

# Inleiding Programmeren

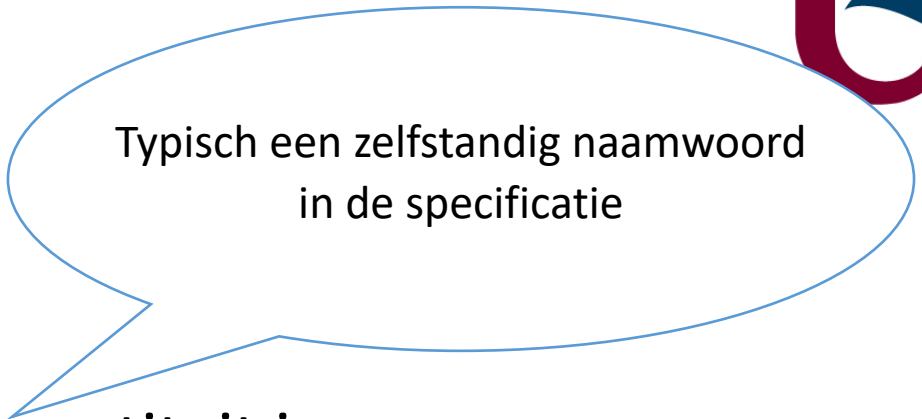
## Inleiding klassen

Tom Hofkens

# Vandaag

- Klassen
  - Definitie van een klasse in C++
    - Member variabelen en functies
  - Public en private members
  - Constructor en destructor

# Classes



Typisch een zelfstandig naamwoord  
in de specificatie

- Idee:
  - Een klasse representeert een entiteit in een programma
    - Als je “het” kan beschouwen als een aparte entiteit, dan is het waarschijnlijk een goed idee om het als een klasse te implementeren
    - Voorbeeld: vector, matrix, input stream, string, robot arm, device driver, foto op het scherm, dialog box, graaf, window, klok
  - Een klasse is een user-defined type dat specificeert hoe objecten van dit type gebruikt kunnen worden
  - In C++ (en in de meeste modern programmeertalen) is een klasse een van de belangrijkste bouwstenen om grote programma's te maken
    - En ook heel nuttig voor kleine programma's

# Members en toegang tot members

- Definitie van een klasse:

```
class X {           // de naam van deze klasse is X
                   // data members (slagen informatie op)
                   // function members (gebruiken en/of manipuleren de data)
};
```

- Voorbeeld

```
class X {
public:
    // data member
    int m;
    // function member = methode
    int mf(int v) { int old = m; m=v; return old; }
};

X var;                // var is a variable of type X
var.m = 7;            // access var's data member m
int x = var.mf(9);    // call var's member function mf()
```



# Klassen

- Een klasse is een user-defined type

```
class X { // naam van de klasse is X
public:   // public members - interface voor gebruikers van de klasse
           // (toegankelijk voor allen)
           // functies
           // types
           // data (over het algemeen best private)
private: // private members - implementatie details
           //(enkel toegankelijk voor member functies van deze klasse)
           // functies
           // types
           // data
};
```

# Struct en class

- Class members zijn private by default:

```
class X {  
    int mf();  
    // ...  
};
```

- betekent

```
class X {  
private:  
    int mf();  
    // ...  
};
```

- Dus

```
X x;           // variable x of type X  
int y = x.mf(); // error: mf is private (i.e., inaccessible)
```

# Struct en class

- Een struct is eigenlijk een klasse waarbij alle members public zijn by default:

```
struct X {  
    int m;  
    // ...  
};
```

betekent

```
class X {  
public:  
    int m;  
    // ...  
};
```

- **structs** worden voornamelijk gebruikt wanneer de member variabelen geen beperkingen hebben

# Structs

```
// simplest Date (just data)
struct Date {
    int year;
    int month;
    int day;
};
```

```
Date my_birthday;    // a Date variable (object)
```

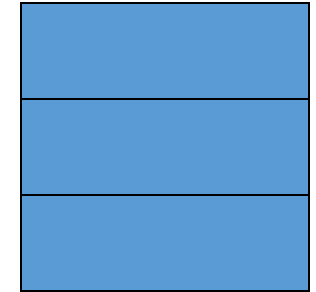
```
my_birthday.year = 12;
my_birthday.month = 30;
my_birthday.day = 1950;    // oeps! (geen dag 1950 in maand 30)
                           // gaat later voor problemen zorgen!
```

