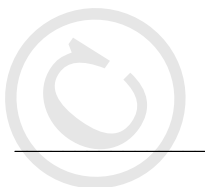


Student-notities
De programmeertaal C++
Voorbeeld-hoofdstuk

03. De class

 **at computing**
The Linux/UNIXperts

Nijmegen



Copyright © AT Computing 1992-2000
Versie: 2f

Student-notities

- De student-notities in dit voorbeeld-hoofdstuk zijn fragmenten uit het dictaat (578 pagina's) dat bij deze cursus wordt meegeleverd. □

In C++ wordt de beschrijving van een datatype een *class* genoemd. Een class is een samengesteld datatype zoals een *structure* in de programmeertaal C. Naast datamembers worden in een class ook de functies gedeclareerd die de datamembers mogen manipuleren. Op deze wijze worden data en toegestane bewerkingen op die data samengevoegd, hetgeen misverstanden (en misbruik) voorkomt. De toegestane bewerkingen op een class worden de *interface* of het *contract* genoemd.

Een *object* is een variabele van een class. Bijvoorbeeld: het dak van mijn huis (een concreet dak) is een object van de class *dak* (een algemene beschrijving van het datatype). Een object wordt ook wel een *instantiatie* van de class genoemd.

Data abstractie en Encapsulatie

Data abstractie

- nieuwe datatypen ontstaan door:
 - koppeling elementen in één klasse: class
 - koppeling functies aan die class

Encapsulatie (Data hiding)

- data elementen ontoegankelijk maken
- interne functies ontoegankelijk maken

Class interface

- de niet-interne functies vormen de service van de class

Figuur 1.

Student-notities

In diverse talen is het mogelijk vast te leggen dat een object een samenstelling is van bepaalde basiselementen. COBOL en Pascal kennen bijvoorbeeld de *record*-structuur; het equivalent in C is de *structure*.

De *structure* is een goed middel om diverse gerelateerde variabelen te groeperen, zoals de volgende omschrijving voor een handelsartikel¹:

```
struct Art{
    char *naam;
    float prijs;
    char btwcode;
};

struct Art spaghetti = { "spaghetti", 2.65, 'H' };
```

Om de *members* van *structure* Art te kunnen gebruiken, moeten twee dingen genoemd worden: een variabele van type `struct Art` én het betreffende member. Deze dienen door de memberoperator te zijn verbonden:

```
cout << spaghetti.naam << "\n";

spaghetti.prijs = 2.67;
```

De memberoperator kan twee vormen hebben, een punt (.) zoals hierboven, of een pijltje (->) om de notatie bij pointers naar structures wat duidelijker weer te geven:

```
struct Art *ptp;

ptp = &spaghetti;

cout << ptp->naam << "\n";
```



1. Merk op dat we in de student-notities consequent een ander voorbeeld hanteren dan op de sheets.

Data abstractie in C/C++ de structure

- Een **struct** is een samengesteld datatype

```
struct Boek {  
    char *titel;  
    char *auteur;  
    long isbn;  
};
```

- **Boek** is een abstract datatype
- De members van structure **Boek** worden gebruikt via de member-operator:

```
Boek eenroman;  
  
eenroman.titel = "Bint";  
  
cout << "ISBN: " << eenroman.isbn;
```

- Het keyword **struct** is bij variabele-declaratie niet nodig in C++ (in tegenstelling tot in C)

Figuur 2.

Student-notities

Het is volstrekt duidelijk dat `spaghetti.naam`, `spaghetti.prijs` en `spaghetti.btwcode` aan elkaar gerelateerd zijn via de structure `Art spaghetti`. Wat er echter niet staat, is door *wie* (welke code) de members van zo'n structure allemaal gebruikt worden en dat is erg belangrijk in verband met het software-onderhoud. Als namelijk het member `prijs` in plaats van een float een double zou worden, moet *iedereen* zijn code aanpassen.

