan dat het einde van de character string aangegeven wordt door een null character `'\0'`. Vaak werden deze variabelen dan aan functies doorgegeven door middel van character pointers (`char*`). Deze manier van het opslaan van character strings heeft vele nadelen (waarschijnlijk heb je zelf meerdere malen programma's zien vastlopen door het verkeerd gebruik van deze character strings). In de ANSI/ISO standaard library is een nieuw type `string` opgenomen waarin character strings op een veilige manier opgeslagen kunnen worden. Het bewerken van deze `string`s is ook veel eenvoudiger dan strings opgeslagen in chararacter array's. Het type `string` komt in de eerste practicumopdracht uitgebreid aan de orde.

Bijvoorbeeld het vergelijken van "oude" strings:

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

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

```
#include <iostream>
#include <string>
using namespace std;
// ...
    string str;
    cin>>str;
    if (str=="Hallo") {
```

```
        // invoer is Hallo
    }
```

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

In de ANSI/ISO C++ standaard library is ook het type `vector` opgenomen. Dit type is bedoeld om het oude type array `[]` te vervangen. Bij het onderwijsdeel SOPX3E1C1 in EH3C&D of PROGMI1T3 in IH1 komen we hier uitgebreid op terug.

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

In C mag elke functienaam maar 1 keer gedefinieerd worden. Dit betekent dat drie functies om de absolute waarde te bepalen van variabelen van de types `int`, `double` en `complex` (zelf gemaakt) allemaal een verschillende naam moeten hebben. In C++ mag een functienaam meerdere keren gedefinieerd worden (function name overloading). De compiler zal aan de hand van de gebruikte argumenten de juiste functie selecteren. Dit maakt deze functies eenvoudiger te gebruiken omdat de gebruiker (de programmeur die deze functies aanroept) slechts 1 naam hoeft te onthouden.[12] Dit is vergelijkbaar met het gebruik van ingebouwde operator + die zowel gebruikt kan worden om integers als om floating point getallen op te tellen.

Voorbeeld zonder function name overloading:

```
int abs_int(int i) {
    if (i<0) return -i; else return i;
}
double abs_double(double f) {
    if (f<0) return -f; else return f;
}
struct Complex {      /* Een Complex getal bestaat uit: */
    double[13] r;     /*    een reeel deel en           */
    double i;         /*    een imaginair deel.         */
};
double abs_complex(struct Complex c) {
    return sqrt(c.r*c.r+c.i*c.i)
}
```

Hetzelfde voorbeeld met function name overloading:

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

---

[12] Je kan echter ook zeggen dat overloading het analyseren een programma moeilijker maakt omdat nu niet meer meteen duidelijk is welke functie aangeroepen wordt. Om deze reden is het niet verstandig het gebruik van overloading te overdrijven.

[13] In de propedeuse heb je vaak gebruik gemaakt van het type `float` om floating point getallen op te slaan. Dit type is echter niet erg nauwkeurig (8 significante cijfers). Het type `double` gebruikt twee keer zoveel bits als een `float` (vandaar de naam) en is dus veel nauwkeuriger (17 significante cijfers). Omdat de huidige processoren net zo snel kunnen rekenenen met het type `double` als met het type `float` wordt in de meeste programma's het type `double` gebruikt. Als je grote hoeveelheden (niet zo nauwkeurige) floating point getallen moet opslaan kun je echter beter het type `float` gebruiken omdat je dan minder geheugenruimte nodig hebt.

```
struct Complex {      /* Een Complex getal bestaat uit: */
    double r;         /*    een reeel deel en           */
    double i;         /*    een imaginair deel.         */
};
double abs(struct Complex c) {
    return sqrt(c.r*c.r+c.i*c.i)
}
```

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

```
double in;
cin>>in;              // lees in
cout<<abs(in)<<endl;  // druk de absolute waarde van in af
```

De compiler bepaalt nu zelf aan de hand van het type van de gebruikte parameter (`in`) welk van de drie bovenstaande `abs` functies aangeroepen wordt.

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

Het is in C++ mogelijk om voor de laatste parameters van een functie default waarden te definiëren.

Voorbeeld van het gebruik van default parameters:

```
void print(int i, int talstelsel=10) {
    // ...
}
// ...
print(5, 2);          // uitvoer: 101
print(5);             // uitvoer: 5
```

De functie print heeft twee parameters. De eerste parameter wordt afgedrukt in het als tweede parameter opgegeven talstelsel. Als de functie met maar één parameter wordt aangeroepen wordt als tweede parameter 10 gebruikt zodat het getal in het decimale talstelsel wordt afgedrukt.

1.12 Naam van struct.

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

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

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

```
struct Tijdsduur td1;
```

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

```
typedef struct Tijdsduur TTijdsduur;
```

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

```
TTijdsduur td2;
```

In C++ is de naam van een struct meteen een typenaam. Je kunt variabelen van het type `struct Tijdsduur` dus eenvoudig als volgt definiëren:

```
Tijdsduur td3;
```

In hoofdstuk 3 (blz. 19) zul je zien dat C++ het mogelijk maakt om op een veel betere manier een tijdsduur te definiëren (als abstract data type).

1.13 C++ als een betere C.

Als we de verbeteringen die in C zijn doorgevoerd bij de definitie van C++ toepassen in het voorbeeld programma van blz. 7 dan ontstaat het volgende C++ programma:

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

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

// Deze functie drukt een Tijdsduur af
void drukaf(Tijdsduur td) {
    if (td.uur==0)
        cout<<"           ";
    else
        cout<<setw(3)<<td.uur<<" uur en ";
    cout<<setw(2)<<td.min<<" minuten"<<endl;
}

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

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

int main () {
    ifstream fin("lijst.txt");
    if (fin!=0) {
        const int MAX(100);
        Tijdsduur rij[MAX];
        int aantal(0);
        while (fin>>rij[aantal].uur>>rij[aantal].min && aantal<MAX)
```

```
            ++aantal;
        if (fin!=0)
                cout<<"De file is niet volledig uitgelezen!\n"<<endl;
        drukaf(rij, aantal);
        cout<<"De totaal gewerkte tijd is:"<<endl;
        drukaf(som(rij, aantal));
    }
    else
        cout<<"Het bestand lijst.txt kan niet worden geopend!"<<endl;
    cin.get();
    return 0;
}
```

Verklaring van de verschillen:

- Hier is gebruik gemaakt van de standaard I/O library van C++ in plaats van de standaard C I/O library. In de eerste regel wordt de file `iostream` “included”. Dit is nodig om gebruik te kunnen maken van functies, objecten en typen die in de standaard C++ I/O library zijn opgenomen. In dit programma maak ik gebruik van `cout` (om te schrijven naar het scherm). Om gebruik te kunnen maken van een zogenaamde I/O manipulator wordt in de tweede regel de file `iomanip` "included". In dit programma maak ik gebruik van de manipulator `setw()` waarmee de breedte van een outputveld gespecificeerd kan worden. Om gebruik te kunnen maken van de file I/O functies uit de standaard C++ I/O library wordt in de tweede regel de file `fstream` “included”. In dit programma maak ik gebruik van `ifstream` (om te lezen uit een file[14]). In de vierde regel wordt aangegeven dat de namespace `std` wordt gebuikt. Dit is nodig om de functies, types enz die in de bovenstaande 3 include files zijn gedefinieerd te kunnen gebruiken.
- Voor commentaar aan het einde van de regel is hier `//...` gebruikt in plaats van `/*...*/`.
- Als typenaam voor de parameter `td` van de functie `drukaf` is hier `Tijdsduur` in plaats van `struct Tijdsuur` gebruikt. Dit zelfde geldt voor de parameter `trij` in de twee volgende functies.
- Om de datavelden van een tijdsduur af te drukken is hier gebruik gemaakt van `cout` in plaats van `printf`. Merk op dat nu het type van de datavelden **niet** opgegeven hoeft te worden. Bij het gebruik van `printf` werd het type van de datavelden in de format string met `%d` aangegeven. De breedte van het uitvoerveld wordt nu met de I/O manipulator `setw` opgegeven in plaats van in de format string van `printf`.
- De functie om een rij af te drukken heeft hier de naam `drukaf` in plaats van `drukafRij`. De functie om een tijdsduur af te drukken heet ook al `drukaf`, maar in C++ is dit geen probleem. Ik heb hier dus function name overloading toegepast.
- De lokale variabele `teller` is hier in het `for` statement gedefinieerd in plaats van op een aparte regel.
- De lokale variabele `teller` is hier geïnitialiseerd met de notatie `(0)` in plaats van `=0`.
- Hier is een `ifstream` gebruikt in plaats van een `FILE*`.
- De constante `MAX` is hier als `const int` gedefinieerd in plaats van met `#define`.
- De lokale variabele `aantal` is hier geïnitialiseerd met de notatie `(0)` in plaats van `=0`.
- Het inlezen uit de file en het testen of het einde van de file bereikt is gaat veel eenvoudiger met de variabele `fin` van het type `ifstream` dan met de variabele `fp` van het type `FILE*`. Hier zijn geen speciale functies zoals `fscanf` en `feof` nodig.

Merk op dat dit programma geen files met meer dan `MAX` gewerkte tijden kan verwerken. Bij het onderwijsdeel SOPX3E1C1 in EH3C&D of PROGMI1T3 in IH1 kun je leren hoe je dit door gebruik te maken van dynamisch geheugentoewijzing kunt verbeteren.

Merk op dat de vernieuwingen die in C++ zijn ingevoerd ten opzichte van C, namelijk het gebruik van abstracte data typen en object georiënteerde technieken, in dit programma nog **niet** toegepast zijn. In dit

[14] Zie eventueel TICPP Chapter02.html#Heading95.

programma wordt C++ dus op een C manier gebruikt. Dit is voor kleine programma's geen probleem. Als een programma echter groter is of als het uitbreidbaar of onderhoudbaar moet zijn kun je beter gebruik maken van de object georiënteerde technieken die C++ biedt. Deze technieken zullen vanaf hoofdstuk 3 van dit dictaat uitgebreid besproken worden.

2 Van C naar C++. Dynamic memory allocation en references.

In dit hoofdstuk worden enkele belangrijke wijzigingen en toevoegingen van C++ ten opzichte van C besproken. Er wordt echter nog niet ingegaan op de meest belangrijke toevoegingen: data abstractie, object oriëntatie en generiek programmeren. De volgende onderwerpen worden in dit hoofdstuk besproken:

- Dynamic memory allocation. (Zie eventueel TICPP Chapter04.html#Heading190 en Chapter13.html.)
- Reference variables. (Zie eventueel TICPP Chapter03.html#Heading123 en Chapter11.html#Heading326.)
- Parameter passing mechanisms. (Zie eventueel TICPP Chapter08.html#Heading260, Chapter08.html#Heading263 en Chapter11.html#Heading330.)

2.1 Dynamische geheugen allocatie (`new` en `delete`).

Je bent gewend om variabelen globaal of lokaal te definiëren. De geheugenruimte voor globale variabelen wordt gereserveerd zodra het programma start en pas weer vrijgegeven bij het beëindigen van het programma. De geheugenruimte voor een lokale (zogenaamde automatic) variabele wordt, zodra die variabele gedefinieerd wordt, gereserveerd op de stack. Als het blok (compound statement), waarin de variabele gedefinieerd is, wordt beëindigd dan wordt de gereserveerde ruimte weer vrijgegeven. Vaak wil je zelf bepalen wanneer ruimte voor een variabele gereserveerd wordt en wanneer deze ruimte weer vrijgegeven wordt. In een programma met een GUI (grafische gebruikers interface) wil je bijvoorbeeld een variabele (struct) aanmaken voor elk window dat de gebruiker opent. Deze variabele kan weer worden vrijgegeven zodra de gebruiker dit window sluit. Het geheugen dat bij het openen van het window was gereserveerd kan dan (her)gebruikt worden bij het openen van een (ander) window.

Dit kan in C++ met de operatoren `new` en `delete`. Voor het dynamisch aanmaken en verwijderen van array's (waarvan de grootte dus tijdens het uitvoeren van het programma bepaald kan worden) beschikt C++ over de operatoren `new[]` en `delete[]` (zie eventueel TICPP Chapter13.html#Heading393). De operatoren `new` en `new[]` geven een pointer naar het nieuw gereserveerde geheugen terug. Deze pointer kan dan gebruikt worden om dit geheugen te gebruiken. Als dit geheugen niet meer nodig is dan kan dit worden vrijgegeven door de operator `delete` of `delete[]` uit te voeren op de pointer die bij `new` of respectievelijk `new[]` is teruggegeven. De met `new` aangemaakte variabelen bevinden zich in een speciaal geheugengebied "heap" genaamd. Tegenover het voordeel van dynamische geheugenallocatie, een grotere flexibiliteit, staat het gevaar van een geheugenlek (memory leak). Een geheugenlek ontstaat als een programmeur vergeet een met `new` aangemaakte variabele weer met `delete` te verwijderen.

In C werden de functies `malloc` en `free` gebruikt om geheugenruimte op de "heap" te reserveren en weer vrij te geven. Deze functies zijn echter niet "type veilig" omdat het return type van `malloc` een `void*` is die vervolgens door de gebruiker naar het gewenste type moet worden omgezet. De compiler merkt het dus niet als de gebruikte "type aanduidingen" niet overeenkomen. Om deze reden zijn in C++ nieuwe memory allocatie operatoren (`new` en `delete`) toegevoegd. Bij het ontwerpen van nieuwe software kan je het best van deze nieuwe operatoren gebruik maken.

Voorbeeld met `new` en `delete`:

```
double* dp(new double);       // reserveer een double
int i; cin>>i;
double* drij(new double[i]);  // reserveer een array met i doubles
// ...
```

```
delete dp;       // geef de door dp aangewezen geheugenruimte vrij
delete[] drij;   // idem voor de door drij aangewezen array
```

In paragraaf 3.25 zul je zien hoe, door het gebruik van dynamische geheugenallocatie in plaats van het gebruik van een statische array, geen grens gesteld hoeft te worden aan het aantal elementen in een array. De enige grens is dan de grootte van het beschikbare (virtuele) werkgeheugen.

2.2 Reference variabelen. (Zie eventueel TICPP Chapter11.html#Heading326.)

Een *reference* is niets anders dan een alias (andere naam) voor de variabele waarnaar hij refereert. Alle operaties die op een variabele zijn toegestaan zijn ook toegestaan op een reference die naar die variabele verwijst[15].

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

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

In C worden bij het aanroepen van een functie de parameters gekopieerd. Dit betekent dat de parameters die in een functieaanroep zijn gebruikt na afloop van de functie **niet** veranderd kunnen zijn[16]. Je kunt de parameters in de functiedefinitie dus gewoon als een lokale variabele gebruiken. Een parameter die in een definitie van een functie gebruikt wordt, wordt formele parameter genoemd. Een parameter die bij een aanroep van een functie gebruikt wordt, wordt actuele parameter genoemd. Bij een functieaanroep wordt dus de **waarde** van de actuele parameter naar de formele parameter gekopieerd. Het is dus geen probleem als de actuele parameter een constante is.

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

[15] Een reference lijkt een beetje op een pointer maar er zijn toch belangrijke verschillen:
- Als je de variabele waar een pointer `p` naar wijst wilt benaderen moet je de notatie `*p` gebruiken maar als je de variabele waar een reference `r` naar refereert wilt benaderen kun je gewoon `r` gebruiken.
- Een pointer die naar de variabele `v` wijst kun je later naar een andere variabele laten wijzen maar een reference die naar de variabele `v` refereert kun je later niet naar een andere variabele laten refereren.
- Een pointer kan ook naar geen enkele variabele wijzen (de waarde is dan `0`) maar een reference verwijst altijd naar een variabele.

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

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

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

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

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

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

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

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

De C++ Builder compiler geeft de volgende foutmeldingen:

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

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

Er is nog een andere reden om een reference parameter in plaats van een “normale” value parameter te gebruiken. Bij een reference parameter wordt zoals we hebben gezien een adres (op de meeste systemen 4 bytes) gekopieerd terwijl bij een value parameter de waarde (aantal bytes afhankelijk van het type) gekopieerd wordt. Als de waarde veel geheugenruimte in beslag neemt dan is het om performance redenen beter om een reference parameter te gebruiken.[17] Om er voor te zorgen dat deze functie toch aangeroepen kan worden met een constante (zonder dat er een tijdelijke kopie wordt gemaakt) kan deze parameter dan het beste als const reference parameter gedefinieerd worden. Het wordt dan ook onmogelijk om in de functie een waarde aan de formele parameter toe te kennen.

```
void drukaf(Tijdsduur td) {           // kopieer een Tijdsduur
    // ...
```

[17] Later zullen we zien dat er nog een veel belangrijke reden is om een reference parameter in plaats van een gewone parameter te gebruiken. (Zie polymorphism blz. 62.)

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

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

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

2.5 Parameter FAQ.[18]

Q: Wanneer gebruik je een `T&` parameter in plaats van een `T` parameter?
A: Gebruik een `T&` als je wilt dat een toekenning aan de formele parameter (de parameter die gebruikt wordt in de functiedefinitie) ook een verandering van de actuele parameter (de parameter die gebruikt wordt in de functieaanroep) tot gevolg heeft.

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

Q: Wanneer gebruik je een `T*` parameter in plaats van een `T&` parameter?
A: Gebruik een `T*` als je ook “niets” door wilt kunnen geven. Een reference **moet** namelijk altijd ergens naar verwijzen. Een pointer kan ook nergens naar wijzen (de waarde van de pointer is dan `0`).

2.6 Reference return type.

Als een functie een waarde teruggeeft door middel van een `return` statement dan wordt de waarde van de expressie achter het `return` statement gekopieerd naar de plaats waar de functie aangeroepen is. Door nu een reference als return type te gebruiken kun je een adres in plaats van een waarde teruggeven. Dit adres kun je dan gebruiken om een waarde aan toe te kennen.

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

int main() {
    int x(3), y(4);
    max(x,y)=0;          // y is nu 0
    max(x,y)++;          // x is nu 4
// ...
```

Pas op dat je geen reference naar een lokale variabele teruggeeft. Na afloop van een functie worden de lokale variabelen uit het geheugen verwijderd. De referentie verwijst dan naar een niet meer bestaande variabele.

```
Tijdsduur& som(const Tijdsduur& td1, const Tijdsduur& td2) {
    Tijdsduur td;
```

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

```
    td.uur=td1.uur+td2.uur;
    td.min=td1.min+td2.min;
    td.uur+=td.min/60;
    td.min%=60;
    return td;           // Een gemene fout!
}
// ...
c=som(a, b);
```

Deze fout (het zogenaamde “dangling reference problem”) is erg gemeen omdat de lokale variabele `td` natuurlijk niet echt uit het geheugen verwijderd wordt. De geheugenplaats wordt alleen vrijgegeven voor hergebruik. Dit heeft tot gevolg dat de bovenstaande aanroep van `som` meestal gewoon werkt[19]. De fout in de definitie van de functie `som` komt pas aan het licht als we de functie bijvoorbeeld als volgt aanroepen:

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

Hetzelfde gevaar is trouwens ook aanwezig als je een pointer (bijvoorbeeld `Tijdsduur*`) als return type kiest (het zogenaamde “dangling pointer problem”). Je moet er dan voor oppassen dat de pointer niet naar een variabele wijst die na afloop van de functie niet meer bestaat[20].

3 Objects and classes.

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

3.1 Object Oriented Design (OOD) en Object Oriented Programming (OOP).

In deze paragraaf zullen we eerst ingaan op de achterliggende gedachten van OOD en OOP en pas daarna de implementatie in C++ bespreken.

Een van de specifieke problemen bij het ontwerpen van software is dat het nooit echt af is. Door het gebruik van de software ontstaan bij de gebruikers weer ideeën voor uitbreidingen en/of veranderingen. Ook de veranderende omgeving van de software (bijvoorbeeld: operating system en hardware) zorgen ervoor dat software vaak uitgebreid en/of veranderd moet worden. Dit aanpassen van bestaande software wordt *software maintenance* genoemd. Er werken momenteel meer software engineers aan het onderhouden van bestaande software dan aan het ontwikkelen van nieuwe applicaties. Ook is het bij het ontwerpen van (grote) applicaties gebruikelijk dat de klant zijn specificaties halverwege het project aanpast. Je moet er bij het ontwerpen van software dus al op bedacht zijn dat deze software eenvoudig aangepast en uitgebreid kan worden (design for change).

Bij het ontwerpen van hardware is het vanzelfsprekend om gebruik te maken van (vaak zeer complexe) *componenten*. Deze componenten zijn door anderen geproduceerd en je moet er dus voor betalen, maar dat is geen probleem want je kunt het zelf niet (of niet goedkoper) maken. Bij het gebruik zijn alleen de specificaties van de component van belang, de interne werking (implementatie) van de component is voor de gebruiker van de component niet van belang. Het opbouwen van hardware met bestaande componenten maakt het ontwerpen en testen eenvoudiger en goedkoper. Als bovendien voor standaard interfaces wordt gekozen kan de hardware ook aanpasbaar en uitbreidbaar zijn (denk bijvoorbeeld aan een PC met PCI bus). Bij het ontwerpen van software is het niet gebruikelijk om gebruik te maken van

[19] De C++ Builder 6 compiler herkent het dangling reference probleem en geeft de volgende foutmelding: “Attempting to return a reference to local variable '`td`'”.

[20] Vreemd genoeg herkent de C++ Builder 6 compiler het dangling pointer probleem niet. Wel wordt de volgende warning gegeven: “Suspicious pointer conversion”.

componenten die door anderen ontwikkeld zijn (en waarvoor je dus moet betalen). Vaak worden wel bestaande datastructuren en algoritmen gekopieerd maar die moeten vaak toch aangepast worden. Het gebruik van functie libraries is wel gebruikelijk maar dit zijn in feite eenvoudige componenten. Een voorbeeld van een complexe component zou bijvoorbeeld een editor (inclusief menu opties: openen, opslaan, opslaan als, print, printer set-up, bewerken (knippen, kopiëren, plakken), zoeken en zoek&vervang en inclusief een knoppenbalk) kunnen zijn. In alle applicaties waarbij je tekst moet kunnen invoeren kun je deze editor dan hergebruiken. Er zijn geen databoeken vol met software componenten die we kunnen kopen en waarmee we vervolgens zelf applicaties kunnen ontwikkelen. Dit heeft volgens mij verschillende redenen:

- Voor een hardware component moet je betalen. Bij een software component vindt men dit niet normaal. Zo hoor je vaak zeggen: “Maar dat algoritme staat toch gewoon in het boek van ... dus dat kunnen we toch zelf wel implementeren.”
- Als je van een bepaalde hardware component 1000 stuks gebruikt dan moet je er ook 1000 maal voor betalen. Bij een software component vindt men dit niet normaal en er valt eenvoudiger mee te sjoemelen.
- Programmeurs denken vaak: maar dat kan ik zelf toch veel eenvoudiger en/of sneller programmeren.
- De omgeving (taal, operating system, hardware enz.) van software is divers. Dit maakt het produceren van software componenten niet eenvoudig. Heeft u deze component ook in C++ voor Linux in plaats van in Delphi voor Windows?[21]

De object georiënteerde benadering is vooral bedoeld om het ontwikkelen van herbruikbare software-componenten mogelijk te maken. In deze onderwijseenheid zul je kennismaken met deze object georiënteerde benadering. Wij hebben daarbij gekozen voor de taal C++ omdat dit één van de meest gebruikte object georiënteerde programmeertalen is. Het is echter niet de bedoeling dat je na deze onderwijseenheid op de hoogte bent van alle aspecten en details van C++. Wel zul je na afloop van deze onderwijseenheid de algemene ideeën achter OOP begrijpen en die toe kunnen passen in C++. De object georiënteerde benadering is een (voor jou) nieuwe manier van denken over hoe je informatie in een computer kunt structureren.

