Programmeren met stijl in C++

Jan Cnops en Kris Coolsaet
15 februari 2016

Universiteit Gent
Faculteit Ingenieurswetenschappen en Architectuur
Vakgroep Industriële Technologie en Constructie
Academiejaar 2015-2016

INLEIDING

Een programmeertaal kennen is één ding — kunnen programmeren is iets anders. Een taal zoals C++ bevat verschillende manieren waarmee je hetzelfde idee kan uitdrukken, en hoewel ze allemaal wellicht tot een 'werkend' programma zullen leiden, zijn ze toch niet allemaal even 'goed'. Je hebt misschien een methode gebruikt die veel minder efficiënt is, die wel veertig lijnen vraagt in plaats van tien of die obscure C++-opdrachten gebruikt waarvan niemand nog de betekenis kent. Het belangrijkste kenmerk van een programma is leesbaarheid. Code die een paar regels langer is dan andere kan betere code zijn omdat ze duidelijker is. Soms gebruikt men methodes die iets meer computertijd vragen omdat ze duidelijker zijn en gemakkelijker te testen.

Vaak is je programma enkel begrijpelijk voor jezelf. Nochtans wordt er tegenwoordig geen enkel programma van professionele kwaliteit nog door één enkele programmeur geschreven (en in grotere projecten wordt het altijd door iemand anders gecontroleerd en getest). Softwareontwikkeling is een groepsgebeuren geworden. Je programma's moeten dus niet alleen duidelijk zijn voor je computer, maar ook voor je medewerkers en opvolgers. Vergeet trouwens de belangrijkste 'verborgen' medewerker niet: je toekomstige zelf!

Deze nota's introduceren een aantal programmeertechnieken waar wij voorstander van zijn en die wij natuurlijk ook zelf in alle lessen en oefeningensessies toepassen. Ze zijn gegroeid uit een jarenlange programmeerervaring van verschillende mensen. Een aantal van deze stijlelementen zijn trouwens opgenomen als onderdeel van programmeertalen zoals Java en C♯ – de jongere neven van C en C++. Dit geeft aan hoe ook internationale experts het op dit gebied met ons eens zijn.

Deze regels zijn wat ons betreft helemaal niet vrijblijvend! Wij beschouwen ze als even belangrijk als de kennis van de programmeertaal zelf en we *verplichten* je om ze in al je programma's toe te passen. Dit lijkt misschien streng, maar het zal gemakkelijker zijn om later in het professionele leven eventjes de teugels te vieren dan omgekeerd. Wellicht zal je daar niet eens meer de neiging toe voelen want een consistente programmeerstijl heeft vele voordelen: je programmeert vlugger, je maakt minder fouten en je spoort eventuele fouten ook veel sneller op. En zoals men in het onderwijs nogal veel hoort verklaren: je zal er ons later dankbaar voor zijn!

HOOFDSTUK 1
VORM

1.1 BLADSPIEGEL

Een goed programma heeft een bladspiegel waaruit je onmiddellijk de interne structuur van het programma kan afleiden. Bovendien helpen sommige programmeeromgevingen en editors je. Je kan netjes indenteren, bijvoorbeeld met de `tab`toets, en in sommige gevallen kan je een overzicht maken van je programma, met een lijst van procedures en klassen. Maar dan moet je je wel aan de regels houden.

Het sleutelwoord is *indentatie*: pas de linkermarge steeds zodanig aan dat de binnenkant van een samengestelde opdracht (zoals een `if`, `while` of `for`) zich meer naar rechts bevindt dan de hoofding van de opdracht zelf. De standaardbreedte hiervoor is bij de meeste editors 3 of 4 spaties: minder is niet duidelijk, en meer zorgt dat je in de problemen komt bij verneste opdrachten.

```
cin >> getal;
while (getal != 0){
    aantal++;
    if (getal % 2 == 0)
        someven=someven + getal;
    else
        somoneven=somoneven + getal;
    cin >> getal;
}
```

De *eindaccolade* van (de binnenkant van) een opdracht plaats je onder de eerste letter van de hoofding van de opdracht. Zo zie je onmiddellijk waar ze bij hoort. De *beginaccolade* plaats je op de hoofdinglijn zoals hierboven. Hier is één uitzondering op. Bij het definiëren van functies en klassen plaatsen sommigen de beginaccolade aan het begin van een nieuwe lijn. Er zijn programma's die code samenvatten door alleen functiehoofdingen te geven en die deze conventie gebruiken. Als je dit doet plaats je de beginaccolade onder de eerste letter van de functieopdracht (of de `class`declaratie). De eerste regel code volgt dan ofwel op dezelfde lijn, ofwel op de volgende. Er zijn dus drie mogelijkheden:

```
int main(){
    cout<<"dag wereld";
}

int main()
{   cout<<"dag wereld";
}

int main()
{
    cout<<"dag wereld";
}
```

Volgens bovenstaande regels moet je een `if`opdracht die binnen een `if` of een `else` ligt verder indenteren. Er is een uitzondering op de regel: als we drie of meer

gelijkwaardige mogelijkheden hebben schrijven we ze met dezelfde indentatie. Een voorbeeld:

```
if (getal==0)
    cout<<"het is nul";
else if (getal<0)
    cout<<"het is kleiner dan nul";
else
    cout<<"het is groter dan nul";
```

Wanneer er zich slechts één opdracht binnen de `if`-, `while`- of `for`structuur bevindt, dan mag je de accolades weglaten. De indentatie moet je echter behouden. Wanneer het om een hele korte opdracht gaat dan kan je het geheel eventueel op één lijn plaatsen. Overdrijf hier echter niet. Dus niet

```
while (i--) if (tab[i]>0) aant++;        //VERKEERD
```

maar wel

```
while (i--)
    if (tab[i]>0)
        aant++;
```

Vermijd ook om meer dan één opdracht op dezelfde lijn te plaatsen. Is een opdracht te lang om op één lijn te passen, indenteer dan het tweede stuk en splits de lijn op een logische plaats. Schrijf dus niet

```
if ((get >= 0 && get < 10) || (get       //VERKEERD
>= 20 && get < 30)) {                    //VERKEERD
    cout << get << endl;                 //VERKEERD
}                                        //VERKEERD
```

maar wel

```
if ( (get >= 0 && get < 10) ||
     (get >= 20 && get < 30) ) {
    cout << get << endl;
}
```

Merk hier ook op dat de voortzetting van een opdrachtlijn op een eigen manier inspringt: verder naar rechts dan gewone indentatie. Er zijn geen vaste regels voor, maar men probeert het netjes te houden: bij meervoudige condities zet men de condities onder elkaar, bij een lange parameterlijst de parameters van de tweede lijn onder die van de eerste, bij een toekenning begint men rechts van het gelijkheidsteken (of iets verder, zoals bij het drukken van formules gebruikelijk is):

```
if ( (get >= 0 && get < 10) ||
     (get >= 20 && get < 30) )
    stelNieuweLijstOp(oudeLijst, aantalElementen,
                      dag, verder, lijsttype);
else
    dagorder=voorstel*kernfunctie - 23
             + dagtekening/hoofdpremie;
```

Afzonderlijke procedure- en functiedefinities moeten in één oogopslag kunnen bekeken worden. Als vuistregel mag één definitie niet meer dan één scherm beslaan — dus ongeveer 20 lijnen (dit is een oude regel). Heb je meer lijnen nodig, dan betekent dit wellicht dat je je programma nog verder in afzonderlijke deelprogramma's moet opsplitsen. Voor de hoofding van een functiedefinitie zet je een blanco lijn.