Date:

my\_birthday: year

month

day





# Classes

```
// simple Date (control access)
```

```
class Date {
```

```
    int year, month, day;
```

```
public:
```

```
    Date(int y, int m, int d); // constructor: check for valid date and initialize
```

```
    // access functions:
```

```
    void add_day(int n); // increase the Date by n days
```

```
    int getMonth() { return month; }
```

```
    int getDay() { return day; }
```

```
    int getYear() { return year; }
```

```
};
```

my\_birthday: year

month

day

Date:

|      |
|------|
| 1950 |
| 12   |
| 30   |

# Classes

```
// simple Date (control access)
```

```
class Date {
```

```
    int year, month, day;
```

```
public:
```

```
    Date(int y, int m, int d); // constructor: check for valid date and initialize
```

```
    // access functions:
```

```
    void add_day(int n); // increase the Date by n days
```

```
    int getMonth() { return month; }
```

```
    int getDay() { return day; }
```

```
    int getYear() { return year; }
```

```
};
```

```
Date my_birthday {12, 30, 1950};
```

```
Date my_birthday {1950, 12, 30};
```

```
cout << my_birthday.getMonth() << endl;
```

10

```
my_birthday.month = 14;
```

my\_birthday: year

month

day

Date:

|      |
|------|
| 1950 |
| 12   |
| 30   |

```
// error in constructor!
```

```
// ok
```

```
// we can read
```

```
// error: Date::m is private
```

# Belangrijke methodes

- Constructor
- Getters en setters
- Destructor

# Classes: constructor oproepen

```
// OFWEL function style  
Date d = Date(2024,2,1);
```

# Classes: constructor oproepen

```
// OFWEL function style  
Date d = Date(2024,2,1);
```

```
// OFWEL verkorte function style  
Date d(2024,2,1);
```

# Classes: constructor oproepen

```
// OFWEL function style  
Date d = Date(2024,2,1);
```

```
// OFWEL verkorte function style  
Date d(2024,2,1);
```

```
// OFWEL uniform style  
Date d = Date{2024,2,1};
```

# Classes: constructor oproepen

```
// OFWEL function style  
Date d = Date(2024,2,1);
```

```
// OFWEL verkorte function style  
Date d(2024,2,1);
```

```
// OFWEL uniform style  
Date d = Date{2024,2,1};
```

```
// OFWEL verkorte uniform style  
Date d{2024,2,1};
```

# Classes: constructor oproepen

```
// OFWEL function style  
Date d = Date(2024,2,1);
```

```
// OFWEL verkorte function style  
Date d(2024,2,1);
```

```
// OFWEL uniform style  
Date d = Date{2024,2,1};
```

```
// OFWEL verkorte uniform style  
Date d{2024,2,1};
```

```
// OFWEL implicit (let op volgorde van parameters!)  
Date d = {2024,2,1};
```



# Classes: constructor oproepen

```
// OFWEL implicit (let op volgorde van parameters!)
Date d = {2024,2,1};

// als je dit niet wil: maak de constructor explicit
class Date {
    int year,month,day;
public:
    explicit Date(int y, int m, int d);
};

// werkt niet meer
Date d = {2024,2,1}; // ERROR
```

# Classes: constructor oproepen

*// waarom explicit gebruiken?*

```
class Date {  
    int year, month, day;  
public:  
    Date(int y) {  
        year = y;  
        month = 1;  
        day = 1;  
    }  
};
```

 werkt dit?

```
Date leeftijd = 35;
```

# Classes: constructor oproepen

*// waarom explicit gebruiken?*

```
class Date {  
    int year, month, day;  
public:  
    Date(int y) {  
        year = y;  
        month = 1;  
        day = 1;  
    }  
};
```



werkt dit?

```
Date leeftijd = 35;           // werkt! Maar doet niet wat je verwacht
```

```
// je bedoelde:  
int leeftijd = 35;
```