De manier waarop we problemen oplossen met object georiënteerde technieken lijkt op de manier van probleem oplossen die we in het dagelijks leven gebruiken. Een object georiënteerde applicatie is opgebouwd uit deeltjes die we *objecten* noemen. Elk object heeft zijn eigen taak. Een object kan aan een ander object vragen of dit object wat voor hem wil doen, dit gebeurt door het zenden van een “*message*” aan dat object. Aan deze message kunnen indien nodig *argumenten* worden meegegeven. Het is de verantwoordelijkheid van het ontvangende object (de “*receiver*”) hoe het de message afhandelt. Zo kan dit object de verantwoordelijkheid afschuiven door zelf ook weer een message te versturen naar een ander object. Het algoritme dat de ontvanger gebruikt om de message af te handelen wordt de “*method*” genoemd. Het object dat de message verstuurt is dus niet op de hoogte van de door de ontvanger gebruikte method. Dit wordt “*information hiding*” genoemd. Het versturen van een message lijkt in eerste instantie op het aanroepen van een functie. Toch zijn er enkele belangrijke verschillen:

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

In een programma kunnen zich meerdere objecten van een bepaald object type bevinden, vergelijkbaar met meerdere variabelen van een bepaald variabele type. Zo’n object type wordt “*class*” genoemd. Elk

[21] Er zijn de laatste jaren diverse alternatieven ontwikkeld voor het maken van taalonafhankelijke componenten. Microsoft heeft de COM (Component Object Model) architectuur ontwikkeld die het ontwikkelen van zogenaamde ActiveX componenten mogelijk maakt. Deze componenten kunnen in diverse talen gebruikt worden maar zijn wel gebonden aan het Windows platform. In de UNIX wereld is CORBA (Common Object Request Broker Architecture) ontwikkeld die het ontwikkelen van componenten mogelijk maakt.

object behoort tot een bepaalde class, een object wordt ook wel een "*instance*" van de class genoemd. Bij het definiëren van een nieuwe class hoef je niet vanuit het niets te beginnen maar je kunt deze nieuwe class afleiden (laten overerven) van een al bestaande class. De class waarvan afgeleid wordt, wordt de "*base class*" genoemd en de class die daarvan afgeleid wordt, wordt "*derived class*" genoemd. Als van een afgeleide class weer nieuwe classes worden afgeleid ontstaat een hiërarchisch netwerk van classes. Een derived class overerft alle eigenschappen van een base class (overerving = "*inheritance*"). Als een object een message ontvangt wordt de bijbehorende method als volgt bepaald ("*method binding*"):

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

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

De bij object georiënteerd programmeren gebruikte technieken komen niet uit de lucht vallen maar sluiten aan bij de historische evolutie van programmeertalen. Sinds de introductie van de computer zijn programmeertalen steeds *abstracter* geworden. De volgende abstractie technieken zijn achtereenvolgens ontstaan:

- **Functies en procedures.**
  In talen zoals Basic, Fortran, C, Pascal, Cobol enz. kan een stukjes logisch bij elkaar behorende code geabstraheerd worden tot een functie of procedure. Deze functies of procedures kunnen lokale variabelen bevatten die buiten de functie of procedure niet zichtbaar zijn. Deze lokale variabelen zitten ingekapseld in de functie of procedure, dit wordt "*encapsulation*" genoemd. Als de specificatie van de functie bekend is kun je de functie gebruiken zonder dat je de implementatie details hoeft te kennen of te begrijpen. Dit is dus een vorm van information hiding. Tevens voorkomt het gebruik van functies en procedures het dupliceren van code, waardoor het wijzigen en testen van programma's eenvoudiger wordt. De programmacode wordt korter, maar dit gaat ten koste van de executietijd.
- **Modules.**
  In programmeertalen zoals Modula 2 (en zeer primitief ook in C) kunnen een aantal logisch bij elkaar behorende functies, procedures en variabelen geabstraheerd worden tot een module. Deze functies, procedures en variabelen kunnen voor de gebruikers van de module zichtbaar (=public) of onzichtbaar (=private) gemaakt worden. Functies, procedures en variabelen die private zijn kunnen alleen door functies en procedures van de module zelf gebruikt worden. Deze private functies, procedures en variabelen zitten ingekapseld in de module. Als de specificatie van de public delen van de module bekend is kun je de module gebruiken zonder dat je de implementatie details hoeft te kennen of te begrijpen. Dit is dus een vorm van data en information hiding.
- **Abstract data types (ADT's).**
  In programmeertalen zoals Ada kan een bepaalde datastructuur met de bij deze datastructuur behorende functies en procedures ingekapseld worden in een ADT. Het verschil tussen een module en een ADT is dat een module alleen gebruikt kan worden en dat een ADT geïnstantieerd kan

worden. Dit wil zeggen dat er meerdere variabelen van dit ADT (=type) aangemaakt kunnen worden. Als de specificatie van het ADT bekend is kun je dit type op dezelfde wijze gebruiken als de ingebouwde typen van de programmeertaal (zoals `char`, `int` en `double`) zonder dat je de implementatie details van dit type hoeft te kennen of te begrijpen. Dit is dus weer een vorm van information hiding. Op het begrip ADT komen we nog uitgebreid terug.

- **Generieke functies en data types.**
  In C++ kunnen door het gebruik van *templates* generieke functies en data types gedefinieerd worden. Een generieke functie is een functie die gebruikt kan worden voor meerdere parameter types. In C kan om een array met `int`'s te sorteren een functie geschreven worden. Als we vervolgens een array met `double`'s willen sorteren moeten we een nieuwe functie definiëren. In C++ kunnen we met behulp van templates een *generieke functie* definiëren waarmee array's van elk willekeurig type gesorteerd kunnen worden. Een generiek type is een ADT dat met verschillende andere types gecombineerd kan worden. In C++ kunnen we bijvoorbeeld een ADT array definieren waarin we variabelen van het type `int` kunnen opslaan. Door het gebruik van een template kunnen we een generiek ADT array definiëren waarmee we dan array's van elk willekeurig type kunnen gebruiken. Op generieke functies en data types komen we nog uitgebreid terug.
- **Classes.**
  In programmeertalen zoals SmallTalk en C++ kunnen een aantal logisch bij elkaar behorende ADT's van elkaar worden afgeleid door middel van inheritance. Door nu programma's te schrijven in termen van base classes en de in deze base classes gedefinieerde messages in afgeleide (Engels: derived) classes te implementeren kan je deze base classes gebruiken zonder dat je de implementatie van deze classes hoeft te kennen of te begrijpen en kun je eenvoudig van implementatie wisselen. Tevens kun je code die alleen messages aanroept van de base class zonder opnieuw te compileren hergebruiken voor alle direct of indirect van deze base class afgeleide classes. Dit laatste wordt mogelijk gemaakt door de message pas tijdens run time aan een bepaalde method te verbinden en wordt "*polymorphism*" genoemd. Op deze begrippen komen we later nog uitgebreid terug.

Tijdens de propedeuse en in het H1 project heb je geleerd hoe je vanuit een probleemstelling voor een programma de functies en procedures kan vinden. Deze methode bestond uit het opdelen van het probleem in deelproblemen, die op hun beurt weer werden opgedeeld in deelproblemen enz. net zo lang tot de oplossing eenvoudig werd. Deze methode heet functionele decompositie. De software ontwikkelmethoden die hiervan zijn afgeleid worden "structured analyse and design" genoemd. Bij het gebruik van object georiënteerde programmeertalen ontstaat al snel de vraag: "Hoe vind ik uitgaande van de probleemstelling de in dit programma benodigde classes en het verband tussen die classes?" Deze vraag wordt in deze onderwijseenheid niet beantwoord en is afhankelijk van de gebruikte OOA (Object Oriented Analyse) en OOD (Object Oriented Design) methode. Bij C&D komen deze onderwerpen in semester H3 bij SOPX3 aan de orde. Bij TI komen deze onderwerpen bij verschillende onderwijsdelen in H2 aan de orde.

3.2 ADT's (Abstract Data Types).

De eerste stap op weg naar het object georiënteerde denken en programmeren is het leren programmeren met ADT's. Een ADT[22] is een user-defined type dat voor een gebruiker niet te onderscheiden is van ingebouwde types (zoals `int`). Een ADT koppelt een bepaalde datastructuur (interne variabelen) met de bij deze datastructuur behorende bewerkingen (functies). Deze functies en variabelen kunnen voor de gebruikers van het ADT zichtbaar (=public) of onzichtbaar (=private) gemaakt worden. Functies en variabelen die private zijn kunnen alleen door functies van het ADT zelf gebruikt worden. De private functies en variabelen zitten ingekapseld in het ADT. Als de specificatie van het ADT bekend is kun je dit type op dezelfde wijze gebruiken als de ingebouwde typen van de programmeertaal (zoals `char`, `int` en `double`) zonder dat je de implementatie details van dit type hoeft te kennen of te begrijpen. Dit is een vorm van information hiding.

[22] De naam ADT wordt ook vaak gebruikt voor een formele wiskundige beschrijving van een type. Ik gebruik het begrip ADT hier alleen in de betekenis van "user-defined type".

In C is het niet mogelijk om ADT's te definiëren. Als je in C een programma wilt schrijven waarbij je met breuken in plaats van met floating point getallen wilt werken[23], dan kun je dit (niet in C) opgenomen "type" `Breuk` alleen maar op de volgende manier definiëren:

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

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

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

Het normaliseren[24] van de breuk zorgt ervoor dat 3/8 + 1/8 als resultaat 1/2 in plaats van 4/8 heeft. Dit zorgt ervoor dat een overflow minder snel optreedt als met het resultaat weer verder wordt gerekend.

Deze manier van werken heeft de volgende nadelen:

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

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

In C++ kunnen we door gebruik te maken van een *class* een eigen type `Breuk` als volgt declareren:

```
class Breuk {             // Op een object van de class Breuk
public:                   // kun je de volgende bewerkingen uitvoeren:
    void leesin();                //  inlezen vanuit het toetsenbord.
```

[23] Een belangrijke reden om te werken met breuken in plaats van floating point getallen is het voorkomen van afrondingsproblemen. Een breuk zoals 1/3 moet als floating point getal afgerond worden tot bijvoorbeeld: 0.333333333. Deze afrondig kan ervoor zorgen dat een berekening zoals 3*(1/3) niet zoals verwacht de waarde 1 maar de waarde 0.999999999 oplevert. Ook breuken zoals 1/10 moeten afgerond worden als ze als binair floating point getal worden weergegeven.

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

```
    void drukaf() const[25];          //  afdrukken op het scherm.
    void plus(const Breuk& b);    //  een Breuk erbij optellen.
private:                    // Een object van de class Breuk heeft privé:
    int boven;                    //  een teller,
    int onder;                    //  een noemer en
    void normaliseer();           //  een functie normaliseer.
};
```

De class `Breuk` koppelt een bepaalde datastructuur (2 interne integer variabelen genaamd `boven` en `onder`) met de bij deze datastructuur behorende bewerkingen[26] (functies: `leesin`, `drukaf`, `plus` en `normaliseer`). Deze functies en variabelen kunnen voor de gebruikers van de class `Breuk` zichtbaar (`public`) of onzichtbaar (`private`) gemaakt worden. Functies en variabelen die private zijn kunnen alleen door functies van de class `Breuk` zelf gebruikt worden[27]. Deze private functies en variabelen zitten ingekapseld in de class `Breuk`.

Functies de opgenomen zijn in een class worden *memberfuncties* genoemd. De memberfunctie `plus` kan als volgt gedefinieerd worden[28]:

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

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

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

Een object heeft drie kenmerken:

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

---

[25] `const` memberfunctie wordt pas behandeld op blz. 35

[26] De verzameling van public functies wordt ook wel de *interface* van de class genoemd. Deze interface definieert hoe je objecten van deze class kunt gebruiken (welke memberfuncties je op de objecten van deze class kan aanroepen).

[27] Variabelen die private zijn kunnen door het gebruik van een pointer en bepaalde type-conversies toch door andere functies benaderd worden. Ze staan namelijk gewoon in het werkgeheugen en ze zijn dus altijd via software toegankelijk. Het gebruik van private variabelen is alleen bedoeld om “onbewust” verkeerd gebruik tegen te gaan.

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

- *Identiteit* (Engels: *identity*).
  Om objecten met dezelfde waarde toch uit elkaar te kunnen houden heeft elk object een eigen identiteit. De twee objecten van de class `Breuk` in het voorgaande voorbeeld hebben bijvoorbeeld elk een eigen naam (`a` en `b`) waarmee ze geïdentificeerd kunnen worden.

Bij de memberfunctie `plus` wordt bij de definitie met de zogenaamde qualifier `Breuk::` aangegeven dat deze functie een member is van de class `Breuk`. De memberfunctie `plus` kan alleen op een object van de class `Breuk` uitgevoerd worden. Dit wordt genoteerd als: `a.plus(b);` Het object `a` wordt de *receiver* (ontvanger) van de memberfunctie genoemd. Als in de definitie van de memberfunctie `plus` de datamembers `boven` en `onder` gebruikt worden dan zijn dit de datamembers van de receiver (in dit geval object `a`). Als in de definitie van de memberfunctie `plus` de memberfunctie `normaliseer` gebruikt wordt dan wordt deze memberfunctie op de receiver uitgevoerd (in dit geval object `a`). De betekenis van const achter de declaratie van de memberfunctie `drukaf` komt later (blz. 35) aan de orde.

Deze manier van werken heeft de volgende voordelen:

- Een programmeur die gebruik maakt van de class (het type) `Breuk` kan **géén** waarde toekennen aan de private datavelden `boven` en `onder`. Alleen de memberfuncties `leesin`, `drukaf`, `plus` en `normaliseer` kunnen een waarde toekennen aan deze datavelden. Als ergens een fout ontstaat omdat het dataveld `onder` van een `Breuk` op nul is gezet kunnen alleen de memberfuncties van `Breuk` de boosdoeners zijn.
- Een programmeur die gebruik maakt van de class `Breuk` kan er **niet** voor kiezen om zelf de code voor het optellen van breuken “uit te vinden” in plaats van gebruik te maken van de memberfunctie `plus`. Als we kunnen “garanderen” dat de functies `plus` en `normaliseer` correct zijn, dan kunnen we dus garanderen dat alle optellingen van breuken correct en genormaliseerd zullen zijn. Iedere programmeur die breuken gebruikt kan ze namelijk alleen optellen door gebruik te maken van de memberfunctie `plus`.
- Iedere programmeur die gebruikt maakt van de class `Breuk` zal **niet** zelf nieuwe bewerkingen (zoals bijvoorbeeld het vermenigvuldigen van breuken) definiëren. Alleen de programmeur die verantwoordelijk is voor het onderhouden van de class `Breuk` (en de bijbehorende bewerkingen) kan dit doen.

Bij het ontwikkelen van kleine programma’s zijn deze voordelen niet zo belangrijk maar bij het ontwikkelen van grote programma’s zijn deze voordelen wel erg belangrijk. Door gebruik te maken van de `class Breuk` met bijbehorende memberfuncties in plaats van de `struct Breuk` met de bijbehorende functies wordt het programma **beter onderhoudbaar** en **eenvoudiger uitbreidbaar**.

De hierboven gedefinieerde class `Breuk` is erg beperkt. In de inleiding van dit hoofdstuk heb ik geschreven: “*Een ADT is een user-defined type dat voor een gebruiker niet te onderscheiden is van ingebouwde types (zoals `int`).*” Een ADT `Breuk` zal dus als volgt gebruikt moeten kunnen worden:

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

Je ziet dat het zelf gedefinieerde type `Breuk` nu op precies dezelfde wijze als het ingebouwde type `int` te gebruiken is. Dit maakt het voor programmeurs die het type `Breuk` gebruiken erg eenvoudig. In het vervolg zal ik bespreken hoe de class `Breuk` stap voor stap uitgebreid en aangepast kan worden zodat

uiteindelijk een ADT `Breuk` ontstaat. Dit blijkt nog behoorlijk lastig te zijn en je zou jezelf af kunnen vragen: "*Is het al die inspanningen wel waard om een `Breuk` zo te maken dat je hem net zoals een `int` kan gebruiken?*" Bedenk dan het volgende: het maken van de class `Breuk` is dan wel een hele inspanning maar iedere programmeur kan vervolgens dit zelf gedefinieerde type, als hij of zij de naam `Breuk` maar kent, als vanzelf (intuïtief) gebruiken. Er is daarbij geen handleiding of helpfile nodig, omdat het type `Breuk` zich net zo gedraagt als het ingebouwde type `int`. De class `Breuk` hoeft maar één keer gemaakt te worden, maar zal talloze malen gebruikt worden. Met een beetje geluk zul je als ontwerper van de class `Breuk` later zelf ook gebruiker van de class `Breuk` zijn zodat je zelf de vruchten van je inspanningen kunt plukken.

3.3 Voorbeeld class `Breuk` (eerste versie).

Dit voorbeeld is een eerste uitbreiding van de op blz. 27 gegeven class `Breuk` (versie 0). In deze versie zul je leren:

- hoe je een object van de class `Breuk` kunt initialiseren (door middel van *constructors*).
- hoe je kunt definiëren wat er moet gebeuren als een object van de class `Breuk` wordt vrijgegeven (door middel van een *destructor*).
- hoe je memberfuncties kunt definiëren die ook voor constante objecten van de class `Breuk` gebruikt kunnen worden.
- hoe je automatische conversie van type `X` naar het type `Breuk` kunt laten plaatsvinden (door middel van constructors).
- wat je moet doen om objecten van de class `Breuk` te kunnen toekennen en kopiëren (niets!).

Ik zal nu eerst de complete source code presenteren van een programma waarin het type `Breuk` gedeclareerd, geïmplementeerd en gebruikt wordt. Meteen daarna zal ik één voor één op de bovengenoemde punten ingaan.

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

// Class interface vertelt wat je met een object van de class kunt doen.
class Breuk {
public:
    Breuk();                  // constructors zie blz. 32
    Breuk(int t);
    Breuk(int t, int n);
    ~Breuk();                 // destructor zie blz. 34
    int teller() const;       // const memberfunctie zie blz. 35
    int noemer() const;
    void plus(const Breuk& b);
    void abs();
private:
    int boven;
    int onder;
    void normaliseer();
};

// Hulpfunctie: bepaalt de grootste gemene deler.
int ggd(int n, int m) {
    assert(n>=0 && m>=0);[29]
    if (n==0) return m;
```

[29] De standaard functie `assert` doet niets als de, als parameter opgegeven, expressie true oplevert maar breekt het programma met een passende foutmelding af als dit niet zo is. Je kunt zogenaamde "assertions" gebruiken om tijdens de ontwikkeling van het programma te controleren of aan een bepaalde voorwaarden (waarvan je "zeker" weet dat ze geldig zijn) wordt voldaan.

```
    if (m==0) return n;
    while (m!=n)
        if (n>m) n-=m;
        else m-=n;
    return n;
}

// Class implementatie vertelt hoe de class in elkaar zit. Dit is
// voor gebruikers van de class niet van belang.
Breuk::Breuk(): boven(0), onder(1) {
}
Breuk::Breuk(int t): boven(t), onder(1) {
}
Breuk::Breuk(int t, int n): boven(t), onder(n) {
    normaliseer();
}
Breuk::~Breuk() {
    cout<<"Er wordt een Breuk object verwijderd"<<endl;
}
int Breuk::teller() const {
    return boven;
}
int Breuk::noemer() const {
    return onder;
}
void Breuk::plus(const Breuk& b) {
    boven=boven*b.onder + onder*b.boven;
    onder*=b.onder;
    normaliseer();
}
void Breuk::abs() {
    boven=boven<0?-boven:boven;
}
void Breuk::normaliseer() {
    assert(onder!=0);
    if (onder<0) {
        onder=-onder;
        boven=-boven;
    }
    int d(ggd(boven<0?-boven:boven,onder));
    boven/=d;
    onder/=d;
}

int main() {
    {               // nutteloos haakje? Zie blz. 34
        Breuk b1(4);
        cout<<"b1(4) ==> "<<b1.teller()<<'/'<<b1.noemer()<<endl;
    }
    Breuk b2(23, -5);
    cout<<"b2(23, -5) ==> "<<b2.teller()<<'/'<<b2.noemer()<<endl;
    Breuk b3(b2);    // kan dit zomaar? Zie blz. 34
    cout<<"b3(b2) ==> "<<b3.teller()<<'/'<<b3.noemer()<<endl;
    b3.abs();
    cout<<"b3.abs() ==> "<<b3.teller()<<'/'<<b3.noemer()<<endl;
    b3=b2;           // kan dit zomaar? Zie blz. 34
    cout<<"b3=b2 ==> "<<b3.teller()<<'/'<<b3.noemer()<<endl;
    b3.plus(5);
    cout<<"b3.plus(5) ==> "<<b3.teller()<<'/'<<b3.noemer()<<endl;
    cin.get();
```

```
    return 0;
}
```

Uitvoer:

```
b1(4) ==> 4/1
Er wordt een Breuk object verwijderd
b2(23, -5) ==> -23/5
b3(b2) ==> -23/5
b3.abs() ==> 23/5
b3=b2 ==> -23/5
Er wordt een Breuk object verwijderd
b3.plus(5) ==> 2/5
Er wordt een Breuk object verwijderd
Er wordt een Breuk object verwijderd
```

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

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

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

Als een `Breuk` bij het aanmaken niet geïnitialiseerd wordt, dan wordt de constructor zonder parameters aangeroepen. In de definitie van deze constructor worden dan de datamembers `boven` met `0` en `onder` met `1` geïnitialiseerd. Dit gebeurt in een zogenaamde *initialisation list*. Na het prototype van de constructor volgt een `:` waarna de datamembers één voor één geïnitialiseerd worden door middel van de `(...)` notatie.

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

In dit geval is er verder geen code in de constructor opgenomen. Door tussen de `{}` bijvoorbeeld een output opdracht op te nemen zou je een melding op het scherm kunnen geven telkens als een `Breuk` aangemaakt wordt. Deze constructor wordt dus aangeroepen als je een `Breuk` aanmaakt zonder deze te initialiseren zowel bij het aanmaken van een lokale variabele op de stack als bij het aanmaken van een dynamische variabele op de heap met `new`. Na het uitvoeren van de onderstaande code hebben de breuk `b` en de breuk waar `bp` naar wijst beide de waarde 0/1.

