

Programmation Orientée Objet : Constructeurs/Destructeurs

Jean-Cédric Chappelier

Faculté I&C

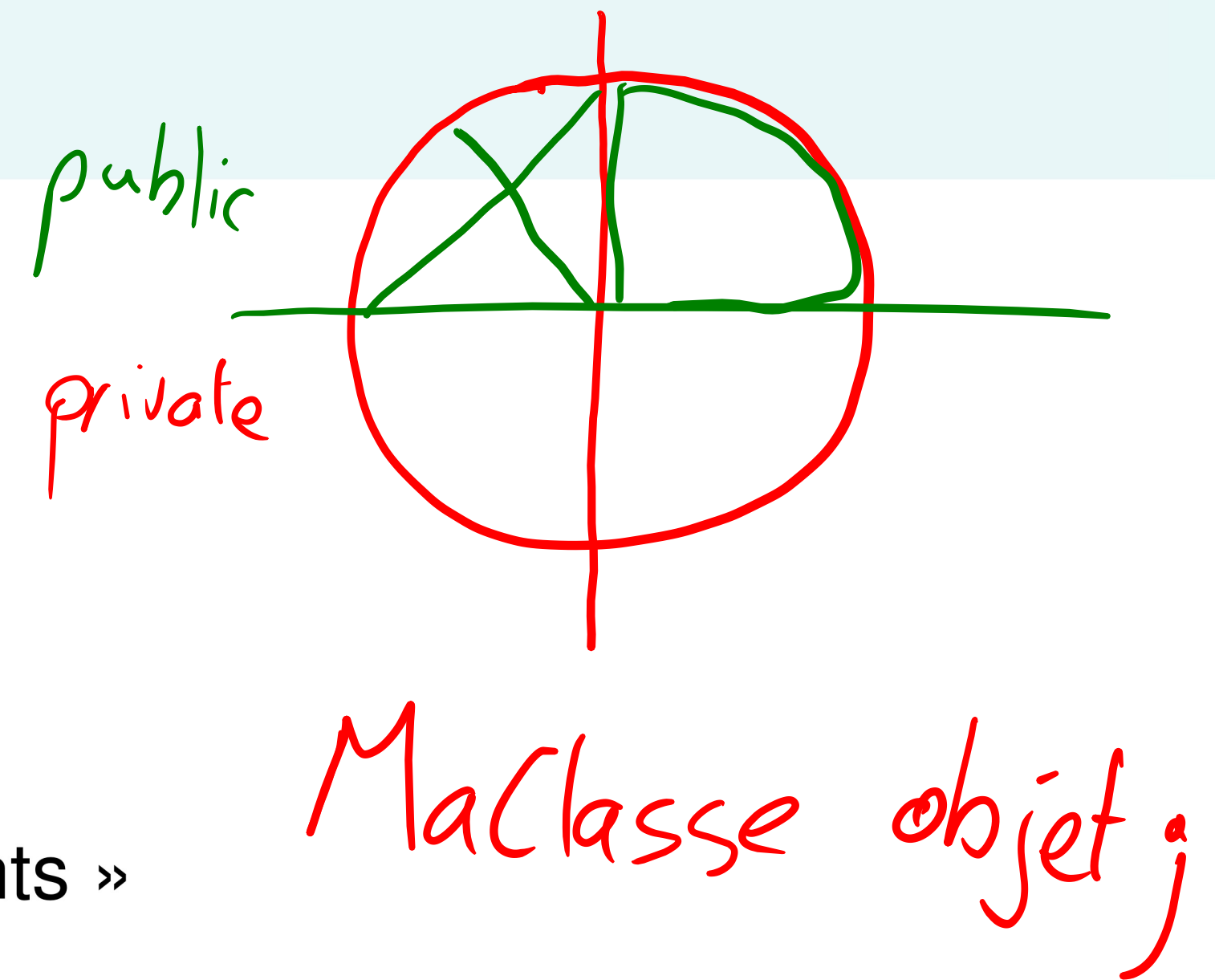
Organisation du travail (semestre)

