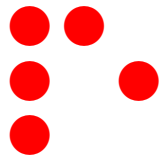


---

## Programmeermethoden



Object-geOrienteerd Programmeren & Life

week 6: 10–14 oktober 2011 (Den Haag: 17–21)

<http://www.liacs.nl/home/kosters/pm/>

Ontbind een getal in factoren, zoals  $84 = 2^2 \times 3^1 \times 7^1$ :

```
void ontbinden (int getal) {
    int teller = 0; // hoe vaak past deler in getal?
    int deler = 2; // kandidaat deler
    while ( getal != 1 ) {
        if ( getal % deler == 0 ) {
            teller = 0;
            while ( getal % deler == 0 ) {
                getal = getal / deler;
                teller++;
            }//while
            cout << deler << " tot de " << teller << "-de macht ";
        }//if
        deler++;
    }//while
}//ontbinden
```

```
int a, b;
int peter (int r, int s) {
    s--; return r+s+2; } // peter
int ellen (int p, int q) {
    int a = 7; p++; q -= 2;
    for ( a = 2; a < q; a++ ) p = p + peter (p,q);
    cout << a << p << q << endl; return a+p+q; } // ellen
```

1. Reken `a = 2; b = 6; cout << ellen (a,b) << endl;`  
`cout << a << b << endl;` door.
2. Nu met vier `&`'s toegevoegd in de headings.
3. Vervang in `ellen` het statement `p = p + peter (p,q);`  
door het statement `p = p + peter (q,p);`, en met vier `&`'s.  
Wat zijn de mogelijke waarden voor `a` na afloop?

We willen willekeurige (**random**) getallen maken. Dit gaat met het volgende recept, de lineaire congruentie methode (zie Knuth). Kies een startwaarde  $x_{\text{oud}}$  (de “seed”). Pas dan herhaald toe

$$x_{\text{nieuw}} = (a \times x_{\text{oud}} + c) \text{ modulo } m$$

met vaste  $a$ ,  $c$  en  $m$ ; en voor modulo lees: % uit C++. Verschuif hierbij steeds  $x_{\text{nieuw}}$  naar  $x_{\text{oud}}$ .

Vaak:  $c = 1$ ,  $m$  een macht van 10, en  $a$  modulo 200 = 21 (zie dictaat). Dan krijg je zoveel mogelijk verschillende getallen, voordat het zich gaat herhalen.

```
// geef random getal tussen 0 en 999
int randomgetal ( ) {
    static int getal = 42;
    getal = ( 221 * getal + 1 ) % 1000; // niet aan knoeien
    return getal;
} //randomgetal
```

Een **static** variabele wordt eenmalig geïnitialiseerd, en blijft behouden tussen functie-aanroepen.

En een random getal uit {1, 2, 3, 4, 5, 6}? Doe:

```
cout << 1 + randomgetal ( ) % 6 << endl;
Of beter 1 + randomgetal ( ) / 167 ?
```

(En in `cstdlib` zit `rand ( ) ...`)

```
string filenaam; // gebruik <string>
ifstream invoer; // gebruik <fstream>
...
cin >> filenaam;
invoer.open (filenaam.c_str ( )); // (*)
if ( invoer.fail ( ) ) {
    cout << filenaam << " niet te openen" << endl;
    return 1; // of exit (1);
} //if
```

In bovenstaand programma maken we een object `filenaam` van klasse `string` (voor de naam van de file) en een object `invoer` van klasse `ifstream` (voor de file). In regel (\*) koppelen we ze, door de **methode** `open` te gebruiken.

C++ is — in tegenstelling tot C — een **object-georiënteerde (OO)** programmeertaal, net als Java.

In een OO-programma hebt je **objecten** (Adam, Eva, Bonzo) van verschillende **klassen** (Mens, Hond). Klassen hebben hun eigen **methoden** (praten, slapen, blaffen).

Een voorbeeld: de klasse `double`. Met `double x, y;` maak je twee objecten (variabelen) van deze klasse. En `cout << x;` vraagt `x` zich af te laten drukken. En `x = x - y;` vraagt `x` zich met de waarde van `y` af te laten.

Denk ook aan files met methodes (**member-functies**) als `get` en `put`.

```
class wagon {
    public:
        double hoogte, breedte, lengte;
        double inhoud ( ) { // member-functie
            return hoogte * breedte * lengte;
        } // inhoud
}; // wagon

...
wagon bert;
bert.hoogte = 3.5;
bert.breedte = 4.0;
bert.lengte = 20.5;
cout << "Inhoud" << bert.inhoud ( ) << endl;
```

Hier is bert een **object** van **klasse** (= **type**) wagon.