```
Breuk b1;            // roep constructor zonder parameters aan
Breuk* bp=new Breuk; // idem
```

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

```
Breuk b2(3,6);       // roep constructor met twee int parameters aan
```

Deze constructor `Breuk(int t, int n);` heb ik als volgt gedefinieerd:

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

Door het definiëren van constructors kun je er dus voor zorgen dat elk object bij het aanmaken “geïnitialiseerd” wordt. Fouten die voortkomen uit het gebruik van een niet geïnitialiseerde variabele worden hiermee voorkomen. Het is dus verstandig om voor elke class één of meer constructors te definiëren.[30]

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

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

Aan het einde van de functie `main` wordt de memberfunctie `plus` als volgt aangeroepen:

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

Uit de uitvoer blijkt dat dit “gewoon” werkt: `b3` wordt verhoogd met 5/1. Op zich is dit vreemd want de memberfunctie `plus` is gedefinieerd met een `const Breuk&` als argument en niet met een `int` als argument. Als je de memberfunctie `plus` aanroept op een `Breuk` met een `int` als argument gebeurt het volgende: eerst “kijkt” de compiler of in de class `Breuk` de memberfunctie `plus(int)`, `plus(int&)` of `plus(const int&)` gedefinieerd is[31]. Als dit het geval is dan wordt deze memberfunctie aangeroepen. Als dit niet het geval is dan “kijkt” de compiler of er in de class `Breuk` een memberfunctie is met een argument van een ander type waarnaar het type `int` omgezet kan worden[32]. In dit geval “ziet” de compiler de memberfunctie `Breuk::plus(const Breuk&)`. De compiler “vraagt zich nu dus af” hoe een variabele van het type `int` omgezet kan worden naar het type `Breuk`. Of met andere woorden hoe een `Breuk` gemaakt kan worden en geïnitialiseerd kan worden met een `int`. Of met andere woorden of de constructor `Breuk(int)` bestaat. Deze constructor bestaat in dit geval. De compiler maakt nu een tijdelijke naamloze variabele aan van het type `Breuk` (door gebruik te maken van de constructor `Breuk(int)`) en geeft een reference naar deze variabele door aan de memberfunctie `plus`. De memberfunctie `plus` telt de waarde van deze variabele op bij de receiver. Na afloop van de memberfunctie `plus` wordt de tijdelijke naamloze variabele weer vrijgegeven.

Als je dus een constructor `Breuk(int)` definieert, wordt het type `int` indien nodig automatisch omgezet naar het type `Breuk`. Of algemeen: als je een constructor `X(Y)` definieert, wordt het type `Y` indien nodig automatisch omgezet naar het type `X`[33].

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

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

- door gebruik te maken van een initialisation list.
- door gebruik te maken van assignments in het code blok van de constructor.

---

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

[31] Slechts 1 van deze 3 mag gedefinieerd zijn anders ontstaan compiler errors omdat de compiler deze types bij aanroep niet kan onderscheiden.

[32] Als dit er meerdere zijn, dan zijn er allerlei conversie regels in de C++ standaard opgenomen om te bepalen welke conversie gekozen wordt. Ik zal dat hier niet bespreken. Wel moet je weten dat een conversie naar een ingebouwd type altijd voorgaat ten opzichte van een conversie naar een zelfgedefinieerd type.

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

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

Dus gebruik:

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

in plaats van:

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

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

Een class kan naast een aantal constructors ook één destructor hebben. De destructor heeft als naam, de naam van de class voorafgegaan door het teken `~`. Als programmeur hoef je niet zelf de destructor aan te roepen. De compiler zorgt ervoor dat de destructor aangeroepen wordt net voordat het object opgeruimd wordt. Dit is:

- aan het einde van het blok waarin de variabele gedefinieerd is voor een lokale variabele.
- aan het einde van het programma voor globale variabele.
- bij het aanroepen van `delete` voor dynamische variabelen.

Als geen destructor gedefinieerd is dan wordt door de compiler een *default destructor* aangemaakt. Deze default destructor roept voor elk dataveld de destructor van dit veld aan (=*memberwise destruction*). In dit geval heb ik de destructor `~Breuk` een melding op het scherm laten afdrukken. Dit is in feite nutteloos en ik had net zo goed de door de compiler gegenereerde default destructor kunnen gebruiken.

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

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

- een object geïnitialiseerd wordt met een object van dezelfde class.
- een object als value parameter wordt doorgegeven aan een functie.
- een object als waarde wordt teruggegeven vanuit een functie.

De compiler zal als de programmeur geen copy constructor definieert zelf een *default copy constructor* genereren. Deze default copy constructor kopieert elk deel waaruit de class bestaat vanuit de een naar de andere (=*memberwise copy*). Het is ook mogelijk om zelf een copy constructor te definiëren (wordt later behandeld zie eventueel blz. 51) maar voor de class `Breuk` voldoet de door de compiler gedefinieerde copy constructor prima.

3.9 Default assignment operator.

Als geen assignment operator (`=`) gedefinieerd is dan wordt door de compiler een *default assignment operator* aangemaakt. Deze default assignment operator roept voor elk dataveld de assignment operator van dit veld aan (=*memberwise assignment*). Het is ook mogelijk om zelf een assignment operator te definiëren (wordt later behandeld zie eventueel blz. 52) maar voor de class `Breuk` voldoet de door de compiler gedefinieerde assignment operator prima.

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

Een object (variabele) van de class (het zelf gedefinieerde type) `Breuk` kan ook als constante gedefinieerd worden. Bijvoorbeeld:

```
Breuk b(1,3);                 // variabele b met waarde 1/3
const Breuk halve(1,2);       // constante halve met waarde 1/2
```

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

```
halve=b;
//  Error: Cannot modify a const object
```

Stel jezelf nu eens de vraag welke memberfuncties je aan mag roepen op het object `halve`[34]. De memberfunties `teller` en `noemer` kunnen zonder problemen worden aangeroepen op een constante breuk omdat ze de waarde van de breuk niet veranderen. De memberfunties `plus` en `abs` mogen echter niet op het object `halve` worden aangeroepen omdat deze memberfuncties dit object zouden wijzigen en een constant object mag je vanzelfsprekend niet veranderen. De compiler kan niet (altijd[35]) controleren of het aanroepen van een memberfunctie een verandering in het object tot gevolg heeft. Daarom mag je een memberfunctie alleen aanroepen voor een `const` object als je expliciet aanduidt dat deze memberfunctie het object niet verandert. Dit doe je door het keyword `const` achter de memberfunctie te plaatsten. Bijvoorbeeld:

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

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

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

Je kunt de memberfunctie `teller` nu dus ook aanroepen voor constante breuken:

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

Het aanroepen van de memberfunctie `plus` voor de `const Breuk halve` geeft echter een foutmelding[36]:

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

---

[34] Zie blz. 30 voor de declaratie van de class `Breuk`.

[35] De compiler kan dit zeker niet als de class definitie seperaat van de applictie gecompileerd wordt. Zie blz. 54.

[36] De Borland C++ Builder 6 compiler geeft slechts een warning in plaats van een error. Dit betekent dat de constante `halve` door het aanroepen van de functie `plus` gewijzigd kan worden. Dit gaat lijnrecht in tegen de C++ standaard! Als we Borland om opheldering vragen (door op F1 te drukken) geven zij de volgende toelichting: *Non-const function called for const object is an error, but was reduced to a warning to give existing programs a chance to work.* Microsoft Visual C++ 6 geeft wel een error: *'plus' : cannot convert 'this' pointer from 'const class Breuk' to 'class Breuk &'. Conversion loses qualifiers.* Cryptisch maar wel correct.

Als je probeert om in een `const` memberfunctie toch de receiver te veranderen krijg je de volgende foutmelding:

```
int Breuk::teller() const {
    boven=1;                  // teller probeert vals te spelen
//  Error: Cannot modify a const object
}
```

Door het toepassen van function overloading is het mogelijk om 2 functies te definiëren met dezelfde naam en met dezelfde parameters waarbij de ene `const` is en de andere niet, maar dit is erg gedetailleerd en zal ik hier niet behandelen[37].

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

- *Vraag-functies*.
  Deze memberfuncties kunnen gebruikt worden om de toestand van een object op te vragen. Deze memberfuncties hebben over het algemeen wel een return type, geen parameters en zijn const memberfuncties.
- *Doe-functies*.
  Deze memberfuncties kunnen gebruikt worden om de toestand van een object te veranderen. Deze memberfuncties hebben over het algemeen geen return type (`void`), wel parameters en zijn non-const memberfuncties.

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

Veel programmeurs denken dat het gebruik van eenvoudige memberfuncties zoals `teller` en `noemer` te veel vertraging opleveren in hun applicatie. Dit is meestal ten onrechte! Maar voor die gevallen waar het echt nodig is biedt C++ de mogelijkheid om functies (ook memberfuncties) als "*inline*" te definiëren. Dit betekent dat de compiler een aanroep naar deze functie niet vertaalt naar een "jump to subroutine" machinecode maar probeert om de machinecode waaruit de functie bestaat rechtstreeks op de plaats van aanroep in te vullen. (Net zoals macro's in assembler.) Dit heeft wel tot gevolg dat de code omvangrijker wordt. Memberfuncties mogen ook in de class declaratie gedefinieerd worden. Ze zijn dan "vanzelf" inline. Als de memberfunctie in de class declaratie alleen maar gedeclareerd is dan kan de definitie van die functie voorafgegaan worden door het keyword `inline` om de functie inline te maken.

Eerste manier om de memberfuncties `teller` en `noemer` inline te maken:

```
class Breuk {
    // ...
    int teller() const {
        return boven;
    }
    int noemer() const {
        return onder;
    }
    // ...
};
```

Tweede manier om de memberfuncties `teller` en `noemer` inline te maken:

```
class Breuk {
    // ...
    int teller() const;
```

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

```
    int noemer() const;
    // ...
};

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

3.12 Class invariant.

In de memberfuncties van de class `Breuk` wordt ervoor gezorgd dat elk object van de class `Breuk` een noemer >0 heeft en de ggd (grootste gemene deler) van de teller en noemer 1 is. Door de noemer altijd >0 te maken kan het teken van de `Breuk` eenvoudig bepaald worden (=teken van teller). Door de `Breuk` altijd te normaliseren wordt onnodige overflow van de datamembers `boven` en `onder` voorkomen. Een voorwaarde waaraan elk object van een class voldoet wordt een *class invariant* genoemd. Als je ervoor zorgt dat de class invariant aan het einde van elke public memberfunctie geldig is en als alle datamembers private zijn, dan weet je zeker dat voor elk object van de class `Breuk` deze invariant altijd geldig is[38]. Dit vermindert de kans op het maken van fouten.

3.13 Voorbeeld class `Breuk` (tweede versie).

Dit voorbeeld is een uitbreiding van de op blz. 30 gegeven class `Breuk` (versie 1). In deze versie zul je leren hoe je een object van de class `Breuk` kunt optellen bij een ander object van de class `Breuk` met de operator `+=` in plaats van met de memberfunctie `plus` (door middel van *operator overloading*). Het optellen van een `Breuk` bij een `Breuk` gaat nu op precies dezelfde wijze als het optellen van een `int` bij een `int`. Dit maakt het type `Breuk` voor programmeurs erg eenvoudig te gebruiken. Ik zal nu eerst de complete source code presenteren van een programma waarin het type `Breuk` gedeclareerd, geïmplementeerd en gebruikt wordt. Meteen daarna zal ik op het bovengenoemde punt ingaan.

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

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

// ...

void Breuk::operator+=(const Breuk& right) {
    boven=boven*right.onder + onder*right.boven;
    onder*=right.onder;
```

[38] Door het definieren van invarianten (en pre- en postcondities) wordt het zelfs mogelijk om op een formele wiskundige manier te bewijzen dat een ADT correct is. Ik zal dit hier verder niet behandelen, in H4 komen we er nog op terug.

```
    normaliseer();
}

int main() {
    Breuk b1(14, 4);
    cout<<"b1(14, 4) ==> "<<b1.teller()<<'/'<<b1.noemer()<<endl;
    Breuk b2(23, -5);
    cout<<"b2(23, -5) ==> "<<b2.teller()<<'/'<<b2.noemer()<<endl;
    b1+=b2;
    cout<<"b1+=b2 ==> "<<b1.teller()<<'/'<<b1.noemer()<<endl;
    cin.get();
    return 0;
}
```

Uitvoer:

```
b1(14, 4) ==> 7/2
b2(23, -5) ==> -23/5
b1+=b2 ==> -11/10
```

3.14 Operator overloading. (Zie eventueel TICPP Chapter12.html.)

In de taal C++ kun je de betekenis van operatoren (zoals bijvoorbeeld `+=`) definiëren voor zelf gedefinieerde typen. Als het statement:

```
b1+=b2;
```

vertaald moet worden, waarbij `b1` en `b2` objecten zijn van de class `Breuk`, zal de compiler "kijken" of in de class `Breuk` de `operator+=` memberfunctie gedefinieerd is. Als dit niet het geval is levert het bovenstaande statement de volgende (niet erg duidelijke) foutmelding op:

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

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

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

De memberfunctie `operator+=` in deze tweede versie van `Breuk` heeft dezelfde implementatie als de memberfunctie `plus` in de eerste versie van `Breuk` (zie blz. 28).

Je kunt voor een zelf gedefinieerd type alle operatoren zelf definiëren behalve de operator `.` waarmee een member geselecteerd wordt en de operator `?:`[39]. Dus zelfs operatoren zoals `[]` (array index), `->` (pointer dereferentie naar member) en `()` (functie aanroep) kun je zelf definiëren! Je kunt de prioriteit van operatoren niet zelf definiëren. Dit betekent dat `a+b*c` altijd wordt geïnterpreteerd als `a+(b*c)`. Ook de associativiteit van de operatoren met gelijke prioriteit kun je niet zelf definiëren. Dit betekent dat `a+b+c` altijd wordt geïnterpreteerd als `(a+b)+c`. Ook is het niet mogelijk om de operatoren die al gedefinieerd zijn voor de ingebouwde typen zelf te (her)definiëren. Het zelf definiëren van operatoren die in de taal C++ nog niet bestaan bijvoorbeeld `@` of `**` is niet mogelijk. Bij het definiëren van operatoren voor zelf gedefinieerde typen moet je er natuurlijk wel voor zorgen dat het gebruik voor de hand liggend is. De compiler heeft er geen enkele moeilijkheid mee als je bijvoorbeeld de memberfunctie `Breuk::operator+=` definieert die de als argument meegegeven `Breuk` van de receiver aftrekt. De programmeurs die de class `Breuk` gebruiken zullen dit minder kunnen waarderen.

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

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

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

Dit komt doordat de `Breuk::operator+=` memberfunctie geen return type heeft (`void`). Het resultaat van de bewerking `b+=c` moet dus niet alleen in het object `b` worden opgeslagen maar ook als resultaat worden teruggegeven. De `operator+=` memberfunctie kan dan als volgt gedeclareerd worden:

```
Breuk operator+=(const Breuk& right);
```
[40]

De definitie is dan als volgt:

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

Ik zal nu eerst bespreken hoe je de receiver (`DIT_OBJECT`) kunt benaderen en daarna zal ik weer op de definitie van de `operator+=` terugkomen.

3.15 `this` pointer.

Elke memberfunctie kan beschikken over een impliciet argument genaamd `this` die het adres bevat van het object waarop de memberfunctie wordt uitgevoerd. Met deze pointer kun je bijvoorbeeld het object waarop een memberfunctie wordt uitgevoerd als return waarde van deze memberfunctie teruggeven. Bijvoorbeeld:

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

3.16 Reference return type (deel 2).

In de bovenstaande `Breuk::operator+=` memberfunctie zal bij het uitvoeren van het `return` statement een kopie van het huidige object worden teruggegeven. Dit is echter niet correct. Want als we deze operator als volgt gebruiken: `(a+=b)+=c;` dan wordt eerst `a+=b` uitgerekend en een kopie van `a` teruggegeven, `c` wordt dan bij deze kopie van `a` opgeteld waarna de kopie wordt verwijderd. Dit is natuurlijk niet de bedoeling, het is de bedoeling dat `c` bij `a` wordt opgeteld. We kunnen dit probleem voorkomen door een `const Breuk` als return type te gebruiken. `a+=(b+=c)` werkt dan nog steeds goed maar `(a+=b)+=c` is dan volgens de C++ standaard niet langer correct omdat de `operator+=` op

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

een `const` object wordt uitgevoerd[41]. Het maken van een kopie is echter toch al zinloos. Op blz. 22 hebben we al gezien dat het maken van deze kopie voorkomen kan worden door een reference als return type te gebruiken. In dit geval kunnen we eenvoudig een reference naar het object zelf teruggeven. Hiermee zorgen we er meteen voor dat de expressie `(a+=b)+=c` gewoon goed (zoals bij integers) werkt. De `operator+=` memberfunctie kan dan als volgt gedeclareerd worden:

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

De definitie is dan als volgt:

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

3.17 Operator overloading (deel2).

Behalve de operator `+=` kun je ook andere operatoren overloaden. Voor de class `Breuk` kun je bijvoorbeeld de operator `+` definiëren, zodat objecten `b1` en `b2` van de class `Breuk` gewoon door middel van `b1+b2` opgeteld kunnen worden.

```
class Breuk {
public:
    Breuk();
    Breuk(int t);
    Breuk(int t, int n);
    Breuk& operator+=(const Breuk& right);
    const Breuk operator+(const Breuk& right) const;
    // ...
};

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

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

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

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

[41] Helaas geeft de C++ Builder 6 compiler bij deze fout geen error (zoals volgens de standaard zou moeten) maar alleen een warning.

3.18 `operator+` FAQ.

Q: Waarom gebruik je `const Breuk` in plaats van `Breuk` als return type bij `operator+`?
A: Dit heb ik afgekeken van Scott Meyers (zie de boeken: *Effective C++* en *More Effective C++*). Als `a`, `b` en `c` van het type `int` zijn dan levert de expressie `(a+b)+=c` de volgende (onduidelijke) foutmelding op "Error: Lvalue required". Dit is goed omdat het optellen van `c` bij een tijdelijke variabele (de som `a+b` wordt namelijk aan een tijdelijke variabele toegekend) zinloos is. De tijdelijke variabele wordt namelijk meteen na het berekenen van de expressie weer verwijderd. Waarschijnlijk heeft de programmeur `(a+=b)+=c` bedoeld. Als `a`, `b` en `c` van het type `Breuk` zijn en we kiezen als return type van `Breuk::operator+` het type `Breuk` dan zal de expressie `(a+b)+=c` zonder foutmelding vertaald worden. Als we echter als return type `const Breuk` gebruiken dan levert deze expressie wel een foutmelding op omdat de `operator+=` niet op een `const` object uitgevoerd kan worden.[42] Als je het zelfgedefinieerde type `Breuk` zoveel mogelijk op het ingebouwde type `int` wilt laten lijken (en dat wil je) dan moet je dus `const Breuk` in plaats van `Breuk` als return type van de `operator+` gebruiken.[43]

Q: Kun je de `operator+` niet beter een reference return type geven zodat het maken van de kopie bij return voorkomen wordt?
A: Nee! We hebben een lokale kopie aangemaakt waarin de som is berekend. Als we een reference teruggeven naar deze lokale kopie ontstaat na return een zogenaamde dangling reference (zie blz. 23) omdat de lokale variabele na afloop van de memberfunctie opgeruimd wordt.

Q: Kun je in de `operator+` de benodigde lokale variabele niet met `new` aanmaken zodat we toch een reference kunnen teruggeven? De met `new` aangemaakte variabele blijft immers na afloop van de `operator+` memberfunctie gewoon bestaan.
A: Ja dit kan wel, maar je kunt het beter **niet** doen. De met `new` aangemaakte variabele zal namelijk (zo lang het programma draait) nooit meer vrijgegeven worden. Dit is een voorbeeld van een *memory leak*. Elke keer als er twee breuken opgeteld worden neemt het beschikbare geheugen af!

Q: Kun je de implementatie van operator+ niet nog verder vereenvoudigen tot:
```
return *this+=right;
```
A: Nee! Na de bewerking `c=a+b;` is dan niet alleen `c` gelijk geworden aan de som van `a` en `b` maar is ook `a` gelijk geworden aan de som van `a` en `b` en dat is natuurlijk niet de bedoeling.

Q: Kun je bij een `Breuk` ook een `int` optellen?
```
Breuk a(1,2);
Breuk b(a+1);
```
A: Ja. De expressie `a+1` wordt geïnterpreteerd als `a.operator+(1)`. De compiler zal dan “kijken” of de memberfunctie `Breuk::operator+(int)` gedefinieerd is. Dit is hier niet het geval. De compiler “ziet” dat er wel een memberfunctie `Breuk::operator+(const Breuk&)` gedefinieerd is en zal vervolgens “kijken” of de `int` omgezet kan worden naar een `Breuk`. Dit is in dit geval mogelijk door gebruik te maken van de constructor `Breuk(int)`. De compiler maakt dan met deze constructor een tijdelijke variabele van het type `Breuk` aan en initialiseert deze variabele met `1`. Vervolgens wordt een reference naar deze tijdelijke variabele als argument aan de `operator+` memberfunctie meegegeven. De tijdelijke variabele wordt na het uitvoeren van de `operator+` memberfunctie weer opgeruimd.

---

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

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

Q: Kun je bij een `int` ook een `Breuk` optellen?
```
Breuk a(1,2);
Breuk b(1+a);
```
A: Nee nu niet. De expressie `1+a` wordt geïnterpreteerd als `1.operator+(a)`. De compiler zal dan "kijken" of de het ingebouwde type `int` het optellen met een `Breuk` heeft gedefinieerd. Dit is vanzelfsprekend niet het geval. De compiler "ziet" dat wel gedefinieerd is hoe een `int` bij een `int` opgeteld moet worden en zal vervolgens "kijken" of de `Breuk` omgezet kan worden naar een `int`. Dit is in dit geval ook niet mogelijk[44]. De compiler genereert de volgende foutmelding:
```
//  Error: illegal structure operation.
```
Als je echter de `operator+` niet als memberfunctie definieert maar in plaats daarvan de globale `operator+` overloadt, dan wordt het wel mogelijk om een `int` bij een `Breuk` op te tellen.