MOOC		déc.	cours 1 h Jeudi 8-9	exercices 2 h Jeudi 9-11
1	19.02.26		0	Intro + compil. séparée
2	26.02.26	1. Intro POO	0	Intro POO
3	05.03.26	2. Constructeurs/Destructeurs	0	Constructeurs
4	12.03.26	3. Surcharge des opérateurs	0	Surcharge
5	19.03.26	4. Héritage	0	Héritage
6	26.03.26	5. Polymorphisme	0	Polymorphisme 1
7	02.04.26		1	Polymorphisme 2 / Collections hétérogènes
-	09.04.26		-	vacances Pâques
8	16.04.26	6. Héritage multiple	1	Héritage multiple
9	23.04.26		-	- Midtem
10	30.04.26	(7. Etude de cas)	-	Templates
11	07.05.26		-	Structure de données abstraites ; Bibliothèques
12	14.05.26		-	(Ascension)
13	21.05.26	(7. Etude de cas)	-	Bibliothèques (fin) + Révisions
14	28.05.26		-	- Examen

Objectifs de la leçon d'aujourd'hui

- ▶ Concepts fondamentaux
- ▶ Compléments non abordés dans le MOOC
- ▶ Étude de cas

Concepts fondamentaux



- ▶ Rôles des constructeurs :
initialiser les instances
- ▶ Ecriture des constructeurs :
utilisez **la liste d'initialisation**, « section deux-points »
- ▶ Ecriture des constructeurs :
il n'est très souvent **pas** nécessaire d'écrire le constructeur de copie
(pensez alors au destructeur et à `operator=`)
- ▶ Rôles des destructeurs :
faire, *si nécessaire*, ce que l'on doit faire *avant* la disparition de l'instance

règle
des 3



C++11

Initialisation avec des listes



C++11 a généralisé la notion de listes de valeurs.

Nous avons par exemple vu :

```
vector<int> ages({ 20, 35, 26, 38, 22 });
```

Vous pouvez aussi munir vos classes d'une construction par listes avec le constructeur suivant :

NomClasse(initializer_list<type> const&)

(nécessite un `#include <initializer_list>`)

Si nécessaire, parcourez la liste avec un « *range-based for* » (ou utilisez le constructeur d'une autre classe qui supporte les listes d'initialisation, telles que `vector` ou `array`).

Exemple :

```
class A {  
public:  
    A(initializer_list<double> const& liste) {  
        for (auto a : liste) { cout << a << endl; }  
    }  
};
```

: contenu(liste)
✓

A b({ 3.4, 4.5 });



C++11

Constructeur de déplacement



C++11 fournit également un moyen de gérer les instances temporaires (« *r-values* ») : le *constructeur de déplacement* (« *move constructor* »)

Ce constructeur permet d'initialiser une instance en *déplaçant* les attributs d'une autre instance, temporaire, du même type.

Syntaxe :

```
NomClasse(NomClasse && autre)
: ...
{ ... }
```

Exemple (pas très pertinent) :

```
Rectangle(Rectangle && autre)
: hauteur(move(autre.hauteur))
, largeur(move(autre.largeur))
{ }
```

Note : l'exemple ci-dessus n'est pas très pertinent car la classe en question est petite. L'intérêt est ici limité. Gérer la « *move semantics* » prend plus de sens pour des classes ayant (potentiellement) de **gros contenus** (typiquement au travers d'un *pointeur*).

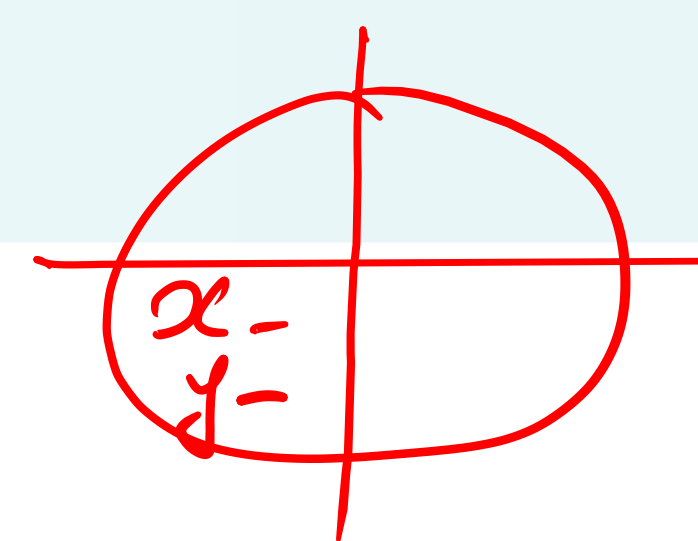
Et c'est de toutes manières un sujet avancé !

f (cree.complexe(...))

g (3.5)

Rectangle rz (cree.rectangle(...));

Etude de cas



Comment initialiser nos nombres complexes ?