Een object `ernie` van klasse `wagon` bestaat uit drie `double`'s, die zijn hoogte, breedte en lengte aanduiden.

En je kunt (via de methode `inhoud`) om zijn inhoud vragen. Deze functies worden eenmalig opgeslagen, niet in ieder object opnieuw.

Let op de punt-notatie: `ernie.breedte`. En voor functies (methoden) `ernie.inhoud ( )`.

Tevens kan bestaan (**overloading** van inhoud en lengte):

```
class tanker {  
    public:  
        double straal, lengte; // zelfde namen  
        double inhoud ( ) { // als zo-even!  
            return straal * straal * lengte;  
        }//inhoud  
}; //tanker
```

```
class wagon {
    public:
        double hoogte, breedte, lengte;
        double belasting (double);
        // functie-prototype van deze methode
}; // wagon
double wagon::belasting (double percentage) {
    return percentage * breedte * lengte;
} // wagon::belasting
```

Hierbij is :: de **binary scope resolution operator**.  
Met ernie van klasse wagon (dus wagon ernie;):

```
cout << "Belasting: "  
      << ernie.belasting (0.5) << endl;
```

Het benutten van de (member-)variabelen van een object gaat meestal met speciaal geschreven functies: **reader** (*getter, accessor*) en **writer** (*setter, mutator*) methodes.

```
class tanker { ... als vroeger ...
    double geefstraal ( ) { // reader
        return straal;
    } //geefstraal
}; //tanker
```

Gebruik nu `zeppo.geefstraal ( )` in plaats van `zeppo.straal` (met `zeppo` van type `tanker`). Analoog `writer`'s.

Een uitbreiding voor `tanker`:

```
double tanker::geefdiameter ( ) { // reader
    return 2.0 * straal;
} //geefdiameter
```

Met behulp van reader's en writer's kun je (member-)variabelen van een object afschermen/verbergen:

```
class tanker {
    public:
        double geefstraal ( ) { // reader
            return straal;
        } //geefstraal
        void maaklang (double t) { // writer
            lengte = t;
        } //maaklang
    private:
        double straal, lengte;
}; //tanker
```

Nu mag `chico.straal` zelfs niet meer gebruikt worden; het *moet* via `chico.geefstraal ( )` (met `chico` van type `tanker`). En `chico.lengte = 42.1;` *moet* nu `chico.maaklang (42.1);` worden.

```
class tanker { public:  
    double straal, lengte;  
    tanker ( ) {  
        straal = 1.0; lengte = 37.0;  
    } //default constructor  
    tanker (double s, double t) {  
        straal = s; lengte = t;  
    } //constructor  
}; //tanker
```

Als je nu een nieuwe variabele maakt van klasse tanker kun je die meteen initialiseren:

```
tanker harpo; // met default constructor  
tanker groucho (7.0,12.12); // met andere constructor
```

Een **constructor** wordt “nooit” direct aangeroepen, maar automatisch gebruikt bij het ontstaan van objecten.

De klasse personenwagon wordt **afgeleid** (= **derived**) van de **ouder** (= **superklasse**) wagon:

```
class personenwagon : public wagon {
    // we erven "alles" van wagon
    public:
        // default constructor:
        personenwagon ( ) { passagiers = 0; }
        personenwagon (int aantal) {
            passagiers = aantal;
        } //constructor
        int hoeveel ( ) { // reader
            return passagiers;
        } //hoeveel
    private:
        int passagiers;
}; //personenwagon
```

Ook **multiple inheritance/overerving** is mogelijk:

```
class gehakt : public dier, eten { ... };
```

Stel we hebben een klasse `voertuig`, met variabelen `gewicht` en `maxsnelheid`, en een methode `belasting ( )`. Er zijn afgeleide klassen `fiets` (met eigen methode `belasting ( )`) en `auto` (met een extra variabele `soort`).

Met `rijwiel` van type `fiets` mag je gebruik maken van `rijwiel.belasting ( )`. Je krijgt dan de `belasting` speciaal voor een `fiets`. Als je toch de `belasting` als voor een `voertuig` wilt laten berekenen: `rijwiel.voertuig::belasting ( )`.