3.19 Operator overloading (deel 3).

Naast het definiëren van operatoren als memberfuncties van zelf gedefinieerde typen kun je ook de globale operatoren overloaden[45]. De globale `operator+` is onder andere gedeclareerd voor het ingebouwde type `int`:

```
const int operator+(int, int);
```

Merk op dat deze globale operator twee parameters heeft, dit in tegenstelling tot de memberfunctie `Breuk::operator+` die slechts één parameter heeft. Bij deze memberfunctie wordt de parameter opgeteld bij de receiver. Een globale operator heeft geen receiver dus zijn voor een optelling twee parameters nodig. Een expressie zoals: `a+b` zal dus als `a` en `b` beide van het type `int` zijn geïnterpreteerd worden als `operator+(a, b)`. Je kunt nu door gebruik te maken van operator (=function) overloading zelf zoveel globale `operator+` functies definiëren als je maar wilt.

Als je dus een `Breuk` bij een `int` wilt kunnen optellen, kun je de volgende globale `operator+` definiëren:

```
const Breuk operator+(int left, const Breuk& right) {
    return right+left;
}
```

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

In plaats van zowel een memberfunctie `Breuk::operator+(const Breuk&)` en een globale `operator+(int, const Breuk&)` te declareren kun je ook één globale `operator+(const Breuk&, const Breuk&)` declareren.

Voorbeeld van het overloaden van de globale `operator+` voor objecten van de class `Breuk`.

```
class Breuk {
public:
    Breuk(int t);
    Breuk& operator+=(const Breuk& right);
    // ...
};
```

[44] Op blz. 44 zal ik bespreken hoe je deze type conversie indien gewenst zelf kunt definiëren.

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

```
const Breuk operator+(const Breuk& left, const Breuk& right) {
    Breuk copyLeft(left);
    copyLeft+=right;
    return copyLeft;[46]
}

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

De unary operatoren (dat zijn operatoren met 1 operand, zoals `!`) en de assignment operatoren (zoals `+=`) kunnen het beste als memberfunctie overloaded worden. De overige binary operatoren kunnen het beste als gewone functie overloaded worden. In dit geval wordt namelijk (indien nodig) de linker operand of de rechter operand geconverteerd naar het benodigde type.

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

Bij het overloaden van de `operator++` en `operator--` ontstaat een probleem omdat beide zowel een *prefix* als een *postfix* operator variant kennen[47]. Dit probleem is opgelost door de postfix versie te voorzien van een (dummy) `int` argument.

Voorbeeld van het overloaden van `operator++` voor objecten van de class `Breuk`. De implementie van deze memberfuncties wordt op blz. 45 gegeven.

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

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

---

[46] Voor de fijnproever: de implementatie van `operator+` kan ook in 1 regel:
```
const Breuk operator+(const Breuk& left, const Breuk& right) {
   return Breuk(left)+=right;
}
```
Voor de connaisseur (echte kenner): De implementatie kan ook zo:
```
const Breuk operator+(Breuk left, const Breuk& right) {
   return left+=right;
}
```

[47] Voor wie het niet meer weet: Een postfix operator `++` wordt na afloop van de expressie uitgevoerd dus `a=b++;` wordt geïnterpreteerd als `a=b;b=b+1;`. De prefix operator `++` wordt voorafgaand aan de expressie uitgevoerd dus `a=++b;` wordt geïnterpreteerd als `b=b+1;a=b;`.

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

Een constructor van class `Breuk` met 1 argument van het type `T` wordt door de compiler gebruikt als een variabele van het type `T` moet worden geconverteerd naar het type `Breuk` (zie blz. 33). Door het definiëren van een conversie operator voor het type `T` kan de programmeur ervoor zorgen dat objecten van de class `Breuk` door de compiler (indien nodig) omgezet kunnen worden naar het type `T`.

Stel dat je wilt dat een `Breuk` die op een plaats wordt gebruikt waar de compiler een `int` verwacht, door de compiler omgezet wordt naar het "gehele deel" van de `Breuk` dan kun je dit als volgt implementeren:

```
class Breuk {
    // ...
    operator int () const;
    // ...
};

Breuk::operator int () const {
    return boven/onder;
}
```

Pas op bij conversies: definieer geen conversie waarbij “informatie” verloren gaat! Is het dan wel verstandig om een `Breuk` automatisch te laten converteren naar een `int`?

3.22 Voorbeeld class `Breuk` (derde versie).

Dit voorbeeld is een uitbreiding van de op blz. 37 gegeven class `Breuk` (versie 2). In deze versie zijn de operator `==` en de operator `!=` toegevoegd. We zullen een nieuwe vriend (*friend*) leren kennen die ons helpt bij het implementeren van deze operatoren. Ook zul je leren hoe je een object van de class `Breuk` kunt wegschrijven en inlezen door middel van de `iostream` library, zodat de operator `<<` voor wegschrijven en de operator `>>` voor inlezen gebruikt kan worden (door middel van operator overloading). De memberfuncties `teller` en `noemer` zijn nu niet meer nodig. Het wegschrijven van een `Breuk` (bijvoorbeeld op het scherm) en het inlezen van een `Breuk` (bijvoorbeeld van het toetsenbord) gaat nu op precies dezelfde wijze als het wegschrijven en het inlezen van een `int`. Dit maakt het type `Breuk` voor programmeurs erg eenvoudig te gebruiken. Ik zal nu eerst de complete source code presenteren van een programma waarin het type `Breuk` gedeclareerd, geïmplementeerd en gebruikt wordt. Meteen daarna zal ik op bovengenoemde punten één voor één ingaan.

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

class Breuk {
public:
    Breuk();
    Breuk(int t);
    Breuk(int t, int n);
    Breuk& operator+=(const Breuk& right);
    Breuk& operator++();         // prefix
    const Breuk operator++(int); // postfix
    // ...
    // Er zijn nog veel uitbreidingen mogelijk
    // ...
private:
    int boven;
    int onder;
    void normaliseer();
```

```
friend ostream& operator<<(ostream& left, const Breuk& right);
friend bool operator==(const Breuk& left, const Breuk& right);
};

istream& operator>>(istream& left, Breuk& right);
bool operator!=(const Breuk& left, const Breuk& right);
const Breuk operator+(const Breuk& left, const Breuk& right);
// ...
// Er zijn nog veel uitbreidingen mogelijk
// ...

int ggd(int n, int m) {
    // ...
}

Breuk::Breuk(): boven(0), onder(1) {
}
Breuk::Breuk(int t): boven(t), onder(1) {
}
Breuk::Breuk(int t, int n): boven(t), onder(n) {
    normaliseer();
}
Breuk& Breuk::operator+=(const Breuk& right) {
    boven=boven*right.onder + onder*right.boven;
    onder*=right.onder;
    normaliseer();
    return *this;
}
Breuk& Breuk::operator++() {
    boven+=onder;
    return *this;
}
const Breuk Breuk::operator++(int) {
    Breuk b(*this);
    ++(*this);
    return b;
}
void Breuk::normaliseer() {
    // ...
}

const Breuk operator+(const Breuk& left, const Breuk& right) {
    return Breuk(left)+=right;
}
ostream& operator<<(ostream& left, const Breuk& right) {
    return left<<right.boven<<'/'<<right.onder;
}
istream& operator>>(istream& left, Breuk& right) {
    int teller;
    if (left>>teller)
        if (left.peek()=='/') {
            left.get();
            int noemer;
            if (left>>noemer) right=Breuk(teller, noemer);
            else right=Breuk(teller);
        }
        else right=Breuk(teller);
    else right=Breuk();
    return left;
}
```

```
bool operator==(const Breuk& left, const Breuk& right) {
    return left.boven==right.boven && left.onder==right.onder;
}
bool operator!=(const Breuk& left, const Breuk& right) {
    return !(left==right);
}

int main() {
    Breuk b1, b2;
    cout<<"Geef Breuk: ";
    cin>>b1;
    cout<<"Geef nog een Breuk: ";
    cin>>b2;
    cout<<b1<<" + "<<b2<<" = "<<(b1+b2)<<endl;
    Breuk b3(18, -9);
    if (-2!=b3)
        cout<<"Error."<<endl;
    else
        cout<<"OK."<<endl;
    cout<<b3++<<endl;
    cout<<b3<<endl;
    b3+=5;
    cout<<b3<<endl;
    cin.get();
    cin.get();
    return 0;
}
```

Ik heb ervoor gekozen om de globale `operator==` te overloaden in plaats van een memberfunctie `Breuk::operator==` te definiëren. Dit heeft als voordeel dat zowel het linker als het rechter argument indien nodig naar het type `Breuk` geconverteerd kan worden. Dus zowel de expressies `b==3` als `3==b` kunnen worden gebruikt als `b` van het type `Breuk` is. De implementatie van deze globale `operator==` is als volgt:

```
bool operator==(const Breuk& left, const Breuk& right) {
    return left.boven==right.boven && left.onder==right.onder;
}
```

Dit levert bij compilatie echter de volgende fouten op:

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

De globaal gedefinieerde `operator==` is namelijk geen memberfunctie en heeft dus geen toegang tot de private velden van de class `Breuk`. Maar gelukkig is er een vriend die ons hier te hulp komt: de *friend function*.

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

Een functie kan als friend van een class gedeclareerd worden. Deze friend functies hebben dezelfde rechten als memberfuncties van de class. Vanuit een friend functie van een class heb je dus toegang tot de private members van die class. Omdat een friend functie geen memberfunctie is van de class, is er geen receiver object. Je moet dus, om de private members te kunnen gebruiken, zelf in de friend functie aangeven welk object je wilt gebruiken. Ook memberfuncties van een andere class kunnen als friend functies gedeclareerd worden. Als een class als friend gedeclareerd wordt dan betekent dit dat alle memberfuncties van die class friend functies zijn.

De zojuist besproken globale `operator==` kan dus eenvoudig als friend van de class `Breuk` gedeclareerd worden waardoor de compiler errors als sneeuw voor de zon verdwijnen.

```
class Breuk {
public:
    // ...
private:
    // ...
friend bool operator==(const Breuk& left, const Breuk& right);
};
```

Het maakt niets uit of een friend declaratie in het private of in het public deel van de class geplaatst wordt. Hoewel een friend function binnen de class gedeclareerd wordt is een friend function géén memberfunctie maar een globale (normale) functie. In deze friend functie kun je de private members `boven` en `onder` van de class `Breuk` zonder problemen gebruiken.

In eerste instantie lijkt het er misschien op dat een friend function in tegenspraak is met het principe van information hiding. Dit is echter niet het geval; een class beslist namelijk zelf wie zijn vrienden zijn. Voor alle overige functies (geen member en geen friend) geldt nog steeds dat de private datamembers en private memberfuncties ontoegankelijk zijn. Net zoals in het gewone leven moet een class zijn vrienden met zorg selecteren. Als je elke functie als friend van elke class declareert worden het principe van information hiding natuurlijk wel overboord gegooid.

De globale `operator!=` is als volgt overloaded voor het type `Breuk`:

```
bool operator!=(const Breuk& left, const Breuk& right) {
    return !(left==right);
}
```

Bij de implementatie is gebruik gemaakt van de al eerder voor het type `Breuk` overloaded globale `operator==`. Het is dus niet nodig om deze `operator!=` als friend function van de class `Breuk` te definiëren.

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

Het object `cout` dat in de headerfile `iostream.h` gedeclareerd is, is van het type `ostream`[48]. Om er voor te zorgen dat je een `Breuk b` op het scherm kunt afdrukken door middel van: `cout<<b`; moet je (zoals je nu zo langzamerhand wel zult begrijpen) de globale `operator<<` overloaden[49].

```
ostream& operator<<(ostream& left, const Breuk& right) {
    left<<right.boven<<'/'<<right.onder;
    return left;
}
```

Deze globale operator gebruikt de private velden van de class `Breuk` en moet daarom als friend van deze class gedeclareerd worden. Als eerste parameter is een `ostream&` gebruikt omdat de operator het als parameter meegegeven object (in ons geval `cout`) moet kunnen aanpassen. Als return type is het

[48] Dit is niet de waarheid. Maar dit is echter niet van belang voor de hier besproken theorie. Het object `cout` is in werkelijkheid van een class die is afgeleid van de class `ostream`. Het begrip afleiden komt in hoofdstuk 5 uitgebreid aan de orde.

[49] We hebben hier geen keuze omdat het definiëren van de memberfunctie `ostream::operator<<(const Breuk&)` geen reële mogelijkheid is. De class `ostream` is namelijk in de `iostream` library gedeclareerd en deze library kunnen (en willen) we natuurlijk niet aanpassen.

type `ostream&` gebruikt. De als parameter meegegeven `ostream&` wordt ook weer teruggegeven. Dit heeft tot gevolg dat je verschillende `<<` operatoren achter elkaar kunt "rijgen". Bijvoorbeeld:

```
cout<<"De breuk a = "<<a<<" en de breuk b = "<<b<<endl;
```

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

```
ostream& operator<<(ostream& left, const Breuk& right) {
    return left<<right.boven<<'/'<<right.onder;
}
```

Op vergelijkbare wijze kun je de globale `operator>>` overloaden zodat de objecten `a` en `b` van de class `Breuk` als volgt ingelezen kunnen worden.

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

Voor het type van de eerste parameter en het return type kun je in dit geval `istream` gebruiken.

Als je deze laatste versie van `Breuk` vergelijkt met versie 0 op blz. 27 dan is eigenlijk het enige belangrijke verschil dat het zelf gedefinieerde type `Breuk` nu op precies dezelfde wijze als de ingebouwde typen te gebruiken is. Je hebt er ongeveer 20 pagina's gedetailleerde informatie voor door moeten worstelen om dit te bereiken. Dat dit toch de moeite waard is heb ik op een van de eerste van die 20 pagina's al een keer benadrukt, maar het zou kunnen zijn dat je door alle tussenliggende details het uiteindelijke nut van alle inspanning vergeten bent. Dus zal ik hier nogmaals antwoord geven op de vraag: *"Is het al die inspanningen wel waard om een `Breuk` zo te maken dat je hem net zoals een ingebouwd type kan gebruiken?"* Ja! Het maken van de class `Breuk` is dan wel een hele inspanning maar iedere programmeur kan vervolgens dit zelf gedefinieerde type, als hij of zij de naam `Breuk` maar kent, als vanzelf (intuïtief) gebruiken. Er is daarbij geen handleiding of helpfile nodig is, omdat het type `Breuk` zich net zo gedraagt als een ingebouwd type. De class `Breuk` hoeft maar één keer gemaakt te worden, maar zal talloze malen gebruikt worden.

3.25 Voorbeeld class `Vector`.

Het in C (en dus ook in C++) ingebouwde *array* type heeft een aantal nadelen en beperkingen. De belangrijkste daarvan zijn:

- De grootte van de array moet bij het compileren van het programma bekend zijn. In de praktijk komt het vaak voor dat pas bij het uitvoeren van het programma bepaald kan worden hoe groot de array moet zijn.
- Er vindt bij het gebruiken van de array geen controle plaats op de gebruikte index. Als je element N benadert uit een array die N-1 elementen heeft, dan krijg je geen foutmelding maar wordt de geheugenplaats "achter" het einde van de array benaderd. Dit is vaak de oorzaak van fouten die lang onopgemerkt kunnen blijven en dan (volgens de wet van Murphy op het moment dat het het slechtst uitkomt) plotseling voor de dag kunnen komen. Deze fouten zijn vaak heel moeilijk te vinden omdat de oorzaak van de fout niets met het gevolg van de fout te maken heeft.

In dit voorbeeld zul je zien hoe ik zelf een eigen array type genaamd `Vector` heb gedefinieerd[50] waarbij:

- de grootte pas tijdens het uitvoeren van het programma bepaald kan worden.
- bij het indexeren de index wordt gecontroleerd en een foutmelding wordt gegeven als de index zich buiten de grenzen van de `Vector` bevindt.

[50] In de ISO/ANSI standaard C++ library is ook een type `vector` opgenomen. Dit type wordt bij het onderwijsdeel SOPX3E1C1 in EH3C&D of PROGMI1T3 in IH1 besproken.

De door de compiler gegenereerde copy constructor, destructor en `operator=` blijken voor de class `Vector` niet correct te werken. Ik heb daarom voor deze class zelf een copy constructor, destructor en `operator=` gedefinieerd. Na het voorbeeld worden de belangrijkste aspecten toegelicht en worden verwijzingen naar het TICPP boek gegeven.

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

class Vector {
public:
    explicit Vector(int s);
    Vector(const Vector& v);
    Vector& operator=(const Vector& r);
    ~Vector();
    int& operator[](int index);
    const int& operator[](int index) const;
    int length() const;
    bool operator==(const Vector& r) const;
    bool operator!=(const Vector& r) const;
    // ...
    // Er zijn vele uitbreidingen mogelijk.
private:
    int size;
    int* data;
};

Vector::Vector(int s): size(s), data(new int[s]) {
}
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;
}
Vector::~Vector() {
    delete[] data;
}
int& Vector::operator[](int index) {
    assert(index>=0 && index<size);
    return data[index];
}
const int& Vector::operator[](int index) const { //[51]
```

[51] Het overloaden van de `operator[]` is noodzakelijk omdat ik vanuit deze operator een reference terug wil geven zodat met deze reference in de vector geschreven kan worden bijvoorbeeld `v[12]=144;` Als ik deze operator als const zou hebben gedefinieerd (wat in eerste instantie logisch lijkt, de `operator[]` verandert immers niets in de vector) dan zou deze operator ook gebruikt kunnen worden voor een const vector. Met de teruggegeven reference kun je dan in een const vector schrijven en dat is natuurlijk niet de bedoeling. Om deze reden heb ik de `operator[]` als non const gedefinieerd. Dit heeft tot gevolg dat de `operator[]` niet meer gebruikt kan worden voor const

(wordt vervolgd...)

```
    assert(index>=0 && index<size);
    return data[index];
}
int Vector::length() const {
    return size;
}
bool Vector::operator==(const Vector& r) const {
    if (size!=r.size)
        return false;
    for (int i(0);i<size;++i)
        if (data[i]!=r.data[i])
            return false;
    return true;
}
bool Vector::operator!=(const Vector& r) const {
    return !(*this==r);
}

int main() {
    cout<<"Hoeveel elementen moet de vector bevatten? ";
    int i;
    cin>>i;
    if (i>0) {
        Vector v(i);
        for (int j(0); j<v.length(); ++j) {
            v[j]=j*j;    // vul v met kwadraten
        }
        Vector w(v);
        cout<<"v[12] = "<<v[12]<<endl;
        cout<<"w[12] = "<<w[12]<<endl;
        v[0]=4;
        cout<<"v[0] = "<<v[0]<<endl;
        if (v==w)
            cout<<"v is nu gelijk aan w."<<endl;
        else
            cout<<"v is nu ongelijk aan w."<<endl;
        w=v;
        cout<<"w = v is uitgevoerd."<<endl;
        if (v!=w)
            cout<<"v is nu ongelijk aan w."<<endl;
        else
            cout<<"v is nu gelijk aan w."<<endl;
    }
    else
        cout<<"Doe niet zo negatief!"<<endl;
    cin.get();
    cin.get();
    return 0;
}
```

[51] (...vervolg)
vectors. Dit is weer teveel van het goede want nu kun je ook niet meer lezen m.b.v `operator[]` uit een const vector bijvoorbeeld `i=v[12];`. Om het lezen uit een const vector toch weer mogelijk te maken heb ik naast de non-const `operator[]` nog een const `operator[]` gedefinieerd. Deze const `operator[]` geeft een const reference terug en zoals je weet kan een const reference alleen gebruikt worden om te lezen. (Als je deze voetnoot na één keer lezen begrijpt is er waarschijnlijk iets niet helemaal in orde :-).

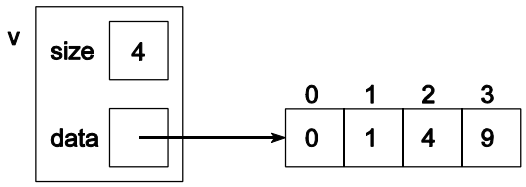
3.26 `explicit` constructor. (Zie eventueel TICPP Chapter12.html#Heading373.)

De constructor `Vector(int)` is `explicit` gedeclareerd om te voorkomen dat de compiler deze constructor gebruikt om een `int` "automatisch" om te zetten naar een `Vector`. Zie blz. 33.

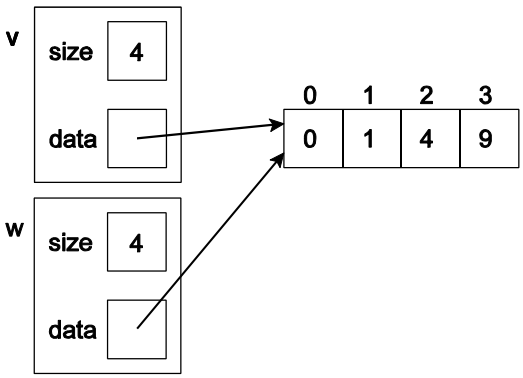
3.27 Copy constructor en default copy constructor. (Zie eventueel TICPP Chapter11.html#Heading331.)

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

De compiler zal als de programmeur geen copy constructor definieert zelf een default copy constructor genereren. Deze default copy constructor kopieert elk deel waaruit de class bestaat vanuit de een naar de andere (=memberwise copy). Naast de default copy constructor genereert de compiler ook (indien niet door de programmeur gedefinieerd) een default assignment operator en een default destructor. De default assignment operator doet een memberwise assignment en de default destructor doet een member-wise destruction.



Dat je voor de class Vector zelf een destructor moet definiëren waarin je het in de constructor met `new` gereserveerde geheugen met `delete` weer vrij moet geven zal niemand verbazen. Dat je voor de class `Vector` zelf een copy constructor en `operator=` moet definiëren ligt misschien minder voor de hand. Ik zal eerst bespreken wat het probleem is bij de door de compiler gedefinieerde default copy constructor en `operator=`. Daarna zal ik bespreken hoe we zelf een copy constructor en een operator kunnen declareren en implementeren. Een `Vector v` met 4 elementen gevuld met kwadraten is hierboven schematisch weergegeven.



De door de compiler gegenereerde copy constructor zal een memberwise copy uitvoeren. De datamembers `size` en `data` worden dus gekopieerd. Als je de `Vector v` naar de `Vector w` kopieert door middel van het statement `Vector w(v);`[52] dan ontstaat de hiernaast weergegeven situatie.

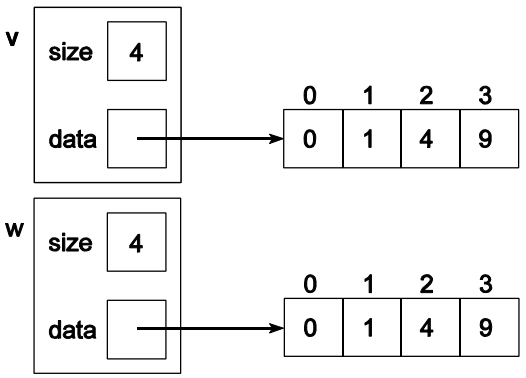
Dit is niet goed omdat als je nu de kopie wijzigt (bijvoorbeeld `w[2]=8;`) dan zal ook het origineel (`v[2]`) gewijzigd zijn en dat is natuurlijk niet de bedoeling.

[52] Dit statement kan ook als volgt geschreven worden `Vector w = v;` Ook in dit geval wordt de copy-constructor van de class Vector aangeroepen en dus niet de `operator=` memberfunctie. Het gebruik van het `=` teken bij een initialisatie is verwarrend omdat het lijkt alsof er een assigment wordt gedaan terwijl in werkelijkheid de copy-constructor wordt aangeroepen. Om deze reden raad ik je aan om bij initialisatie altijd de notatie `Vector w(v);` te gebruiken. Ook bij de ingebouwde types, bijvoorbeeld `int i(0);`

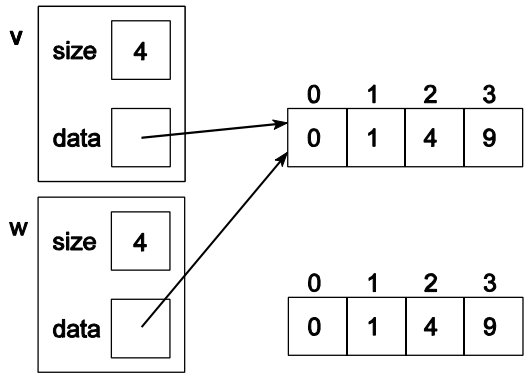
De gewenste situatie na het kopiëren van de `Vector v` naar de `Vector w` is hiernaast weergegeven. Om deze situatie te bereiken moeten je zelf de copy constructor van de class Vector declareren:

