Hoofdstuk 13

Migreren van C naar C++

In dit hoofdstuk vindt u een overzicht van de programmeertaal C++. Het biedt tevens een introductie in de toepassing van C++ als objectgeoriënteerde programmeertaal. In het hoofdstuk wordt een reeks programma's gepresenteerd, en de elementen van elk programma worden zorgvuldig uitgelegd. De programma's worden geleidelijk ingewikkelder en de voorbeelden in de latere paragrafen illustreren enkele concepten van objectgeoriënteerd programmeren.
De voorbeelden in dit hoofdstuk geven eenvoudige, directe, praktische ervaring met belangrijke functies van de taal C++. Het hoofdstuk laat u kennismaken met stream-I/O, overloading van operatoren en functies, referentieparameters, klassen, constructors, destructors, sjablonen en overerving. Beheersing van de afzonderlijke onderwerpen vereist een grondige bestudering van een begeleidend boek, zoals Pohl, *C++ for C Programmers*, 2nd ed (Redwood City, CA: Benjamin/Cummings, 1993) of Pohl, *Object-Oriented Programming Using C++*, 2nd ed (Reading, MA: Addison-Wesley, 1997).
Objectgeoriënteerd programmeren wordt geïmplementeerd door de **class-**constructie. De **class**-constructie in C++ is een uitbreiding van **struct** in C. De latere voorbeelden in dit hoofdstuk laten zien hoe C++ OOP (objectgeoriënteerde programmeer)-concepten verwezenlijkt, zoals data-hiding, ADT's, overerving en typehiërarchieën.

13.1Uitvoer

Programma's moeten communiceren om van nut te zijn. Ons eerste voorbeeld is een programma dat de zin 'C++ is an improved C.' in het scherm afdrukt. Het volledige programma staat in het bestand improved.cpp

```
// A first C++ program illustrating output.
//    Title:  Improved
//    Author: Richmond Q. Programmer

#include <iostream.h>

    int main()
    {
       cout << "C++ is an improved C.\n";
    }
```

Het programma drukt in het scherm af.

```
C++ is an improved C.
```

Bespreking van het programma *improved*

```
// A first C++ program illustrating output.
```

De dubbele schuine streep is een nieuw commentaarsymbool. Het commentaar loopt tot het eind van de regel. De oude commentaarsymbolen in C, /* */, zijn nog steeds beschikbaar voor meerregelig commentaar.

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

De header *iostream.h* zorgt voor I/O-faciliteiten voor C++.

```
int main()
```

In C++ betekenen de lege haakjes altijd **main(void)**, nooit **main(…)**. In C++ wordt het overbodige **void** niet gehanteerd voor de declaratie van de lege argumentenlijst.

```
cout << "C++ is an improved C.\n";
```

Dit statement drukt af naar het scherm. De identifier **cout** is de naam van de standaarduitvoerstream. De operator **<<** geeft de string "**C++ is an improved C.\n**" door aan de standaarduitvoer. Op deze manier gebruikt, wordt de *uitvoeroperator* **<<** aangeduid als de *put to*-operator of *invoeg*operator.

We kunnen ons eerste programma als volgt herschrijven:

```
// A first C++ program illustrating output.

#include <iostream.h>

main()
{
    cout << "C++ is an improved C." << endl;
}
```

Hoewel deze notatie afwijkt van de eerste versie, produceert dit programma dezelfde uitvoer. In deze versie wordt **main()** gedeclareerd zonder de expliciete declaratie van **int** als het type returnwaarde, en maakt het gebruik van het feit dat dit returntype impliciet is. In dit voorbeeld maken we tweemaal gebruik van de uitvoeroperator **<<** *put to*. Elke keer dat de **<<** wordt gebruikt met **cout**, gaat het afdrukken verder vanaf de plaats waar het was gebleven. In dit geval legt de identifier **endl** een nieuwe regel op, die wordt gevolgd door een *flush*. De **endl** wordt een *manipulator* genoemd.

13.2Invoer

We gaan een programma schrijven om de afstand van de aarde naar de maan in mijlen om te zetten naar kilometers. In mijlen is deze afstand ongeveer 238.857. Dit getal is een integer. Om mijlen te converteren naar kilometers vermenigvuldigen we met de conversiefactor 1,609, een reëel getal.
Ons conversieprogramma maakt gebruik van variabelen voor de opslag van integerwaarden en reële waarden. In C++ moeten alle variabelen worden gedeclareerd

voordat ze worden gebruikt, maar in tegenstelling tot in C hoeven ze niet per se bovenaan in een blok te staan. Declaraties mogen worden geplaatst in uitvoerbare statements. Het bereik van variabelen is van het punt van de declaratie tot en met het eind van het blok waarbinnen ze worden gedeclareerd. U moet identifiers kiezen om hun toepassing in het programma aan te geven. Op deze manier fungeren ze als documentatie, waardoor het programma beter leesbaar is.
Deze programma's gaan uit van een **int** met een lengte van vier bytes, maar op sommige machines moeten deze variabelen als **long** worden gedeclareerd. U kunt de constante **INT_MAX** in *limits.h* controleren.

In het bestand moon.cpp:

```
// The distance to the moon converted to kilometers.
//    Title: moon

#include <iostream.h>


int main()
{
    const int moon = 238857;

    cout << "The moon's distance from Earth is " << moon;
    cout << " miles." << endl;
    int moon kilo = moon * 1.609;
    cout << "In kilometers this is " << moon kilo;
    cout << " km." << endl;
}
```

De uitvoer van het programma is:

```
The moon's distance from Earth is 238857 miles.
In kilometers this is 384320 km.
```

Analyse van het programma *moon*

```
    const int moon = 238857;
```

Het keyword **const** is nieuw in C++. Het vervangt in een aantal gevallen het gebruik van de preprocessoropdracht **define** om benoemde constanten te creëren. Als dit type modifier wordt ingezet, weet de compiler dat de geïnitialiseerde waarde van **moon** niet kan worden gewijzigd. Het maakt van **moon** dus een symbolische constante.

```
    cout << "The moon's distance from Earth is " << moon;
```

De stream-I/O in C++ kan onderscheid maken tussen een verscheidenheid aan eenvoudige waarden zonder dat hij extra opmaakinformatie nodig heeft. In dit voorbeeld wordt de waarde van **moon** afgedrukt als een integer.

```
    int moon kilo = moon * 1.609;
```

Declaraties kunnen worden geplaatst ná uitvoerbare statements. Hierdoor kunnen declaraties van variabelen dichter bij de toepassing ervan staan.