Als je de constructor voor `fiets` “aanroept”, wordt automatisch eerst die voor `voertuig` uitgevoerd.

Stel we willen met gehele getallen van “willekeurige” lengte werken, zoals 1234567891011121314151617181920. Grote getallen dus. We maken daartoe een klasse gg met methoden als drukaf ( ), maak (int m), kopie (gg & getal) en telop (gg & getal). Zie de vierde programmeeropgave.

Je kunt dan een programma schrijven als

```
gg x; gg y; // int x; int y;
x.maak (1); y.kopie (x); // x = 1; y = x;
for ( int i = 1; i <= 1000; i++ ) {
    x.telop (y); // x = x + y;
    y.kopie (x); // y = x;
    x.drukaf ( ); // cout << x;
} //for
```

Dit berekent uiteindelijk  $2^{1000}$  (het kan anders en beter).

OOP

De Marx brothers



Zeppo, Harpo, Chico en Groucho “**password**” Marx  
“Time flies like an arrow; fruit flies like a banana.”

```
class wagon { ... pointer-details: zie later ☒
    virtual void naam ( ) { cout << "Wagon"; }
}; //wagon
class personenwagon : public wagon { ...
    virtual void naam ( ) { cout << "Personenwagon"; }
    // naam wordt polymorf gebruikt
}; //personenwagon

wagon* treinwijzer; // pointer naar wagon
if (mooi_weer)
    treinwijzer = new wagon;
else
    treinwijzer = new personenwagon;
// en nu late binding = runtime binding
// of je "Wagon" of "Personenwagon" gaat zien
// hangt van het weer af!
treinwijzer->naam ( );
```

Overigens mag je  $p = w$ ; *niet* zeggen (met personenwagon  $p$ ; en wagon  $w$ );, en  $w = p$  *wel*; je verliest dan wel informatie.

Als je zegt `w1 = w2;` of `w1 = p;` (met `w1` en `w2` van klasse `wagon`, en `p` van klasse `personenwagon`), wordt de inhoud van de betreffende objecten gekopieerd. Zoals gezegd, bij `w1 = p` gaat informatie verloren. Verder worden in geval van pointers als member-variabelen alleen de pointers gekopieerd (arrays *wel* geheel).

Wil je ook datgene waar de pointer naar wijst meekopiëren (een zogeheten **diepe kopie**), dan moet je daar zelf voor zorgen.

Let op: constructor, `this` (zie later), copy constructor (niet hier), overladen van `=` (zie later).

Elke member-functie kan beschikken over een impliciete pointer, die **this** heet, waarvan de waarde een pointer is naar het adres van het object waar de functie behoort.

```
class wagon { ...
    int inhoud ( ) { ... return ... }
    void schrijfinhoud ( ) {
        cout << inhoud ( );
    }//schrijfinhoud
}; //wagon
```

Nu is `cout << inhoud ( ) << endl;` hier precies hetzelfde als `cout << (*this).inhoud ( ) << endl;`, en dat is weer hetzelfde als `cout << this->inhoud ( ) << endl;`.

De this-pointer wordt vaak gebruikt om te kijken of een object dat als parameter binnenkomt verschilt van het “huidige”, bijvoorbeeld bij kopiëren: je wilt als het “`x = x;`” is niet eerst de oude `x` weggooien, en eigenlijk gewoon niets doen!

De test `if ( this != &argument )` is dan ook een veelgezien constructie.

Iedere goede klasse heeft naast een aantal constructoren ook één **destructor**, die aan het eind van het bestaan van een object van die klasse automatisch wordt aangeroepen.

```
class wagon { public:
    int* codes; // pointer
    wagon (int serie[ ], int n) { // constructor
        codes = new int[n]; // dynamisch array!
        for ( int i = 0; i < n; i++ ) codes[i] = serie[i]; }
    ~wagon ( ) { // destructor
        delete [ ] codes; }
}; //wagon
```

Let er op dat een destructor ook wordt aangeroepen wanneer locale variabelen aan het einde van een functieaanroep worden vrijgegeven. Er ontstaan soms problemen als niet-diepe kopieën gemaakt zijn, of als je per ongeluk call by value gebruikt (geef dus objecten call by reference (met een &) door!).

Objecten (klassen) kunnen op allerlei manieren de toegangsrechten tot member-variabelen en member-functies regelen. Deze kunnen `private`, `public` of `protected` zijn; verder kan ook `private`, `public` of `protected` geerfd worden ... keuze genoeg! Ook kunnen klassen bevriend met elkaar zijn: het zijn dan `friend`'s.