Binnen een programma kan je hier en daar een blanco lijn toevoegen: dit heeft ongeveer hetzelfde effect als een nieuwe paragraaf beginnen in een tekst. Doe het dan ook als je een reeks bij elkaar horende opdrachten hebt, zoals een rij declaraties.

Binnen opdrachten kan je ook blanco's gebruiken om alles meer leesbaar te maken. We geven hier geen vaste regels (experten zijn het er niet over eens), maar waarschijnlijk is het best om blanco's te gebruiken om elementen van de opdracht in groepjes in te delen. Sleutelwoorden van C++, zoals `if`, `while` en `for` zet je best voor de duidelijkheid tussen blanco's. Teveel wit kan storend zijn. Vergelijk maar eens de drie mogelijkheden voor een programmafragment dat te maken heeft met de vergelijking $ax^2 + bx + c$.

```
if((b*b-4*a*c>0)&&(a!=0))
    brandpt=-b/a+2*a;
else
    brandpt=0;

if ((b*b-4*a*c > 0) && (a!=0))
    brandpt= -b/a + 2*a;
else
    brandpt=0;

if ( ( b * b - 4 * a * c > 0 ) && ( a != 0) )
    brandpt = - b / a + 2 * a;
else
    brandpt = 0;
```

1.2 NAAMKEUZE

De regels voor naamgeving in Java zijn zeer streng, en leiden vaak tot lange namen (dat is ook wel nodig als je veel namen hebt). In C++ gaat het er iets losser aan toe, maar toch zijn een aantal dingen belangrijk. Een programma wordt heel wat leesbaarder als je de juiste namen kiest voor je variabelen en functies. Zorg ervoor dat de namen *zinvol* en *leesbaar* zijn. Wil je voorbeelden van onduidelijke naamgeving, kijk dan eens in `string.h`, waar functies zoals `strncmp` en `strcspn` gedefinieerd zijn.

Wat hoofdletters betreft zijn er in C++ twee regels: de namen van constanten worden in hoofdletters geschreven, en klassennamen beginnen soms met een hoofdletter. Bij namen die uit meerdere woorden bestaan gebruikt men klassiek (ook in C) onderstreepjes `titel_hoofdstuk`. Voornamelijk onder invloed van Java gebruikt men ook vaak hoofdletters (dit staat bekend als *camel case*, zoals in `titelHoofdstuk`).

Je mag rustig afkortingen gebruiken, maar maak ze niet te kort. Een variabele die het product van een aantal ingelezen waarden moet bevatten noem je `product` of `prod`, maar niet `p` (en zeker niet `w`). Let echter op met nietszeggende lange namen zoals `geheelgetal`. Wellicht zal je programma heel wat andere gehele getallen te verwerken krijgen. Ook een naam als `tabel_van_ingevoerde_waarden` slaat nergens op. Dat een variabele een tabel is zie je wel aan de indexen die er achter staan, en een variabele kan moeilijk iets anders dan waarden bevatten. `invoer` lijkt een iets logischer keuze voor deze variabele.

Voorbeelden van domme namen zijn: `dummy`, `tabel`, `tab`, `getal`, `teller`, `pointer`, `ptr`, `temp` (behalve als het over temperatuur gaat), `hulp`, enzovoorts.

In enkele situaties zijn zeer korte variabelennamen toegelaten. Eenlettervariabelen mogen in drie gevallen:

(1) De lopende index van een `for`lus heet meestal `i` als het een getal is (en vernestelde lussen krijgen dan `j`, `k`, en zo verder). Natuurlijk, als de index een 'echte' betekenis heeft krijgt hij een echte naam, maar het heeft geen zin om namen als `teller` of `index` te gebruiken. Als het gaat om een iterator gebruik je `it`

(2) Dikwijls heeft de parameter van een functie geen andere betekenis dan dat het een parameter is. In de functiedefinitie krijgt deze dan een enkele letter. Je schrijft dan bijvoorbeeld

```
double kwadraat(double x){
    return x*x;
}
```

Overdrijf hier niet in: als er verscheidene parameters zijn, dan hebben ze meestal wel een betekenis. Schrijf dus

```
double macht(double x,unsigned int exponent){
    double numacht=x, uit=1;
    while (exponent>0){
        if (exponent % 2==0)
            uit*=numacht;
        numacht*=numacht;
        exponent/=2;
    }
    return uit;
}
```

en niet

```
double macht(double x,unsigned int n){      //VERKEERD
    double y=x, u=1;                        //VERKEERD
    while (n>0){                            //VERKEERD
        if (n % 2==0)                       //VERKEERD
            u*=y;                           //VERKEERD
        y*=y;                               //VERKEERD
        n/=2;                               //VERKEERD
    }                                       //VERKEERD
}                                           //VERKEERD
```

Vaak duidt de gebruikte letter, of de korte naam, het type van de parameter aan: `x` voor een double, `n` voor een geheel getal, `c` of `ch` voor een karakter, `str` voor een string en `l` voor een gelinkte lijst. Deze conventie wordt ook gebruikt bij de range-based `for`lus.

(3) Als het programma gebaseerd is op een tekst of op wiskundige formules waar korte namen gebruikt worden. Het heeft geen zin om, als in de tekst een gewichtenmatrix `w` voorkomt, deze in het programma `gewichtenmatrix` te noemen. Ook `x`, `y` en `z` voor reële coördinaten van een punt zijn voorbeelden van klassieke notaties.

Een (lid)functienaam moet aangeven wat ze doet. Zeker als een functie twee dingen doet moeten ze allebei in de naam staan. Als je een container schrijft met een functie die een gegeven sleutel zoekt en ineens dan ook maar duplicaten verwijdert noem je ze niet `zoek` maar `zoekEnVerwijderDuplicaten`. Als je vindt dat daardoor de naam vam je functie te lang wordt dan is er iets mis: je functie doet te veel dingen. Als je een functienaam hebt `zoekEnVerwijderDuplicatenEnBerekenDeDatumVanPasen` moet je niet de naam veranderen maar het ontwerp van de functie.

