SOPX2E1
PROGMI1

Object Georiënteerd Programmeren in C++

bd.thrijswijk.nl

THRijswijk

Werkvormen

84 SBU

- 14 uur theorie
- 14 uur practicum

Inhoud

- **OOP = Object Oriënted Programming**
  - C++ (niet volledig)
- **OOD = Object Oriënted Design**
  - Aanpasbaarheid
  - Uitbreidbaarheid
  - Herbruikbaarheid

Leermiddelen

- **Dictaat**
  - studiewijzer
  - theorie
  - voorbeelden
  - practicumhandleiding
- **(eventueel)** Boek
  - http://mindview.net/Books/TICPP/ThinkingInCPP2e.html
  - toelichting op theorie
  - extra voorbeelden
  - Ti en C&D studenten hebben dit boek later nodig!
- **http://bd.thrijswijk.nl/sopx2/**
  - uitgebreide studiewijzer + planning
  - sheets
  - practicumhandleiding + planning
  - sourcecode van alle voorbeelden
  - links en extra cursus

Een stapje verder...

met programmeren

- Van **gestructureerd** programmeren naar **object georiënteerd** programmeren
- **C++** is een uitbreiding op **C**
  - alles wat in C kan, kan ook in C++
  - veel wat in C kan, kan in C++ beter (struct, array, c-string enz).

P-fase is het **fundament** voor dit onderwijsdeel
zie dictaat blz.7

Waarom een stapje verder?

- Waarom OOP/OOD?
- Waarom met C++?

En dan?

- E: SOPX3 (alleen voor **C&D**)
  - datastructuren en databases.
  - OOD/OOA met UML
- TI: ...

Geschiedenis

Van programmeertalen

- ±1945 assembler
- ±1957 FORTRAN
- ±1960 Algol60

Structured Design
(functionele decompositie)

Software Crisis

veel software is te laat, te duur en niet goed

Doelen OOD/P

- maken van grote **uitbreidbare** en **onderhoudbare** software systemen
- maken van **herbruikbare** software **componenten**

Geschiedenis

Van OOP

- ±1967 Simula
- ±1976 Smalltalk
- ±1984 C++ (Sept 1998 std C++)
- ±1995 Java (Sun)
- ±2000 C# (Microsoft)

Inleiding C++

C++ is designed to:

- be a **better C**
- support **data abstraction**
- support **object-oriented** programming
- support **generic** programming

Initialisatie

```
...
int i=3;
```

initialisatie in **C**

```
...
int i(3);
...
for (int i(0);i<10;++i) {
```

initialisatie in **C++**

const

```
int i(3);
const int j(24);
```

Een constante moet je initialiseren:
```
const int k;
```
error: Constant variable 'k' must be initialized

Een constante mag je niet veranderen:
```
j=7;
```
error: Cannot modify a const object

const * const

```
const int* p(&i);
// p wijst naar i en je kan i niet via p wijzigen.
// Let op: je kan i zelf wel wijzigen!
i=4;      // Goed
*p=5;     // Cannot modify a const object
p=&j;     // Goed

int* const q(&i);
// q wijst naar i en je kan q nergens anders
// meer naar laten wijzen. Let op: je kan i wel
// via q (of rechtstreeks) wijzigen.
i=4;      // Goed
*q=5;     // Goed
q=&j;     // Cannot modify a const object

const int* const r(&i);
// r wijst naar i en je kan i niet via r wijzigen
// en je kan r nergens anders meer naar laten
// wijzen. Let op: je kan i zelf wel wijzigen!
i=4;      // Goed
*r=5;     // Cannot modify a const object
r=&j;     // Cannot modify a const object
```

THRijswijk

bool

- In C word het type **int** gebruikt om booleanse waarden in op te slaan:
  0 = false en ongelijk 0 = true
- In C++ is een speciaal type **bool** gedefinieerd.
  Mogelijke waarden: **false true**

```
int a(5);
bool b(true);

b=a<3;
if (b) {
//...
}
```

THRijswijk

ISO/ANSI std include files

- **#include <complex>**
  in plaatst van
  #include <complex.h>
- **#include <cmath>**
  in plaats van
  #include <math.h>
- **using namespace std;**

THRijswijk

I/O libaries

```
#include <stdio.h>
...
   double d;
   scanf("%d",d);
   printf("d=%lf\n",d);
...
```