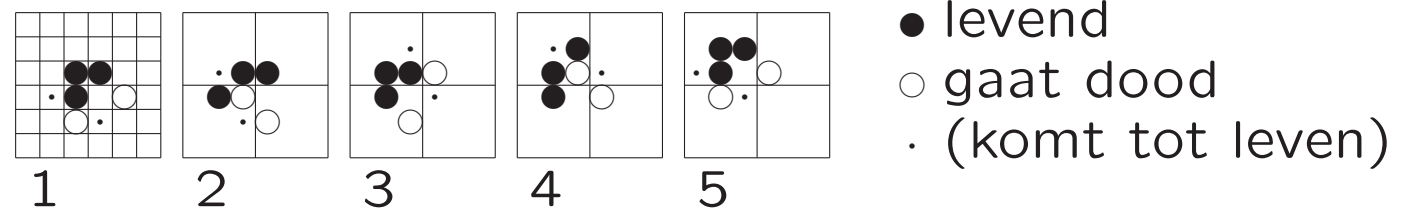
Het is mogelijk operatoren (zoals `=`, `+` of `<<`) te overladen, oftewel bij te definiëren:

```
ostream & operator << (ostream & uitvoer, wagon & w) {
    uitvoer << "[" << w.naam ( ) << " ";
    return uitvoer;
} //operator <<
```

Nu mag je gewoon `uitvoer << w;` (met wagon `w`;) zeggen.

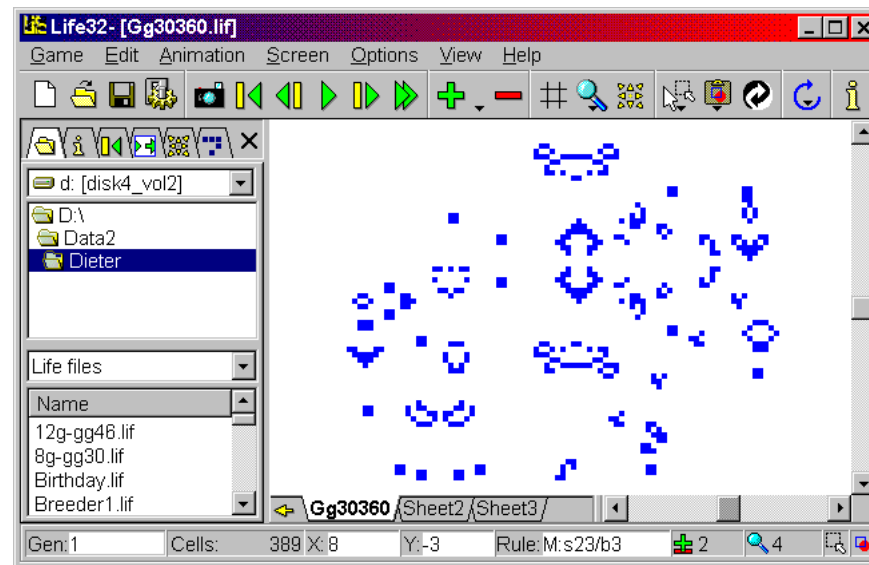
**Life** is een “cellulaire automaat”, in 1970 bedacht door John Horton Conway.

In een 2-dimensionaal oneindig groot rooster beginnen we met een eindig aantal levende vakjes oftewel cellen. Een levend vakje met minder dan 2 of meer dan 3 burens (van de 8) gaat dood, met precies 2 of 3 levende burens overleeft het. In een dood vakje met precies 3 levende burens ontstaat leven. Dit leidt tot de volgende generatie. Let erop dat dit voor alle vakjes tegelijk gebeurt.