We gaan nu een programma schrijven dat een reeks waarden converteert van mijlen naar kilometers. Het programma wordt interactief. De gebruiker typt een waarde in mijlen, waarna het programma deze waarde naar kilometers converteert en afdrukt.

In het bestand mi_km.cpp:

```
// Miles are converted to kilometers.
//   Title: mi km

#include <iostream.h>

const double m to k = 1.609;

inline int convert(int mi) { return (mi * m to k); }


int main()
{
    int miles;

    do {
       cout << "Input distance in miles: ";
       cin >> miles;
       cout << "\nDistance is " << convert(miles) << " km."
            << endl;
    } while (miles > 0);
}
```

Dit programma maakt gebruik van de invoerstreamvariabele **cin**, die normaliter standaardinvoer is. De *invoeroperator* **>>** wordt de *get from*-operator of *extractie*operator genoemd, die de waarden van de invoerstream toekent aan een variabele. Dit programma illustreert invoer en uitvoer.

Bespreking van het programma *mi_km*

- `const double m to k = 1.609;`

In C++ wordt minder dan in C vertrouwd op de preprocessor. In plaats van bijvoorbeeld **define** te gebruiken worden speciale constanten, zoals de conversiefactor 1,609, toegekend aan variabelen die worden opgegeven als constanten.

- `inline int convert(int mi) { return (mi * m_to_k); }`

Het nieuwe keyword **inline** geeft op dat een functie moet worden gecompileerd, indien mogelijk als inline-code. Hierdoor hoeven geen functies te worden aangeroepen en het is beter dan de definitie van macro's in C. **inline** moet met mate worden gebruikt en

alleen voor korte functies. Merk tevens op hoe de parameter **mi** wordt gedeclareerd binnen de functiehaakjes. In C++ worden *functieprototypen* gehanteerd om functies te definiëren en te declareren. Dit wordt in de volgende paragraaf nader uitgelegd.

```
■ do {
    cout << "Input distance in miles: ";
    cin >> miles;
    cout << "\nDistance is " << convert(miles) << " km."
         << endl;
  } while (miles > 0);
```

Het programma vraagt de gebruiker herhaaldelijk om een afstand in mijlen. Het programma wordt beëindigd door een waarde die kleiner dan of gelijk is aan nul. De waarde die in de standaardinvoerstream wordt geplaatst, wordt automatisch geconverteerd naar een integerwaarde die wordt toegekend aan **miles**.

13.3Functies

De syntaxis van functies in C++ gaf aanleiding tot de nieuwe functieprototypesyntaxis die u in Standard C-compilers aantreft. Het komt erop neer dat de typen parameters binnen de header-haakjes worden opgesomd. Door het type en aantal argumenten expliciet weer te geven, zijn sterke typecontrole en toekenningscompatibele conversies mogelijk in C++.
In C++ kunnen functies over argumenten beschikken die direct by reference worden aangeroepen. Call-by-reference-parameters worden gedeclareerd met de syntaxis:

type& identifier

Parameters in C++-functies kunnen ook standaardwaarden hebben. Deze worden gegeven in de functiedeclaratie binnen de parameterlijst door

= expression

toe te voegen ná de parameter.
In het voorbeeld hierna worden deze punten geïllustreerd:

In het bestand add3.cpp:

```
// Use of a default value

#include <iostream.h>

inline void add3(int& s, int a, int b, int c = 0)
{
    s = a + b + c;
}

inline double average(int s) { return s / 3.0; }
```

```
int main()
{
    int   score 1, score 2, score 3, sum;

    cout << "\nEnter 3 scores: ";
    cin >> score 1 >> score 2 >> score 3;
    add3(sum, score 1, score 2, score 3);
    cout << "\nSum = " << sum;
    cout << "\nAverage = " << average(sum) << endl;
    add3(sum, 2 * score 1, score 2);  // use of default value 0
    cout << "\nWeighted Sum = " << sum << ".";
    cout << "\nWeighted Average = " << average(sum) << ".\n";
}
```

Analyse van het programma *add3*

- ```
  inline void add3(int& s, int a, int b, int c = 0)
  {
     s = a + b + c;
  }
  ```

De variabele **s** is call-by-reference. Een in te voegen werkelijk argument moet een **lvalue** zijn, omdat dat het adres is dat wordt gebruikt als de procedure wordt aangeroepen.

- ```
  add3(sum, score 1, score 2, score 3);
  ```

De variabele **sum** is passed-by-reference. Daarom wordt hij direct gemanipuleerd en kan hij worden ingezet om een resultaat uit de berekening van de functie te verkrijgen.

- ```
  add3(sum, 2 * score 1, score 2);  // use of default value 0
  ```

In deze aanroep van **add3()** worden drie argumenten gehanteerd. Het vierde argument neemt standaard de waarde nul aan.

13.4Klassen en abstracte datatypen

Nieuw in C++ is het type **class**. Een **class** is een uitbreiding van het concept van **struct** in traditioneel C. Een **class** zorgt voor de middelen om een user-defined datatype en geassocieerde functies en operatoren te implementeren. Daarom kan een **class** worden gebruikt om een ADT te implementeren. We gaan een klasse, **string** genaamd, schrijven die een beperkte vorm van string implementeert.

In het bestand my_string.cpp:

```
// An elementary implementation of type string.

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

```
const int max_len = 255;


class string {
public:         // universal access
    void assign(const char* st)
       { strcpy(s, st); len = strlen(st); }
    int  length() { return len; }
    void print() { cout << s << "\nLength: " << len << "\n"; }
private:        // restricted access to member functions
    char s[max_len];    // implementation by character array
    int  len;
};
```

In dit voorbeeld ziet u twee belangrijke toevoegingen aan het structuurconcept van traditioneel C: ten eerste beschikt het over leden die functies zijn, zoals **assign**; en ten tweede heeft het zowel public als private leden. Het keyword **public** duidt op de zichtbaarheid van de leden die erop volgen. Zonder dit keyword zijn de leden private voor de klasse. Private leden kunnen alleen door andere lidfuncties van de klasse worden gebruikt. Public leden zijn beschikbaar voor elke functie binnen het bereik van de klassendeclaratie. Privacy zorgt ervoor dat een deel van de implementatie van een klassentype 'hidden' kan zijn. Deze beperking voorkomt onvoorziene aanpassingen aan de datastructuur. Beperkte toegang, of *data-hiding*, is een functie van objectgeoriënteerd programmeren.
Door de declaratie van lidfuncties kan het ADT bepaalde functies op zijn eigen private representatie laten inwerken. Zo geeft de lidfunctie **length** de lengte van de gedefinieerde string terug als het aantal tekens tot aan het eerste teken met de waarde nul (dit teken wordt dus niet meegerekend). De lidfunctie **print()** voert zowel de string als de lengte ervan uit. De lidfunctie **assign()** slaat een tekenstring op in de hidden variabele **s**, berekent zijn lengte en slaat die op in de hidden variabele **len**.
We kunnen dit datatype **string** nu gebruiken alsof het standaard deel uitmaakt van de taal. Het volgt de standaardregels voor het bereik van een blokstructuur van C. Andere code die van dit type gebruikmaakt, is een *client*. De client kan alleen de public leden gebruiken om variabelen van het type **string** te beïnvloeden.