C I/O library

2x run-time error

```
#include <iostream>
using namespace std;
...
   double d;
   cin>>d;
   cout<<"d="<<d<<endl;
...
```

C++ I/O library

THRijswijk

string

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

C char [ ]

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

C++ string

THRijswijk

Functionname overloading

```
int abs_int(int i) {
   if (i<0) return -i;else return i;
}
double abs_double(double f) {
   if (f<0) return -f;else return f;
}
double abs_complex(struct Complex c) {
   return sqrt(c.real*c.real+
      c.imag*c.imag)
}
```

met function name overloading:

```
int abs(int i) {
   if (i<0) return -i;else return i;
}
double abs(double f) {
   if (f<0) return -f;else return f;
}
double abs(struct Complex c) {
   returnsqrt(c.real*c.real+
      c.imag*c.imag)
}
```

THRijswijk

Functionname overloading

```
int abs(int i);
double abs(double f);
double abs(struct Complex c);

...
   double in;
   cin>>in;
   cout<<abs(in)<<endl;
...
   int i;
   cin>>i;
   cout<<abs(i)<<endl;
...
```

THRijswijk

Default parameters

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

THRijswijk

struct = type

```
struct Tijdsduur {
   int uur;
   int min;
};
```

Gebruik struct in **C**

```
...
struct Tijdsduur td1;
```

Alternatief gebruik struct in **C**

```
...
typedef struct Tijdsduur TTijdsduur;
TTijdsduur td2;

...
Tijdsduur td3;
```

Gebruik struct in **C++**

THRijswijk

Globaal en lokaal geheugen

een terugblik

```
int global;

void f(int parameter) {
   int local1;
   ...
}

void main() {
   int local2;
   ...
   f(13);
}
```

THRijswijk

Globaal of lokaal geheugen?

```
struct Window {
   int length, width, x, y;
   ...
}

Window* open() {
// hier moet een Window gemaakt
// worden
}

void close(Window* wp) {
// hier moet een Window opgeruimd
// worden
}
```

THRijswijk

Dynamisch geheugen!

```
struct Window {
   int length, width, x, y;
   ...
}

Window* open() {
   return new Window;
}

void close(Window* wp) {
   delete wp;
}
```

THRijswijk

Dynamisch geheugen

```
double* dp(new double);
int i; cin>>i;
double* drij(new double[ i ]);
...
delete dp;
delete[ ] drij;
```

PAS OP!

- Memory leak (delete vergeten!)
- Memory corruption (delete teveel of verkeerde pointer)
- Undefined behaviour (gebruik van een deleted variabele)

THRijswijk

Reference

```
int  i;
int& j(i); //init verplicht!

i=3;
cout<<j<<endl;
// een reference is een "pseudoniem"
```

Gebruik van reference

- Globale variabele
- Lokale variabele
- Functie parameter
- Functie return

THRijswijk

Reference parameter

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

Call by reference in **C**

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

Call by reference in **C++**

THRijswijk

const reference parameter

```
void f(const Matrix& m) {
   ...
}
```

Reference return

```
int a[100];

int& index(int i) {
   assert(i>=0 && i<100);
   return a[i];
}
...
cout<<index(3)<<endl;
index(3)=2;
```

THRijswijk

Parameters

Welk type kies je?

- **T** als kopie nodig (of snel) is
- **T&** als output parameter
- **T*** als alternatief voor T& als er ook “niets” doorgegeven moet kunnen worden.
- **const T&** als kopie niet nodig (of traag) is
- **const T*** als alternatief voor const T& als er ook “niets” doorgegeven moet kunnen worden.

THRijswijk

Welke software hoeft nooit uitgebreid of veranderd te worden?

THRijswijk

Welke klant past nooit zijn specificaties (halverwege het project) aan?

THRijswijk

28

Object Oriented Programming is a new way of thinking

About how we can **structure information** inside a computer

THRijswijk

29

Programming paradigms

- **imperative** (C Pascal)
- functional (LISP)
- logic (Prolog)
- **object oriented** (C++, Java)
- **generic** (ADA, C++)

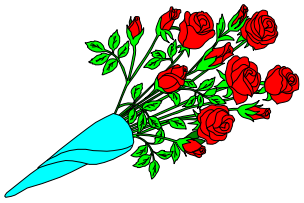
The style of problem solving embodied in the OO technique is frequently the method to address problems in **everyday life**.