$\text{Complexe}(\text{double } a, \text{double } b)$
: $x-(a), y-(b)$ }

$\text{Complexe}(\text{double } \rho, \text{double } \theta)$
: $x-(\rho * \cos \theta) \dots$

Etude de cas

Comment initialiser nos nombres complexes ?

Première idée : parties réelle et imaginaire

Ensuite :

1^{re} question : constructeur par défaut ?

- ▶ En **a**-t-on un ?
- ▶ En **veut**-on un ?

2^e question : plongement des réels ?

3^e question : constructeur de copie ?

4^e question : autres constructeurs ? (ρ, θ) ?

Premier constructeur

Idée la plus simple : par coordonnées cartésiennes

```
class Complexe {  
public:  
    // constructeurs  
    Complexe(double abscisse, double ordonnee)  
        : x_(abscisse), y_(ordonnee)  
    {}  
    //...  
};
```

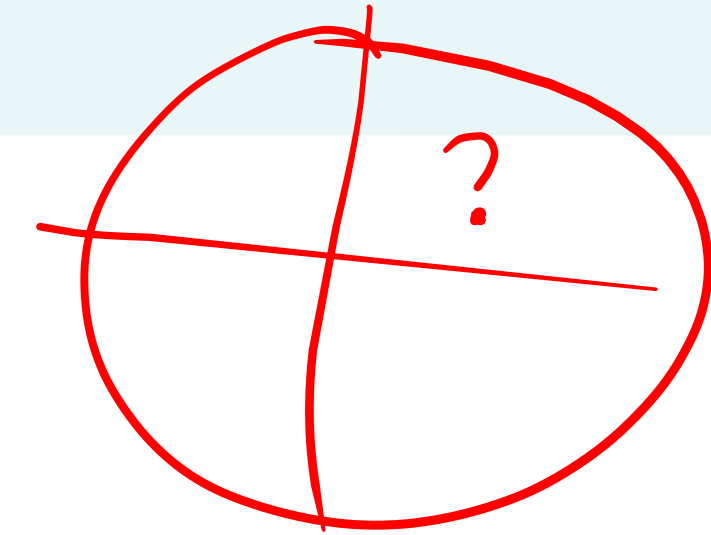
Utilisation :

```
Complexe z1(1.1, 2.2);
```

~~double f();
Complexe z3();~~

Complexe z2;

Constructeur par défaut ?



A-t-on déjà un constructeur par défaut ?

Peut-on écrire

Complexe z2;

?

👉 **NON !** Car le constructeur par défaut par défaut n'est plus fourni, vu que nous avons déclaré un autre constructeur !

⇒ on doit donc ajouter un constructeur par défaut si l'on en veut un.

Veut-on un constructeur par défaut ?