# Classes: constructor implementeren

```
// OFWEL als een gewone functie  
Date(int y, int m, int d){  
    year = y; // of Date::year = y;  
    ...  
}
```

# Classes: constructor implementeren

```
// OFWEL
Date(int y, int m, int d){
    year = y;
    ...
}
// OFWEL met initializer lists oude stijl
Date(int y, int m, int d): year(y), month(m), day(d) {
    ...
}
```



wordt uitgevoerd  
vóór de body van de constructor

# Classes: constructor implementeren

```
// OFWEL
Date(int y, int m, int d){
    year = y;
    ...
}
// OFWEL met initializer lists oude stijl
Date(int y, int m, int d): year(y), month(m), day(d) {
    ...
}
// OFWEL met initializer lists nieuwe stijl
Date(int y, int m, int d): year{y}, month{m}, day{d} {
    ...
}
```

# Classes: constructor implementeren

```
// OFWEL
Date(int y, int m, int d){
    year = y;
    ...
}
// OFWEL met initializer lists oude stijl
Date(int y, int m, int d): year(y), month(m), day(d) {
    ...
}
// OFWEL met initializer lists nieuwe stijl
Date(int y, int m, int d): year{y}, month{m}, day{d} {
    ...
}
// OFWEL een mengeling
Date(int y, int m, int d): year{y} {
    if(d <= 31){
        day = d;
    }
}
```

# Meerdere constructors

```
class Date {  
    int year;  
    int month;  
    int day;  
  
public:  
    Date() { set_date(1,1,1); }  
    Date(int y, int m, int d) { set_date(j,m,d); }  
    void set_date(int y, int m, int d);  
    void get_date(int& y, int& m, int& d) const;  
    void print() const;  
};
```

default  
constructor

hergebruik  
van code!

hier zit de logica  
voor een correcte datum

- Geen ambiguïteit

- Date d1; // eerste variant
- Date d2(1977,1,2) // tweede variant
- Date\* pd1 = new Date(2013,7,13); // tweede variant
- Date\* pd2 = new Date(); // eerste variant



# Meerdere constructors: let op!

- Als je geen constructor toevoegt:
  - C++ voegt een “default constructor” toe

```
class Date {  
    ...  
};  
  
=  
  
class Date {  
public:  
    Date() {}  
    ...  
};  
  
Date d;
```

- Als je wel een constructor toevoegt doet C++ dat NIET

```
class Date {  
public:  
    Date(int y, ...) { ... }  
    ...  
};
```

Date d;

**error: no matching function for call to 'Date::Date()'**

# Getters en setters

- Voor elk data member moet je beslissen of je die wil laten ...

- lezen
- schrijven

```
class Date {  
    int getYear();  
    void setYear(int y);  
};
```

- ... door een gebruiker van je klasse: getter/setter
- NIET default getters en setters voor alles aanmaken!!!
- Default geen enkele en als je die nodig hebt, voorzie je die.

# Destructor

- Wordt aangeroepen wanneer een variabele out-of-scope gaat; omgekeerde van een constructor
  - Zoals constructor, maar naam is naam klasse voorafgegaan door ~

```
struct StackNode {
    StackNode* next=nullptr;
    int content;
};

class Stack {
    StackNode* root=nullptr;
public:
    ~Stack(); // declaratie van de destructor
    void push(int c);
    bool is_empty() const {
        return root==nullptr;
    }
    int pop();
};
```

# Destructor

```
// Maak stack terug leeg
Stack::~~Stack() {
    while (root != nullptr) {
        StackNode* temp = root->next;
        delete root;
        root = temp;
    }
}
```

# Destructor: gebruik

- Wanneer het object verdwijnt, moeten we nog wat “opkuisen”
  - Dynamisch gereserveerd geheugen
  - Garanderen van een invariant
    - Bvb persoon: delete een persoon
      - verwijder uit lijst kinderen van vader, moeder
      - verwijder als vader/moeder bij de kinderen