THRijswijk

30

Voorbeeld

Jij wilt je oma een bosje bloemen sturen.



object
message + arguments
recievers responsibility
method (information hiding)

THRijswijk

31

Wat is het verschil tussen een message en een functie?

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

THRijswijk

32

Waarom weet ik zoveel van mijn bloemiste?

class
instance (object)
hierarchy
inheritance (base and derived)

THRijswijk

33

Method binding

- search in **class of receiver** object
- if not found search in the **base class** of the class of the receiver
- if not found search in the **base class of the base class** of the class of the receiver
- etc.

A method in a base class can be **overriden** by a method in the derived class

THRijswijk

34

Verband tussen classes

composition
(containment, aggregation, part of)

... has a ...

inheritance
(generalization, kind of)

... is a ...

THRijswijk

35

Steeds meer abstractie

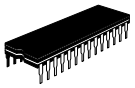
- **functions**
  - avoid duplicating code
  - information hiding
- **modules**
  - data and information hiding
- **abstract data types**
  - instantiation
- **generic functions en ADT's.**
  - templates
- **classes**
  - messages
  - inheritance
  - polymorphism

THRijswijk

36

Doel OOP

Construeren van
"reuseable software components"



Software IC

THRijswijk

37

Wat is de kern van OOP

classes
inheritance
messages
polymorphism

responsibility driven design

THRijswijk

38

Breuk

Versie 0

```
class Breuk {
public:
   void leesin();
   void drukaf() const;
   void plus(const Breuk& b);
private:
   int boven;
   int onder;
   void normaliseer();
};

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

datamembers en memberfuncties
private en public

THRijswijk

39

Breuk

Versie 0

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

Kenmerken van een object

- state
- behaviour
- identity

THRijswijk

40

Breuk

Versie 1

Class **interface** vertelt wat je met een object van de class kunt doen.

```
class Breuk {
public:
   Breuk();
   Breuk(int t);
   Breuk(int t, int n);
   ~Breuk();
   int teller() const;
   int noemer() const;
   void plus(const Breuk& b);
   void abs();
private:
   int boven;
   int onder;
   void normaliseer();
};
```

constructors en destructor
vraag en doe functies

THRijswijk

41

Breuk

Versie 1

Class **implementatie** vertelt je hoe de class in elkaar zit. Dit is voor gebruikers van de class niet van belang.

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

Breuk::Breuk(int t): boven(t), onder(1) {
}

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

Breuk::~Breuk() {
    cout<<"Breuk object verwijderd"<<endl;
}
```

initialisation list

THRijswijk

42

Constructors en destructor

De compiler roept deze memberfuncties automatisch aan.

```
{
   Breuk a;
   ...
   ...
   ...
   ...
}
```

Reserve memory on stack and call constructor **Breuk()** for object a

Call destructor **~Breuk()** for object a and free memory on stack

THRijswijk

43

Constructors en destructor

De compiler roept deze memberfuncties automatisch aan.

Reserve memory on heap and call constructor **Breuk(2,3)**

```
Breuk* bp(new Breuk(2,3));
...
...
...
...
delete bp;
```

Call destructor **~Breuk()** for object and free memory on heap

THRijswijk

44

Constructors en type conversie

Reserve memory on stack and call constructor Breuk(5) for object

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

b3.plus(**Breuk(5)**);

Call destructor ~Breuk() for object and free memory on stack

THRijswijk

45

Breuk

Versie 1

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

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

Breuk b(3,4);
const Breuk half(1,2);
cout<<half.teller()<<'/'<<half.noemer()<<endl;

half=b3;
```

Error: Cannot modify a const object

```
half.plus(b3);
```

Warning: Non-const function Breuk::plus(const Breuk &) called for const object (This is an error, but was reduced to a warning to give existing programs a chance to work.)

const memberfunctie
inline functie

THRijswijk

Breuk

Versie 1