Ook de plaats waarop je een variabele declareert is belangrijk. Declareer een variabele *altijd op de plaats waar je ze gaat gebruiken*. Dit zeggen we niet zomaar:

wie een (stukje) programma leest moet onthouden welke variabelen er bestaan en wat voor type ze zijn: dus wil je dat er op elk ogenblik zo weinig mogelijk variabelen relevant zijn, en dat de declaratie gemakkelijk terug te vinden is. Variabelen die in een heel codestuk (bijvoorbeeld een functiedefinitie) belangrijk blijven declareer je in het begin van dat stuk. Maar variabelen die je binnen een lusblok (of een `if`- of een `else`blok) nodig hebt, declareer je binnen dat blok. Variabelen die je maar voor een paar opdrachten nodig hebt declareer je vlak voor die opdrachten. En ja, in dit laatste geval mag je een domme naam gebruiken. Voor domme variabelen geldt dat je niet zuinig hoeft te zijn: het is een slecht idee om een domme variabele die toevallig al gedefinieerd is te hergebruiken. Volgende code roteert een tabel een plaats naar links en verwisselt dan de elementen paarsgewijs

```
int hulp,i;                                    //VERKEERD
hulp=tab[0];                                   //VERKEERD
for (i=0;i<tab.size()-1;i++){                  //VERKEERD
    tab[i]=tab[i+1];                           //VERKEERD
}                                              //VERKEERD
tab[tab.size()-1]=hulp;                        //VERKEERD
for (i=0;i<tab.size()-1;i+=2){                 //VERKEERD
    hulp=tab[i];                               //VERKEERD
    tab[i]=tab[i+1];                           //VERKEERD
    tab[i+1]=hulp;                             //VERKEERD
}                                              //VERKEERD
```

Goede code ziet er zo uit:

```
int hulp=tab[0];
for (i=0;i<tab.size()-1;i++){
    tab[i]=tab[i+1];
}
tab[tab.size()-1]=hulp;
for (int i=0;i<tab.size()-1;i+=2){
    int hulp=tab[i];
    tab[i]=tab[i+1];
    tab[i+1]=hulp;
}
```

(Je kan natuurlijk `swap` gebruiken maar dan heeft het voorbeeld geen zin meer.

Zorg ervoor dat de namen die je gebruikt allemaal op dezelfde wijze gevormd worden. Als je in een programma het aantal studenten en het aantal vakken moet bijhouden, gebruik dan niet de combinatie `vak_aant` en `aantalStudenten`. Als je afkortingen gebruikt, wees dan altijd even bondig: korte en heel lange namen door elkaar werkt verwarrend.

Namen van functies en procedures moeten duidelijk maken wat er gebeurt. Pure queries (die een informatieve waarde teruggeven maar geen veranderingen aanbrengen) en actieve procedures die dingen wijzigen moeten zoveel mogelijk gescheiden worden.

In C oude stijl gaf een procedure wel eens een getal terug om te melden of alles goed was gegaan (0 betekende alles oké), en werd zo een functie die een `int` teruggaf; leesfuncties gaven het aantal ingelezen karakters terug.

Dit is geen nette techniek, al had ze een voordeel: je kon in een `while`lus het klaarzetten van de voorwaarde en het testen ervan samen vlakbij de `while` zetten. Je kan ze nog wel terugvinden in de gebruiksbibliotheken.

HOOFDSTUK 2
INHOUD

2.1 CONSTANTEN

Je kan numerieke constanten die je in je programma gebruikt ook een naam geven met behulp van een `const`declaratie. Namen van constanten noteer je, zoals gezegd, steeds in hoofdletters. Wees niet zuinig met constanten: als je een numerieke waarde twee of meer keer moet schrijven in een programma is het bijna altijd een goed idee er een constante van te maken. Hetzelfde geldt voor de grootte van een tabel, en constante strings die twee keer voorkomen. Er is een school die eist dat *alle* strings samengezet worden en als constanten gedefinieerd. Ook kan het goed zijn een constante die één keer voorkomt met `const` te definiëren: ze valt dan tenminste op.

Er zijn twee redenen hiervoor. Ten eerste maakt het een programma leesbaarder. Als je in een programma BTW moet berekenen is een uitdrukking als `taks=BTWVOET*prijs;` duidelijker dan `taks=20.5*prijs;`. Een programma met constanten is ook beter aanpasbaar. Soms verandert de BTW-voet, en als je overal manueel de 20.5 in je programma verandert kan je in de problemen komen als er ergens een 20.5 staat die geen BTW-voet is, of als je de BTW-voet al ergens verrekend hebt. Vergelijk de veranderingen in

```
cout<<"Prijs zonder BTW van 20.5% is"   //VERKEERD
    <<prijs*0.829876                    //VERKEERD
    <<endl;                             //VERKEERD
```

met deze in

```
const double BTWVOET=20.5;

double prijszonder=prijs*100.0/(100.0+BTWVOET);
cout<<"Prijs zonder BTW van "<<BTWVOET<<"% is"
    <<prijszonder
    <<endl;
```

Het tweede argument is minder belangrijk bij een constante zoals π.

2.2 TYPES

Hoewel er hiervoor in C++ ook nog andere mogelijkheden bestaan, gebruiken wij enkel `int` voor gehele getallen, en `double` voor reële getallen. Occasioneel kan het nuttig zijn om `float` of `short` te gebruiken (als er echt *heel* weinig geheugen is) maar daar hoef je je meestal niets van aan te trekken. `unsigned` gebruiken kan zijn nut hebben, vooral om duidelijk te maken dat negatieve waarden echt onmogelijk zijn en bij bitbewerkingen.

Het type `char` gebruiken we enkel voor lettertekens, niet voor kleine gehele getallen. We maken trouwens helemaal geen gebruik van het feit dat lettertekens eigenlijk voorgesteld worden door kleine gehele getallen die hun ASCII-code bevatten. Dus niet:

```
if (ch >= 'A' && ch <= 'Z')             //VERKEERD
```

```
    ch += 32;                                    //VERKEERD
```

maar wel

```
if (ch >= 'A' && ch <= 'Z')
    ch += 'a' - 'A';
```

2.3 LOGISCHE UITDRUKKINGEN

Het type `bool` wordt gebruikt voor logische *variabelen*. Verder heb je logische *uitdrukkingen* zoals `i==0` of `b*b>5`. Je kan de waarde van zo'n uitdrukking direct in een logische variabele steken. Bijvoorbeeld:

```
bool positief=(a>=0);
```

Logische variabelen vormen een logische uitdrukking op zich. Het heeft geen zin om te schrijven `if ((a>=0)==true))`, en dus heeft het ook geen zin om te schrijven `if (positief==true)`. Een echt gruwelijke fout is om logische waarden toe te kennen met een `if-else`. Schrijf dus *nooit*

```
if (a>0 || a<-10)                                //VERKEERD
    gedaan=false;                                //VERKEERD
else                                             //VERKEERD
    gedaan=true;                                 //VERKEERD
```

maar gewoon

```
gedaan=!(a>0 || a<-10);
```

Meestal is het netter om de negatie weg te werken en gewoon te schrijven `gedaan=(a>=-10 && a<=0)`. De ronde haakjes rond de logische uitdrukkingen zijn niet altijd echt nodig. Zet ze echter altijd: ze maken duidelijk dat het om een logische uitdrukking gaat.