De volgende oplossing in C++ heeft ook functiedeclaraties in de structure-declaratie staan; dit zijn declaraties van functies die de datamembers gebruiken. Met andere woorden: zo leggen we vast welke functies de datamembers manipuleren en relateren we deze functies aan de structure:

```
struct Art{
    char *naam;
    float prijs;
    char btwcode;

    char *get_naam(); // return pointer naar naam
    float get_prijs(); // return prijs
    float get_btwpercentage(); // return btw percentage
};
```

De functies in de declaratie van `struct Art` heten *memberfuncties* en worden net als *datamembers* gebruikt via de memberoperator.

```
Art spaghetti = { "spaghetti", 2.65, 'H' };
float c, p;

c = spaghetti.get_btwpercentage();
cout << "Het btwpercentage = " << c << "\n";

p = spaghetti.get_prijs();
cout << "De prijs = " << p << "\n";
```



Data abstractie in C++ memberfuncties

- Naast datamembers bestaan er memberfuncties

```
struct Boek {  
    char *titel;  
    char *auteur;  
    long isbn;  
  
    void set_isbn(long i) {  
        isbn = i;  
    }  
    void print() {  
        cout << titel << ' ' <<  
             << auteur << ' ' <<  
             << isbn;  
    }  
};
```

- Ook de memberfuncties worden gebruikt via de member-operator:

```
Boek roman, *rp;  
  
roman.set_isbn(122233432);  
roman.print();  
  
rp = &roman;  
rp->print();
```

Figuur 3.

Student-notities



at computing
The Linux/UNIXperts

Data abstractie in C++

- Memberfuncties horen bij een variabele

```
print(); // waarschijnlijk fout  
roman.print(); // goed
```

- Dit voorkomt namespace pollution

```
struct Persoon {  
    char *naam;  
    char *titel;  
    long sofinr;  
  
    void print() {  
        cout << naam << ' ' <<  
            titel << ' ' <<  
            sofinr;  
    }  
};
```

```
Persoon guust;  
Boek bint;  
...  
bint.print();  
guust.print();
```

- Voordeel: gelijke functionaliteit == gelijke naam.

Vergelijk C oplossing:

```
print_Boek(&bint);  
print_Persoon(&guust);
```

Figuur 4.

Student-notities

De memberfuncties uit een class kunnen op twee manieren gedefinieerd worden:

- *Binnen* de klassedefinitie.
In dit geval staat de functiedefinitie (body) in de class-declaratie, zoals bij het class `Disk`-voorbeeld hierboven.
- *Buiten* de klassedefinitie.
De functiedefinitie staat dan niet in de class-declaratie maar erbuiten en wordt aangeduid door de classnaam en de *scope-operator* `::` vóór de functienaam op te nemen bij de functiedefinitie. De functie moet nog wel binnen de klassedefinitie *gedeclareerd* worden. Zo weet de compiler dat de definitie nog wél een keer moet komen. De scope-operator wordt later nog verder behandeld.

Als de definitie ontbreekt in de class-declaratie, zoals bij class `Art`, dan dient de functie dus als volgt gedefinieerd te worden:

```
class Art{
    char *naam;
    float prijs;
    // ...
public:
    // definities buiten de class:
    float get_prijs(); // definities komen nog

    // definities binnen de class:
    char *get_naam() { return naam; }
};

float Art::get_prijs() {
    return prijs;
}
```

De functienaam wordt voorafgegaan door `Art::`, dit zijn respectievelijk de naam van de class en de *scope-operator*.² Deze notatie zorgt ervoor dat de functie `get_btwpercentage()` gekoppeld wordt aan class `Art`. Deze functie kan dan niet gebruikt worden zonder een object van klasse `Art` te gebruiken:

```
float x;

x = get_prijs(); // error
```



2. De scope-operator kun je uitspreken als "z'n": "Art z'n getprijs".

Data abstractie in C++

Een memberfunctie ligt in de **scope** van zijn class

- scope operator `::` koppelt member aan class

- volledige naam van member **print** uit

class **Boek** is: **Boek::print**

Twee manieren van memberfunctie definitie

boek.h:

```
struct Boek {
    char *titel;
    char *auteur;
    long isbn;

    void set_isbn(long i) { // definitie
        isbn = i;
    }
    void print(); // declaratie
                // verplicht
};
```

boek.c:

```
#include "boek.h"
void Boek::print() // definitie
{
    cout << titel << ' ' << auteur
        << ' ' << isbn;
}
```

Figuur 5.