```
// Test of the class string.

int main()
{
    string  one, two;
    char    three[40] = {"My name is Charles Babbage."};

    one.assign("My name is Alan Turing.");
    two.assign(three);
    cout << three;
    cout << "\nLength: " << strlen(three) << endl;
    // Print shorter of one and two.
    if (one.length() <= two.length())
       one.print();
    else
```

```
        two.print();
}
```

De variabelen **one** en **two** zijn van het type **string**. De variabele **three** is van het type pointer naar **char** en is niet compatibel met **string**. De lidfuncties worden aangeroepen met behulp van de dot-operator of 'structuurlidoperator'. Zoals uit de definities ervan is af te leiden, werken deze lidfuncties in op de hidden private lidvelden van de benoemde variabelen. Men kan niet binnen **main** de expressie **one.len** schrijven om dit lid te benaderen. De uitvoer van dit voorbeeldprogramma is:

```
My name is Charles Babbage.
Length: 27
My name is Alan Turing.
Length: 23
```

13.5Overloading

De term *overloading* verwijst naar het meerdere betekenissen geven aan een operator of een functie. De geselecteerde betekenis hangt af van de typen argumenten die worden gebruikt door de operator of functie. We gaan de functie **print** uit het vorige voorbeeld overloaden. Dit wordt een tweede definitie van de functie **print**.

```
class string {
public:         // universal access
    .....
    void print() { cout << s << "\nLength: " << len << "\n"; }
    void print(int n)
    {
       for(int i = 0; i < n; ++i)
          cout << s << endl;
    }
    .....
}
```

Deze versie van **print** gebruikt één argument van het type **int**. Hij drukt de string **n** keer af.

```
three.print(2);    // print string three twice
three.print(-1);   // string three is not printed
```

Het is ook mogelijk om de meeste C-operatoren te overloaden. We gaan + overloaden om twee strings samen te voegen. We hebben hiervoor twee nieuwe keywords nodig: **friend** en **operator**. Het keyword **operator** gaat vooraf aan het operator-token en vervangt wat anders een functienaam in een functiedeclaratie zou zijn. Het keyword **friend** geeft een functie toegang tot de private leden van een klassenvariabele. Een **friend**-functie is geen lid van de klasse, maar heeft de privileges van een lidfunctie in de klasse waarin hij wordt gedeclareerd.

In het bestand ovl_string.cpp:

```
// Overloading the operator +

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

const int max len = 255;

class string {
public:
    void assign(const char* st) { strcpy(s, st); len = strlen(st); }
    int  length() { return len; }
    void print() { cout << s << "\nLength: " << len << endl; }
    friend string operator+(const string& a, const string& b);
private:
    char s[max len];
    int  len;
};

string operator+(const string& a, const string& b)   // overload +
{
    string temp;

    temp.assign(a.s);
    temp.len = a.len + b.len;
    if (temp.len < max len)
       strcat(temp.s, b.s);
    else
       cerr << "Max length exceeded in concatenation.\n";
    return temp;
}
void print(const char* c)   // file scope print definition
{
    cout << c << "\nLength: " << strlen(c) << "\n";
}

int main()
{
    string   one, two, both;
    char     three[40] = {"My name is Charles Babbage."};

    one.assign("My name is Alan Turing.");
    two.assign(three);
    print(three);         // file scope print called
    // Print shorter of one and two.
    if (one.length() <= two.length())
       one.print();       // member function print called
    else
       two.print();
    both = one + two;     // plus overloaded to be concatenate
```

```
    both.print();
}
```

Analyse van de functie operator+

```
string operator+(const string& a, const string& b)
```

Plus wordt overloaded. De twee argumenten die hij gebruikt, zijn beide strings. De argumenten zijn call-by-reference. Het gebruik van **const** duidt erop dat de argumenten niet kunnen worden aangepast.

```
string temp;
```

De functie moet een waarde van het type **string** teruggeven. Deze lokale variabele wordt gebruikt om de samengevoegde stringwaarde op te slaan en terug te geven.

```
temp.assign(a.s);
temp.len = a.len + b.len;
if (temp.len < max len)
   strcat(temp.s, b.s);
```

De string **a.s** wordt gekopieerd in **temp.s** door de aanroep van de libraryfunctie **strcpy()**. De lengte van de resulterende samengevoegde string wordt getest om te controleren of hij de maximumlengte voor strings niet overschrijdt. Als de lengte acceptabel is, wordt de standaardlibraryfunctie **strcat()** aangeroepen met de hidden stringleden **temp.s** en **b.s**. De verwijzingen naar **temp.s**, **a.s** en **b.s** zijn toegestaan, omdat deze functie een **friend** is van de klasse **string**.

```
cerr << "Max length exceeded in concatenation.\n";
```

De standaardfoutstream **cerr** wordt gebruikt om een foutboodschap af te drukken. Er vindt nu geen samenvoeging plaats. Alleen de eerste string wordt teruggegeven.

```
return temp;
```

Aan de operator is een returntype van **string** teruggegeven, en aan **temp** is de juiste samengevoegde string toegekend.

13.6Constructors en destructors

Een *constructor* is een lidfunctie die als taak heeft om een variabele van zijn klasse te *initialiseren*. In OOP-jargon wordt zo'n variabele aangeduid met de term *object*. In veel gevallen houdt dit dynamische toewijzing van opslagruimte in. Constructors worden elke keer aangeroepen als er een object van zijn geassocieerde klasse wordt gecreëerd. Een *destructor* is een lidfunctie die als taak heeft een variabele van zijn klasse *vrij te geven*. De destructor wordt impliciet aangeroepen als een automatisch object buiten bereik raakt.
We gaan ons **string**-voorbeeld aanpassen door dynamisch opslagruimte toe te kennen voor elke **string**-variabele. We vervangen de private arrayvariabele met een pointer. De

opnieuw gemodelleerde klasse gebruikt een constructor om dynamisch een correcte hoeveelheid opslagruimte toe te kennen met de operator **new**.