Let op bij voorwaarden met `double`variabelen. Afrondingen kunnen ervoor zorgen dat gelijkheden niet opgaan. Het programmafragment

```
double x=1.0;                                    //VERKEERD
while (x!=0){                                    //VERKEERD
    x-=0.1;                                      //VERKEERD
    cout<<x<<endl;                               //VERKEERD
}                                                //VERKEERD
```

levert op mijn computer een oneindige lus op.

Tenslotte: je compiler laat toe om gehele getallen en dergelijke te beschouwen als logische uitdrukkingen (ze zijn onwaar als ze nul zijn). Een goede programmeur doet dit *niet*. schrijf dus niet `while (i)...`, maar `while (i!=0)...`.

2.4 DECLARATIES

C++ staat toe om meerdere variabelen in één keer te declareren. Gebruik dit niet om tegelijkertijd tabellen en gewone variabelen te introduceren. Je kan ook variabelen ineens initialiseren. Bovendien zet je in een declaratie het sterretje bij de gedeclareerde naam en niet bij het type Schrijf die drie soorten niet in dezelfde opdracht. Dus niet

```
int punt[AANTPUNT], code[AANTCODE],              //VERKEERD
    aantfact=5, discr, mediaan=0, alfa;          //VERKEERD
vector<int>* ptr,v;                              //VERKEERD
```

maar

```
int punt[AANTPUNT], code[AANTCODE];
int aantfact=5, mediaan=0;
int discr, alfa;
vector<int>* ptr;
vector<int> v;
```

Als je meerdere tabellen in de declaratie initialiseert, gebruik je voor elke tabel een aparte declaratieopdracht. Het is niet wenselijk om 'voor de zekerheid' alle variabelen op voorhand de waarde 0 te geven.

2.5 TOEWIJZINGEN

We beschouwen een toewijzing als een opdracht en niet als een uitdrukking. Dus staat een toewijzing altijd alleen en maakt geen deel uit van een andere uitdrukking. In klassieke C-programma's wordt wel eens een toewijzing als een conditionele uitdrukking gebruikt: als er nul wordt toegewezen is de uitdrukking onwaar. Een voorbeeld: schrijf een positief geheel getal `a` in bits uit (minst significante bit links!). In C wordt dit

```
cout<<a%2;                                  //VERKEERD
while (a/=2)                                //VERKEERD
    cout<<a%2;                              //VERKEERD
```

Nette C++-programmeurs schrijven dan

```
cout<<a%2;
a/=2;
while (a!=0){
    cout<<a%2;
    a/=2;
}
```

Dit is langer, maar gemakkelijker om te lezen.

Overdrijf niet met het gebruik van '++' en '−−' binnenin andere uitdrukkingen. Beperk je tot verhogen en verlagen van indices van tabellen. Je mag deze uitdrukkingen natuurlijk wel los gebruiken, maar probeer maar eens te begrijpen wat

```
while (n > 0)                               //VERKEERD
    som += n * n--;                         //VERKEERD
```

betekent.

Tenslotte: als je een waarde twee keer nodig hebt, steek je ze altijd in een variabele. Dit is duidelijker (je ziet meteen dat het *dezelfde* waarde is die je gebruikt), en ook efficiënter. Vooral als je een functie oproept, die misschien veel rekenwerk vergt, is dat nuttig (en bij recursie kan je verschrikkelijke resultaten krijgen als je het niet doet: een programma 1000000000000 keer trager laten lopen is geen goed idee). Schrijf dus niet

```
x=25.0;                                     //VERKEERD
som=0;                                      //VERKEERD
while (cos(x)>-0.9){                        //VERKEERD
    som+=cos(x);                            //VERKEERD
    x-=0.01;                                //VERKEERD
};                                          //VERKEERD
```

maar wel

```
x=25.0;
som=0;
double cosx=cos(x);
while (cosx>-0.9){
    som+=cosx;
    x-=0.01;
    cosx=cos(x);
};
```

Eén uitzondering op deze regel: de voorwaarde in een `for` wordt meestal verschillende keren getest, en toch mag je er eenvoudige uitdrukkingen inzetten.

```
for (int i=0; i<aantkol*aantrij;i++)
    ...
```

mag dus nog net. De reden hiervoor: de meeste compilers zijn slim genoeg om hier zelf een variabele aan te maken. Doe dit echter nooit met functieoproepen!

HOOFDSTUK 3
SAMENGESTELDE OPDRACHTEN

Van alle samengestelde opdrachten die C++ rijk is, gebruiken wij enkel de `if`-, `while`- en `for`opdracht. De `switch`structuur is erg uitzonderlijk, maar mag wel. Andere constructies moet je steeds door combinaties van deze vier opdrachten vervangen. Verbannen zijn `do`, `continue`, `break`[1] en *–horresco referens–* `goto`.

De opdrachten `throw`, `try` and `catch` dienen voor foutafhandeling en worden hier niet behandeld. Met `if` zijn er meestal weinig problemen. Twee opmerkingen toch: schrijf alleen opdrachten binnen de `if-else`structuur die er binnen thuis horen. Je schrijft dus niet

```
if (disc>0){                                    //VERKEERD
    afstand=sqrt(x*x+y*y);                      //VERKEERD
    aantal=2;                                   //VERKEERD
}                                               //VERKEERD
else{                                           //VERKEERD
    afstand=sqrt(x*x+y*y);                      //VERKEERD
    aantal=0;                                   //VERKEERD
};                                              //VERKEERD
```

maar wel

```
afstand=sqrt(x*x+y*y);
if (disc>0)
    aantal=2;
else
    aantal=0;
```

Ten tweede: wees erg zuinig op de `(...? ...: ...)`-vorm. Deze is over het algemeen erg slecht leesbaar. Vergelijk het bovenstaande fragment maar eens met

```
afstand=sqrt(x*x+y*y);
aantal=(disc>0 ? 2:0);
```

Voor een simpele uitdrukking als deze is dit nog net aanvaardbaar, maar als het een beetje ingewikkelder wordt moet je de `if-else` voluit schrijven, of de spatiëring ervan gebruiken:

```
afstand=sqrt(x*x+y*y);
aantal=(disc>0 ?
    2:
    0);
```

Lussen verdienen wel wat meer uitleg.

3.1 FOR, FOR_EACH OF WHILE?

Als je een lus schrijft moet je je afvragen of de lus een vooraf bepaald aantal keer wordt uitgevoerd of dat de lus probeert een bepaalde voorwaarde waar te maken.

1. Hierop is één uitzondering: binnen een `switch`structuur gebruik je een `break` op het einde van elke `case`.

In het eerste geval gebruik je een `for`lus (of zijn variant, de `for_each`lus) in het tweede geval een `while`.