Student-notities

Wat verdient nu de voorkeur: de definitie van de memberfuncties binnen of buiten de class zetten? Het is gebruikelijk om deze buiten de class te zetten. De class-declaraties (met daarbinnen dus alleen de functiedeclaraties) worden meestal in een headerfile gezet. De definitie van de memberfuncties staat meestal in een aparte sourcefile. Dat heeft als voordeel dat er in de headerfile alleen maar declaraties staan en geen definities.

Als de class deel uitmaakt van een commerciële softwarebibliotheek, kan bovendien de sourcefile gecompileerd worden geleverd. Als de functiedefinities in de headerfiles zouden staan, dan zou de sourcecode daarmee zichtbaar worden voor de klant.

Aan de andere kant zijn memberfuncties soms zo klein dat het gemakkelijker is om ze even in de headerfile uit te schrijven.

Het overladen van memberfuncties is ook mogelijk:

```
class Art{
    //...
public:
    //...
    void fill(char *n, float f, char b){
        naam = n; prijs = p; btwcode = b;
    }
    void fill(Art ander){
        prijs = ander.prijs;
        //...
    }
};
```

De memberfunctie `fill()` is overladen en kan op twee manieren aangeroepen worden. Een `class Art`-object kan nu worden gevuld door het opgeven van een lijstje waarden voor de members, of door een eerder gemaakt `Art`-object door te geven waarvan de waarden worden gekopieerd.

Merk overigens op dat als een memberfunctie een argument krijgt van zijn eigen type deze functie de `private`-edeeltes van zijn argument mag gebruiken!

Met andere woorden: de bescherming van `private` geldt op het niveau van de class, niet per object.



Memberfunctie overloading

```
struct Datum {
    void print(FILE *fp) {
        fprintf(fp, .....);
    }
    void print() {
        print(stdout);
    }
    void print(int fd) {
        write(fd, .....);
    }
    void print(char *s) {
        sprintf(s, .....);
    }
    //...
};

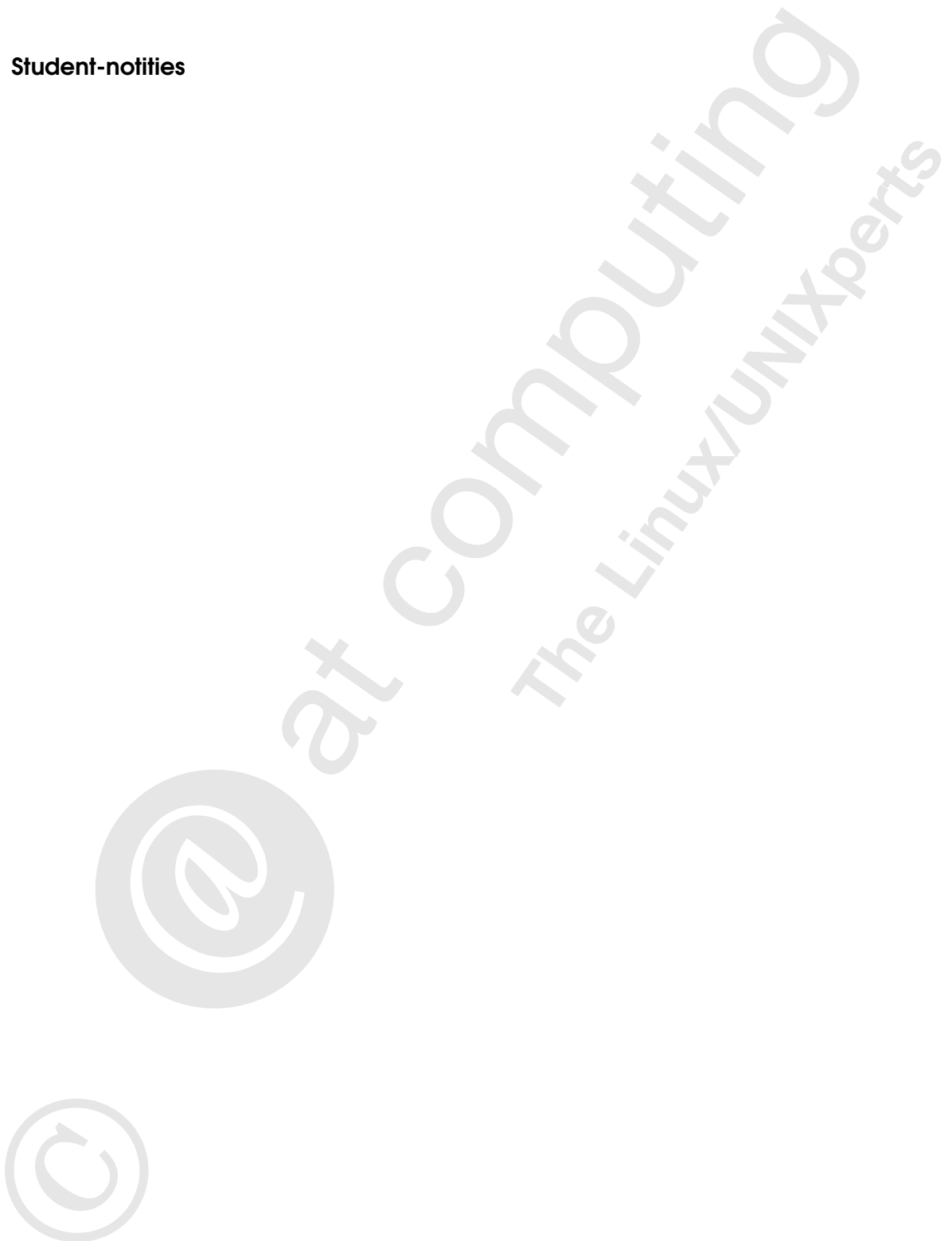
int main()
{
    Datum idb;
    idb.set(1, 1, 1970);

    char buf[32];

    idb.print(1);           //low level out
    idb.print(stderr);     //stdio stderr
    idb.print();           //stdio stdout
    idb.print(buf);        //naar char[]
}
```