```
void main() {
   { // nutteloos haakje?
      Breuk b1(4);
      cout<<"b1(4) ==> "
          <<b1.teller()<<'/'<<b1.noemer()<<endl;
   }
   Breuk b2(23, -5);
   cout<<"b2(23, -5) ==> "
       <<b2.teller()<<'/'<<b2.noemer()<<endl;
   Breuk b3(b2);
   cout<<"b3(b2) ==> "
       <<b3.teller()<<'/'<<b3.noemer()<<endl;
   b3.abs();
   cout<<"b3.abs() ==> "
       <<b3.teller()<<'/'<<b3.noemer()<<endl;
   b3=b2;
   cout<<"b3=b2 ==> "
       <<b3.teller()<<'/'<<b3.noemer()<<endl;
   b3.plus(5);
   cout<<"b3.plus(5) ==> "
       <<b3.teller()<<'/'<<b3.noemer()<<endl;
}
```

THRijswijk

Breuk

Versie 1

Output:

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

THRijswijk

Wat krijg je “gratis” bij elke class?

Voor elke class maakt de compiler zelf een:

- constructor zonder argument (**default constructor**). Deze constructor roept de default constructor aan van alle data members.
- **copy constructor**. Deze constructor roept de copy constructor aan van alle data members.
- assignment operator (**operator=**). Deze assignment operator roept de assignment operator aan van alle data members.
- **destructor**. Deze destructor roept de destructor aan van alle data members.

Je kunt al deze functies ook **zelf** definieren, de compiler maakt dan niets aan!

THRijswijk

Operator overloading

a.**operator+=**(b);

a+=b;

Je kunt nu binnen de class de function **operator+=** definiëren.

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

THRijswijk

Operator overloading

a.**operator+=**(b.**operator+=**(c));

a+=b+=c;

Werkt niet als operator+= niets teruggeeft (**void** return type)!

Welk object moet je teruggeven?
het **receiver** object zelf
Welk return type kies je?

- Breuk
- **Breuk&**
- const Breuk&

THRijswijk

Breuk

Versie 2

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

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

...
```

THRijswijk

Operator overloading

a.**operator=**(b.**operator+**(c));

a=b+c;

- Wat moet de operator+ memberfunctie doen?
- Wat moet het return type zijn?
  - Breuk
  - **const Breuk**
  - Breuk&
  - const Breuk&

THRijswijk

Operator overloading

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

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

...
```

THRijswijk

Probleem

a.**operator=**(b.**operator+**(3));

a=b+3;

a.**operator=**(3.**operator+**(b));

a=3+b; **Compile error: illegal structure operation**

THRijswijk

55

Oplossing

We hebben gezien dat je operator functies als memberfuncties kunt definiëren. Je kunt echter ook de **globale operator functies** overloaden.

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

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

THRijswijk

56

Oplossing

Breuk(3)

a.**operator=**(**operator+**(b,3));

a=b+3;

Breuk(3)

a.**operator=**(**operator+**(3,b));

a=3+b;

THRijswijk

57

++

a=b;
b+=1;

a=b++;

- maak een kopie van object
- verhoog object met 1
- return kopie

b+=1;
a=b;

a=++b;

- verhoog object met 1
- return object

THRijswijk

58

Operator overloading

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

int parameter is **dummy**

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

const Breuk Breuk::operator++(int) {
   Breuk b(*this);
   ++(*this);
   return b;
}
```

THRijswijk

59

Type conversie

We hebben gezien dat een int automatisch (impliciet) naar een Breuk geconverteerd kan worden door middel van de **constructor** Breuk(int);

int naar Breuk

Je kunt ook een **conversie operator** definiëren waarmee een Breuk automatisch (impliciet) naar een int geconverteerd kan worden.

Breuk naar int

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

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

THRijswijk

60

Pas op!

Bij type conversie

Definieer **geen** conversie waarbij “informatie” verloren gaat!

Is het verstandig om een int automatisch te laten converteren naar een Breuk?

Is het verstandig om een Breuk automatisch te laten converteren naar een int?

THRijswijk

61

==

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

Waarom heb ik de **globale** operator== overloaded?

Waarom kan dit niet zo:

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

Als dit echt nodig is kunnen we een functie friend maken van een class:

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

THRijswijk

62

Friend

Vriendschap in C++ gaat wel erg ver ...

A friend is some one who may touch your private parts.

Geldt bij het gebruik van friend functions nog steeds het principe van information hiding?

THRijswijk

63

Uitvoer d.m.v. <<

We willen een Breuk net zo eenvoudig als een int kunnen afdrukken:
cout<<"b="<<b<<endl;

cout is een object van de class ostream.
Je moet dus de volgende operator overloaden:
ostream& operator<<(ostream& left,
const Breuk& right)

- Waarom const Breuk& parameter?
- Waarom ostream& parameter?
- Waarom ostream& return?

Vaak wordt een friend functie gebruikt.