Een `for`lus heeft een range. Vaak is dat een *teller* die voorafbepaalde waarden aanneemt. Meestal is dit heel duidelijk: de teller neemt alle even waarden aan tussen 1 en 365, of gaat van nul tot aan het aantal studenten (sprong 1), enzovoorts. Soms is het minder duidelijk en neemt de teller bijvoorbeeld alle strikt positieve waarden aan met kwadraat kleiner dan 729. Dan kan je kiezen tussen een `while` en een `for`. Bij een dubbele voorwaarde gebruik je bijna altijd een `while`. Dit is bijvoorbeeld het geval als je achteraf nog moet nakijken *waarom* de lus gestopt is. Zo schrijf je, om te weten of er een nul in een tabel staat

```
int i=0;
while (i<tab.size() && tabel[i]!=0)
    i++;
bool erisnul=(i<tab.size());
```

De teller mag binnen een `for`lus niet van waarde veranderen. Deze techniek wordt door slechte programmeurs gebruikt om een `for` vroegtijdig te beëindigen:

```
for (int i=0;i<tab.size();i++){          //VERKEERD
    ...                                  //VERKEERD
    if (tabel[i]==0)                     //VERKEERD
        i=tab.size();                    //VERKEERD
}                                        //VERKEERD
```

is even erg als een `break` gebruiken! Hier hoort duidelijk een `while` te staan, die er hetzelfde uitziet als het vorige voorbeeld.

De teller wordt binnen de `for`opdracht gedeclareerd. Soms heb je de waarde van de teller na de `for` nog nodig. Dat is geen reden om de teller niet binnen de lus te declareren. Ofwel kan je de eindwaarde op voorhand berekenen (en dan is het veel beter dit met een aparte opdracht te doen), ofwel moet je een `while` gebruiken.

In C++11 is de mogelijkheid om `for`lussen te schrijven uitgebreid. Dit komt de duidelijkheid ten goede en, zoals we weten, duidelijkheid is zeer belangrijk.

In veel gevallen zal de range van een `for`lus gegeven worden door een *container* zoals een `vector`: het is de bedoeling om alle elementen van die container één keer te bezoeken in de natuurlijke volgorde. In dat geval gebruik je de *range-based* `for`lus. Een voorbeeld: je hebt een vector `tab` die elementen van een klasse `Ding` bevat en je wil op al deze dingen de lidfunctie `foo()` loslaten. Dat levert

```
for(auto&& d : tab)
    d.foo();
```

Let hierbij op de dubbele ampersand na `auto`. Deze heeft dezelfde betekenis als ampersands bij een functiedefinitie. `auto` wordt door de compiler vertaald naar `Ding`. Schrijf je gewoon '`auto`' dan wordt elk Ding gekopieerd naar de variabele `d`, wat als Dingen groot zijn erg veel tijd en ruimte kan kosten en trouwens niet altijd mogelijk is, bijvoorbeeld als elk `Ding` een `unique_ptr` is. Bovendien wordt dan `foo()` toegepast op de kopie. Als dit `d` verandert zullen de elementen in `tab` niet mee veranderen. De reden waarom je *twee* ampersands gebruikt is zeer technisch (zoek maar eens *proxy iterator* op als je de details wil weten). Het is eigenlijk alleen belangrijk als je een vector van `bool`s hebt, maar het is goed om de gewoonte te kweken om altijd een dubbele ampersand na `auto` te gebruiken.

Soms is echter de *plaats* van het element in de container belangrijk. In dit geval gebruik je een `for_each`lus. Een eenvoudig voorbeeld: je wil de elementen van `tab` uitschrijven met komma's tussen de verschillende waarden. Daarom moeten we het eerste element anders behandelen dan de volgende. Dit levert

```
if (tab.size() > 0)
    cout<<tab[0];
for_each(++tab.begin(),tab.end(),
    [](Ding& a){cout<<",";a;}
    );
```

Merk op dat de lambdafunctie thuishoort op een aparte lijn om de structuur visueel weer te geven en dat `for_each` gedefinieerd wordt in de `algorithm` header. Er is weinig verschil tussen een `for_each`lus en een klassieke `for`lus. Vergelijk bovenstaande code maar met

```
if (tab.size() > 0)
    cout<<tab[0];
for (auto it=++tab.begin(); it!=tab.end(); it++){
        cout<<","<<*it;
};
```

Hier gebruiken we `auto` zonder ampersand, omdat het hier gaat over de iterator die naar de objecten wijst en niet over de objecten zelf. Beide vormen zijn bruikbaar. Als we de iterator (bij vectoren kunnen we ook een index gebruiken) expliciet nodig hebben, zoals bij ons vroegere voorbeeld waar we de elementen van een vector roteerden, is de `for_each`vorm niet bruikbaar.

En dan is er nog één speciaal geval: de oneindige lus. Deze wordt gebruikt bij allerhande serverprogramma's die moeten blijven draaien. Deze wordt traditioneel in C als een `for`lus geschreven:

```
for(;;)
```

en niet als een `while`, zoals je misschien zou denken.

3.2 BINNEN OF BUITEN DE LUS?

Wanneer je in een lus een `if`opdracht nodig hebt om een bijzonder geval op te vangen, dan ben je wellicht op de verkeerde weg. Het volgende fragment drukt een tabel van getallen af, gescheiden door komma's. Na het laatste element mag er geen komma staan.

```
for (int i=0; i < AANT; i++) {          //VERKEERD
    cout << tab[i];                     //VERKEERD
    if (i != AANT-1)                    //VERKEERD
        cout << ", ";                   //VERKEERD
}                                       //VERKEERD
```

Dit werkt wel maar is onduidelijk. Schrijf wel

```
for (int i=0; i < AANT-1; i++)
    cout << tab[i] << ", ";
cout << tab[AANT-1];
```

Wanneer je in een herhaling het laatste geval op een speciale manier moet behandelen, haal dit geval dan uit de lus en verwerk het achteraf. Op dezelfde manier kan je een speciaal begingeval best vóór de lus plaatsen.