Figuur 6.

Student-notities



Samenvatting

- Een structure mag memberfuncties bevatten (en dus gedrag vertonen)
- Memberfuncties kunnen overladen worden
- Bij variabelen declaraties geen keyword **struct**

Figuur 7.

Student-notities

Door memberfuncties te definiëren is veel duidelijker geworden welke acties er allemaal geoorloofd zijn op de structure. In feite geven we te kennen dat een structure-variabele een object is dat gemanipuleerd kan worden via de interfacefuncties. Of — in Smalltalk-jargon — we sturen het object een *message*. Natuurlijk kan iedereen die dat wil de structure nog steeds via z'n datamembers benaderen, waardoor het bovenstaande niet bestand is tegen misbruik:

```
spaghetti.btwcode = '?'; // via de achterdeur
```

Encapsulatie in C++

- **Encapsulatie:** data en functies ontoegankelijk maken
- Encapsulatie niet mogelijk met de structure uit C en dat kan tot fouten en misbruik leiden

```
/* C++ voorbeeld voor valuta
 * zonder gebruik van encapsulatie
 */
struct Currency {
    // datamember:
    char oms[4];    // b.v. "EUR"

    // interface functie:
    void set(char *c) {
        strncpy(oms, c, 3);
        oms[3] = '\0';
    }
};

Currency deens;

deens.set("DKKR"); // ok, wordt "DKK"

strcpy(deens.oms, "DKKR");
// fout, core dump?
```

Figuur 8.

Student-notities

De class is in feite een structure, met members die niemand mag gebruiken tenzij daar expliciet toestemming voor wordt gegeven:

```
class Art{
    char *naam;
    float prijs;
    char btwcode;
public:
    char *get_naam();
    float get_prijs();
    float get_btwpercentage();
};

float c, p;
// ...

// ok, deze members zijn public
c = spaghetti.get_btwpercentage();
p = spaghetti.get_prijs();

// sorry, btwcode is niet public
spaghetti.btwcode = '?'; // error
```

De gebruiker van class Art kan nu alleen nog maar die members gebruiken die expliciet public verklaard zijn.