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

THRijswijk

Invoer d.m.v. >>

We willen een Breuk net zo eenvoudig als een int kunnen inlezen:
cin>>b1>>b2;

cin is een object van de class istream. Je moet dus de volgende operator overloaden:
istream& operator>>(istream& left,
Breuk& right)

- Waarom Breuk& parameter?
- Waarom istream& parameter?
- Waarom istream& return?

THRijswijk

Invoer d.m.v. >>

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

Merk op: deze globale operator>> hoeft **geen** friend te zijn van de class Breuk.

THRijswijk

Breuk Versie 3

```
class Breuk {
public:
   Breuk();
   Breuk(int t);
   Breuk(int t, int n);
   Breuk& operator+=(const Breuk& right);
   Breuk& operator++();
   const Breuk operator++(int);
private:
   int boven;
   int onder;
   void normaliseer();
friend ostream& operator<<(ostream& left,
                           const Breuk& right);
friend bool operator==(const Breuk& left,
                       const Breuk& right);
};

istream& operator>>(istream& left,
                    Breuk& right);
bool operator!=(const Breuk& left,
                const Breuk& right);
const Breuk operator+(const Breuk& left,
                      const Breuk& right);
```

THRijswijk

Breuk Versie 3

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

THRijswijk

Breuk Versie 3

```
void Breuk::normaliseer() {
   assert(onder!=0);
   if (onder<0) {
      onder=-onder;
      boven=-boven;
   }
   int d(ggd(boven<0?-boven:boven,onder));
   boven/=d;
   onder/=d;
}

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

THRijswijk

Breuk Versie 3

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

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

istream& operator>>(istream& left,
                    Breuk& right) {
...
}

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

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

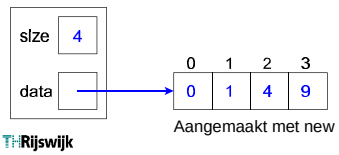
THRijswijk

ADT vector

Een "betere" array

- De grootte van een vector kan tijdens **run-time** bepaald worden
- Bij het gebruik van [...] wordt **gecontroleerd** of de index wel binnen de grenzen van de array ligt

Implementatie van Vector:



Aangemaakt met new

THRijswijk

Vector

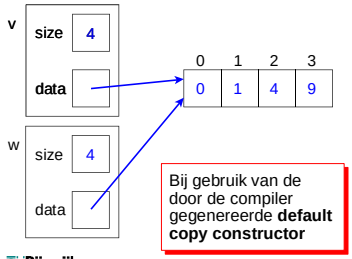
```
class Vector {
public:
   explicit Vector(int s);
   Vector(const Vector& v);
   Vector& operator=(const Vector& r);
   ~Vector();
   int& operator[](int index);
   const int& operator[](int index) const;
   int length() const;
   bool operator==(const Vector& r) const;
   bool operator!=(const Vector&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

Vector::Vector(int s):
      size(s), data(new int[s]) {
}
Vector::~Vector() {
   delete[] data;
}
```

THRijswijk

Copy constructor

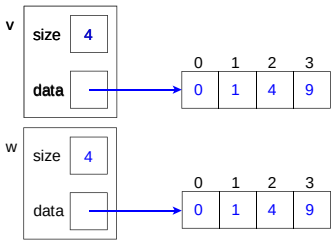
```
Vector v(4);
for (int j(0); j<v.length(); ++j) {
   v[j]=j*j;   // vul v met kwadraten
}
Vector w(v);
```



Bij gebruik van de door de compiler gegenereerde **default copy constructor**

Copy constructor

Gewenst gedrag

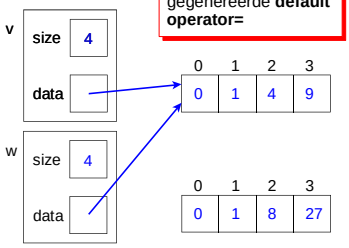


Je moet in dit geval de copy constructor dus **zelf** definiëren.

operator=

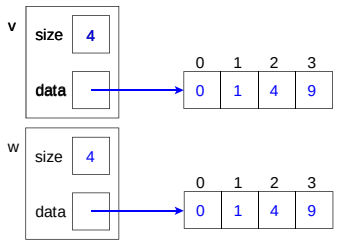
```
Vector v(4);
Vector w(4);
for (int j(0); j<4; ++j) {
   v[j]=j*j;      // vul v met kwadraten
   w[j]=j*j*j;    // vul w met derde machten
}
w=v;
```

Bij gebruik van de door de compiler gegenereerde **default operator=**



operator=

Gewenst gedrag



Je moet in dit geval de operator= dus **zelf** definiëren.

Copy constructor en operator=

Implementatie

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

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

Voorkomt een lugubere fout bij **v=v;**

Wanneer zelf definiëren

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

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

Seperate compilation

MEMCELL.H