# Klassen: implementatie vs interface

- Klassen in C++ hebben veel meer code dan Python: ctor/get/set

```
class Date:
    def __init__(self, y, m, d):
        self.year = y
        self.month = m
        self.day = d
```

```
class Date {
    Date(int y, int m, int d): year(y), ...{}

    int getYear() {
        return year;
    }

    void setYear(int y) {
        ...
    }

    int getMonth() {
        ...
    };

    void setMonth(int m) {
        ...
    }
}
```

o.a.  
daarom scheiden van  
implementatie en  
interface

# Klassen: implementatie vs interface

Date:

my\_birthday: year

1950

month

12

day

30

```
class Date {
    int year;
    int month;
    int day;

public:
    Date();
    Date(int y, int m, int d);
    void set_date(int y, int m, int d);
    void get_date(int& y, int& m, int& d) const;
    void print() const;
};
```

enkel declaratie,  
geen definitie

enkel WAT je kan doen,  
niet HOE je het kan doen

zoals een ADT  
moet zijn dus!

enkel contract,  
geen implementatie

# Klassen: implementatie vs interface

```
class Date {  
    int year;  
    int month;  
    int day;  
  
public:  
    Date();  
    Date(int d, int m, int d);  
    void set_date(int y, int m, int d);  
    void get_date(int& y, int& m, int& d) const;  
    void print() const;  
};
```

```
Date::Date(int d, int m, int d): year(y), ...
```

```
Date::set_date(int d, int m, int d) {...}
```

hier staat de declaratie

hier staat de definitie

methode en geen  
functie



# Klassen: implementatie vs interface

```
class Date {  
    int year;  
    int month;  
    int day;  
  
public:  
    Date();  
    Date(int d, int m, int d);  
    void set_date(int y, int m, int d);  
    void get_date(int& y, int& m, int& d) const;  
    void print() const;  
};  
  
int getMonth(){ return month; }  
  
int Date::season(){...}
```

welke error?

# Voorbeelden (cplusplus.org)

```
1 // example: class constructor
2 #include <iostream>
3 using namespace std;
4
5 class Rectangle {
6     int width, height;
7 public:
8     Rectangle (int,int);
9     int area () {return (width*height);}
10 };
11
12 Rectangle::Rectangle (int a, int b) {
13     width = a;
14     height = b;
15 }
16
17 int main () {
18     Rectangle rect (3,4);
19     Rectangle rectb (5,6);
20     cout << "rect area: " << rect.area() << endl;
21     cout << "rectb area: " << rectb.area() << endl;
22     return 0;
23 }
```

# Meerdere constructors (cplusplus.org)

```
1 // overloading class constructors
2 #include <iostream>
3 using namespace std;
4
5 class Rectangle {
6     int width, height;
7 public:
8     Rectangle ();
9     Rectangle (int,int);
10    int area (void) {return (width*height);}
11 };
12
13 Rectangle::Rectangle () {
14     width = 5;
15     height = 5;
16 }
17
18 Rectangle::Rectangle (int a, int b) {
19     width = a;
20     height = b;
21 }
22
23 int main () {
24     Rectangle rect (3,4);
25     Rectangle rectb;
26     cout << "rect area: " << rect.area() << endl;
27     cout << "rectb area: " << rectb.area() << endl;
28     return 0;
29 }
```

# Voorbeelden initialisatie

```
1 // classes and uniform initialization
2 #include <iostream>
3 using namespace std;
4
5 class Circle {
6     double radius;
7 public:
8     Circle(double r) { radius = r; }
9     double circum() {return 2*radius*3.14159265;}
10 };
11
12 int main () {
13     Circle foo (10.0);    // functional form
14     Circle bar = 20.0;    // assignment init.
15     Circle baz {30.0};    // uniform init.
16     Circle qux = {40.0}; // POD-like
17
18     cout << "foo's circumference: " << foo.circum() << '\n';
19     return 0;
20 }
```

Als je dit niet wil,  
explicit VOOR de  
constructor zetten

impliciete  
conversie

impliciete  
conversie

Plain Old Data

# Klassen in klassen

```
1 // member initialization
2 #include <iostream>
3 using namespace std;
4
5 class Circle {
6     double radius;
7     public:
8     Circle(double r) : radius(r) { }
9     double area() {return radius*radius*3.14159265;}
10 };
11
12 class Cylinder {
13     Circle base;
14     double height;
15     public:
16     Cylinder(double r, double h) : base (r), height(h) {}
17     double volume() {return base.area() * height;}
18 };
```

# Object-georiënteerd programmeren: begrippen

- Encapsulatie
- Compositie
- Invariant

# Object-georiënteerd programmeren: begrippen

- **Encapsulatie**: data en functies die deze data manipuleren met elkaar verbinden en beschermen van inferentie en foutief gebruik van buitenaf. *Data hiding* (beschermen van data door private te maken) is hierbij een belangrijke strategie.



dus geen getters en setters  
van alle data members!