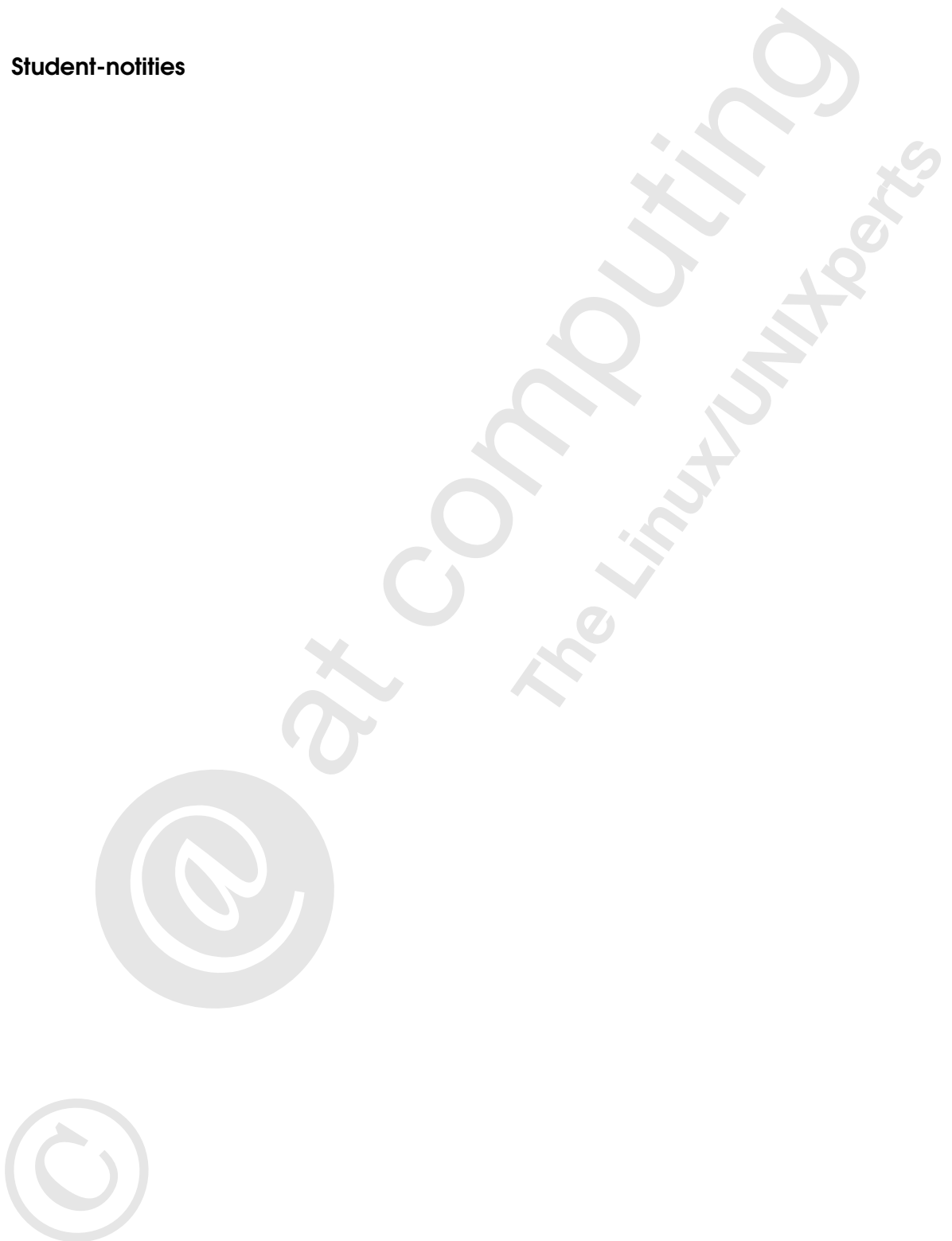
Encapsulatie in C++

- Encapsulatie kan in C++ worden aangegeven met:
public
private
keywords in de structure
- **private** members mogen alleen gebruikt worden door de memberfuncties van de structure zelf
- **public** members mogen door alle functies gebruikt worden (bieden dus geen encapsulatie)

```
struct Currency {  
private:  
    /* deze members zijn  
    * slechts toegankelijk  
    * voor Currency::functies  
    */  
public:  
    /* deze members zijn toegankelijk  
    * voor alle functies (zoals in C)  
    */  
};
```

Figuur 9.