```
// An implementation of dynamically allocated strings.

class string {
public:
    string(int n) { s = new char[n + 1]; len = n; }  // constructor
    void assign(const char* st)
       { strcpy(s, st); len = strlen(st); }
    int  length() { return len; }
    void print() { cout << s << "\nLength: " << len << "\n"; }
    friend string operator+(const string& a, const string& b);
private:
    char* s;
    int   len;
};
```

Een constructor is een lidfunctie waarvan de naam gelijk is aan die van de klasse. Het keyword **new** is een toevoeging aan de taal C. Het is een unaire operator die als argument een datatype gebruikt dat een arraygrootte kan bevatten. Het reserveert de juiste hoeveelheid vrij geheugen om dit type op te slaan en geeft de pointer-waarde terug die naar dit deel van het geheugen wijst. In het voorgaande voorbeeld zouden er **n + 1** bytes in het geheugen worden gereserveerd. De declaratie

```
string a(40), b(100);
```

zou dus 41 bytes reserveren voor de variabele **a**, waarnaar wordt gewezen door **a.s**, en 101 bytes voor de variabele **b**, waarnaar wordt gewezen door **b.s**. Er moet één byte worden toegevoegd voor de end-of-string-waarde 0. Opslagruimte verkregen door **new** is persistent en wordt niet automatisch teruggegeven als een blok wordt verlaten. Als gewenst is dat die opslagruimte wordt vrijgegeven, moet er een destructor-functie in de klasse worden geplaatst. Een destructor wordt geschreven als een normale lidfunctie waarvan de naam gelijk is aan die van de klasse die wordt voorafgegaan door het tildesymbool: ~. Normaliter gebruikt een destructor de unaire operator **delete** of **delete[]**, een andere uitbreiding van de taal, om automatisch opslagruimte vrij te geven die is geassocieerd met een pointer-expressie. De **delete[]** wordt gebruikt wanneer **new *type* [size]** is gebruikt voor reservering.

```
// Add as a member function to class string.
~string() { delete []s; }   // destructor
```

Het is gebruikelijk om de constructor te overloaden door een aantal van dat soort functies te schrijven om zich aan te passen aan meer dan één initialisatiestijl. Bekijk de initialisatie van een string met een pointer naar **char**-waarde. Zo'n constructor is

```
string(const char* p)
{
    len = strlen(p);
    s = new char[len + 1];
```

```
    strcpy(s, p);
}
```

Een gangbare declaratie om deze versie van de constructor aan te roepen, is

```
char*   str = "I came on foot.";
string  a("I came by bus."), b(str);
```

Het zou ook gewenst zijn om over een constructor zonder argumenten te beschikken:

```
string() { len = 255; s = new char[255]; }
```

Deze zou worden aangeroepen door declaraties zonder argumenten tussen haakjes en zou, standaard, 255 bytes in het geheugen reserveren. Nu zouden alle drie constructors worden aangeroepen in de volgende declaratie:

```
string  a, b(10), c("I came by horse.");
```

De overloaded constructor wordt geselecteerd door de vorm van elke declaratie. De variabele **a** heeft geen parameters, waardoor er aan deze variabele 255 bytes worden toegewezen. De variabele **b** heeft een integer-parameter, zodat er 11 bytes voor worden gereserveerd. De variabele **c** heeft een pointer-parameter voor de letterlijke string "I came by horse.", zodat er 17 bytes voor worden vrijgemaakt, met deze letterlijke string gekopieerd in zijn private **s**-lid.

13.7Objectgeoriënteerd programmeren en overerving

Een nieuw concept in OOP is het *overerving*smechanisme. Dit is het mechanisme van het *afleiden* van een nieuwe klasse vanaf een bestaande klasse, de *basisklasse* genaamd. De afgeleide klasse voegt iets toe aan of wijzigt de overgeërfde basisklassenleden. Dit wordt toegepast om code en interface te delen en om een hiërarchie van verwante typen te creëren.
Hiërarchie is een methode om met complexiteit om te gaan. Het classificeert objecten. De periodieke tabel der elementen kent bijvoorbeeld elementen die een gas zijn. Deze hebben eigenschappen die worden gedeeld door alle elementen in die classificatie. Inerte gassen zijn een belangrijke speciale klasse van gassen. De hiërarchie is hier als volgt: een inert gas, zoals argon, is een gas dat op zijn beurt een element is. Deze hiërarchie biedt een uitstekende manier om het gedrag van inerte gassen te begrijpen. We weten dat ze zijn opgebouwd uit protonen en elektronen, omdat deze beschrijving wordt gedeeld met alle elementen. We weten dat ze bij kamertemperatuur in gasvorm verkeren, omdat dit gedrag op alle gassen van toepassing is. Verder weten we dat ze niet chemisch reageren met andere elementen, omdat dit gedrag wordt gedeeld met alle inerte gassen.
We gaan ons hoofd eens buigen over een database voor een universiteit. Het administratief hoofd moet de verschillende typen studenten bijhouden. De basisklasse die we moeten ontwikkelen, legt een beschrijving van 'student' vast. Twee hoofdcategorieën van student zijn graduate (post-doctoraal student) en undergraduate (doctoraal student).
Hier ziet u de OOP-ontwerpmethodiek:

OOP-ontwerpmethodiek

1. Bepaal een toepasselijke reeks typen.
2. Ontwerp hun onderlinge samenhang.
3. Gebruik overerving om code te delen.

Hier ziet u een voorbeeld van het afleiden van een klasse:

```
enum support { ta, ra, fellowship, other };
enum year { fresh, soph, junior, senior, grad };

class student {
public:
    student(char* nm, int id, double g, year x);
    void  print();
private:
    int       student id;
    double    gpa;
    year      y;
    char      name[30];
};

class grad student: public student {
public:
    grad student
       (char* nm, int id, double g, year x, support t,
        char* d, char* th);
    void print();
private:
    support   s;
    char      dept[10];
    char      thesis[80];
};
```

In dit voorbeeld is **grad_student** de afgeleide klasse en **student** de basisklasse. Het gebruik van het keyword **public** na de dubbelepunt in de header van de afgeleide klasse betekent dat de public leden van **student** moeten worden overgeërfd als public leden van **grad_student**. Private leden van de basisklasse kunnen niet worden benaderd in de afgeleide klasse. Public overerving houdt tevens in dat de afgeleide klasse **grad_student** een subtype van **student** is.
Een overervingsstructuur biedt een ontwerp voor het totale systeem. Een database die alle mensen in een universiteit bevat, zou kunnen worden afgeleid van de basisklasse **person**. De relatie tussen **student** en **grad_student** zou kunnen worden uitgebreid tot extramurale studenten, als een andere belangrijke categorie objecten. Op soortgelijke wijze zou **person** de basisklasse kunnen zijn voor een aantal categorieën van werknemers.

13.8Polymorfisme

Een *polymorfische* functie kent vele vormen. Een voorbeeld in Standard C is de deeloperator. Als de argumenten voor de deeloperator integers zijn, kan integer-deling worden toegepast. Als één of beide argumenten echter van het type floating-point zijn, wordt de floating-point-deling gehanteerd.
In C++ is een functienaam of operator overloadbaar. Een functie wordt aangeroepen op basis van zijn *signatuur*, gedefinieerd als de lijst met argumenttypen in zijn parameterlijst.

```
a / b        // divide behavior determined by native coercions
cout << a    // overloading << the shift operator for output
```

In de deelexpressie hangt het resultaat af van de argumenten die automatisch naar het breedste type worden gedwongen. Als beide argumenten integer zijn, is het resultaat een integer-deling. Als één of beide argumenten floating-point zijn, is het resultaat een floating-point-getal. In het uitvoerstatement roept de shift-operator **<<** een functie aan die in staat is een object van het type **a** uit te voeren.
Polymorfisme lokaliseert verantwoordelijkheid voor gedrag. De client-code hoeft meestal niet te worden aangepast als er extra functionaliteit wordt toegevoegd aan het systeem via door ADT geleverde verbeteringen in de code.
In C zou de techniek voor de implementatie van een pakket routines om een ADT-vorm te leveren vertrouwen op een uitgebreide structurele beschrijving van een willekeurige vorm.

```
struct shape {
    enum{CIRCLE, ·····} e val;
    double center, radius;
·····
};
```

zou over alle leden beschikken die noodzakelijk zijn voor elke willekeurige vorm die op dit moment in ons systeem kan worden getekend, plus een enumerator-waarde zodat hij kan worden geïdentificeerd. De area-routine zou dan worden geschreven als

```
double area(shape* s)
{
    switch(s -> e val) {
    case CIRCLE:  return (PI * s -> radius * s -> radius);
    case RECTANGLE: return (s -> height * s -> width);
    ·····
}
```

Vraag: Wat is er nodig om aan deze C-code een nieuw vorm toe te voegen?
Antwoord: Een extra case in de codebody en extra leden in de structuur. Helaas zouden deze in de gehele codebody doorwerken. Elke routine die zo wordt gestructureerd, moet een extra case hebben, zelfs als die case alleen een label toevoegt aan een reeds bestaande case. Dus wat conceptueel een lokale verbetering is, vereist globale wijzigingen.
OOP-coderingstechnieken in C++ voor hetzelfde probleem maken gebruik van een vormhiërarchie. De hiërarchie is die waar *circle* en *rectangle* worden afgeleid van *shape*. Tijdens het revisieproces worden codeverbeteringen in de nieuwe afgeleide

klasse ingevoegd, zodat extra beschrijving wordt gelokaliseerd. De programmeur overschrijft de betekenis van eventuele aanpaste routines – in dit geval de nieuwe berekening van het oppervlak. Client-code die niet gebruikmaakt van het nieuwe type wordt niet beïnvloed. Client-code die door het nieuwe type wordt verbeterd, wordt normaliter minimaal gewijzigd.
C++-code die dit ontwerp volgt, gebruikt **shape** als een *abstracte basisklasse*. Dit is een klasse die één of meer pure virtuele functies bevat. Een pure virtuele functie kent geen definitie. De definitie wordt geplaatst in een afgeleide klasse.

```
// shape is an abstract base class

class shape {
public:
    virtual double area() = 0;     // pure virtual function
};

class rectangle: public shape {
public:
    rectangle(double h, double w): height(h), width(w) {}
    double area() { return (height * width); }    // overridden
private:
    double height, width;
};

class circle: public shape {
public:
    circle(double r): radius(r) {}
    double area() { return ( 3.14159 * radius * radius); }
private:
    double radius;
};
```

Client-code voor het berekenen van een willekeurig oppervlak is polymorfisch. De juiste **area()**-functie wordt tijdens runtime geselecteerd.

```
shape*  ptr shape;
    .....
    cout << " area = " << ptr shape -> area();
    .....
```

Stel nu eens dat we onze hiërarchie met typen verbeteren door een **square**-klasse in het leven te roepen.

```
class square: public rectangle {
public:
    square(double h): rectangle(h,h) {}
    double area() { return rectangle::area(); }
};
```

De client-code blijft ongewijzigd. Dit zou niet het geval zijn geweest met de niet-OOP-code.

13.9Sjablonen

C++ maakt gebruik van het keyword **template** om te voorzien in *parametrisch polymorfisme*. Met parametrisch polymorfisme kan dezelfde code voor verschillende typen worden gebruikt, wanneer het type een parameter is van de codebody. De code wordt generiek geschreven om in te werken op **class T**. De sjabloon wordt gebruikt om verschillende echte klassen te genereren als **class T** wordt vervangen door een echt type.
Een belangrijke toepassing van deze techniek is het schrijven van generieke *container-klassen*. Een container-klasse wordt gebruikt om data van een bepaald type te bewaren. Stacks, vectoren, trees en lijsten zijn alle voorbeelden van standaard container-klassen. We gaan een stack-container-klasse ontwikkelen als een geparametriseerd type.

In het bestand stack.cpp:

```
// template stack implementation

template <class TYPE>
class stack {
public:
    stack(int size = 1000) :max len(size)
       { s = new TYPE[size]; top = EMPTY; }
    ~stack() { delete []s; }
    void reset() { top = EMPTY; }
    void push(TYPE c) { s[++top] = c; }
    TYPE pop() { return s[top--]; }
    TYPE top of() { return s[top]; }
    bool empty() { return top == EMPTY; }
    bool full() { return top == max len - 1; }
private:
    enum  {EMPTY = -1};
    TYPE* s;
    int   max len;
    int   top;
};
```

De syntaxis van de klassendeclaratie wordt voorafgegaan door