# Object-georiënteerd programmeren: begrippen

- **Compositie**: door samenstelling nieuwe, complexere datatypes maken.

de relatie HEEFT-EEN

```
struct Date {  
    int day;  
    int month;  
    int year;  
};
```

een datum HEEFT een dag

volgende de week de  
relatie IS-EEN



# Object-georiënteerd programmeren: begrippen

- De notie van een “correcte datum” is een speciaal geval van het idee van een correcte waarde
- We proberen onze types zodanig te implementeren zodat de waarden gegarandeerd correct zijn **OP ELK MOMENT** in de code
  - Anders moeten we dat de hele tijd checken
- Een regel die bepaalt wat correct is, wordt een “**invariant**” genoemd
  - Dag, maand positief getal ; jaar niet gelijk aan 0 ; dag niet meer dan aantal dagen die maand



wat is een invariant voor datum?

# Object-georiënteerd programmeren: begrippen

- We proberen onze types zodanig te implementeren zodat de waarden gegarandeerd correct zijn
  - We implementeren member functies zodanig dat geldt: “Als de invariant voldaan is voordat de functie wordt uitgevoerd (preconditie), dan is ze nog steeds voldaan nadat de functie werd uitgevoerd (postconditie)”
- Als we niet zulk een invariant kunnen bedenken, hebben we waarschijnlijk te maken met een gewone data structuur
  - Gebruik een struct
  - Denk goed na over invarianten voor jouw klassen (dit vermijdt buggy code)

# Invariant datum

private; kan van buitenaf niet  
gewijzigd worden

```
class Date {  
    int year;  
    int month;  
    int day;  
  
public:  
    void set_date(int y, int m, int d);  
    void get_date(int& y, int& m, int& d) const;  
    void print() const;  
};
```

```
int nr_of_days(int month, int year) {  
    if (month==1 or month==3 or month==5 or month==7 or month==8 or month==10 or month==12) return 31;  
    if (month==2) {  
        if (year%4==0 && (year%100!=0 or year%400==0)) return 29; else return 28;  
    }  
    return 30;  
}
```

# Invariant datum

```
bool Date::set_date(int y, int m, int d) {  
    if (y <= 0) {  
        cerr << "Enkel data vanaf jaar 1 zijn toegestaan" << endl;  
        return false;    }  
    if (!(1 <= m && m <= 12)) {  
        cerr << "Maand moet een getal van 1 t.e.m. 12 zijn" << endl;  
        return false;    }  
    if (d < 1) {  
        cerr << "Dag moet minstens 1 zijn" << endl;  
        return false;    }  
    if (d > nr_of_days(m,y)) {  
        cerr << "Maand " << m << " in jaar " << y << " heeft slechts "  
            << nr_of_days(m,j) << " dagen." << endl;  
        return false;  
    }  
    day = d;  
    month = m;  
    year = j;  
    return true;  
}
```

de error  
stream

# Klassen

- Waarom ons druk maken over public/private onderscheid?
- Waarom maken we niet alles public?
  - Om een “clean interface” te hebben
    - Data en complexe hulp functies worden private
  - Om een invariant te kunnen garanderen
    - Enkel een beperkte set functies kan data manipuleren
  - Om debuggen te vereenvoudigen
    - Maar een paar functies hebben toegang tot de data
  - Om makkelijk de representatie te kunnen veranderen
    - Maar een aantal functies moeten veranderd worden
    - Je weet niet wie een publieke functie/data gebruikt