```
Vector::Vector(const Vector&);
```

Ik zal nu eerst de problematiek van de assignment operator bespreken omdat ik de `operator=` bij de implementatie van de copy constructor goed kan gebruiken.



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



Voor de door de compiler gegenereerde assignment operator geldt ongeveer hetzelfde verhaal als voor de door de compiler gegenereerde copy constructor. Na het statement `w=v;` zal de situatie zoals hiernaast weergegeven ontstaan. Je ziet dat de door de compiler gegenereerde assignment operator niet alleen onjuist werkt maar bovendien een memory leak veroorzaakt.

Als je de assignment operator voor de class `Vector` zelf wilt definiëren moet je de memberfunctie `operator=` declareren.

Deze memberfunctie kun je dan als volgt implementeren[53]:

```
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;
}
```

Voor het return type van `operator=` heb ik `Vector&` gebruikt. Dit zorgt ervoor dat assignment operatoren achter elkaar “geregen” kunnen worden, bijvoorbeeld: `u=v=w` zie blz. 22. De implementatie van de `operator=` moet de `Vector r` toekennen aan de receiver. Als eerste wordt de `size` van de receiver vergeleken met de `size` van `r`. Je zou misschien kunnen denken dat ik deze test voor optimalisatie doeleinden heb toegevoegd[54]. Dit is echter niet zo, het is een beveiliging tegen een hele smerige fout waar ik aan het einde van deze paragraaf op terugkom. Als de `size`’s gelijk zijn, dan kan de data

[53] Bij SOPX3E1C1 in EH3C&D zullen we deze implementatie nogmaals bekijken.

[54] Optimaliseer een programma alleen als bij testen is aangetoond dat het programma te traag is, en als snellere hardware niet beschikbaar of betaalbaar is. Optimaliseer een functie alleen als bij testen is aangetoond dat die functie een aanzienlijk gedeelte van de executietijd (van het te trage programma) opslurpt. Vaak blijkt dat door de keuze van een beter algoritme veel meer winst te behalen valt dan door kleine optimalisaties van een bepaald algoritme. Dit komt in het tweede gedeelte van dit kwartaal uitgebreid aan de orde.

meteen gekopieerd worden. Als de `size`'s ongelijk zijn wordt het pas leuk. Eerst moet de array waar de pointer `data` van de receiver naar wijst vrijgegeven worden. Dan moet een nieuwe array die net zo groot is als die van `r` gereserveerd worden. De pointer `data` van de receiver moet naar deze array wijzen. Vervolgens wordt de `size` van de receiver gelijk gemaakt aan de `size` van `r` waarna de inhoud van de array van `r` gekopieerd wordt naar de array van de receiver.

Stel nu dat ik het vergelijken van de `size`'s achterwege gelaten zou hebben. In dat geval zou bij het uitvoeren van het statement `v=v;` een erg lugubere fout ontstaan. De array van het linker argument zal eerst vrijgegeven worden waarna een nieuwe array voor het linker argument wordt gereserveerd. Waarna de inhoud van de array van het rechter argument naar de array van het linker argument wordt gekopieerd. Omdat het rechter argument gelijk is aan het linker argument en de array van het linker argument net is vrijgegeven, gebruiken we bij het kopiëren dus net vrijgegeven geheugen. Dit geheugen kan ondertussen voor andere doeleinden gebruikt zijn! De kans dat dit gebeurt is echter klein. Er moet net een interrupt tussendoor komen die tot gevolg heeft dat het net vrijgegeven geheugen gebruikt wordt. Dit heeft tot gevolg dat deze fout lang onopgemerkt kan blijven en dan (volgens de wet van Murphy op het moment dat je net voorgedragen bent voor een promotie mits je programma goed door de allerlaatste test komt) plotseling voor de dag kan komen. Deze fout is heel moeilijk te vinden omdat de oorzaak van de fout niets met het gevolg van de fout te maken heeft. Bij het implementeren van de `operator=` memberfunctie moet je dus altijd oppassen voor self assignment `a=a`.

De copy constructor kan nu eenvoudig door gebruik te maken van de zojuist gedefinieerde `operator=` geïmplementeerd worden:

```
Vector::Vector(const Vector& v): size(v.size), data(new int[v.size]) {
    *this=v;
}
```

Natuurlijk kun je de copy constructor ook als volgt definiëren:

```
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];
}
```

Het hier zelf gedefinieerde type `Vector` heeft een aantal voordelen ten opzichte van het ingebouwde array type. De belangrijkste zijn dat het aantal elementen pas tijdens het uitvoeren van het programma bepaald wordt en dat bij het gebruik van de `operator[]` de index wordt gecontroleerd. Het type `Vector` heeft echter ook een groot nadeel ten opzichte van het ingebouwde array type. Het type `Vector` heeft namelijk altijd elementen van het type `int` terwijl met het ingebouwde type array, array's met elementen van elk gewenst type gedefinieerd kunnen worden. In hoofdstuk 4 zul je leren hoe je het type `Vector` zodanig kunt declareren dat je het type van de elementen pas bij het gebruik van de `Vector` hoeft op te geven (net zoals bij het ingebouwde array type).

3.29 Wanneer moet je zelf een destructor, copy constructor en `operator=` definiëren.

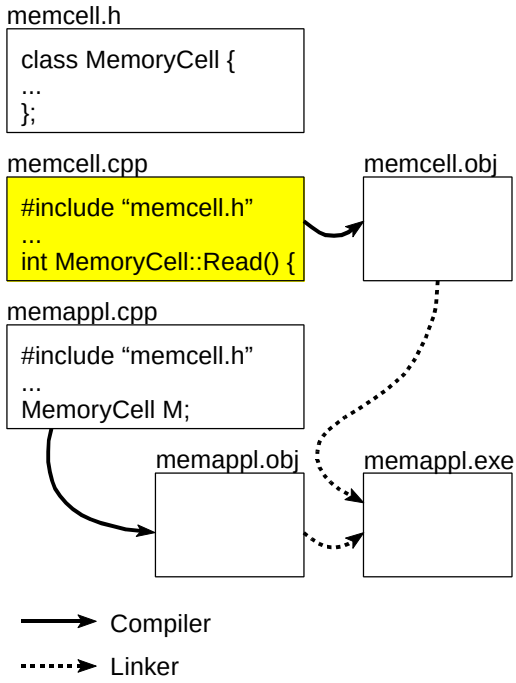
Een class moet een *zelf* gedefinieerde copy constructor, `operator=` en destructor bevatten als:

- die class een pointer bevat en
- als bij het kopiëren van een object van de class niet de pointer, maar de data waar de pointer naar wijst moet worden gekopieerd en
- als bij een toekenning aan een object van de class niet de pointer, maar de data waar de pointer naar wijst moet worden toegekend en
- als bij het "destructen" van een object van de class niet de pointer, maar de data waar de pointer naar wijst moet worden "destructed".

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

3.30 Voorbeeld separate compilation van class `MemoryCell`.

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



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

```
// Dit is file memcell.h
// prevent multiple inclusion.
#ifndef _memcell_ [55]
#define _memcell_
class MemoryCell {
public:
     int Read() const;
     void Write(int x);
private:
     int StoredValue;
};
#endif
```

```
// Dit is file memcell.cpp
#include "memcell.h"
int MemoryCell::Read() const {
     return StoredValue;
}
void MemoryCell::Write(int x) {
     StoredValue=x;
}
```

```
// Dit is file memappl.cpp
#include <iostream>
using namespace std;
#include "memcell.h"
int main() {
     MemoryCell M;
     M.Write(5);
     cout<<"Cell contents are "<<M.Read()<<endl;
// ...
```

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

4 Templates.

Een van de belangrijkste doelen van C++ is het ondersteunen van het hergebruik van code. In dit hoofdstuk wordt één van de taalconstructies die C++ biedt om hergebruik van code mogelijk te maken, de *template*, besproken. De template maakt het mogelijk om functies of classes te schrijven die werken met een nog onbepaald type. We noemen dit *generiek* programmeren. Pas tijdens het (her)gebruik van de functie of class moeten we specificeren welk type gebruikt moet worden.

4.1 Template functies.

Op blz. 21 heb ik de functie `swapInts` besproken waarmee twee `int` variabelen verwisseld kunnen worden:

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

Als je twee `double` variabelen wilt verwisselen kun je deze functie niet rechtstreeks gebruiken. Waarschijnlijk ben je op dit moment gewend om de functie `swapInts` op de volgende wijze te “hergebruiken”:

- maak een kopie van de functie met behulp van de editor functies “knippen” en “plakken”.
- vervang het type `int` door het type `double` met behulp van de editor functie “zoek en vervang”.

Deze vorm van hergebruik heeft de volgende nadelen:

- Telkens als je zelf een nieuw type definieert (bijvoorbeeld `Tijdsduur`) waarvoor je de functie `swap` ook wilt kunnen gebruiken zul je opnieuw moeten knippen, plakken, zoeken en vervangen.
- Bij een wat ingewikkelder algoritme, bijvoorbeeld sorteren, is het niet altijd duidelijk welke `int` je wel en welke `int` je niet moet vervangen in `double` als je in plaats van een `int` array een `double` array wilt sorteren. Hierdoor kunnen in een goed getest algoritme toch weer fouten opduiken.
- Als er zich in het algoritme een logische fout bevindt of als je het algoritme wilt vervangen door een efficiëntere versie, dan moet je de benodigde wijzigingen in elke gekopieerde versie aanbrengen.

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

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

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

Na het keyword `template` volgt een lijst van template parameters tussen `<` en `>`. Een template parameter zal meestal een type zijn[56]. Dit wordt aangegeven door het keyword `typename`[57] gevolgd door een naam voor de parameter. Ik heb hier de naam `T` gebruikt maar ik had net zo goed de naam `VulMaarIn` kunnen gebruiken. De template parameter moet[58] in de parameterlijst van de functie gebruikt worden. Deze template functie definitie genereert nog geen enkele machinecode instructie. Het is alleen een "mal" waarmee (automatisch) functies aangemaakt kunnen worden.

Als je nu de functie `swap` aanroept zal de compiler zelf afhankelijk van het type van de gebruikte parameters de benodigde "versie" van `swap` genereren[59] door voor het template argument (in dit geval `T`) het betreffende type in te vullen.

Dus de aanroep:

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

heeft tot gevolg dat de volgende functie gegenereerd wordt[60]:

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

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

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

dan heeft dit tot gevolg dat een tweede functie `swap` gegenereerd wordt[61]:

```
void swap(Breuk& p, Breuk& q) {
```

---

[56] Een template kan ook "normale" parameters hebben. Zie blz. 58.

[57] In plaats van `typename` mag ook `class` gebruikt worden. Omdat het keyword `typename` pas laat in de ISO/ANSI C++ standaard is opgenomen gebruiken veel C++ programmeurs en C++ boeken in plaats van `typename` nog steeds het "verouderde" `class`. Het gebruik van `typename` is op zich duidelijker omdat bij het gebruik van de template zowel zelfgedefineerde types (classes) als ingebouwde typen (zoals `int`) gebruikt kunnen worden.

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

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

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

[61] Hier blijkt duidelijk het belang van function name overloading (zie blz. 15).

```
    Breuk t(p);
    p=q;
    q=t;
}
```

Het gebruik van een template functie heeft de volgende voordelen:

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

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

- Doordat de compiler de volledige template functie definitie nodig heeft om een functie aanroep te kunnen vertalen moet de template functie definitie in een headerfile (`.h` file) opgenomen worden en “included” worden in elke `.cpp` file waarin de template functie gebruikt wordt. Het is niet mogelijk om de template functie afzonderlijk te compileren tot een `.obj` file en deze later aan de rest van de code te “linken” zoals dit met een “gewone” functie wel kan.

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

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

In het voorgaande hoofdstuk heb ik het zelf gedefinieerde type `Vector` besproken. Dit type heeft enkele voordelen ten opzichte van het ingebouwde array type maar heeft als nadeel dat de elementen alleen maar van het type `int` kunnen zijn. Als je een `Vector` met elementen van het type `double` nodig hebt, dan kun je natuurlijk gaan kopiëren, plakken, zoeken en vervangen maar daar zitten weer de in de vorige paragraaf besproken nadelen aan. Als je verschillende versies van Vector “met de hand” genereert moet je bovendien elke versie een andere naam geven omdat een class naam uniek moet zijn. In plaats daarvan kun je ook het template mechanisme gebruiken om een `Vector` met elementen van het type `T` te definiëren, waarbij het type `T` pas bij het gebruik van de template class `Vector` wordt bepaald. Bij het gebruik van de template class `Vector` kan de compiler niet (snel) zelf bepalen wat het type `T` moet zijn. Vandaar dat je dit bij het gebruik van de template class `Vector` zelf moet specificeren. Bijvoorbeeld:

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

4.3 Voorbeeld template class `Vector`.

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

template <typename T> class Vector {
public:
    explicit Vector(int s);
    Vector(const Vector<T>& v);
    Vector<T>& operator=(const Vector<T>& r);
    ~Vector();
    T& operator[](int index);
```

```
    const T& operator[](int index) const;
    int length() const;
    bool operator==(const Vector<T>& r) const;
    bool operator!=(const Vector<T>& r) const;
private:
    int size;
    T* data;
};

template <typename T> Vector<T>::Vector(int s):
        size(s), data(new T[s]) {
}
// ... enz. ...

int main() {
    cout<<"Hoeveel elementen moet de vector bevatten? ";
    int i; cin>>i;
    Vector<double> v(i);
    for (int j(0); j<v.length(); ++j)
        v[j]=sqrt(j);                  // Vul v met wortels
    cout<<"v[12] = "<<v[12]<<endl;
    Vector<int> w(i);
    for (int t(0); t<w.length(); ++t)
        w[t]=t*t;                      // Vul w met kwadraten
    cout<<"w[12] = "<<w[12]<<endl;
// ...
```

Een template kan ook meerdere parameters hebben. Een template parameter kan in plaats van een typename parameter ook een “normale parameter” zijn. Zo zou je ook de volgende template class kunnen definiëren:

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

Deze template class kan dan als volgt gebruikt worden:

```
FixedVector<Breuk, 300> vb;   // Een vector met 300 breuken.
```

Er zijn grote verschillen tussen deze template class `FixedVector` en de eerder gedefinieerde template class `Vector`:

- De `size` moet bij de laatste template tijdens het compileren bekend zijn. De compiler genereert (instantieert) namelijk de benodigde “versie” van de template class en vult daarbij de opgegeven template parameters in.
- De compiler genereert een nieuwe “versie” van de class telkens als deze class gebruikt wordt met andere template parameters. Dit betekent dat `FixedVector<int, 3> v` en `FixedVector<int, 4> w` verschillende type’s zijn. Dus expressies zoals `v!=w` en `v=w` enz. zijn dan niet toegestaan. Telkens als je een `FixedVector` met een andere `size` definieert, wordt er weer een nieuw type met bijbehorende machinecode voor alle memberfuncties gegenereerd[62]. Bij de template class `Vector` wordt maar één “versie” aangemaakt als de variabelen `Vector<int> v(3)` en `Vector<int> w(4)` gedefinieerd worden. De expressies `v!=w` en `v=w` enz. zijn dan wel toegestaan.

[62] Dit is niet de waarheid. Een memberfunctie van een template class wordt niet gegenereerd als de template geïnstantieerd wordt maar pas als de compiler daadwerkelijk een aanroep naar de betreffende memberfunctie moet vertalen. Dit is hier echter niet van belang.

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

4.4 Standaard Templates.

In september 1998 is de ISO/ANSI C++ standaard officieel vastgesteld. In deze standaard zijn een groot aantal standaard templates voor datastructuren en algoritmen opgenomen. Deze in de standaard opgenomen verzameling tempates is grotendeels afkomstig uit de STL (Standard Template Library) een verzameling templates die in het begin van de jaren '90 ontwikkeld werd door Alex Stepanov en Meng Lee van Hewlett Packard Laboratories en sinds 1994 via internet gratis is verspreid. In deze standaard library is ook een generiek type `vector` opgenomen dat wij bij het onderwijsdeel SOPX3E1C1 in EH3C&D of PROGMI1T3 in IH1 zullen bespreken.

4.5 Template details.

Over templates valt nog veel meer te vertellen:

- Template specialisation. Een speciale versie van een template die alleen voor een bepaald type (bijvoorbeeld `int`) of voor bepaalde typen (bijvoorbeeld `T*`) wordt gebruikt. De laatste vorm wordt *partial* specialisation genoemd.
- Template memberfuncties. Een class (die zelf geen template hoeft te zijn) kan memberfuncties hebben die als template gedefinieerd zijn.
- Default waarden voor template parameters. Net zoals voor gewone functie parameters kun je voor template parameters ook default waarden (of typen) specificeren.

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

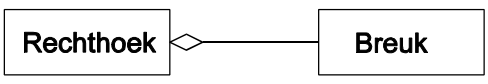
5 Inheritance.

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

De in het vorige hoofdstuk gedefinieerde template class `Vector` lijkt op het eerste gezicht al een prima herbruikbare software component. Als je echter een variant van deze template class `Vector` wilt definiëren, dan heb je op dit moment geen andere keuze dan de template class `Vector` te kopiëren, te voorzien van een andere naam bijvoorbeeld `MyVector` en de benodigde (kleine) wijzigingen in deze kopie aan te brengen. Deze manier van genereren van varianten produceert, zoals je al weet, een minder goed onderhoudbaar programma. Je zult in dit hoofdstuk leren hoe je door het toepassen van een object georiënteerde techniek (inheritance) een onderhoudbare variant van een herbruikbare software component kan maken. Door middel van deze techniek kun je dus software componenten niet alleen hergebruiken op de manier zoals de ontwerper van de component dat bedoeld heeft, maar kun je de software component ook naar je eigen wensen omvormen.

Hergebruik door middel van *composition* is niet specifiek voor OOP. Ook bij de gestructureerde programmeer methode paste je deze vorm van hergebruik al toe. Het hergebruik van een software component door middel van composition is niets anders als het gebruiken van deze component als onderdeel van een andere (nieuwe) software component. Als je bijvoorbeeld rechthoeken wilt gaan gebruiken waarbij de lengte en de breedte als breuk moeten worden weergegeven, dan kun je het ADT `Breuk` als volgt (her)gebruiken:

```
class Rechthoek {
public:
    // ...
```



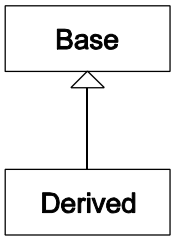
```
private:
    Breuk lengte;
    Breuk breedte;
};
```

We zeggen dan dat de class `Rechthoek` een HAS-A (heeft een of meer) relatie heeft met de class `Breuk`. Schematisch kan dit zoals hierboven getekend worden weergegeven.[63] Bij gestructureerd programmeren is dit de enige relatie die software componenten met elkaar kunnen hebben. Bij object georiënteerd programmeren bestaat ook de z.g IS-A relatie waarop ik nu uitvoerig zal ingaan.

Om de verschillende begrippen te introduceren zal ik niet meteen gebruik maken van een praktisch voorbeeld. Nadat ik de verschillende begrippen geïntroduceerd heb zal ik in een uitgebreid praktisch voorbeeld uit de elektrotechniek laten zien hoe deze begrippen in de praktijk kunnen worden toegepast, zie blz. 65. Ook zal ik dan bespreken wat de voordelen van een object georiënteerde benadering zijn ten opzichte van een gestructureerde of ADT benadering.

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

Door middel van *overerving* (Engels: *inheritance*) kun je een “nieuwe variant” van een bestaande class definiëren zonder dat je de bestaande class hoeft te wijzigen en zonder dat er code gekopieerd wordt. De class die als uitgangspunt gebruikt wordt, wordt de *base* class genoemd (of ook wel parent class of super class). De class die hiervan afgeleid (Engels: derived) wordt, wordt de *derived* class genoemd (of ook wel child class of sub class). Schematisch kan dit zoals hiernaast getekend worden aangegeven. In C++ code wordt dit als volgt gedeclareerd:




```
class Base {
    // ...
};
class Derived: public[64] Base { // Derived is afgeleid van Base
    // ...
};
```

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

De derived class erft alle datamembers van de base class over. Dit wil zeggen dat een object van een derived class (minimaal) dezelfde datamembers heeft als een object van de base class. Private datamembers uit de base class zijn in objecten van de derived class wel aanwezig maar kunnen vanuit memberfuncties van de derived class **niet** rechtstreeks bereikt worden. In de derived class kun je bovendien “extra” datamembers toevoegen.

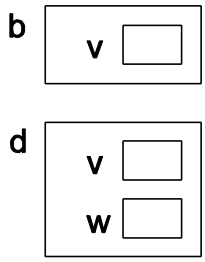
Als de objecten `b` en `d` op onderstaande wijze gedefinieerd zijn dan kan de structuur van deze objecten weergegeven worden zoals daarnaast getekend is. Het object `b` bevat alleen een datamember `v` en een object `d` bevat zowel een datamember `v` als een datamember `w`. Het datamember `v` is alleen toegankelijk vanuit de memberfuncties van de class `Base` en het datamember `w` is alleen toegankelijk vanuit de memberfuncties van de class `Derived`.

---

[63] De hier gebruikte tekennotatie heet UML (Unified Modelling Language) en is een standaard notatie die veel bij object georiënteerd ontwerpen wordt gebruikt. Bij het onderwijsdeel SOPX3E1C1 in EH3C&D of bij verschillende onderwijsdelen in IH2 komen we hier uitgebreid op terug.