```
template <class identifier >
```

Deze identifier is een sjabloonargument dat staat voor een willekeurig type. In de klassendefinitie kan het sjabloonargument worden gebruikt als een typenaam. Dit argument wordt geïnstantieerd in de werkelijke declaraties. Hier ziet u een voorbeeld van een **stack**-declaratie waarin dit wordt toegepast

```
stack<char> stk ch;              // 1000 element char stack
stack<char*>  stk str(200);      // 200 element char* stack
stack<complex> stk_cmplx(100);   // 100 element complex stack
```

Dit mechanisme zorgt ervoor dat we klassendeclaraties, waarbij alleen de typedeclaratie zou veranderen, niet opnieuw hoeven te schrijven.
Als we zo'n type verwerken, moet de code altijd gebruikmaken van vishaken als onderdeel van de declaratie. Hier ziet u twee functies die de **stack**-sjabloon benutten:

```
// Reversing a series of char* represented strings

void reverse(char* str[], int n)
{
    stack<char*> stk(n);   // this stack holds char*
    for (int i = 0; i < n; ++i)
       stk.push(str[i]);
    for (i = 0; i < n; ++i)
       str[i] = stk.pop();
}
```

In de functie **reverse()** wordt **stack<char*>** gebruikt om **n** strings in te voegen, waarna ze in omgekeerde volgorde worden weergegeven.

```
// Initializing a stack of complex numbers from an array
void init(complex c[], stack<complex>& stk, n)
{
    for (int i = 0; i < n; ++i)
       stk.push(c[i]);
}
```

In de functie **init()** wordt een **stack<complex>**-variabele by reference doorgegeven en worden **n** complexe getallen op deze stack geplaatst. U ziet dat we in dit voorbeeld **bool** gebruiken. Dat is een basistype in C++ dat als waarden **true** en **false** kent.

13.10 C++-excepties

C++ introduceert een contextgevoelig mechanisme voor de afhandeling van excepties. De context voor het werpen van een exceptie is een **try**-blok. Gedeclareerde handlers die het keyword **catch** gebruiken, treft u aan het eind van een **try**-blok aan.
Er wordt een exceptie geworpen door het gebruik van de expressie **throw**. De exceptie wordt afgehandeld door het aanroepen van een correcte *handler* die wordt geselecteerd uit een lijst met handlers die direct ná het **try**-blok van de handler staat. Hier ziet u een eenvoudig voorbeeld:

```
// stack constructor with exceptions

stack::stack(int n)
{
    if (n < 1)
       throw (n);       // want a positive value
    p = new char[n];    // create a stack of characters
    if (p == 0)         // if new returns 0 when it fails
       throw ("FREE STORE EXHAUSTED");
}
```