```
#ifndef _memcell_
#define _memcell_
class MemoryCell {
public:
   int Read() const;
   void Write(int x);
private:
   int StoredValue;
};
#endif
```

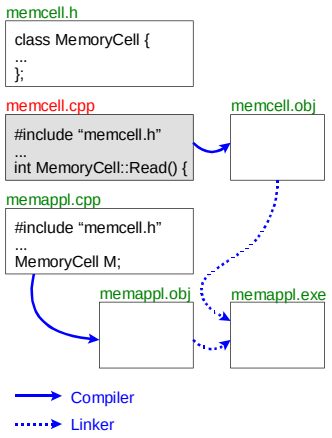
MEMCELL.CPP

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

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

APPL.CPP

Seperate compilation



Template functie

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

Een template is een “mal” waarmee diverse functies “gemaakt” kunnen worden

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

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

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

Template class

```
template <typename T> class Vector {
public:
   explicit Vector(int s);
   Vector(const Vector<T>& v);
   Vector<T>& operator=(const Vector<T>& r);
   ~Vector();
   T& operator[](int index);
   const T& operator[](int index) const;
   int length() const;
   bool operator==(const Vector<T>& r) const;
   bool operator!=(const Vector<T>& r) const;
private:
   int size;
   T* data;
};

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

// ... enz. ...
```

THRijswijk

Gebruik van template class

Bij gebruik van een template class moet je de template parameters opgeven.

```
int i; cin>>i;
Vector<double> v(i);
for (int j(0); j<v.length(); ++j)
   v[j]=sqrt(j);                  // Vul v met wortels
cout<<"v[12] = "<<v[12]<<endl;


Vector<int> w(i);
for (int t(0); t<w.length(); ++t)
   w[t]=t*t;                      // Vul w met kwadraten
cout<<"w[12] = "<<w[12]<<endl;
```

THRijswijk

Hergebruik van classes

composition
(containment, aggregation, part of)

... has a ...

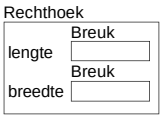
inheritance
(generalization, kind of)

... is a ...

THRijswijk

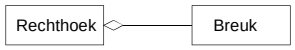
Composition

```
class Rechthoek {
public:
   ...
private:
   Breuk lengte;
   Breuk breedte;
};
```



Een Rechthoek **heeft een** lengte en een breedte van het type Breuk.

Schematische weergave:



THRijswijk

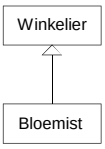
Inheritance

```
class Winkelier {
...
};

class Bloemist: public Winkelier {
...
};
```

Een Bloemist **is een** Winkelier.

Schematische weergave:



THRijswijk

ADT aanpak

enum Soort {sintBernard, tekkel};

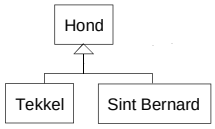
```
class Hond {
private:
   Soort s;
   ...
public:
   void haalKrant();
   void blaf();
   ...
};
void Hond::haalKrant() {
   ...
   blaf();
}
void Hond::blaf() {
   switch(s) {
      case sintBernard:
         cout<<"WOEF WOEF";
         break;
      case tekkel:
         cout<<"kef kef";
         break;
   }
}
```

THRijswijk

OO aanpak

Een Tekkel **is een** Hond

Een SintBernard **is een** Hond




```
class Hond {
private:
   ...
public:
   void haalKrant();
   virtual void blaf();
   ...
};

void Hond::haalKrant() {
   ...
   blaf();
}
void Hond::blaf() {
   cout<<"blaf blaf";
}
```

THRijswijk

OO Aanpak

```
class SintBernard : public Hond {
private:
   Whisky vat;
public:
   virtual void blaf();
};

void SintBernard::blaf() {
   cout<<"WOEF WOEF";
}
```

```
class Tekkel : public Hond {
public:
   virtual void blaf();
};

void Tekkel::blaf() {
   cout<<"kef kef";
}
```

THRijswijk

Polymorphism

```
...
void doeJeWerk(Hond& h) {
   h.haalKrant();
   ...
}
...