# Klassen: voordeel

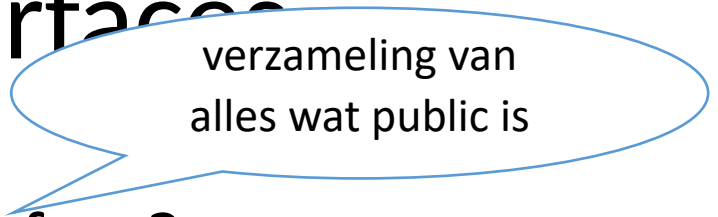
- Klassen laten ons toe om de interne werking van de klasse af te schermen
- We kunnen de interne werking dan ook veranderen, zolang de interface maar hetzelfde blijft

wanneer geheugen  
beperkt is (sensoren bv)

- Voorbeeld: datum

```
class Date {  
    int date_int; // day + month * 32 + year * 32 * 13  
  
public:  
    Date() { set_date(1,1,1); }  
    Date(int y, int m, int d) { set_datum(d,m,j); }  
    bool set_date(int y, int m, int d);  
    void get_date(int& y, int& m, int& d) const;  
    void print() const;  
};
```

# Klassen en interfaces



verzameling van  
alles wat public is

- Wat is een goede interface?
  - Minimaal
    - Zo klein mogelijk
  - Compleet
    - Maar niet kleiner
  - Type safe
    - Pas op voor verwarrende volgordes van variabelen
    - Pas op voor te algemene types
  - *Const correct*

# Const

```
class Date {  
    int year;  
    int month;  
    int day;  
  
public:  
    Date();  
    Date(int d, int m, int d);  
    void set_date(int y, int m, int d);  
    void get_date(int& y, int& m, int& d) const;  
    void print() const;  
};
```

gaat (mogelijks)  
object wijzigen

dat is een contract!

gaat object NIET wijzigen



# Const

```
class Date {  
    int year;  
    int month;  
    int day;  
  
public:  
    Date();  
    Date(int d, int m, int d);  
    void set_date(int y, int m, int d);  
    void get_date(int& y, int& m, int& d) const;  
    void print() const;  
};  
  
Date d1;  
const Date d2;  
  
d1.print(); // ok  
d2.print(); // ok  
  
d1.set_date(2024,1,1); // ok  
d2.set_date(2024,1,1); // error
```

# Klassen – Even samenvatten

- Klasse:
  - Samengesteld datatype
    - Variabelen van een klasse noemen we *member variables*
  - Samen met functies
    - Functies in een klasse noemen we *member functions*
- Public – private
  - Onderscheid tussen deel van buitenaf zichtbaar (denk aan contract) versus hoe het geïmplementeerd is
- Klassen staan ons toe complexiteit af te schermen
  - Betere code, makkelijker te onderhouden