```
void g()
{
    try {
       stack  a(n), b(n);
       .....
    }
    catch (int n) { ... }           // an incorrect size
    catch (char* error) { ... }     // free store exhaustion
}
```

De eerste **throw()** heeft een integer-argument en komt overeen met de **catch(int n)**-signatuur. Van deze handler wordt verwacht dat hij een correcte actie onderneemt als een onjuiste arraygrootte is doorgegeven als argument van de constructor. Een foutboodschap en afbreken zijn bijvoorbeeld normaal. De tweede **throw()** heeft een pointer naar **char**-argument en komt overeen met de **catch(char* error)**-signatuur. Moderne ANSI C++-compilers werpen de standaardexceptie **bad_alloc** als **new** faalt. Oudere systemen gaven de null-pointer-waarde 0 terug als **new** faalde.

13.11 Voordelen van objectgeoriënteerd programmeren

Het centrale element van OOP is de inkapseling van een correcte reeks datatypen en hun bewerkingen. De klassenconstructie, met zijn lidfuncties en dataleden, voorziet in een goede coderingstool. Klassenvariabelen zijn de *objecten* die moeten worden gemanipuleerd.
Klassen zorgen ook voor data-hiding. Toegangsprivileges kunnen worden beheerd en beperkt tot de groep met functies die toegang nodig heeft tot implementatiedetails. Dit bevordert modulariteit en robuustheid.
Een ander belangrijk concept in OOP is de promotie van codehergebruik via het *overerving*smechanisme. Dit is het mechanisme waarbij een nieuwe klasse wordt *afgeleid* van een bestaande klasse, de *basis*klasse genaamd. De basisklasse kan worden uitgebreid of worden aangepast om de afgeleide klasse te creëren. Op deze manier kan er een hiërarchie van verwante datatypen worden gecreëerd die code delen.
Veel nuttige datastructuren zijn varianten van elkaar, en het is vaak vervelend om voor elk hiervan dezelfde code te schrijven. Een afgeleide klasse overerft de beschrijving van de basisklasse. Deze klasse kan vervolgens worden aangepast door het toevoegen van extra leden, het overloaden van bestaande lidfuncties en het wijzigen van toegangsprivileges. Zonder dit mechanisme van hergebruik zou er voor elke kleine aanpassing code moeten worden gekopieerd.
De OOP-programmeertaak is vaak moeilijker dan normaal procedureel programmeren zoals in C. Er is ten minste één extra ontwerpstap vereist voordat de algoritmen kunnen worden gecodeerd die zijn betrokken bij de typehiërarchie die bruikbaar is voor het op handen zijnde probleem. Vaak lost men het probleem algemener op dan strikt noodzakelijk is.
De gedachte is dat OOP op een aantal manieren zijn investering terugbetaalt. De oplossing is meer ingekapseld en daardoor robuuster en eenvoudiger te onderhouden en aan te passen. Bovendien is de oplossing beter herbruikbaar. Op plekken waar de code

een stack nodig heeft, kan die stack eenvoudiger worden geleend van bestaande code. In een normale procedurele taal wordt zo'n datastructuur vaak in het algoritme opgenomen en kan niet worden geëxporteerd.
Al deze voordelen zijn met name van belang bij grote coderingsprojecten waarbij regelmatig overleg tussen programmeurs plaatsvindt. Door de mogelijkheid om in header-bestanden algemene interfaces op te geven voor verschillende klassen, kan elke programmeur aan afzonderlijke codesegmenten werken met een hoge mate van onafhankelijkheid en integriteit.
OOP is voor iedereen weer anders. Pogingen om het te definiëren zijn analoog aan het verhaal van de blinde wijze man die een olifant probeert te beschrijven. We geven nog één vergelijking:

OOP = type-uitbreidbaarheid + polymorfisme

Samenvatting

1. De dubbele schuine streep, //, is een nieuw commentaarsymbool. Het commentaar loopt tot aan het eind van de regel. De oude C-stijl commentaarsymbolen /* */ zijn nog steeds beschikbaar voor meerregelig commentaar.
2. De *iostream.h*-header introduceert I/O-faciliteiten voor C++. De identifier **cout** is de naam van de standaarduitvoerstream. De operator **<<** geeft zijn argument door aan de standaarduitvoer. Als **<<** op deze manier wordt toegepast, wordt hij de *put to*-operator genoemd. De identifier **cin** is de naam van de standaardinvoerstream. De operator **>>** is de invoeroperator, *get from* genaamd, die de waarden van de invoerstream toekent aan een variabele.
3. In C++ wordt minder dan in C op de preprocessor vertrouwd. In plaats van **define** te gebruiken, worden er speciale constanten toegekend aan variabelen die als **const** worden gespecificeerd. Het nieuwe keyword **inline** geeft an dat een functie inline moet worden gecompileerd om functieaanroep-overhead te voorkomen. Als regel geldt dat dit met mate en alleen voor korte functies moet worden gedaan.
4. De syntaxis van functies in C++ gaf aanleiding tot de nieuwe functieprototypesyntaxis die u in Standard C-compilers aantreft. De parametertypen worden weergegeven binnen de haakjes van de header – bijvoorbeeld, **void add3(int&, int, int, int)**. Call-by-reference en standaardparameterwaarden zijn beschikbaar. Door expliciet het type en het aantal argumenten op te sommen zijn sterke typecontrole en toekenningscompatibele conversies in C++ mogelijk.
5. Nieuw in C++ is het type **class**. Een **class** is een uitbreiding van het **struct**-principe in traditioneel C. Het gebruik ervan is een manier om een datatype en geassocieerde functies en operatoren te implementeren. Daarom is een **class** een implementatie van een abstract datatype (ADT). Er zijn twee belangrijke toevoegingen aan het structuurconcept: ten eerste bevat hij leden die functies zijn, en ten tweede maakt hij gebruik van de toegangskeywords **public**, **private** en **protected**. Deze keywords duiden op de zichtbaarheid van de leden die volgen. Public leden zijn beschikbaar voor elke functie binnen het bereik van de klassendeclaratie. Private leden kunnen alleen door andere lidfuncties van de klasse worden gebruikt. Protected leden zijn alleen beschikbaar voor gebruik door andere lidfuncties van de klasse en door afgeleide klassen. Privacy maakt het mogelijk dat een deel van de implementatie van een klassentype 'hidden' kan zijn.

6. De term *overloading* verwijst naar het geven van meerdere betekenissen aan een operator of een functie. De geselecteerde betekenis hangt af van de typen argumenten die door de operator of functie worden gebruikt.
7. Een constructor is een lidfunctie die als taak heeft om een variabele van zijn klasse te initialiseren. In veel gevallen houdt dit dynamische toekenning van geheugen in. Constructors worden aangeroepen als een object van zijn geassocieerde klasse wordt gecreëerd. Een destructor is een lidfunctie die als taak heeft een variabele van zijn klasse vrij te geven. De destructor wordt impliciet aangeroepen als een automatisch object buiten bereik raakt.
8. Het centrale element van objectgeoriënteerd programmeren (OOP) is de inkapseling van een correcte reeks datatypen en hun bewerkingen. Deze user-defined typen zijn ADT's. De klassenconstructie, met zijn lidfuncties en dataleden, zorgt voor een juiste coderingstool. Klassenvariabelen zijn de *objecten* die moeten worden gemanipuleerd.
9. Een ander belangrijk concept in OOP is de promotie van het hergebruiken van code via het *overerving*smechanisme. Dit is het mechanisme waarbij een nieuwe klassen wordt *afgeleid* van een bestaande klasse, de *basisklasse* genaamd. De basisklasse kan worden uitgebreid of worden gewijzigd om de afgeleide klasse te creëren. Op deze manier kan er een hiërarchie van verwante datatypen worden samengesteld die code delen. Deze typehiërarchie kan dynamisch worden gebruikt door **virtual** functies. Virtuele lidfuncties in een basisklasse worden overloaded in een afgeleide klasse. Deze functies staan dynamische ofwel runtime-typering toe. Een pointer naar de basisklasse kan ook verwijzen naar objecten van de afgeleide klassen. Als zo'n pointer wordt ingezet om te wijzen naar de overloaded virtuele functie, bepaalt hij dynamisch welke versie van de lidfunctie moet worden aangeroepen.
10. Een *polymorfische* functie kent vele vormen. Een **virtual**-functie staat runtime-selectie toe van een groep functies die worden overschreven binnen een typehiërarchie. Een voorbeeld in de tekst is de berekening van het oppervlak binnen de **shape**-hiërarchie. Client-code voor het berekenen van een willekeurig oppervlak is polymorfisch. De juiste **area()**-functie wordt tijdens runtime geselecteerd.
11. C++ gebruikt het keyword **template** om te voorzien in *parametrisch polymorfisme*. Parametrisch polymorfisme staat toe dat dezelfde code wordt gebruikt met betrekking tot verschillende typen, waarbij het type een parameter is van de codebody. De code wordt generiek geschreven om in te werken op **class T**. De sjabloon wordt gebruikt om verschillende echte klassen te genereren als **class T** wordt vervangen door een echt type.
12. C++ introduceert een contextgevoelig mechanisme voor de afhandeling van excepties. De context voor het werpen van een exceptie is een **try**-blok. Gedeclareerde handlers die het keyword **catch** gebruiken, treft u aan het eind van een **try**-blok aan. Er wordt een exceptie geworpen door het gebruik van de **throw**-expressie. De exceptie wordt afgehandeld door het aanroepen van een correcte *handler*, die wordt geselecteerd uit een lijst met handlers die direct ná het **try**-blok van de handler staat.

Oefeningen

1. Schrijf, gebruikmakend van stream-I/O, in het scherm de woorden

   ```
   she sells sea shells by the seashore
   ```

(a) alle in één regel, (b) in drie regels, (c) binnen een kader.

2. Schrijf een programma dat afstanden in yards converteert naar afstanden in meters. 1 meter komt overeen met 1,0936 yards. Maak gebruik van **cin** om de afstanden in te lezen. Het programma moet een lus zijn die deze berekening uitvoert totdat het als invoer een waarde ontvangt die kleiner dan of gelijk is aan nul.
3. Het volgende programma leest drie integers in en drukt de som ervan af. Merk op dat de expressie **!cin** wordt gehanteerd om te testen of invoer in **a**, **b** en **c** is gelukt. Om de **for**-lus te verlaten kan de gebruiker **bye** of **exit** typen, of iets anders dat niet naar een integer kan worden geconverteerd. Experimenteer wat met dit programma zodat u de effecten ervan begrijpt.