int main() {
   SintBernard  Boris;
   Tekkel Harry;

   if (!weekend)
      doeJeWerk(Harry);
   else if(zaterdag)
      doeJeWerk(Boris);

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

THRijswijk

Slicing problem

polymorphism werkt alleen bij * en &

```
cass Hond {
public:
   virtual void blaf() {cout<<"blaf blaf"<<endl;}
};
class SintBernard : public Hond {
public:
   virtual void blaf() {cout<<"WOEF"<<endl;}
private:
   Whisky vat;
};

int main() {
   SintBernard boris;
   Hond& rhond(boris);
   rhond.blaf();
   Hond* phond(&boris);
   phond->blaf();
   Hond hond(boris);  // Waar blijft de whisky?
   hond.blaf();
...
```

THRijswijk

Abstract Base Class

```
class Hond {
private:
   ...
public:
   void haalKrant();
   virtual void blaf()=0;
   ...
};

void Hond::haalKrant() {
   ...
   blaf();
}
```

- Er kunnen **geen** objecten (variabelen) van een ABC gedefinieerd worden.
- Een class die overerft van Hond en blaf() override is geen ABC meer.
- Een class die overerft van Hond en blaf() niet override is een ABC.

THRijswijk

Overloading van memberfuncties

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

int main() {
   Class object;
   object.f();
   object.f(3);
...
```

Uitvoer:
Ik ben f()
Ik ben f(int)

THRijswijk

Overerving, Overloading en de hiding-rule

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

class Derived: public Base {
public:
   void f(int i) { // Verberg f() geen goed idee !!!
      cout<<"Ik ben f(int)"<<endl;
   }
};
```

THRijswijk

Overloading en de hiding-rule

```
int main() {
   Base b;
   Derived d;

   b.f();
   d.f(3);

// d.f();
   d.Base::f();
}
```

Error:
Too few parameters in call to 'Derived::f(int)'

Uitvoer:
Ik ben f()
Ik ben f(int)
Ik ben f()

Conclusie:
Overloading en overerving gaan niet goed samen!

THRijswijk

Reden voor de hiding-rule

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

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

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

Uitvoer:
Ik ben f(double)

THRijswijk

Reden voor de hiding-rule

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

// Code van Dewi niet gewijzigd

int main() {
   Derived d;
   d.f(3); // Base::f(int) is hidden. Gelukkig!
// ...
```

Uitvoer:
Ik ben f(double)

Conclusie: De hiding-rule vergroot de **onderhoudbaarheid**!

THRijswijk

Overloading versus Overriding

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

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

THRijswijk

Overloading versus Overriding

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

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

THRijswijk

Virtual destructor

Als een class nu of in de toekomst als base class gebruikt wordt dan moet de destructor virtual zijn zodat van deze class afgeleide classes via een **polymorphic pointer** gedelete kunnen worden.

```
class Hond {
private:
   ...
public:
   virtual ~Hond() {
   }
   void haalKrant();
   virtual void blaf()=0;
   ...
};
```

Pas op: De door de compiler gedefinieerde default destructor is **niet** virtual.

THRijswijk

Virtual destructor

```
class SintBernard : public Hond {
private:
   Whisky vat;
public:
   SintBernard() {
      vat.maakVol();
   }
   virtual ~SintBernard() {
      vat.maakLeeg();
   }
   virtual void blaf();
};

int main() {
   Hond* Boris(new SintBernard);
   Boris->blaf();
   delete Boris;
   cin.get();
   return 0;
}
```

THRijswijk

Afscherming

- **private:**
  - alleen bereikbaar in member functies van de class zelf
- **protected:**
  - alleen bereikbaar in member functies van de class en in member functies van "nakomelingen" van de class
- **public:**
  - altijd bereikbaar via object

THRijswijk

Inheritance

Wanneer gebruiken?

Bedenk altijd dat overerving een **typerelatie** oplevert, als class Derived overerft van class Base dan geldt "**Derived is een Base**".

Dat wil zeggen dat elke bewerking die op een object (variabele) van class (type) Base uitgevoerd kan worden ook op een object (variabele) van class (type) Derived uitgevoerd moet kunnen worden.

In de class Derived moet je alleen data members en member functies **toevoegen** en/of virtual member functies **overriden**.

THRijswijk

Vraag

?

Is een

Struisvogel

een

Vogel?

THRijswijk