Soms gebruik je één lus waar er eigenlijk twee opeenvolgende lussen nodig zijn. Hieronder lezen we een reeks gehele getallen afgesloten met een 0 en drukken daarna het eerste getal uit deze rij af dat kleiner was dan 10 (de invoer moet zeker een getal kleiner dan tien bevatten). Foutief is

```
const int GRENS=10;                          //VERKEERD
res=GRENS;                                   //VERKEERD
cin >> getal;                                //VERKEERD
while (getal != 0){                          //VERKEERD
    if ( res == GRENS &&                     //VERKEERD
         getal < GRENS)                      //VERKEERD
        res=getal;                           //VERKEERD
    cin >> getal;                            //VERKEERD
}                                            //VERKEERD
if (res == GRENS) res=0;                     //VERKEERD
cout << res << endl;                         //VERKEERD
```

wat wel oké is is

```
const int GRENS=10;
int uit;
cin >> getal;
while (getal >= GRENS)
    cin >> getal;
uit=getal;
while (getal != 0)
    cin >> getal;
cout<<uit<<endl;
```

3.3 DE STRUCTUUR VAN EEN `WHILE`

De structuur van een `while`lus ziet er altijd als volgt uit:

```
klaarzetten voorwaarde;
while (voorwaarde voldaan){
    uitvoeren lus;
    opnieuw klaarzetten voorwaarde;
}
```

Beginnende programmeurs willen nogal eens van deze structuur afwijken, wat altijd problemen geeft: ze vergeten dikwijls het eerste geval te behandelen, ze voeren de lus een keer te veel uit, en zo verder. Enkele voorbeelden. Stel dat je getallen moet inlezen tot je een nul tegenkomt. Zolang je geen nul hebt geef je het kwadraat terug. Schrijf dan niet

```
int getal;                                   //VERKEERD
while (getal!=0){                            //VERKEERD
    cin>>getal;                              //VERKEERD
    cout<<getal*getal;                       //VERKEERD
};                                           //VERKEERD
```

(denk maar eens even na waarom dit de mist in gaat. Het kan op twee plaatsen verkeerd lopen). Correct is wel

```
int getal;
cin>>getal;
while (getal!=0){
    cout<<getal*getal;
    cin>>getal;
};
```

Een ander voorbeeld: bereken de som van alle gehele kwadraten kleiner dan 729. Schrijf dan niet

```
int i=0;                                    //VERKEERD
int kwadr=0;                                //VERKEERD
som=0;                                      //VERKEERD
while (kwadr<729){                          //VERKEERD
    i++;                                    //VERKEERD
    kwadr=i*i;                              //VERKEERD
    if (kwadr<729)                          //VERKEERD
        som+=kwadr;                         //VERKEERD
};                                          //VERKEERD
```

hoewel dit tenminste al werkt[2]. Juist is

```
int i=1;
int kwadr=i*i;// kwadr=1 mag ook
som=0;
while (kwadr<729){
    som+=kwadr;
    i++;
    kwadr=i*i;
};
```

3.4 LUSSEN BINNEN LUSSEN

Als je binnen een lus nog eens een lus gebruikt, dan kan het zijn dat je programma traag gaat lopen (10000 keer een stukje code uitvoeren is niet veel, 100000000 keer wel). Soms is daar niets aan te doen, maar soms is het mogelijk je programma te herschrijven zodanig dat je geen binnenlus nodig hebt. Een klassiek voorbeeld is dat van de machtreeks. Als je bijvoorbeeld $1 + x + x^2/2 + \ldots x^n/(n!)$ moet berekenen, moet je niet voor de i-de stap zowel x^i als $i!$ berekenen beginnend vanaf niets: je maakt gebruik van de waarden die je de vorige keer berekend hebt. Je schrijft dus niet

```
reeks=1;
for (int i=1;i<=n;i++){                     //VERKEERD
    double xmacht=1;                        //VERKEERD
    double faculteit=1;                     //VERKEERD
    for (int j=1;j<=i;j++){                 //VERKEERD
        xmacht*=x;                          //VERKEERD
        faculteit*=j;                       //VERKEERD
    }                                       //VERKEERD
    reeks+=xmacht/faculteit;                //VERKEERD
}
```

maar wel

```
reeks=1;
double xmacht=1;
double faculteit=1;
for (int i=1;i<=n;i++){
    xmacht*=x;
    faculteit*=j;
    reeks+=xmacht/faculteit;
}
```

2 We herhalen nog maar eens: het is niet omdat een programma werkt dat het goed is.

En als je je afvraagt waarom `faculteit` als `double` gedefinieerd is terwijl het alleen gehele getallen bevat: dit is nodig omdat $i!$ heel erg groot kan worden: voor i groter dan 12 gaat $i!$ niet meer in een geheel getal van 32 bits. Bij deze machtreeks moesten we bij de kleinste macht van `x` beginnen omdat we de coëfficiënten nog moesten berekenen. Als we bij de grootste macht van `x` kunnen beginnen, is het beter om het geheel als een veelterm beschouwen, en dit doe je met de regel van Horner

$$a_n x^n + a_{n-1}x^{n-1} + \ldots + a_1 x + a_0 = (\ldots(a_n x + a_{n-1})x + \ldots + a_1)x + a_0.$$

In code (merk op dat er $n+1$ elementen coëfficiënten zijn, en niet n)

```
//berekent a[n]*x^n+a[n-1]*x^(n-1)+a[0];
    double result=a[n];
    for (int i=n-1;i>=0;i--)
        result=result*x+a[i];
```

Je gebruikt *geen* tweede lus (en je gebruikt ook niet `pow`, dat al minstens even traag is en bovendien bij gehele `x` afrondingsfouten geeft).

3.5 LUSSEN MET MEERDERE STOPCONDITIES

Niet alleen een `for`lus kan vroegtijdig beëindigd worden, ook een `while` kan op die manier verknoeid worden:

```
    int i=0;                                    //VERKEERD
    while (i<tab.size()){                       //VERKEERD
        ...                                     //VERKEERD
        if (tabel[i]==0)                        //VERKEERD
            i=tab.size();                       //VERKEERD
        else                                    //VERKEERD
            i++;                                //VERKEERD
    }                                           //VERKEERD
```

Je moet er steeds voor zorgen dat *elke* reden waarom een bepaalde lus eventueel kan stoppen, duidelijk zichtbaar is in de conditie bovenaan de lus.

We zien dan ook niet graag dat je een `return`opdracht gebruikt om uit een lus te springen (en trouwens ook niet om vroegtijdig een functie of procedure te verlaten). Een `return`opdracht hoort thuis waar je haar het meest verwacht: onderaan de definitie van een functie of procedure.

We geven nog een ander voorbeeld van een lus die om meerdere redenen kan stoppen. In onderstaand fragment bepalen we of een gegeven getal een priemgetal is.

```
bool isPriem(unsigned int getal){
    unsigned int deler=2;
    while (deler * deler <= getal && getal % deler != 0)
        deler++;
    return (deler*deler >= getal);
}
```

Dergelijke lussen 'met dubbele conditie' zijn een zeer belangrijk (maar niet eenvoudig) hulpmiddel voor de programmeur. Merk op dat er in de meeste gevallen na een dergelijke lus nog een `if`opdracht volgt waarin één van de twee condities opnieuw wordt getest. Denk steeds goed na welke van de twee je nodig hebt. Het is immers altijd mogelijk dat, bij een EN-conditie, de twee voorwaarden tegelijk onwaar worden.