```
#include <iostream.h>

int main()
{
   int   a, b, c, sum;

   cout << "---\n"
           "Integers a, b, and c will be summed.\n"
           "\n";
   for ( ; ; ) {
      cout << "Input a, b, and c: ";
      cin >> a >> b >> c;
      if (!cin)
         break;
      sum = a + b + c;
      cout << "\n"
              "   a + b + c = " << sum << "\n"
              "\n";
   }
   cout << "\nBye!\n\n";
}
```

4. Sommige C++-systemen kennen een ‘groot integer’-type. In GNU C++ wordt dit type bijvoorbeeld aangeduid met de term **Integer**. We schrijven een programma dat gebruikmaakt van dit type om faculteiten te berekenen.

```
#include <assert.h>
#include <iostream.h>
#include <Integer.h>            // valid for GNU g++\

int main()
{
      int       i;
      int       n;
      Integer   product = 1;

      cout << "The factorial of n will be computed.\n"
              "\n"
              "Input n: ";
```

```
    cin >> n;
    assert(cin && n >= 0);
    for (i = 2; i <= n; ++i)
       product *= i;
    cout << "\n"
            "factorial(" << n << ") = " << product << "\n"
            "\n";
}
```

U ziet dat het programma lijkt op een faculteitprogramma dat in C is geschreven. Als we dit programma echter uitvoeren, zien we bijvoorbeeld dat

```
factorial(37) = 13763753091226345046315979581580902400000000
```

Als we gebruikmaken van een type als **int** of **double**, kunnen we niet zulke grote integers genereren, iets wat met een groot-integertype wel kan. Als u beschikt over GNU C++, moet u dit programma eens proberen. Anders moet u nagaan of uw C++-systeem een groot-integertype kent; als dat het geval is, kunt u een faculteitprogramma schrijven dat op dit programma lijkt. Als uw programma 100 faculteit correct uitwerkt, eindigt het met 24 nullen. Is dat het geval?

5. Open een werkend programma, laat elke regel om beurten weg en voer het uit via de compiler. Noteer de foutboodschappen die elk van die verwijderingen veroorzaakt. Gebruik bijvoorbeeld de code uit oefening 3.
6. Schrijf een programma dat interactief vraagt om uw *name* en *age* en antwoordt met

   Hello *name*, next year you will be *next age*.

   waarbij *next_age* gelijk is aan *age + 1*.
7. Schrijf een programma dat een tabel afdrukt met daarin kwadraten, wortels en getallen tot de derde macht. Maak gebruik van tabs of strings met spaties om een keurig uitgelijnde tabel in het scherm te krijgen.

i	i * i	square root	i * i * i
1	1	1.00000	1
2	4	1.41421	8
.....			

8. De verwisselfunctie in C is

```
void swap(int *i, int *j)
{
   int  temp;

   temp = *i;
   *i = *j;
   *j = temp;
}
```

Herschrijf dit programma door gebruik te maken van referentieparameters en test het programma.

```
void swap(int& i, int& j);
```

9. In traditioneel C, maar niet in ANSI C of C++, veroorzaakt de volgende code een fout:

```
#include <math.h>
#include <stdio.h>

int main()
{
   printf("%f is the square root of 2.\n", sqrt(2));
   return 0;
}
```

   Verklaar waarom dit gebeurt en waarom de functieprototypen in C++ dit probleem voorkomen. Herschrijf het programma door gebruik te maken van *iostream.h*.
10. Voeg een lidfunctie **reverse** toe aan de klasse **string** in paragraaf 13.4, 'Klassen en abstracte datatypen'. Deze functie keert de onderliggende representatie van de tekenreeks om die is opgeslagen in het private lid **s**.
11. Voeg een lidfunctie **void print(int pos, int k)** toe aan de klasse **string** in paragraaf 13.4, 'Klassen en abstracte datatypen'. Deze functie overloadt **print()** en heeft als taak de **k** tekens van de string af te drukken, beginnend bij positie **pos**.
12. Overload de operator * in de klasse **string**. Zijn liddeclaratie wordt

```
string string::operator*(int n);
```

   De expressie **s * k** wordt een string die **k** kopieën is van de string **s**. Ga na of dit de opslagcapaciteit overschrijdt.
13. Schrijf een klasse **person** die basisinformatie bevat, zoals naam, geboortedatum en adres. Leidt de klasse **student** af van **person**.
14. Schrijf een klasse **triangle** die overerft van **shape**. Deze klasse moet over zijn eigen **area()**-lidfunctie beschikken.
15. De functie **reverse()** kan als volgt generiek worden geschreven:

```
// generic reversal

template <class T>
void reverse(T v[], int n)
{
   stack<T> stk(n);
   for (int i = 0; i < n; ++i)
      stk.push(v[i]);
   for (i = 0; i < n; ++i)
      v[i] = stk.pop();
}
```

   Probeer dit op uw systeem door hem te gebruiken om een array met tekens om te keren. Doe hetzelfde voor een array met **char***.
16. (S.Clamage.) De volgende drie programma's gedragen zich verschillend:

```
// Function declarations at file scope

int f(int);
double f(double);   // overloads f(int)

double add f()
{
   return (f(1) + f(1.0));   // f(int) + f(double)
}
```

Nu plaatsen we intern één functiedeclaratie.

```
// Function declaration at local scope

int f(int);

double add_f()
{
   double f(double);           // hides f(int)
   return (f(1) + f(1.0));     // f(double) + f(double)
}
```

Nu plaatsen we de andere functiedeclaratie intern.

```
double f(double);

double add f()
{
   int f(int);
   return (f(1) + f(1.0));     // What is called here?
}
```

Schrijf enkele testprogramma's die duidelijk de verschillen in gedrag aantonen.