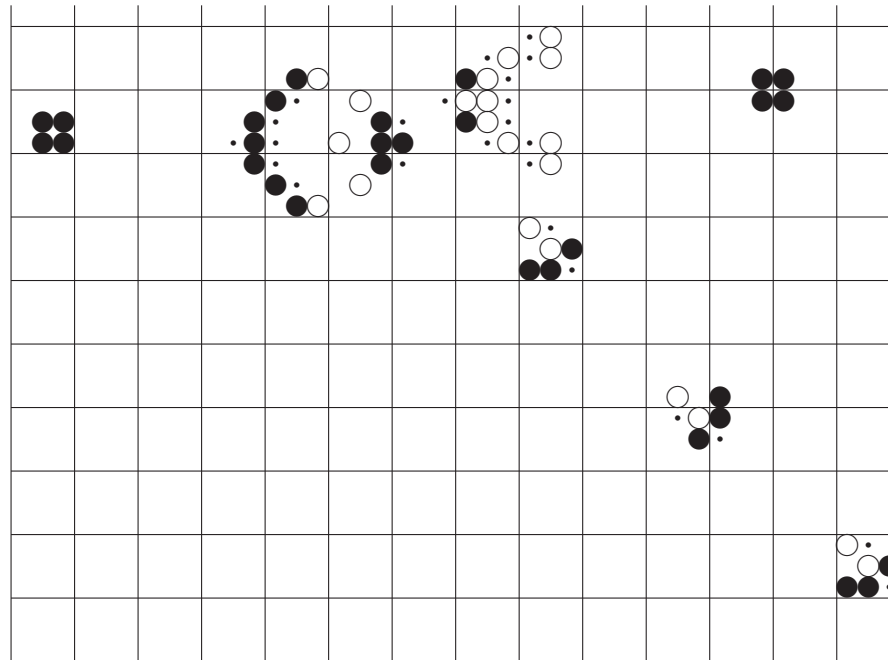


Dit patroon heet **glider**.

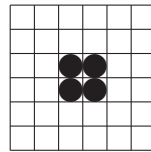
- Wiki: <http://www.conwaylife.com/wiki/>
- Programma (Windows):  
<http://www.liacs.nl/home/kosters/op3pm.php>



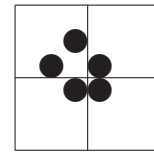
In 1970 wonnen onderzoekers van het M.I.T. in Boston een prijs van \$ 50 met een beginconfiguratie waarbij het aantal levende cellen groter en groter wordt: Gosper's **glider gun**, die elke dertigste generatie een nieuwe glider afvuurt:



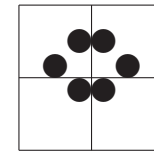
Een **stilleven** is een Life-configuratie die niet verandert:



blok

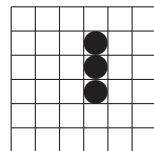


boot

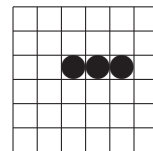


bijenkorf

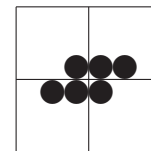
En een **oscillator** repeteert met een zekere periode (stilleven is een periode-0 oscillator):



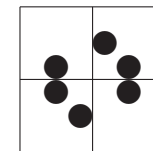
1 blinker



2

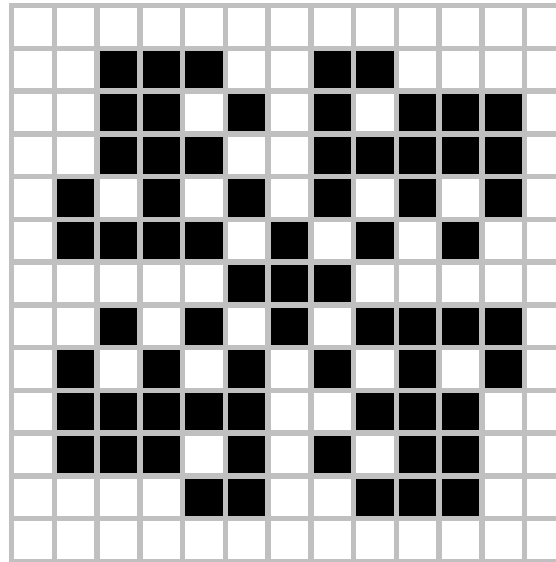


1 pad



2

Een **wees** is een life-(deel)patroon dat nooit kan ontstaan tijdens de ontwikkeling vanuit een beginpatroon. En, minder algemeen, een **Hof van Eden** heeft geen “vader”.



Beluchenko, 2009

Een **breeder** is een life-configuratie die glider guns produceert:



- 
- werk aan de tweede programmeeropgave — de deadline is op vrijdag 14 oktober 2011  
Den Haag: maandag 24 oktober 2011
  - lees Savitch Hoofdstuk 6 en 7.1
  - lees dictaat Hoofdstuk 3.11
  - maak opgaven 26/30 uit het opgavendictaat
  - lees de derde programmeeropgave  
<http://www.liacs.nl/home/kosters/op3pm.php>