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

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

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

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



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

Als de objecten `b` en `d` op onderstaande wijze gedefinieerd zijn dan kan de memberfunctie `getV` zowel op het object `b` als op het object `d` uitgevoerd worden. De memberfunctie `getW` is vanzelfsprekend alleen op het object `d` uit te voeren. De private memberfunctie `setV` is alleen aan te roepen vanuit de andere memberfuncties van de class `Base` en de private memberfunctie `setW` is alleen aan te roepen vanuit de andere memberfuncties van de class `Derived`.

```
class Base {
public:
    // ...
    int getV() const { return v; }[65]
private:
    void setV(int i) { v=i; }
    int v;
};

class Derived: public Base {
public:
    // ...
    int getW() const { return w; }
private:
    void setW(int i) { w=i; }
    int w;
};

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

Een derived class heeft (minimaal) dezelfde datamembers en (minmaal) dezelfde public memberfuncties als de base class waarvan hij is afgeleid. Om deze reden mag je een object van de derived class ook gebruiken als de compiler een object van de base class verwacht.[66] De relatie tussen de derived class en de base class wordt een IS-A (is een) relatie genoemd. De derived class is een (speciaal geval van) base

[65] Alle memberfuncties zijn in dit voorbeeld inline gedefinieerd (zie blz. 36). De enige reden hiervoor is dat ik mezelf wat typewerk heb willen besparen.

[66] Maar pas op voor het slicing probleem dat ik later (blz. 77) zal bespreken.

class. Omdat een derived class datamembers en memberfuncties kan toevoegen aan de base class is het omgekeerde niet waar. Je kunt een object van de base class niet gebruiken als de compiler een object van de derived class verwacht. Een base class IS-NOT-A derived class.

5.2 Polymorphism. (Zie eventueel TICPP Chapter15.html.)

Als de classes `Base` en `Derived` zoals hierboven gedefinieerd zijn, dan kan een pointer van het type `Base*` niet alleen wijzen naar een object van de class `Base` maar ook naar een object van de class `Derived`. Want een `Derived` IS-A (is een) `Base`. Omdat een pointer van het type `Base*` naar objecten van verschillende classes kan wijzen wordt zo'n pointer een *polymorphic* (veelvormige) pointer genoemd. Evenzo kan een reference van het type `Base&` niet alleen verwijzen naar een object van de class `Base` maar ook naar een object van de class `Derived`. Want een `Derived` IS-A (is een) `Base`. Omdat een reference van het type `Base&` naar objecten van verschillende classes kan verwijzen wordt zo'n reference een *polymorphic* (veelvormige) reference genoemd. Later in dit hoofdstuk zal blijken dat *polymorphism* de kern is waar het bij OOP om draait[67]. Door het toepassen van polymorphism kan je software maken die eenvoudig aangepast, uitgebreid en hergebruikt kan worden. (Zie het uitgebreide voorbeeld op blz. 65.)

Als ik nu de volgende functie definieer:

```
void drukVaf(const Base& p) {
    cout<<p.getV();
}
```

Dan kun je deze functie dus niet alleen gebruiken voor een object van de class `Base` maar ook voor een object van de class `Derived`. Dit kan omdat de parameter `p` van de functie polymorphic is. De functie wordt daardoor dus zelf ook polymorphic (veelvormig).

5.3 Memberfunctie overriding.

Een derived class kan, zoals we al hebben gezien, datamembers en memberfuncties toevoegen aan de base class. Een derived class kan bovendien memberfuncties die in de base class geïmplementeerd zijn in de derived class een andere implementatie geven (*overridden*). Dit kan alleen als de base class de memberfunctie *virtual* heeft gedeclareerd.[68]

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

class Derived: public Base {
public:
    virtual[69] void printName() {
        cout<<"Ik ben een Derived."<<endl;
```

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

[68] Dit is niet waar. Maar als je een non-virtual memberfunctie uit de base class in de derived class toch opnieuw implementeert dan wordt de functie in de base class niet overridden maar overloaded. Overlaoding geeft onverwachte (en meestal ongewenste) effecten zoals je later (blz. 73) zult zien.

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

```
    }
};
```

Als via een polymorphic pointer een memberfunctie wordt aangeroepen dan wordt de memberfunctie van de class van het object waar de pointer naar wijst aangeroepen. Omdat een polymorphic pointer tijdens het uitvoeren van het programma naar objecten van verschillende classes kan wijzen, kan de keuze van de memberfunctie pas tijdens het uitvoeren van het programma worden bepaald. Dit wordt "*late binding*" of ook wel "*dynamic binding*" genoemd. Op soortgelijke wijze kan een polymorphic reference verwijzen naar een object van verschillende classes. Als via deze polymorphic reference een memberfunctie wordt aangeroepen dan wordt de memberfunctie van de class van het object waar de reference naar verwijst aangeroepen. Ook in dit geval is er sprake van "late binding".

Voorbeeld van het gebruik van polymorphism:

```
Base b;
Derived d;
Base* bp1(&b);
Base* bp2(&d);
bp1->printName();
bp2->printName();
```

Uitvoer:

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

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

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

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

Voorbeeld van het gebruik van polymorphism:

```
Base b;
Derived d;
Base* bp1(&b);
Base* bp2(&d);
bp1->printName();
bp2->printName();
```

Uitvoer:

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

Aan het herdefiniëren van memberfuncties moeten bepaalde voorwaarden gesteld worden (zoals geformuleerd door Liskov) om er voor te zorgen dat er geen problemen ontstaan bij polymorphic gebruik van de class. Simpel gesteld luidt de regel van Liskov: "*Een object van de derived class moet op alle plaatsen waar een object van de base class verwacht wordt gebruikt kunnen worden.*". Het is belangrijk om goed te begrijpen wanneer inheritance wel/niet gebruikt moet worden. Bedenk dat overerving altijd een *type-relatie* oplevert. Als class `Derived` overerft van class `Base` dan geldt "`Derived` is een `Base`". Dat wil zeggen dat elke bewerking die op een object (variabele) van class (type) `Base` uitgevoerd kan worden ook op een object (variabele) van class (type) `Derived` uitgevoerd moet kunnen worden. In de class `Derived` moet je alleen datamembers en memberfuncties toevoegen en/of virtual memberfuncties overridden, maar nooit memberfuncties van de class `Base` overloaden. Het verkeerd gebruik van inheritance is een van de meest voorkomende fouten bij OOP. Een leuke vraag op dit gebied is: is een struisvogel een vogel? (Oftewel mag een class `struisvogel` overerven van class `vogel`?) Het antwoord is afhankelijk van de declaratie van `vogel`. Als een `vogel` een (non-virtual) memberfunctie `vlieg()` heeft waarmee het dataveld `hoogte > 0` wordt, dan niet! In dit geval heeft de ontwerper van de class `vogel` een fout gemaakt.

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

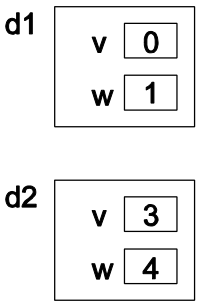
Het is mogelijk om een virtual memberfunctie in een Base class alleen maar te declareren en nog niet te implementeren. Dit wordt dan een "*pure virtual*" memberfunctie genoemd en de betreffende base class wordt een zogenaamde "*Abstract Base Class (ABC)*". Een virtual memberfunctie kan pure virtual gemaakt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

5.5 Constructors en destructors bij inheritance.

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

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

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

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

Derived d1;
```



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

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

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

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

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

5.7 Voorbeeld: ADC kaarten.

Ik zal nu een praktisch programmeerprobleem beschrijven. Vervolgens zal ik een gestructureerde oplossing, een oplossing door middel van een ADT en een object georiënteerde oplossing bespreken.[70] Daarna zal ik deze oplossingen met elkaar vergelijken (nu mag je één keer raden welke oplossing de beste zal blijken te zijn:-).

5.7.1 Probleemdefinitie.

In een programma om een machine te besturen moeten bepaalde signalen via een ADC kaart (ADC = Analoog Digitaal Converter) ingelezen worden. Het programma moet met 2 verschillende typen ADC kaarten kunnen werken. Deze kaarten hebben de typenamen AD178 en NI323. Deze kaarten zijn functioneel gelijk en hebben beide een 8 kanaals 16 bits ADC met instelbare voorversterker. Het initialiseren van de kaarten, het selecteren van een kanaal, het uitlezen van de “sampled” waarde en het instellen van de versterkingsfactor moet echter bij elke kaart op een andere wijze gebeuren (andere adressen, andere bits en/of andere procedures). In de applicatie moeten meerdere ADC kaarten van verschillende typen gebruikt kunnen worden. In de applicatie is alleen de spanning in volts van de verschillende signalen van belang. Voor beide 16 bits ADC kaarten geldt dat deze spanning U als volgt berekend kan worden: U=S*F/6553.5 [V]. S is de “sampled” 16 bits waarde (two’s complement) en F is de ingestelde versterkingsfactor. Hoe kun je in het programma nu het beste met de verschillen tussen de kaarten omgaan.

5.7.2 Een gestructureerde oplossing.

Eerst zal ik bespreken hoe je dit probleem op een gestructureerde manier oplost. Met de methode van functionele decompositie deel ik het probleem op in een aantal deelproblemen. Voor elk deelprobleem definieer ik vervolgens een functie:

- `initCard` voor het initialiseren van de kaart,

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

- `selectChannel` voor het selecteren van een kanaal,
- `getChannel` voor het opvragen van het op dit moment geselecteerde kanaal,
- `setAmplifier` voor het instellen van de versterkingsfactor,
- `sampleCard` voor het uitlezen van een sample en
- `readCard` voor het uitlezen van de spanning in volts.

Voor elke kaart die in het programma gebruikt wordt moeten een aantal gegevens bijgehouden worden zoals: de kaartsoort, de ingestelde versterkingsfactor en het geselecteerde kanaal. Om deze gegevens per kaart netjes bij elkaar te houden heb ik de struct `ADCCard` gedeclareerd. Voor elke kaart die in het programma gebruikt wordt, wordt dan een variabele van dit struct type aangemaakt. Aan elk van de eerder genoemde functies wordt de te gebruiken kaart dan als parameter van het type struct `ADCCard` doorgegeven.

```
enum CardName {AD178, NI323};

struct ADCCard {
    CardName t;      // card type
    double f;        // amplifying factor
    int c;           // selected channel
};

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

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

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

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

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

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

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

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

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

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

Hopelijk heb je al lang zelf de nadelen van deze aanpak bedacht:

- Iedere programmeur die gebruikt maakt van het type `struct ADCCard` kan een waarde toekennen aan de datavelden. Zo zou een programmeur die de struct `ADCCard` gebruikt in plaats van de functie `selectChannel` het statement `++adc.c;` kunnen bedenken om het geselecteerde kanaal met 1 te verhogen. Dit werkt natuurlijk niet omdat dan alleen het in de struct opgeslagen kanaalnummer verhoogd wordt terwijl in werkelijkheid geen ander kanaal geselecteerd wordt.
- Iedere programmeur die gebruikt maakt van het type `struct ADCCard` kan er voor kiezen om zelf de code voor het inlezen van een spanning in volts “uit te vinden” in plaats van gebruik te maken van de functie `readCard`. Er valt dus niet te garanderen dat altijd de juiste formule wordt gebruikt. Ook niet als we wel kunnen “garanderen” dat de functies `readCard` en `sampleCard` correct zijn.
- Iedere programmeur die gebruikt maakt van het type `struct ADCCard` zal zelf nieuwe bewerkingen (zoals bijvoorbeeld het opvragen van de ingestelde versterkingsfactor) definiëren. Het zou beter zijn als alleen de programmeur die verantwoordelijk is voor het onderhouden van het type `struct ADCCard` (en de bijbehorende bewerkingen) dit kan doen.
- Als een nieuw type 8 kanaals 16 bits ADC met instelbare voorversterker (typenaam BB647) in het programma ook gebruikt moet kunnen worden, dan moet ten eerste het enumeratie type `CardName` uitgebreid worden. Bovendien moeten alle functies, waarin door middel van een `switch` afhankelijk van het kaart type verschillende code worden uitgevoerd, gewijzigd en opnieuw gecompileerd worden.

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

5.7.3 Een oplossing door middel van een ADT.

Deze problemen kunnen voorkomen worden als een ADT gebruikt wordt, waarin zowel de data van een kaart als de functies die op een kaart uitgevoerd kunnen worden, ingekapseld zijn[71].

```
enum CardName {AD178, NI323};

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

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

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

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

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

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

int ADCCard::sample() const {
    int sample;
```

[71] De I/O registers van de ADC kaart zelf zijn helaas niet in te kapselen. We kunnen dus niet voorkomen dat een programmeur in plaats van de ADT `ADCCard` te gebruiken rechtstreeks de kaart aanspreekt.

```
//   ... zelfde switch instructie als bij initCard
     return sample;
}

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

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

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

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

Aan deze oplossingsmethode zit echter nog steeds het volgende nadeel:

- Als een nieuw type 8 kanaals 16 bits ADC met instelbare voorversterker (typenaam BB647) in het programma ook gebruikt moet kunnen worden, dan moet ten eerste het enumeratie type `CardName` uitgebreid worden. Bovendien moeten alle memberfuncties, waarin door middel van een `switch` afhankelijk van het kaart type verschillende code worden uitgevoerd, gewijzigd en opnieuw gecompileerd worden.

Op zich is dit nadeel bij deze oplossing minder groot dan bij de gestructureerde oplossing omdat in dit geval alleen de ADT `ADCCard` aangepast hoeft te worden, terwijl in de gestructureerde oplossing de gehele applicatie (kan enkele miljoenen regels code zijn) doorzocht moet worden op het gebruik van het betreffende `switch` statement. Toch kan ook de oplossing door middel van een ADT tot een probleem voor wat betreft de uitbreidbaarheid leiden.

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

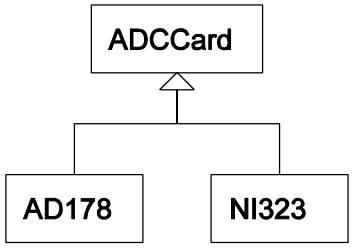
- De prijs van de ADT zal waarschijnlijk zodanig zijn dat zelf maken al snel duurder wordt. Als de prijs van de bovenstaande ADT 250 Euro is (een redelijke schatting), dan betekent dit dat je zelf (als beginnende professionele programmeur) de ADT in één dag moet ontwerpen, implementeren, testen en documenteren om goedkoper uit te zijn.[72]
- De gekochte ADT is hoogst waarschijnlijk uitgebreid getest. Zeker als het product al enige tijd bestaat zullen de meeste bugs er inmiddels uit zijn. De kans dat de applicatie plotseling niet meer werkt als de kaarten in een andere PC geprikt worden is met een zelf ontwikkelde ADT groter dan met een gekochte ADT.
- Als de leverancier van de AD178 kaart een hardware bug oplost waardoor ook de software aansturing gewijzigd moet worden dan zal de leverancier van de ADT (hopelijk) ook een nieuwe versie uitbrengen.

[72] Een dag werk van een beginnende professionele programmeur kost de werkgever ongeveer 100 Euro (exclusief de benodigde koffie).

Het is waarschijnlijk dat de leverancier van de ADT alleen de `adccard.h` file en de `adccard.obj` file aan ons levert maar de source code `adccard.cpp` file niet aan ons beschikbaar stelt[73]. Het toevoegen van de nieuwe ADC kaart (BB647) is dan niet mogelijk.

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

5.7.4 Een object georiënteerde oplossing.



Bij het toepassen van de object georiënteerde benadering constateer ik dat in het programma twee typen ADC kaarten gebruikt worden. Beide kaarten hebben dezelfde functionaliteit en kunnen dus worden afgeleid van dezelfde base class[74]. Dit is hier schematisch weergeven. Ik heb de base class als volgt gedeclareerd:

```
class ADCCard {
public:
     ADCCard();
     virtual ~ADCCard();
     virtual void selectChannel(int channel) =0;
     int getChannel() const;
     virtual void setAmplifier(double factor) =0;
     double read() const;
protected:
     void rememberChannel(int channel);
     void rememberAmplifier(double factor);
private:
     double f;         // amplifying factor
     int c;            // selected channel
     virtual int sample() const =0;
};
```

De memberfuncties `selectChannel`, `setAmplifier` en `sample` heb ik pure virtual (zie blz. 64) gedeclareerd omdat de implementatie per kaart type verschillend is. Dit maakt de class `ADCCard` abstract. Je kunt dus geen objecten van dit type definiëren, alleen references en pointers. Elke afgeleide concrete class (elk kaarttype) moet deze functies “overridden”. De destructor heb ik virtual gedeclareerd[75] om het genereren van een non-virtual default destructor te voorkomen. Waarom dit nodig is zal ik op blz. 78 bespreken. De memberfuncties `getChannel` en `read` heb ik non-virtual gedeclareerd omdat het niet mijn bedoeling is dat een derived class deze functies “override”. Wat er gebeurt als je dit toch probeert zal ik op blz. 73 bespreken. De datamembers `f` en `c` heb ik protected gedeclareerd om er voor te zorgen dat ze vanuit de derived classes bereikbaar zijn. De memberfunctie `sample` heb ik private gedefinieerd omdat deze functie alleen vanuit de memberfunctie `read` gebruikt hoeft te kunnen worden. Derived classes moeten deze memberfunctie dus wel definiëren maar ze mogen hem zelf niet aanroepen! Ook gebruikers van deze derived classes hebben mijn inziens de memberfunctie `sample`

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

[74] Het bepalen van de benodigde classes en hun onderlinge relaties is bij grotere programma’s niet zo eenvoudig. Het bepalen van de benodigde classes en hun relaties wordt OOA (Object Oriented Analyse) en OOD (Object Oriented Design) genoemd. Hoe je dit moet doen wordt in deze onderwijseenheid niet behandeld. Bij E C&D en TI komt dit later wel aan de orde.

[75] Ik heb de destuctor niet *pure* virtual gedefinieerd. Dit wil zeggen dat een class die van de class `ADCCard` overerft de destructor mag overridden maar niet verplicht is om dit te doen.

niet nodig. Ze worden dus door mij verplicht de memberfunctie `read` te gebruiken zodat de returnwaarde altijd in volts is (ik hoop hiermee fouten bij het gebruik te voorkomen).

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

```
class AD178: public ADCCard {
public:
    AD178();
    virtual void selectChannel(int channel);
    virtual void setAmplifier(double factor);
private:
    virtual int sample() const;
};
class NI323: public ADCCard {
public:
    NI323();
    virtual void selectChannel(int channel);
    virtual void setAmplifier(double factor);
private:
    virtual int sample() const;
};
```

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

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

De diverse classes heb ik als volgt geïmplementeerd:

```
ADCCard::ADCCard(): f(1.0), c(1) {
    // voor alle kaarten benodigde code
}
int ADCCard::getChannel() const {
    return c;
}
double ADCCard::read() const {
    return sample()*f/6553.5;
}
void ADCCard::rememberChannel(int channel) {
    c=channel;
}
void ADCCard::rememberAmplifier(double factor) {
    f=factor;
}

AD178::AD178() {
    // de specifieke voor de AD178 benodigde code
}
void AD178::selectChannel(int channel) {
    rememberChannel(channel);
    // de specifieke voor de AD178 benodigde code
}
void AD178::setAmplifier(double factor) {
    rememberAmplifier(factor);
    // de specifieke voor de AD178 benodigde code
}
int AD178::sample() const {
```

```
    int sample;
    // de specifieke voor de AD178 benodigde code
    return sample;
}

NI323::NI323() {
    // de specifieke voor de NI323 benodigde code
}
void NI323::selectChannel(int channel) {
    rememberChannel(channel);
    // de specifieke voor de NI323 benodigde code
}
void NI323::setAmplifier(double factor) {
    rememberAmplifier(factor);
    // de specifieke voor de NI323 benodigde code
}
int NI323::sample() const {
    int sample;
    // de specifieke voor de NI323 benodigde code
    return sample;
}
```

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

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

Deze functie kan ik dan als volgt gebruiken:

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

Merk op dat door de overloaded `operator<<` in de functie `doIt` de in de base class `ADCCard` gedefinieerde memberfunctie `read` aangeroepen wordt die op zijn beurt de in de **derived** class gedefinieerde memberfunctie `sample` aanroept.

5.7.5 Een kaart toevoegen.

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

```
class BB647: public ADCCard {
public:
    BB647();
    virtual void selectChannel(int channel);
    virtual void setAmplifier(double factor);
private:
    virtual int sample() const;
};
```

Met de volgende implementatie:

```
BB647::BB647() {
    // de specifieke voor de BB647 benodigde code
}
void BB647::selectChannel(int channel) {
    rememberChannel(channel);
    // de specifieke voor de BB647 benodigde code
}
void BB647::setAmplifier(double factor) {
    rememberAmplifier(factor);
    // de specifieke voor de BB647 benodigde code
}
int BB647::sample() const {
    int sample;
    // de specifieke voor de NI323 benodigde code
    return sample;
}
```

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

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

Als alle functie en class declaraties in aparte `.h` en alle functie en class definities in aparte `.cpp` file opgenomen zijn dan hoeft alleen de nieuwe class en de nieuwe `main` functie opnieuw vertaald te worden. De rest van het programma kan dan eenvoudig (zonder hercompilatie) mee gelinkt worden. Dit voordeel komt voort uit het feit dat de functie `doIt` polymorphic is. Aan de parameter die gedefinieerd is als een `ADCCard&` kun je objecten van elke van deze class afgeleide classes (`AD178`, `NI323` of `BB647`) gebruiken. Je ziet dat de object georiënteerde oplossing een zeer goed **onderhoudbaar** en **uitbreidbaar** programma oplevert.

5.8 Overloading en overriding van memberfuncties.

Het onderscheid tussen memberfunctie overloading en memberfunctie overriding is van groot belang. Op blz. 15 heb je gezien dat een functienaam meerdere keren gebruikt (overloaded) kan worden. De compiler zal aan de hand van de gebruikte argumenten de juiste functie selecteren. Dit maakt deze functies eenvoudiger te gebruiken omdat de gebruiker (de programmeur die deze functies aanroept) slechts 1 naam hoeft te onthouden. Elke functienaam, dus ook een memberfunctienaam, kan overloaded worden. Een memberfunctie die een andere memberfunctie *overload* heeft dus dezelfde naam. Bij een aanroep van de memberfunctienaam wordt de juiste memberfunctie door de compiler aangeroepen door naar de argumenten bij aanroep te kijken.

Voorbeeld[76] van het gebruik van overloading van memberfuncties:

```
class Class {
public:
```

[76] In dit voorbeeld (en ook in enkele volgende voorbeelden) heb ik de memberfunctie definitie in de class definitie opgenomen. Dit maakt deze memberfuncties inline, zie blz. 36. De enige reden om dit te doen is de besparing van wat typewerk.