Student-notities



Encapsulatie in C++

```
struct Currency {  
private:  
    long val;  
    char oms[4];  
    void readdbms();  
public:  
    void set(char *c) {  
        strncpy(oms, c, 3);  
        oms[3] = '\\0';  
        readdbms(); // zet val  
    }  
};  
  
Currency deens;  
  
deens.set("DKKR"); // ok, wordt "DKK"  
  
strcpy(deens.oms, "DKKR");  
                // fout, private  
  
deens.readdbms(); // fout, private
```

Figuur 10.

Student-notities

Een structure is in feite een class die volledig public is. Ook in een structure mogen de aanduidingen `public`, `private` en `protected` staan. De class en de structure zijn dus qua functionaliteit volledig identiek. In veel objectgeoriënteerde talen wordt de bouwsteen voor abstracte datatypes een *class* genoemd en ook in C++ is dat het geval. Hoewel in C++ in principe dus alles met structures kan worden gedaan, wordt in de praktijk meestal de class gebruikt.

De controles voor toegang tot `public/private` members worden door de compiler uitgevoerd. Het kost dus alleen (zeer weinig) extra compilatietijd, maar zeker geen extra executietijd van het programma want dan is alle `public`, `protected` en `private` informatie verdwenen.

Data hiding kan goed worden bereikt door middel van het `public/private`-mechanisme. De gebruiker van de class is niet in staat de implementatie van de class te gebruiken, laat staan te wijzigen. In het bovenstaande voorbeeld met class `Art` zijn alleen de memberfuncties in staat de `private` members te manipuleren.

Memberfuncties mogen altijd de `private` members gebruiken en veranderen. In zo'n functie zelf hoeft de variabelenaam plus de memberoperator `.` of `->` niet gebruikt te worden.

Encapsulatie in C++

- **struct** members zijn standaard **public**
- In C++ wordt meestal geen **struct** maar **class** gebruikt
- **class** members zijn standaard **private**

```
class Currency {  
    // deze members zijn private  
public:  
    // deze members zijn public  
};
```

- Door eerst de **public** members te noemen wordt de **class** beter leesbaar

```
class Currency {  
public:  
    void set(char *);  
  
private:  
    /* deze members zijn hooguit  
    * informatief voor de class  
    * gebruikers  
    */  
};
```

Figuur 11.

Student-notities

Het kan wenselijk zijn de private members van een class toegankelijk te maken voor een stel nader te specificeren functies. Over het algemeen zijn dat (member)functies die samen met de class waarin ze mogen rondsnuffelen, een service vormen.

Om aan te geven dat een gewone functie of een memberfunctie van een andere class het recht heeft alle members uit een bepaalde class te gebruiken, wordt de *friend*-specificatie opgegeven.

In het navolgende voorbeeld hebben we twee klassen, Matrix en Vector. Daarnaast hebben we een functie solve(), die een stelsel van n vergelijkingen van n variabelen op kan lossen. De coëfficiënten van de vergelijkingen staan dan in Matrix; in Vector staan de waarden die de vergelijkingen op moeten leveren:

```
class Matrix {
public:
    // ...
    void print() const;
    double get(int row, int col) const;
    void put(int row, int col, double val);
    // ...

private:
    int    rows;
    int    cols;
    double *data;
    // ...

};

class Vector {
public:
    // ...
    void print() const;
    double get(int index) const;
    void put(int index, double val);
    // ...

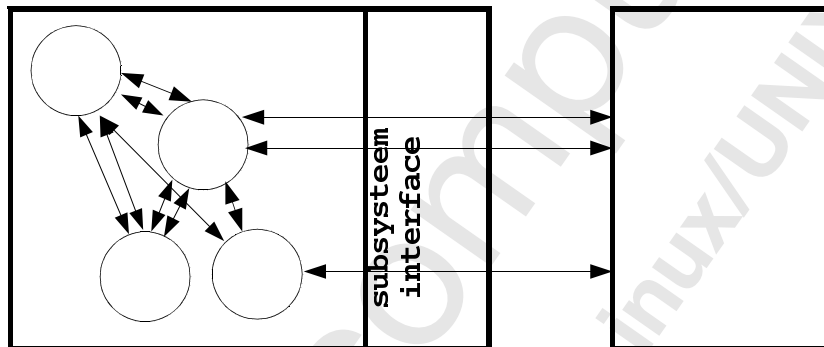
private:
    int    size;
    double *data;
    // ...

};

Vector solve(const Matrix& a,
             const Vector& b);
```