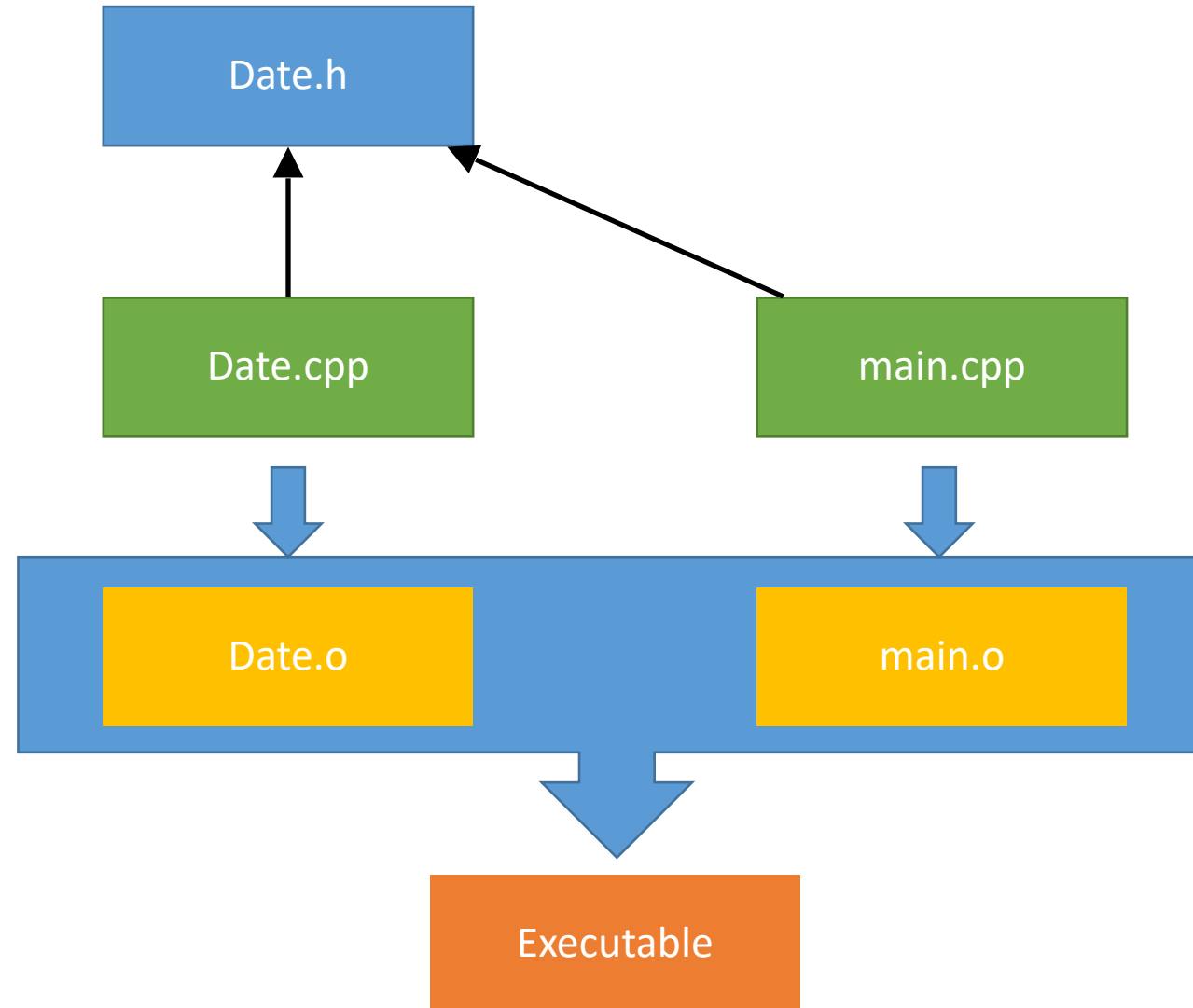
# Object-georiënteerd programmeren: begrippen

- Encapsulatie: data en functies die deze data manipuleren met elkaar verbinden en beschermen van inferentie en foutief gebruik van buitenaf. *Data hiding* (beschermen van data door private te maken) is hierbij een belangrijke strategie.
- Invariant: integriteitseigenschap die steeds gegarandeerd moet worden; member functies moeten zodanig geïmplementeerd zijn dat ze invarianten bewaren (voor aanroep geldt invariant → na aanroep ook)
- Compositie: door samenstelling nieuwe, complexere datatypes maken. Dankzij encapsulatie kunnen we de complexiteit van samengestelde datatypes beheersen.

# Preprocessor, compiler, linker

- We kunnen onze code onderverdeling in logische “pakketjes”
  - Declaraties in een “**header file**”
  - Implementatie in een normale .cpp file die de header #include
- De pre-processor voert de directives uit (#include, #def, ...) per cpp bestand
- De output stream van elk .cpp bestand gaat naar de compiler die er een object bestand van maakt met extentie .o
- De linker hangt die object bestanden vervolgens aan elkaar en maakt de finale executable

# Preprocessor, compiler, linker



# Pre-processor directives

- We kunnen elke functie, klasse, etc. maar 1 maal declareren:

Wordt enkel  
gelezen door de  
compiler als  
DATE\_DATE\_H  
niet gekend is

```
#ifndef DATE_DATE_H  
#define DATE_DATE_H
```

Definieer DATE\_DATE\_H; vanaf nu is deze  
gekend en worden deze regels niet meer  
gelezen

```
class Date {  
    int year;  
    int month;  
    int day;  
public:  
    bool set_date(int y, int m, int d);  
    void get_date(int& y, int& m, int& d) const;  
    void print() const;  
};
```

```
#endif //DATE_DATE_H
```

# Lang leve CLion

- Class aanmaken met data members
- Constructor genereren (of meerdere)
- Destructor genereren
- Getters en setters genereren