```
    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[77] { // 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)'[78]
    d.f(3);
    d.Base::f();[79]
// ...
```

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

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

[79] 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`).

Uitvoer:

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

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

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

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

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

Uitvoer:

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

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

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

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

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

Op blz. 62 heb je gezien dat een **virtual** gedefinieerde memberfunctie in een derived class overridden kan worden. Een memberfunctie die in een derived class een memberfunctie uit de base class *override*

moet dezelfde naam en dezelfde parameters hebben[80]. Alleen memberfuncties van een "ouder of voorouder class" kunnen overridden worden in de "kind class". Als een overridden memberfunctie via een polymorphic pointer of reference (zie blz. 62) aangeroepen wordt, dan wordt tijdens het uitvoeren van het programma bepaald naar welk type object de pointer wijst (of de reference refereert). Pas daarna wordt de in deze class gedefinieerde memberfunctie aangeroepen.

Als er dus twee memberfuncties zijn met dezelfde naam en dezelfde parameters in een base en in een derived class dan wordt bij een aanroep van de memberfunctienaam de juiste memberfunctie:

- door de compiler aangeroepen door de naar het statische type van het object waarop de memberfunctie wordt uitgevoerd te kijken als de memberfuncties niet virtual zijn (er is dan sprake van overloading) of
- naar het dynamische type van het object waarop de memberfunctie wordt uitgevoerd te kijken als de memberfunctie wel virtual zijn (er is dan sprake van overriding).[81] [82]

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

```
class Base {
public:
     void f(int i) const {
          cout<<"Base::f(int) called."<<endl;
     }
     virtual void g(int i) const {
          cout<<"Base::g(int) called."<<endl;
     }
     // ...
};

class Derived: public Base {
public:
     void f(int i) const {
          cout<<"Derived::f(int) called."<<endl;
     }
     virtual void g(int i) const {
          cout<<"Derived::g(int) called."<<endl;
     }
     // ...
};

int main() {
     Base b;
     Derived d;
     Base* pb=&d;
     b.f(3);
     d.f(3);
     pb->f(3);
     b.g(3);
     d.g(3);
     pb->g(3);
```

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

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

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

```
    pb->Base::g(3);
// ...
```

Uitvoer:

```
Base::f(int) called.
Derived::f(int) called.
Base::f(int) called.
Base::g(int) called.
Derived::g(int) called.
Derived::g(int) called.
Base::g(int) called.
```

5.9 Slicing problem. (Zie eventueel TICPP Chapter15.html#Heading450.)

Een object van een class `Derived`, die public overerft van class `Base`, mag worden toegekend aan een object van de class `Base` (`b=d`). Er geldt immers: een `Derived` is een `Base`. Dit levert problemen op als het object van de class `Derived` meer geheugenruimte inneemt dan een object van de class `Base`. Dit is het geval als in class `Derived` (extra) datamembers zijn opgenomen. Deze extra datamembers kunnen niet aan het object van class `Base` toegekend worden omdat het object van class `Base` hier geen ruimte voor heeft. Dit probleem wordt het "*slicing problem*" genoemd. Het is dus aan te raden om nooit een object van een derived class toe te kennen aan een object van de base class.

Voorbeeld van slicing:

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

class Docent: public Mens {
public:
    Docent(): salaris(30000) {
    }
    virtual void printSoort() {
        cout<<"Docent.";
    }
    virtual void printSalaris() {
        cout<<"Salaris = "<<salaris;
    }
    virtual void verhoogSalarisMet(unsigned short v) {
        salaris+=v;
    }
    // ...
private:
    unsigned short salaris[83];
};

int main() {
```

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

```
    Docent Bd;
    Bd.printSoort(); cout<<" ";
    Bd.printSalaris(); cout<<endl;
    Bd.verhoogSalarisMet(10000);
    Bd.printSalaris(); cout<<endl;

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

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

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

Uitvoer:

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

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

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

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

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

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

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

```
    virtual void printSoort() {
        cout<<"Peer."<<endl;
    }
    // ...
};

class FruitMand {
public:
    ~FruitMand() {
        for (int i(0); i<aantal; ++i)
            delete fp[i];
    }
    FruitMand(): aantal(0) {
    }
    void voegToe(Fruit* p) {
        if (aantal<100)
            fp[aantal++]=p;
        else
            cout<<"De mand is al vol."<<endl;
    }
    void printInhoud() {
        cout<<"De fruitmand bevat:"<<endl;
        for (int i(0); i<aantal; ++i)
            fp[i]->printSoort();
    }
private:
    int aantal;
    Fruit* fp[100]; // kan niet meer dan 100 stuks fruit bevatten
};

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

De uitvoer van dit programma is als volgt:

```
De fruitmand bevat:
Appel.
Peer.
Appel.
Er is een Appel verwijderd.
Er is een stuk Fruit verwijderd.
Er is een Peer verwijderd.
Er is een stuk Fruit verwijderd.
Er is een Appel verwijderd.
Er is een stuk Fruit verwijderd.
```

Als in de base class `Fruit` geen virtual destructor gedefinieerd wordt maar een gewone (non-virtual) destructor dan wordt de uitvoer als volgt:

```
De fruitmand bevat:
Appel.
Peer.
Appel.
Er is een stuk Fruit verwijderd.
Er is een stuk Fruit verwijderd.
```

```
Er is een stuk Fruit verwijderd.
```

Dit komt doordat de destructor via een polymorphic pointer (zie blz. 62) aangeroepen wordt. Als de destructor virtual gedefinieerd is dan wordt tijdens het uitvoeren van het programma bepaald naar welk type object (een appel of een peer) deze pointer wijst. Vervolgens wordt de destructor van deze class (`Appel` of `Peer`) aangeroepen[84]. Omdat de destructor van een derived class ook altijd de destructor van zijn base class aanroept (zie blz. 64) wordt de destructor van `Fruit` ook aangeroepen. Als de destructor niet virtual gedefinieerd is dan wordt tijdens het compileren van het programma bepaald van welk type de pointer is. Vervolgens wordt de destructor van deze class (`Fruit`) aangeroepen[85]. In dit geval wordt dus alleen de destructor van de base class aangeroepen.

5.11 Voorbeeld: Impedantie calculator.

In dit uitgebreide praktijkvoorbeeld kun je zien hoe de OOP technieken die je tot nu toe hebt geleerd kunnen worden toegepast bij het maken van een elektrotechnische applicatie.

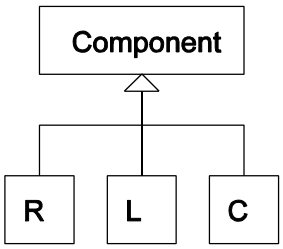
5.11.1 Weerstand, spoel en condensator.

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

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

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

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



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

```
class Component {
public:
     virtual ~Component() {
     }
     virtual complex<double> Z(double f) const=0;
     virtual void print(ostream& o) const=0;
};
```

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

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

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

---

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

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

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

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

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

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

Merk op dat de functie `printImpedanceTable` niet “weet” welke `Component` gebruikt wordt. Dit betekent dat deze polymorphic functie voor alle huidige “soorten” componenten te gebruiken is. De functie is zelf ook voor toekomstige “soorten” componenten bruikbaar. Dit maakt het programma eenvoudig uitbreidbaar.

De uitvoer van het bovenstaande programma is:

```
Impedantie tabel voor: R(100)

freq    Z
   10   (100,0)
  100   (100,0)
 1000   (100,0)
10000   (100,0)
100000  (100,0)
1e+06   (100,0)

Impedantie tabel voor: C(1e-05)

freq    Z
   10   (0,-1591.55)
  100   (0,-159.155)
 1000   (0,-15.9155)
10000   (0,-1.59155)
100000  (0,-0.159155)
1e+06   (0,-0.0159155)

Impedantie tabel voor: L(0.001)

freq    Z
   10   (0,0.0628319)
```

```
  100   (0,0.628319)
 1000   (0,6.28319)
10000   (0,62.8319)
100000  (0,628.319)
1e+06   (0,6283.19)
```

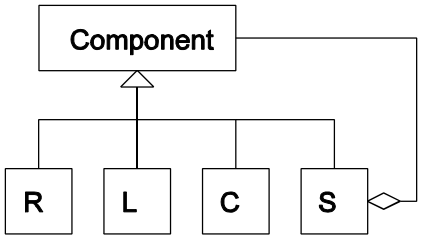
Vraag:
Implementeer nu zelf de classes `R`, `C` en `L`.

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

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

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

5.11.2 Serie- en parallelschakeling.

Natuurlijk wil je het programma nu uitbreiden zodat je ook de impedantie van serieschakelingen kunt berekenen. Je moet jezelf dan de volgende vragen stellen:

- Is een serieschakeling een component?
- Heeft een serieschakeling een (of meer) component(en)?



Dat een serieschakeling **bestaat uit** componenten zal voor iedereen een duidelijk zijn. Het antwoord op de eerste vraag is misschien moeilijker. De ABC `Component` is gedefinieerd als “iets” dat een impedantie heeft en dat geprint kan worden. Als je bedenkt dat een serieschakeling ook een impedantie heeft en ook geprint kan worden zal duidelijk zijn dat een serieschakeling een soort component **is**. De class `S` (serieschakeling) kan dus van de class `Component` afgeleid worden. Dit heeft als bijkomend voordeel dat je het aantal componenten waaruit een serieschakeling bestaat tot 2 kunt beperken. Als je dan een serieschakeling wilt doorrekenen van een `R`, `L` en `C` maak je bijvoorbeeld eerst van de `R` en `L` een serieschakeling, die je vervolgens met de `C` combineert tot een tweede serieschakeling. Op soortgelijke wijze kun je het programma uitbreiden met parallelschakelingen. Met dit programma kun je dan van elk passief elektrisch netwerk de impedantie berekenen.

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

```
int main() {
    R r1(1E2);
    C c1(1E-6);
    L l1(3E-2);
    S s1(r1, c1);
    S s2(r1, l1);
    P p(s1, s2);
    printImpedanceTable(p);
// ...
```

Je ziet dat je de al bestaande polymorphic functie `printImpedanceTable` ook voor objecten van de nieuwe classes `S` en `P` kunt gebruiken!

De uitvoer van het bovenstaande programma is:

```
Impedantie tabel voor: ((R(100)+C(1e-06))//(R(100)+L(0.03)))

freq    Z
   10   (100.016,1.25659)
  100   (101.591,12.5146)
 1000   (197.893,-14.3612)
10000   (101.132,-10.5795)
100000  (100.011,-1.061)
1e+06   (100,-0.106103)
```

Vraag:
Implementeer nu zelf de classes `S` en `P`.

Antwoord:

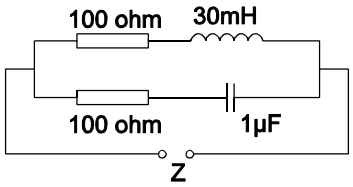
```
class S: public Component { // S = Serieschakeling van 2 componenten
public:
    S(const Component& c1, const Component& c2): comp1(c1), comp2(c2) {
    }
    virtual complex<double> Z(double f) const {
        return comp1.Z(f)+comp2.Z(f);
    }
    virtual void print(ostream& o) const {
      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

```
    const Component& comp2;
    S(const S&);              // voorkom gebruik
    void operator=(const S&); // voorkom gebruik
};

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

5.11.3 Een grafische impedantie calculator.

Op http://bd.thrijswijk.nl/sopx2/prog/impcalc vind je een Windows applicatie waarmee de absolute waarde van de impedantie van een in te voeren netwerk als functie van de frequentie grafisch weergegeven kan worden. Dit programma (her)gebruikt de hierboven gedefinieerde classes R, L, C, S en P. Op de website http://bd.thrijswijk.nl/sopx2 kun je, als je dat leuk vind, zelf leren hoe je met Borland C++ Builder grafische applications kunt maken.

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



5.12 Inheritance details.

Over inheritance valt nog veel meer te vertellen:

- Private en protected inheritance. Een manier van overerven waarbij de *is-een* relatie niet geldt. Kan meestal vervangen worden door composition (*heeft-een*).
- Multiple inheritance. Overerven van meerdere classes tegelijk.
- Virtual inheritance. Speciale vorm van inheritance nodig om bepaalde problemen bij multiple inheritance op te kunnen lossen.

Voor al deze details verwijs ik je naar TICPP Chapter14.html en Chapter15.html.

Practicumhandleiding.

1 Opdracht 1: Gebruik van `string` en `iostream`.

Deze opdracht is bedoeld om je C kennis op te frissen en behandelt daarnaast enkele kleine verbeteringen van C++ ten opzichte van C. In de propedeuse heb je leren werken met character array's voor het opslaan van strings. In de standaard C++ library is het type `string` gedefinieerd dat veel eenvoudiger te gebruiken is dan character array's. In C gebruik je de `stdio` library voor input en output bewerkingen. In de C++ standaard is de `iostream` library gedefinieerd die eenvoudiger te gebruiken is dan de `stdio` library. Deze nieuwe library is bovendien uitbreidbaar (zoals we in opdracht 2 zullen zien).

Lees voordat je met deze opgave gaat beginnen eerst hoofdstuk 1 van dit dictaat door.

1.1 Het `string` type.

We zullen eerst de problemen met strings in C op een rijtje zetten en daarna het nieuwe `string` type uit C++ bespreken.

1.1.1 De problemen met strings in C.

De programmeertaal C heeft geen "echt" string type. Als je in C een string (rij karakters) wilt opslaan dan doe je dat in een character array. In C geldt de afspraak dat elke string wordt afgesloten door een zogenaamd *nul-karakter* `'\0'`. Voorbeeld:

```
char naam[] = "Harry"; // deze array bevat 6 karakters!
```

Een string in C heeft de volgende problemen:
- Een character array is *statisch* (de lengte wordt tijdens het compileren bepaald).
- Het nul-karakter als afsluiting is onhandig (de maximale lengte van de string is 1 minder dan de lengte van de array).
- De operatoren zijn niet bruikbaar (in plaats daarvan moeten de `strxxx` functies gebruikt worden).

```
#include <string.h>
#include <iostream.h>

int main() {
     char naam[10];
     naam = "Harry"; // Error! Zie opmerking 1.
     strcpy(naam, "Harry"); // OK
     cout<<naam<<endl;
     strcpy(naam, "Willem-Alexander"); // Error! Zie opmerking 2.
     strcpy(naam, "Alex"); // OK
     cout<<naam<<endl;
     if (naam == "Alex") { // Error! Zie opmerking 3.
     //  ...
     }
     if (strcmp(naam, "Alex") == 0) { // OK
     //  ...
     }
}
```

Opmerkingen:
1) Deze regel geeft de volgende foutmelding: "Lvalue required". Aan een array variabele (`naam`) kun je namelijk niets toekennen. Als je de array wilt vullen dan moet je dat met de functie `strcpy` (gedefinieerd in `<string.h>`) doen.

2) Deze regel geeft tijdens het compileren geen foutmelding. Tijdens het uitvoeren van het programma kan[86] het programma echter vastlopen of zelfs spontaan de harde schijf gaan formatteren! Dit komt doordat er 17 karakters (even natellen, en het nul-karakter niet vergeten) naar de array `naam` worden gekopieerd. Terwijl er maar 10 karakters gereserveerd zijn. In C wordt hier echter helemaal niet op gecontroleerd en de 7 extra karakters worden gewoon naar het geheugen geschreven (achter de 10 gereserveerde karakters). Dit kan tot gevolg hebben dat andere delen van je programma (of zelfs andere programma's) plotseling niet meer correct werken omdat hun geheugen overschreven wordt.
3) Ook deze regel geeft tijdens het compileren geen foutmelding. De vergelijking `naam == "Alex"` levert echter altijd `false` op! Als je twee array variabelen met elkaar vergelijkt dan worden hun *adressen* met elkaar vergeleken. Als je de *inhoud* van de array's met elkaar wilt vergelijken moet je de functie `strcmp` (gedefinieerd in `<string.h>`) gebruiken.

1.1.2 De oplossing in C++: het type `string`.

De programmeertaal C++ heeft wel een "echt" string type[87]. Dit type is gedefinieerd in de standaard library. Voorbeeld:

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

string naam("Harry"); // deze string bevat 5 karakters.
```

Een `string` in C++ heeft de volgende voordelen ten opzichte van een string uit C:
- Een `string` is *dynamisch* (de lengte kan tijdens het uitvoeren van het programma, indien nodig, worden aangepast, er is dus ook geen maximale lengte!).
- De operatoren zijn gewoon bruikbaar.
- Het type `string` bevat veel extra functionaliteit.

```
#include <string> // Zie opmerking 1.
#include <iostream>
using namespace std;

int main() {
     string naam;
     naam = "Harry"; // Zie opmerking 2.
     cout<<naam<<endl;
     naam = "Willem-Alexander"; // Zie opmerking 3.
     cout<<naam<<endl;
     if (naam == "Willem-Alexander") { // Zie opmerking 4.
          cout<<"Hoi Alex!"<<endl;
     }
     return 0;
}
```

Opmerkingen:
1) De C++ include file `<string>` is dus heel wat anders dan de C include file `<string.h>`. Als je in een C++ programma toch de oude C strings wilt gebruiken (om oude C code te hergebruiken) dan kun je de oude `strxxx` functies includen met de include file `<cstring>`.
2) Je kunt gewoon een waarde toekennen aan een variabele van het type `string` met behulp van de `operator =`.

---

[86] De wet van Murphy zegt: Het programma zal pas vastlopen door deze fout als je het aan de klant demonstreert!

[87] Later zul je leren dat `string` geen "ingebouwd" type is maar een "zelfgemaakt type" een zogenaamde `class`. Voor het gebruik maakt dat echter niet uit.

3) Het type `string` is dynamisch en “groeit” als dat nodig is!
4) Je kunt variabelen van het type `string` gewoon vergelijken met behulp van de operator `==`.

1.1.3 Je eerste stap op weg naar object oriëntatie.

Nu komt de verrassing: een variabele van het type (eigenlijk de class) `string` is geen gewone variabele maar een *object*! Wat dat precies betekent wordt in hoofdstuk 3 van dit dictaat uitgebreid behandeld. Op dit moment zullen we alleen bekijken wat dit betekent voor het gebruik van objecten (variabelen) van de class (het type) `string`.

Objecten zijn vergelijkbaar met gewone variabelen: ze hebben een naam en je kunt er “iets” in opslaan. Maar met objecten kun je iets wat met gewone variabelen niet kan. Je kunt objecten boodschappen (*messages*) sturen.

Je kunt een message naar een object sturen om het object een *vraag* te stellen. Als je wilt weten hoeveel karakters een object van de class `string` bevat dan kun je dat object de message `size` sturen. Het antwoord op deze message is dan een integer die het aantal karakters weergeeft.

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

int main() {
    string naam("Willem-Alexander");
    cout<<"De naam "<<naam<<" bevat "<<naam.size()<<" karakters."<<endl;
    cin.get();
    return 0;
}
```

Uitvoer:
```
De naam Willem-Alexander bevat 16 karakters.
```

De syntax voor het versturen van een message is:
```
naam-van-object.naam-van-message(parameters)
```
Dus eerst de naam van het object waar naar toe de message verstuurd moet worden (dat object wordt de *receiver* van de message genoemd) dan een punt gevolgd door de naam van de message en tot slot een haakje openen en een haakje sluiten met daartussen eventuele *parameters*. De message `size` heeft geen parameters. Het versturen van een message lijkt een beetje op het aanroepen van een functie. In C++ wordt het versturen van een message meestal het aanroepen van een *memberfunctie* genoemd. Toch is er een duidelijk verschil tussen een memberfunctie (message) en een functie: een memberfunctie heeft een receiver en een gewone functie niet!

Je kunt ook een message naar een object sturen om het object iets te laten *doen*. Na afloop van de memberfuntie is het object dan veranderd. Als je bijvoorbeeld iets aan een object van de class `string` wilt toevoegen dan kun je dat object de message `append` sturen. De `string` die moet worden toegevoegd moet je als argument meesturen.