Om je op dit soort lussen te oefenen kan je je eens aan de volgende opgave wagen: Schrijf een programma dat een rij getallen inleest, afgeloten met een nul. Druk daarna op het scherm af hoeveel getallen deze lijst bevat (de nul op het einde niet meegerekend). Stop echter met de invoer zodra er tien getallen zijn ingelezen en zorg dat er nooit een elfde getal aan de gebruiker wordt gevraagd, zelfs al zou je dit in je programma niet gebruiken. Los dit op 'met stijl'.

3.6 VERSCHILLENDE LUSSEN NA ELKAAR

Als je na elkaar twee lussen hebt die evenveel keer worden uitgevoerd kan je ze dikwijls combineren. Het maakt je programma eenvoudiger, en dus leesbaarder. In het volgende fragment lezen we tien getallen $x_0, \ldots, x_9$ in en bepalen de *variantie* van die getallen. Deze wordt gedefinieerd als $\frac{1}{10}\sum_i (x_i - \bar{x})^2$, waarbij $\bar{x}$ het gemiddelde van die 10 getallen is[3]. Verkeerd is

```
const int AANT = 10;                    //VERKEERD
                                        //VERKEERD
for (int i=0; i < AANT; i++)            //VERKEERD
    cin >> tab[i];                      //VERKEERD
                                        //VERKEERD
double gem = 0;                         //VERKEERD
for (int i=0; i < AANT; i++)            //VERKEERD
    gem+=tab[i];                        //VERKEERD
gem/=AANT;                              //VERKEERD
                                        //VERKEERD
double var=0;                           //VERKEERD
for (int i=0; i<AANT; i++)              //VERKEERD
    var+=(gem-tab[i])                   //VERKEERD
          *(gem-tab[i]);                //VERKEERD
var /= AANT;                            //VERKEERD
```

In de plaats schrijf je

```
const int AANT=10;

double gem=0;
for (int i=0; i<AANT; i++) {
    cin>>tab[i];
    gem+=tab[i];
}
gem/=AANT;

var = 0;
for (int i=0; i < AANT; i++)
    var+=(gem-tab[i])
          *(gem-tab[i]);
var/=AANT;
```

Merk op dat we de overblijvende lussen niet nogmaals kunnen combineren omdat we eerst de volledige reeks moeten inlezen om het gemiddelde te bepalen vooraleer we de variantie kunnen berekenen (tenzij we onze statistiek goed kennen ...).

3 Je kan bewijzen dat de variantie in dit geval ook gelijk is aan $\frac{1}{10}(\sum x_i^2) - (\bar{x})^2$. Welk merkwaardig gevolg heeft het gebruik van deze formule voor het programma?

HOOFDSTUK 4
TABELLEN

4.1 TABELELEMENTEN

Tabelelementen zijn volwaardige 'C++-burgers'. Je kan er mee rekenen, je kan ze afdrukken, je kan ze van waarde veranderen en je kan ze rechtstreeks inlezen met >>. Vele programmeurs gebruiken in dit laatste geval echter steeds een overbodige extra variabele. Het is geen goed idee om te schrijven

```
vector<double> tab(AANT);                    //VERKEERD
double getal;                                //VERKEERD
                                             //VERKEERD
for (int i=0; i<AANT; i++){                  //VERKEERD
    cin>>getal;                              //VERKEERD
    tab[i]=getal;                            //VERKEERD
}                                            //VERKEERD
```

Beter is

```
vector<double> tab(AANT);

for (int i=0; i < AANT; i++)
    cin >> tab[i];
```

Laat je door dit voorbeeld niet misleiden. In dit geval zijn we op voorhand zeker dat we precies `AANT` elementen willen inlezen. Wanneer we daarentegen bijvoorbeeld een reeks elementen in een tabel inlezen die afgesloten wordt met een nul, dan gebruik je beter een `while`lus (immers: je kan niet uitrekenen hoe vaak de lus doorlopen wordt, en bovendien zal je waarschijnlijk moeten onthouden hoeveel elementen in de tabel zijn gestoken) en een hulpvariabele. De afsluitende nul sla je dus *niet* op in de tabel, tenzij het expliciet gevraagd is. Merk op dat je daar één plaats mee uitspaart: dat kan belangrijk zijn als een tabel maar net groot genoeg is. Dus niet

```
aant=0;                                      //VERKEERD
cin>>tab[aant];                              //VERKEERD
while (tab[aant]!=0)                         //VERKEERD
    cin>>tab[++aant];                        //VERKEERD
```

maar wel

```
aant=0;
cin>>getal;
while (getal != 0) {
    tab[aant++]=getal;
    cin>>getal;
}
```

Uiteraard ga je in de praktijk meestal gebruik maken dat een vector variabele grootte heeft zodat je in de meeste gevallen beter af bent met de opdracht `push_pack`.

4.2 INDEXVARIABELEN

Je hoeft je ook niet in allerlei bochten te wringen om steeds dezelfde indexvariabele te gebruiken bij dezelfde tabel. Er is geen enkel verband tussen een tabelvariabele en de tellervariabele die je gebruikt om door die tabel te lopen. Je kan het ook niet altijd. Probeer maar eens het volgende programmafragment (dat een tabel omdraait) te herschrijven op zo'n manier dat je altijd dezelfde index gebruikt voor `tab`:

```
vector<double> tab[tab.size()];
...
for (int i=0;i<tab.size()/2;i++){
    double temp=tab[i];
    int j=tab.size()-1-i;
    tab[i]=tab[j];
    tab[j]=temp;
}
```

4.3 WELKE TABELLEN HEB JE NODIG?

Vele programmeurs zijn de eerste keer blijkbaar zodanig onder de indruk van het begrip 'tabel' dat ze naderhand niets anders meer wensen te gebruiken. Ze vergeten daarbij dat vele problemen kunnen worden opgelost zonder ook maar één tabel te gebruiken. Het volgende fragment leest een aantal reële getallen in en drukt hun gemiddelde af[1]:

```
for (int i=0; i<AANT; i++)          //VERKEERD
    cin>>tab[i];                    //VERKEERD
som=0;                              //VERKEERD
for (int i=0; i<AANT; i++)          //VERKEERD
    som+=tab[i];                    //VERKEERD
cout << som/AANT << endl;           //VERKEERD
```

De tabel is overbodig: schrijf gewoon

```
som=0;
for (int i=0; i<AANT; i++){
    cin>>getal;
    som+=getal;
}
cout << som/AANT << endl;
```

Als vuistregel kan je stellen dat je een tabel slechts nodig hebt wanneer je de eerste elementen van de tabel nog dient te gebruiken *nadat* je de laatste hebt ingevuld. Het is niet omdat er in het probleem een 'rij' of 'reeks' ter sprake komt dat je noodzakelijk een tabel nodig hebt.