Encapsulatie en friend functies

Soms is het wenselijk de encapsulatie voor één functie te doorbreken



Uitdelen van extra privileges met keyword **friend**

```
void monteerBougie(Motor& m)
{
    m.bougies++; // bougies is private!!
}

class Motor {
    friend void monteerBougie(Motor&);
private:
    int bougies;
    //...
};
```

Figuur 12.

Student-notities

De functie `solve()` kan natuurlijk via de nodige interfacefuncties (zoals `get()` en `put()`) de `private` members van `Matrix` en `Vector` bekijken. Dit zou wel performance kosten. We kunnen er ook voor kiezen om `solve()` een member van `Matrix` te laten zijn, dan hoeven we alleen nog maar via memberfuncties van `Vector` te werken. Maar omgekeerd kunnen we ook kiezen om `solve()` een member van `Vector` te laten zijn. Beide keuzen zijn op zich geldig. Eigenlijk zou `solve()` dus zowel in de klasse `Matrix` als in `Vector` opgenomen moeten worden. We hebben er nu voor gekozen om `solve()` *niet* in een klasse op te nemen.

Hoe kunnen we er nu voor zorgen dat de encapsulatie van de `private` members van `Matrix` en `Vector` voor `solve()` opgeheven wordt? Hier komt het keyword `friend` om de hoek kijken. Door *binnen* `Matrix` en *binnen* `Vector` te specificeren dat `solve()` een `friend` is, wordt de encapsulatie voor `solve()` opgeheven. Zonder een memberfunctie te zijn, krijgt `solve` dezelfde rechten die een memberfunctie heeft.

Het lijkt alsof de `friend`-constructie een stuk van de encapsulatie teniet doet. Dat is echter niet het geval, want de class bepaalt *zelf* wie zijn `friends` zijn. Een `friend`functie moet daarom beschouwd worden als onderdeel van de interface van de class (eigenlijk als een soort memberfunctie, met als enige verschil dat hij geen `this`-pointer heeft).

Soms is de `friend` erg nuttig, bijvoorbeeld bij operator-overloading. Het criterium dat in het voorbeeld hierboven geldt, is dat `Matrix`, `Vector` en `solve()` *samen* een service verlenen. In feite zijn deze drie samen één systeem.

De `friend`-constructie doet ook geen afbreuk aan het principe van `locality of change`: er is immers in de class waar het om draait, geadmistreerd wie er precies iets mag met de `private` members.

Ook in dat opzicht is de `friend`-functie een apart soort "member".

Niet alleen functies kunnen als `friend` gedeclareerd worden. Ook complete klassen kunnen bij de compiler als `friend` van een bepaalde klasse aangemeld worden.



Encapsulatie en friend functies

friend functies op verschillende niveau's

```
class Garage {
public:
    void geefBeurt(Motor& m) { /*...*/ }
    // ...
};

class Motor {
    friend void monteerBougies(Motor&);
    // gewone functie

    friend void Garage::geefBeurt(Motor&);
    // memberfunctie van andere class

    friend class Wegenwacht;
    // alle memberfuncties van
    // andere class
    // ...
};
```

Let op: de class **deelt** de privileges **uit**,
niet andersom (geen misbruik mogelijk)

Figuur 13.

Student-notities

In het resterende deel van dit hoofdstuk worden tips aangereikt voor het ontwerp van klassen.



at computing
The Linux/UNIXperts