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

int main() {
    string naam("Willem-Alexander");
    naam.append(" en Maxima");
    cout<<naam<<endl;
    cin.get();
    return 0;
```

```
}
```

Uitvoer:
```
Willem-Alexander en Maxima
```

1.1.4 De mogelijkheden van het type `string`.

Voor het complete overzicht verwijs ik je naar de helpfile van de class `string`. Als je in Borland C++ Builder het woordje `string` intypt en op F1 drukt moet je kiezen voor `basic_string`[88] voor het complete overzicht. In deze paragraaf worden enkele voorbeelden gegeven.

- **Vergelijken** met operatoren **`> < >= <= == !=`** (spreekt voor zich).
- **Losse karakters opvragen** met de operator `[]`.
  ```
  char c(s[2]);
  ```
  Als `s` gelijk is aan `Maxima` dan wordt het karakter `c` gelijk aan `x`.
- **Losse karakters vervangen** met de operator `[]`.
  ```
  s[3]='a';
  ```
  Als `s` gelijk is aan `Maxima` dan wordt het `s` gelijk aan `Maxama`.
- **Toekennen** met operator `=` of met `assign`. Met `assign` kun je ook een deel van de ene `string` aan de andere `string` toekennen.
  ```
  s1.assign(s2, 7, 4);
  ```
  s1 wordt gelijk aan het deel van `s2` dat begint op positie 7 en 4 karakters lang is. Als `s2` gelijk is aan `Willem-Alexander` dan zal `s1` gelijk worden aan `Alex`.
- **Toevoegen** met operatoren **`+`** en **`+=`** of met `append`. Met `append` kun je ook een deel van de ene `string` achter de andere `string` plakken.
  ```
  string s1("Willem");
  s1+="-Alexander";
  string s2("Maxima");
  s2+=" en ";
  s2.append(s1, 7, 4);
  ```
  `s2` wordt gelijk aan `Maxima en Alex`
- **Invoegen** met `insert`.
  ```
  s1.insert(4, s2);
  ```
  Voeg `s2` op positie 4 in `s1` in. Als `s1` gelijk is aan `Maxima` en `s2` gelijk is aan `Willem-Alexander` dan wordt `s1` gelijk aan `MaxiWillem-Alexanderma`.
  ```
  s1.insert(4, s2, 7, 4);
  ```
  Als `s1` gelijk is aan `Maxima` en `s2` gelijk is aan `Willem-Alexander` dan wordt `s1` gelijk aan `MaxiAlexma`.
- **Verwijderen** met `erase`.
  ```
  s1.erase(3, 10);
  ```
  Als `s1` gelijk is aan `Willem-Alexander` dan wordt `s1` gelijk aan `Wilder`.
- **Vervangen** met `replace`.
  ```
  s1.replace(0, 4, "Jongs");
  ```
  Als `s1` gelijk is aan `Maxima` dan wordt `s1` gelijk aan `Jongsma`.
- **Gedeelte opvragen** met `substr`.
  ```
  s1=s2.substr(7);
  ```
  `s1` wordt gelijk aan het deel van `s2` dat begint op positie 7. Als `s2` gelijk is aan `Willem-Alexander` dan zal `s1` gelijk worden aan `Alexander`.
  Je kunt ook als tweede parameter opgeven hoeveel karakters er gekopieerd moeten worden:
  ```
  s1=s2.substr(7, 4);
  ```
  `s1` wordt gelijk aan het deel van `s2` dat begint op positie 7 en 4 karakters lang is. Als `s2` gelijk is aan `Willem-Alexander` dan zal `s1` gelijk worden aan `Alex`.

[88] De typenaam `string` blijkt een andere naam te zijn voor het template type `basic_string`. Het begrip template wordt pas in hoofdstuk 4 van dit dictaat behandeld.

- **Zoeken** met `find`.
  ```
  int i(s1.find("Alex"));
  ```
  Als s1 gelijk is aan `Willem-Alexander` dan wordt `i` gelijk aan 7. Als de string `Alex` niet in `s1` voorkomt krijgt `i` de waarde `string::npos` (een in de class `string` gedefinieerde constante). Eigenlijk moeten we deze regel als volgt programmeren:
  ```
  string::size_type i(s1.find("Alex"));
  ```
  In de class `string` wordt namelijk het type `size_type` gedefinieerd dat gebruikt moet worden om indexwaarden van een string in op te slaan.
  Als tweede parameter kun je een index meegeven, het zoeken begint dan bij die index.
  ```
  string::size_type i(s1.find("le"));
  string::size_type j(s1.find("le", 7));
  ```
  Als `s1` gelijk is aan `Willem-Alexander` dan wordt `i` gelijk aan 3 en `j` gelijk aan 8.
  Zoals je hebt gezien doorzoekt `find` de string van voor naar achter. Je kunt de memberfunctie `rfind` gebruiken om van achter naar voor te zoeken.
- **Zoeken** met `find_first_of`.
  ```
  string::size_type i(s1.find_first_of("aeiou"));
  ```
  Zoek de eerste letter uit `s1` die voorkomt in de als parameter meegegeven `string`. Als `s1` gelijk is aan `Maxima` dan wordt `i` gelijk aan 1. Als de string `s1` geen van de karakters `a`, `e`, `i`, `o` of `u` bevat krijgt `i` de waarde `string::npos`.
  Als tweede parameter kun je een index meegeven, het zoeken begint dan bij die index.
  Het gebruik van de functies: `find_last_of`, `find_first_not_of` en `find_last_not_of` spreekt denk ik voor zich.
- **Conversie** naar `const char*` met `c_str()`.
  Op deze manier kun je functies die met een (ouderwetse character array (C-string)) werken toch gebruiken met het nieuwe C++ type `string`.

1.2 De `iostream` library.

Zoals je in paragraaf 1.7 hebt gelezen gebruiken we in C++ de input/output library `iostream` in plaats van de C library `stdio`. Het zal je niet verbazen dat de variabelen `cin` en `cout` die je in paragraaf 1.7 hebt leren kennen geen variabelen maar objecten zijn. Je kunt deze objecten (net als objecten van de class `string`) dus ook messages sturen. Elke class definieert echter zijn eigen messages. Het object `cout` is een object van de class `ostream` en het object `cin` is een object van de class `istream`[89]. Naar een object van de class `string` kun je bijvoorbeeld de message `size` sturen om te vragen hoeveel karakters het object bevat. Naar `cout` kun je deze message echter niet sturen (dit object begrijpt deze message niet). De aanroep `cout.size()` geeft tijdens het compileren de volgende foutmelding: `'size' is not a member of 'ostream'`. Welke messages je naar `cout` en `cin` kunt sturen kun je opzoeken in de helpfile van de class `ostream` respectievelijk `istream`. Bijvoorbeeld:

```
cout.fill('#');
cout.width(10);
int i(189);
cout<<i<<endl;
```

Output:
```
#######189
```

1.3 Voorbeeldprogramma.

Tot slot van deze inleiding volgt nog een voorbeeld waarin met objecten van de class `string` wordt gewerkt. Dit programma kun je op het practicum kopiëren vanaf het Internet
http://bd.thrijswijk.nl/sopx2/pract/opd1a.cpp

[89] Later zal blijken dat dit niet helemaal klopt. Zie ook voetnoot 48.

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

int main () {
    cout<<"Geef je email adres: ";
    string mailAdres;
    cin>>mailAdres;
    string::size_type indexAapje(mailAdres.find("@"));
    if (indexAapje!=string::npos) {
        cout<<"Gebruiker: "
            <<mailAdres.substr(0, indexAapje)
            <<endl;
        cout<<"Machine:   "
            <<mailAdres.substr(indexAapje+1)
            <<endl;
    }
    else {
        cout<<mailAdres<<" is geen geldig email adres!"<<endl;
    }
    cout<<"Druk op de return-toets."<<endl;
    cin.get();
    cin.get();
    return 0;
}
```

1.4 Opdrachtomschrijving.

Deze opdracht bestaat uit de deelopdrachten 1a tot en met 1c. Opdrachten 1b en 1c moet je laten afteke-nen door de docent.

Opdracht 1a.

Compileer en test het bovenstaande programma (http://bd.thrijswijk.nl/sopx2/pract/opd1a.cpp) met behulp van Borland C++ Builder. Op de webpagina http://bd.thrijswijk.nl/sopx2/builder.htm kun je een inleiding in het gebruik van C++ Builder vinden.

Opdracht 1b.

Het adres van een webpagina (een URL) kan opgesplitst worden in 4 delen. Het protocol, de machine, de directory en de file.
Bijvoorbeeld:
`http://bd.thrijswijk.nl/sopx2/pract/opd1a.cpp`
kan worden gesplitst in:

```
protocol:  http
machine:   bd.thrijswijk.nl
directory: sopx2/pract
file:      opdr1a.cpp
```

Schrijf een programma dat een ingetypte URL splitst op de hierboven beschreven methode.

Opdracht 1c.

Soms is alleen 1 van de 4 onderdelen (protocol, machine, directory of file) nodig. Het ontleden van de hele URL is dan overbodig. Schrijf één of meer functies zodanig dat het volgende testprogramma (http://bd.thrijswijk.nl/sopx2/pract/opd1c.cpp) correct werkt. Lees indien nodig eerst paragraaf 1.10 en 1.11 van het dictaat.

```
int main() {
    string tURL("http://bd.thrijswijk.nl/sopx2/pract/opd1c.cpp");
    cout<<"test 1:"<<endl; OntleedUrl(tURL);
    cout<<"test 2:"<<endl; OntleedUrl(tURL, "file");
    cout<<"test 3:"<<endl; OntleedUrl(tURL, "directory");
    cout<<"test 4:"<<endl; OntleedUrl(tURL, "machine");
    cout<<"test 5:"<<endl; OntleedUrl(tURL, "protocol");
    cin.get();
    return 0;
}
```

De uitvoer moet dan zijn:

```
test 1:
protocol:  http
machine:   bd.thrijswijk.nl
directory: sopx2/pract
file:      opdr1c.cpp
test 2:
file:      opdr1c.cpp
test 3:
directory: sopx2/pract
test 4:
machine:   bd.thrijswijk.nl
test 5:
protocol:  http
```

2 Opdracht 2: Abstract Data Type.

Deze opdracht bestaat uit de deelopdrachten 2a t/m 2e. Alle deelopdrachten moet je laten aftekenen door de docent.

Op pagina 7 van dit dictaat wordt gebruik gemaakt van variabelen waarin een tijdsduur in uren en minuten kan worden opgeslagen. Je hebt in de propedeuse geleerd dat een tijdsduur door middel van een struct weergegeven kan worden. In de theorielessen heb ik besproken dat deze manier van werken programma's oplevert die niet goed onderhoudbaar, uitbreidbaar en herbruikbaar zijn. Je kunt een tijdsduur beter als ADT definiëren.

Het ADT `Tijdsduur` moet als volgt gebruikt kunnen worden http://bd.thrijswijk.nl/sopx2/pract/tijd1.cpp:

```
// ...
int main() {
    Tijdsduur t1(3,50);          // t1 is 3 uur en 50 minuten
    cout<<"t1 = "; t1.print(); cout<<endl;
    const Tijdsduur kw(15);      // kw is 15 minuten
    cout<<"kw = "; kw.print(); cout<<endl;
    t1.erbij(kw);                // Tel kw bij t1 op
    cout<<"t1 = "; t1.print(); cout<<endl;
```

```
    Tijdsduur t2(t1);           // t2 is een kopie van t1
    t2.eraf(kw);                // Trek kw van t2 af
    cout<<"t2 = "; t2.print(); cout<<endl;
    t2.maal(7);                 // Vermenigvuldig t2 met 7
    cout<<"t2 = "; t2.print(); cout<<endl;
    Tijdsduur t3(3,-122);       // t3 is 3 uur minus 122 minuten
    cout<<"t3 = "; t3.print(); cout<<endl;
    Tijdsduur t4(3,122);        // t4 is 3 uur plus 122 minuten
    cout<<"t4 = "; t4.print(); cout<<endl;
    cout<<"Druk op de return-toets."<<endl;
    cin.get();
    return 0;
}
```

De uitvoer moet dan zijn:

```
t1 =   3 uur en  50 minuten
kw =             15 minuten
t1 =   4 uur en   5 minuten
t2 =   3 uur en  50 minuten
t2 =  26 uur en  50 minuten
t3 =             58 minuten
t4 =   5 uur en   2 minuten
```

Opdracht 2a.

Ontwerp en implementeer een ADT (Abstract Data Type) genaamd `Tijdsduur` (een aantal uren en minuten). Zorg ervoor dat het bovenstaande programma tijd1.cpp zonder warnings compileert en de gewenste uitvoer produceert. Zoek (indien nodig) inspiratie bij de in de les behandelde class `Breuk`.

Tip: Zorg ervoor dat de opgeslagen minuten altijd >= 0 en < 60 zijn. Het volgende code-fragment kan hierbij gebruikt worden:

```
while (min<0) {
      min+=60;
      uur-=1;
}
while (min>59) {
      min-=60;
      uur+=1;
}
```

Het ADT `Tijdsduur` kan veel eenvoudiger gebruikt worden als we ervoor zorgen dat een variabele van het zelfgemaakte type `Tijdsduur` gebruikt kan worden op dezelfde wijze als een variabele van het standaard type `int`.

Het ADT `Tijdsduur` kan dan eenvoudig als volgt gebruikt worden
http://bd.thrijswijk.nl/sopx2/pract/tijd2.cpp:

```
// ...
int main() {
    Tijdsduur t1(3,50);         // t1 is 3 uur en 50 minuten
    cout<<"t1 = "<<t1<<endl;
    const Tijdsduur kw(15);     // kw is 15 minuten
    cout<<"kw = "<<kw<<endl;
    t1+=kw;                     // Tel kw bij t1 op
    cout<<"t1 = "<<t1<<endl;
```

```
    Tijdsduur t2(t1-kw);        // t2 is t1 min kw
    cout<<"t2 = "<<t2<<endl;
    t2*=7;                      // Vermenigvuldig t2 met 7
    cout<<"t2 = "<<t2<<endl;
    Tijdsduur t3(t2-122);       // t3 is t2 minus 122 minuten
    cout<<"t3 = "<<t3<<endl;
    Tijdsduur t4(722-t1);       // t4 is 722 minuten minus t1
    cout<<"t4 = "<<t4<<endl;
    if (t3!=t4)
        cout<<"t3 is ongelijk aan t4."<<endl;
    if (t3>t4)
        cout<<"t3 is groter dan t4."<<endl;
    cin.get();
    return 0;
}
```

De uitvoer moet dan zijn:

```
t1 =   3 uur en  50 minuten
kw =             15 minuten
t1 =   4 uur en   5 minuten
t2 =   3 uur en  50 minuten
t2 =  26 uur en  50 minuten
t3 =  24 uur en  48 minuten
t4 =   7 uur en  57 minuten
t3 is ongelijk aan t4.
t3 is groter dan t4.
```

Opdracht 2b.

Pas het ADT `Tijdsduur` uit opdracht 2a aan. Zorg ervoor dat het bovenstaande programma tijd2.cpp zonder warnings compileert en de gewenste uitvoer produceert. Zoek (indien nodig) inspiratie bij de in de les behandelde class `Breuk`.

Opdracht 2c.

Voeg de volgende destructor toe aan het ADT `Tijdsduur` uit opdracht 2b.

```
Tijdsduur::~Tijdsduur() {
      cout<<"Er is een tijdsduur verwijderd: "<<*this;
      cout<<" Type return:";
      cin.get();
}
```

Deze destructor zorgt ervoor dat de tekst "Er is een tijdsduur verwijderd: " gevolgd door de waarde van de verwijderde tijdsduur op het scherm afgedrukt wordt en daarna gewacht wordt op het invoeren van een return.

Verklaar de uitvoer!

Een slimme programmeur bedenkt dat het veel handiger is om een tijdsduur (alleen) als een aantal minuten op te slaan. Bij het afdrukken kan dit aantal minuten dan worden omgezet naar een aantal uren en een resterend aantal minuten (kleiner dan 60). Dit heeft twee voordelen:

- Een variabele van het ADT `Tijdsduur` neemt nu minder geheugenruimte in.
- De bewerkingen met variabelen van het ADT `Tijdsduur` kunnen nu eenvoudiger gecodeerd worden. Deze bewerkingen zullen dus sneller uitgevoerd worden.

De programmeurs die het ADT `Tijdsduur` gebruiken mogen natuurlijk niets merken van deze aanpassing. De programma's die gebruik maken van dit ADT moeten natuurlijk wel opnieuw gecompileerd worden.

Opdracht 2d.

Pas het ADT `Tijdsduur` uit opdracht 2c aan zodat dit type nog slechts één private datamember van het type `int` heeft (de tijdsduur uitgedrukt in minuten). Zorg ervoor dat het bovenstaande programma tijd2.cpp zonder warnings compileert en nog steeds de gewenste uitvoer produceert.

Opdracht 2e.

Splits het programma uit opdracht 2d in 2 source files main.cpp en tijdsduur.cpp en 1 header file tijdsduur.h. Zie paragraaf 3.30 in dit dictaat. Je kunt in Borland C++ Builder verschillende source files combineren tot 1 executable door meerdere files toe te voegen aan een project met behulp van de knop  op de knoppenbalk of met de sneltoets Shift+F11. Je kunt ook de project manager opstarten door middel van de menukeuze View, Project Manager.

3 Opdracht 3: Inheritance and polymorphism.

Deze opdracht bestaat uit de deelopdrachten 3a t/m 3c. Opdracht 3b en 3c moet je laten aftekenen door de docent. Over de andere deelopgave kunnen wel vragen worden gesteld.

In een bedrijf werken verschillende soorten werknemers:

- **free lance werknemers**. Deze werknemers verdienen een vast bedrag per gewerkt uur.
- **vaste krachten**. Deze werknemers verdienen een vast bedrag per maand.

Binnen de salarisadministratie van het bedrijf heeft elke werknemer een registratienummer.

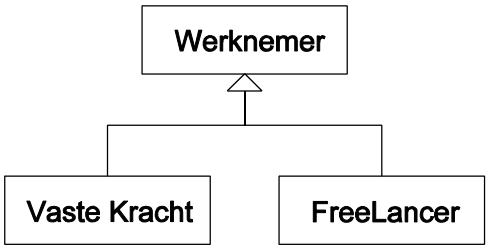
De analist die de specificaties voor het salarisadministratie programma heeft opgesteld heeft bedacht dat er rekening mee moet worden gehouden dat er in de toekomst andere soorten werknemers moeten worden toegevoegd. Bijvoorbeeld stukwerkers die werken voor een vast stukprijs. Het programma moet dus zoveel mogelijk onafhankelijk van het concrete werknemer type gemaakt worden.

Hieronder volgt een voorbeeld van een functie die onafhankelijk van het werknemer type is:

```
void printMaandSalaris(const Werknemer& w) {
    cout<<"Werknemer: "<<w.geefNummer()
        <<" verdient: "<<setw(8)<<setprecision(2)<<fixed
        <<w.geefSalaris()<<" Euro."<<endl;
}
```

Deze functie wordt vanuit de functie `main` als volgt aangeroepen
http://bd.thrijswijk.nl/sopx2/pract/werk1.cpp:

```
int main() {
   FreeLancer f(1, 25.75);    // werknemer 1 verdient 25.75 per uur
   VasteKracht v(2, 1873.53); // werknemer 2 verdient 1873.53 per maand

   f.werkUren(84);            // werknemer 1 werkt (deze maand) 84 uren

   cout<<"Maand 1:"<<endl;
   printMaandSalaris(f);
   printMaandSalaris(v);

   f.werkUren(13);            // werknemer 1 werkt (deze maand) 13 uren

   cout<<"Maand 2:"<<endl;
   printMaandSalaris(f);
   printMaandSalaris(v);

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



Denk er aan hoe een blaft! Bestudeer indien nodig hoofdstuk 5 uit het dictaat. De relatie tussen de verschillende classes is hiernaast gegeven.

Opdracht 3A.

Implementeer de drie benodigde classes: `Werknemer`, `FreeLancer` en `VasteKracht` zodat het bovenstaande hoofdprogramma uitgevoerd kan worden. Denk aan het correct gebruik van `const`. In de class `Werknemer` moet je een virtuele destructor definiëren. Snap je waarom?

De juiste uitvoer is:

```
Maand 1:
Werknemer: 1 verdient: 2163.00 Euro.
Werknemer: 2 verdient: 1873.53 Euro.
Maand 2:
Werknemer: 1 verdient: 334.75 Euro.
Werknemer: 2 verdient: 1873.53 Euro.
```

Merk op dat de functie `printMaandSalaris` niet “weet” welk soort `Werknemer` gebruikt wordt. Dit betekent dat deze polymorphic functie voor alle huidige “soorten” werknemers te gebruiken is. De functie is zelf ook voor toekomstige “soorten” werknemers bruikbaar. Dit maakt het programma eenvoudig uitbreidbaar. Op dit moment is het nodig om het programma uit te breiden met een nieuw “soort” werknemer. Deze stukwerker werkt voor een vast stukprijs.

De `main` functie wordt nu als volgt uitgebreid:

```
int main() {
   FreeLancer f(1, 25.75);    // werknemer 1 verdient 25.75 per uur
   VasteKracht v(2, 1873.53); // werknemer 2 verdient 1873.53 per maand
   StukWerker s(3, 1.05);     // werknemer 3 verdient 1.05 per stuk
```

```
   f.werkUren(84);            // werknemer 1 werkt 84 uren
   s.produceerStuks(1687);    // werknemer 3 produceert 1687 stuks

   cout<<"Maand 1:"<<endl;
   printMaandSalaris(f);
   printMaandSalaris(v);
   printMaandSalaris(s);

   f.werkUren(13);            // werknemer 1 werkt 13 uren
   s.produceerStuks(0);       // werknemer 3 produceert 0 stuks

   cout<<"Maand 2:"<<endl;
   printMaandSalaris(f);
   printMaandSalaris(v);
   printMaandSalaris(s);

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

Opdracht 3B.

Implementeer de benodigde class `StukWerker` zodat het bovenstaande hoofdprogramma uitgevoerd kan worden.

De juiste uitvoer is:

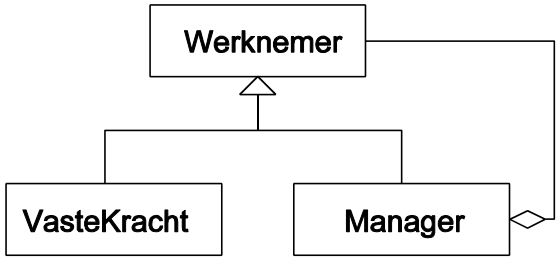
```
Maand 1:
Werknemer: 1 verdient: 2163.00 Euro.
Werknemer: 2 verdient: 1873.53 Euro.
Werknemer: 3 verdient: 1771.35 Euro.
Maand 2:
Werknemer: 1 verdient: 334.75 Euro.
Werknemer: 2 verdient: 1873.53 Euro.
Werknemer: 3 verdient: 0.00 Euro.
```

Er is nog een “soort” werknemer waarmee we het programma kunnen uitbreiden. Een manager is een werknemer die andere werknemers leiding geeft. Het bedrijf dat we als voorbeeld nemen heeft als regel dat een manager nooit aan meer dan 10 werknemers direct leiding geeft. De meeste managers hebben zelf ook weer een manager boven zich. Het bedrijf dat we als voorbeeld nemen heeft een orginele manier bedacht om het salaris van een manager te bepalen. Het salaris van de manager is 2 maal zo hoog als het gemiddelde salaris van de mensen waaraan deze manager direct leiding geeft.

Het programma moet natuurlijk ook het salaris van managers kunnen berekenen. De relatie tussen manager en werknemer is als volgt:

- Een manager is een werknemer.
- Een manager heeft een (of meer) werknemer(s) (onder zich).

De relatie tussen de classes `Werknemer`, `VasteKracht` en `Manager` is hiernaast gegeven.



Het hoofdprogramma wordt nu:

```
int main() {
```

```
   StukWerker s(1, 1.05);     // werknemer 1 verdient 1.05 per stuk
   FreeLancer f(2, 25.75);    // werknemer 2 verdient 25.75 per uur
   VasteKracht v1(3, 1873.53);// werknemer 3 verdient 1873.53 per maand

   Manager m1(4);             // werknemer 4 is de manager van:
   m1.geeftLeidingAan(s);     // -  werknemer 1
   m1.geeftLeidingAan(f);     // -  werknemer 2
   m1.geeftLeidingAan(v1);    // -  werknemer 3

   VasteKracht v2(5, 2036.18);// werknemer 5 verdient 2036,18 per maand
   Manager m2(6);             // werknemer 6 is de manager van:
   m2.geeftLeidingAan(v2);    // -  werknemer 6
   m2.geeftLeidingAan(m1);    // -  werknemer 4

   s.produceerStuks(678);     // werknemer 1 produceert 678 stuks
   f.werkUren(84);            // werknemer 2 werkt 84 uren

   printMaandSalaris(s);
   printMaandSalaris(f);
   printMaandSalaris(v1);
   printMaandSalaris(m1);
   printMaandSalaris(v2);
   printMaandSalaris(m2);

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

Opdracht 3C.

Implementeer de benodigde class `Manager` zodat het bovenstaande hoofdprogramma uitgevoerd kan worden. Maak bij de implementatie van een `Manager` gebruik van een array met maximaal 10 pointers naar `Werknemer`'s.

De juiste uitvoer is:

```
Werknemer: 1 verdient: 711.90 Euro.
Werknemer: 2 verdient: 2163.00 Euro.
Werknemer: 3 verdient: 1873.53 Euro.
Werknemer: 4 verdient: 3165.62 Euro.
Werknemer: 5 verdient: 2036.18 Euro.
Werknemer: 6 verdient: 5201.80 Euro.
```