Soms zijn tabellen een stuk kleiner dan je op het eerste zicht zou denken of duiken er tabellen op uit onverwachte hoek. Om de vijf kleinste waarden te bepalen van tienduizend ingetikte getallen, heb je slechts een tabel van vijf elementen nodig — je hoeft de tienduizend getallen niet eerst op te slaan en te sorteren. Om van zes miljoen kiezers de stemmen te tellen, heb je slechts tabellen nodig waarvan de grootteorde overeenkomt met het aantal kandidaten — niet met het aantal kiezers. Om van zeven miljard aardbewoners te bepalen welke verjaardagsdatum er het meest voorkomt, heb je slechts een tabel nodig van 366 gehele getallen.

1. Een programmeur met stijl merkt trouwens reeds de verdachte tweevoudige `for`lus op. Wanneer je deze twee lussen samenneemt, valt het nog meer op dat de tabel `tab` hier overbodig is.

4.4 COMMENTAAR

Soms is het nuttig om commentaarregels aan je programma toe te voegen. Een programma dat volgens de stijlregels van dit document is opgebouwd, behoeft echter meestal niet veel uitleg. Beperk je tot opmerkingen die misschien niet door iedereen onmiddellijk uit de programmacode kunnen worden afgeleid. Over het algemeen zijn er twee soorten commentaar: detailcommentaar en algemeen commentaar.

Detailcommentaar wordt gegeven om kleine dingen toe te lichten die niet duidelijk worden uit de code: als je een truukje moet gebruiken, of als je een reden hebt om iets eigenaardigs te doen. Ook als de lezer zou denken dat je iets verkeerds gedaan hebt moet je zeker commentaar toevoegen. Bijvoorbeeld:

```
double faculteit;   //double om overflow te voorkomen!
...
bool schrikkeljaar=(jaar%4==0) &&
 (jaar<1582 || //Gregoriaanse kalender ingevoerd in 1582
        (jaar % 100 != 0) || (jaar % 400 != 0));
```

Het is volkomen overbodig om dingen die duidelijk zijn te commentariëren:

```
                                          //VERKEERD
int aantdagen;     //geeft aantal dagen aan
...                                       //VERKEERD
cout<<aantaldagen; //uitschrijven aantal dagen.
                                          //VERKEERD
```

Verder is er nog algemene commentaar. Deze geeft van een stuk code aan wat ze uitvoert. Ook hier geldt dat dingen die al duidelijk zijn niet vermeld worden:

```
                                          //VERKEERD
double faculteit(int n){   //berekent de faculteit van n
...                                       //VERKEERD
```

Waar de je de code echt van dichtbij moet bekijken om te weten wat er gebeurt zet je wel commentaar, zoals in een vorig voorbeeld:

```
//berekent a[n]*x^n+a[n-1]*x^(n-1)+a[0];
    double result=a[n];
    for (int i=n-1;i>=0;i--)
        result=result*x+a[i];
```

In het begin van een programma kan je ook best zetten wat het programma allemaal doet. Vooral uitzonderingsgevallen, en uitleg over hoe het programma moet gebruikt worden moet je becommentariëren. Vaak gebeurt dat niet zozeer in commentaar tussen de code, maar bijvoorbeeld in een tekst die in een hulpfunctie thuishoort. Je kan dan bovenaan naar die hulpfunctie verwijzen.

Bij grotere projecten heeft elk programma ook nog documentatie buiten de code om. Daar staat voor elk onderdeel (zoals functies en procedures) beschreven waar de code voor dient. Het verdient aanbeveling om een documentatiegenerator zoals Doxygen of, voor Java, Javadoc te gebruiken. Hiervoor moet de commentaar bij de code aangevuld worden met tags zoals in onderstaand voorbeeld. Doxygen kan dan het commentaar oogsten en structureren tot een gepaste API in, bijvoorbeeld, LaTeXof HTML.

```
/** \class SAIS
    \brief _S_uffix _A_rray geconstrueerd met
    het _I_nduction _S_ort algoritme
    Induction sort is een O(n) algoritme
    voor de constructie van de SA.
*/
class SAIS:public vector<int>{
public:
/** \fn SAIS
    @param T pointer naar de basisstring voor de suffixtabel
    \warning Voor het algoritme moet unsigned gebruikt worden.
    @param t lengte van de basisstring. Moet opgegeven worden
            omdat de string nulkarakters mag bevatten.
*/
    SAIS(const unsigned char T, int n);
    void defranguleer();///< onbegrijpelijke functie
};
```

Het begin van commentaar bestemd voor Doxygen wordt aangegeven met `///` of met `/**` terwijl tags kunnen worden aangeduid met een backslash of met een @.

Voor kleine projecten volstaat documentatie vanaf het klassenniveau; voor grotere projecten moet een algemeen overzicht gemaakt worden zodat een nieuwkomer niet alle klassen apart moet bekijken maar een zicht krijgt op de voornaamste onderdelen en de belangrijke klassen, waarbij bijkomende klassen zoals technische klassen naar de achtergrond verdwijnen.

HOOFDSTUK 5
STRUCTUUR VAN HET KLASSENMODEL

Het opstellen van het klassenmodel is de kern van objectgericht ontwerp. Het is niet de bedoeling hier er zelfs maar een schets van te geven. Wel een kort lijstje van de meest populaire fouten:

- Niet objectgericht werken. Als je een object wil wijzigen of er informatie van wil vraag je dat aan dat object door middel een lidfunctie. Een veel voorkomende fout is om een lidfunctie te schrijven die niet bij het `*this`-object hoort. Het eigenlijke object wordt dan als parameter opgegeven. Een voorbeeldje: je hebt een keuken waarin een koelkast staat. Nu wil je de instelling veranderen. Veelvoorkomende fout is dan om die verandering in de klasse `Keuken` onder te brengen:

```
class Keuken{                                          //VERKEERD
  void veranderInstelling(Koelkast& koelkast,//VERKEERD
                          int nieweWaarde){ //VERKEERD
       koelkast.instelling=nieuweWaarde);   //VERKEERD
    };                                                 //VERKEERD
};                                                     //VERKEERD
```

  Het komt ook veel voor dat alles wel binnen de klasse `Koelkast` gebeurt, maar dat men het vraagt aan de verkeerde koelkast, zeker in een context waarin men met verschillende koelkasten werkt. Ook hier wordt de eigenlijke koelkast als parameter meegegeven. De fout is gemakkelijk herkenbaar omdat men binnen de lidfunctie het `*this`-object niet gebruikt.

- Zinloze gegevensvelden. Een klasse mag alleen datavelden bevatten die ook zinvol zijn 'in rust', dus als er geen lidfunctie actief is. Als lidfuncties onderling data doorgeven die alleen zinvol zijn binnen de uitvoering van één van die lidfuncties dienen deze als parameters te worden doorgegeven. Deze data in een gegevensveld steken is een overtreding van de modulariteitseis.

- Verkeerde plaatsing van functies binnen een klassenhiërarchie. Vaak heeft men een hiërarchie van klassen met gelijkaardige objecten, waarbij lagere klassen specialisaties zijn van hogere. Als je een lidfunctie toevoegt met een bepaalde eigenschappen dan dient deze zo hoog mogelijk te staan in de hiërarchie.