👉 Oui, cela fait sens pour les complexes : typiquement 0

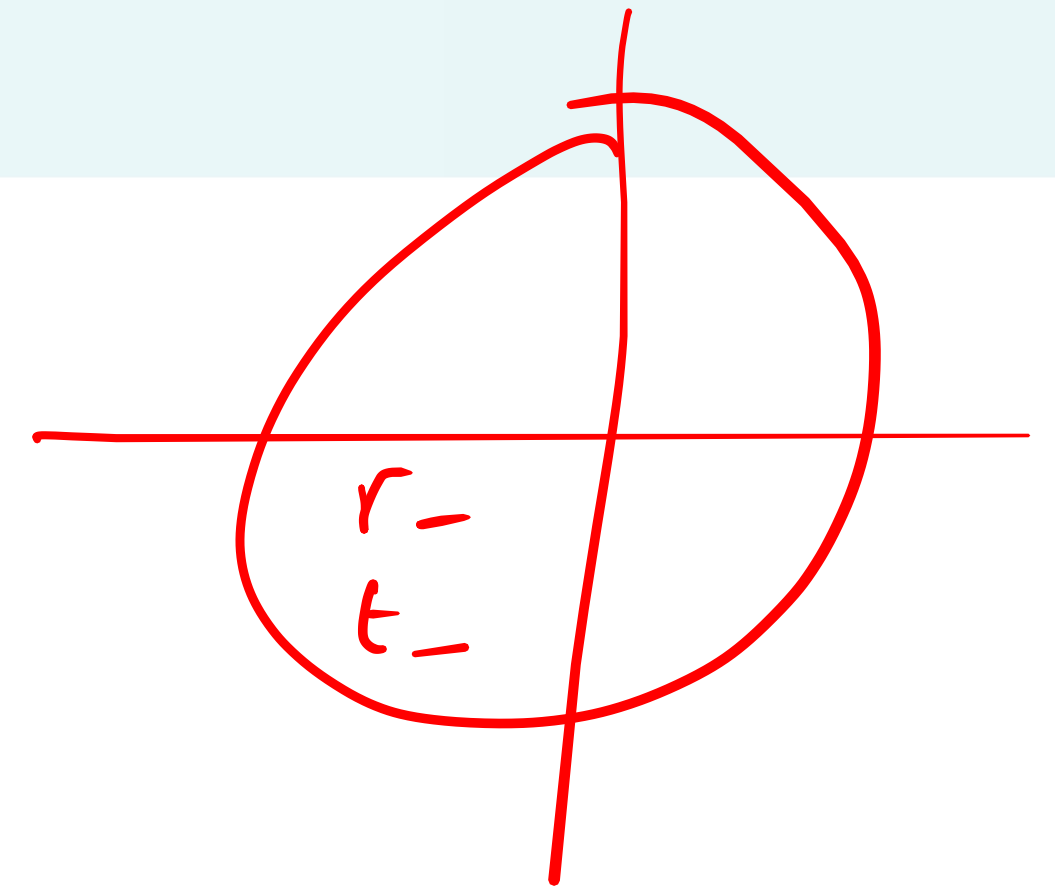
Complexe() : x-(0.0), y-(0.0)}}
Complexe() : Complexe(0.0, 0.0) {}

Constructeur par défaut (1/3)

Nous avons 3 façons de le faire :

1. soit séparément, version 1 :

```
class Complexe {  
public:  
    // constructeurs  
    Complexe(double abscisse, double ordonnee)  
    : x_(abscisse), y_(ordonnee)  
    {}  
    Complexe()  
    : x_(0.0), y_(0.0)  
    {}  
    //...  
};
```



Constructeur par défaut (2/3)

Nous avons 3 façons de le faire :

2. soit séparément, version 2 (C++11 uniquement) :

```
class Complexe {  
public:  
    // constructeurs  
    Complexe(double abscisse, double ordonnee)  
    : x_(abscisse), y_(ordonnee)  
    {}  
    Complexe()  
    : Complexe(0.0, 0.0)  
    {}  
    //...  
};
```

Constructeur par défaut (3/3)

Nous avons 3 façons de le faire :

3. soit regroupé avec notre constructeur précédent, en donnant des valeurs par défaut aux paramètres :

```
class Complexe {  
public:  
    // constructeurs  
    Complexe(double abscisse = 0.0, double ordonnee = 0.0)  
        : x_(abscisse), y_(ordonnee)  
    {}  
    //...  
};
```



3

1

4

(5)

Complexe z1(1.0);

Complexe z2;

Complexe z3(3.1, 4.6);

Complexe z4(z3);

Constructeur par défaut (3/3)

Nous avons 3 façons de le faire :

3. soit regroupé avec notre constructeur précédent, en donnant des valeurs par défaut aux paramètres :

```
class Complexe {  
public:  
    // constructeurs  
    Complexe(double abscisse = 0.0, double ordonnee = 0.0)  
    : x_(abscisse), y_(ordonnee)  
    {}  
    //...  
};
```

Note : nous avons alors dans ce cas *trois* constructeurs (écrits en un seul) !

Plongement des réels ?

Comment faire pour « plonger $\mathbb{R}$ dans $\mathbb{C}$ » ?

Complexe `z3(1.0);`

Avec les solutions 1 et 2 précédentes :
écrire encore un troisième constructeur.

Avec la solution 3 : rien à faire, c'est déjà fait !

Constructeur de copie ?

Peut-on (déjà) construire des complexes comme ceci :

Complexe $z_4(z_3)$;

Réponse : **OUI!**

Nous avons un constructeur de copie fourni par défaut.

Est-ce qu'il nous suffit (copie de surface) ?

👉 Oui ! (donc rien à faire !)

Constructeur en coordonnées polaires ?

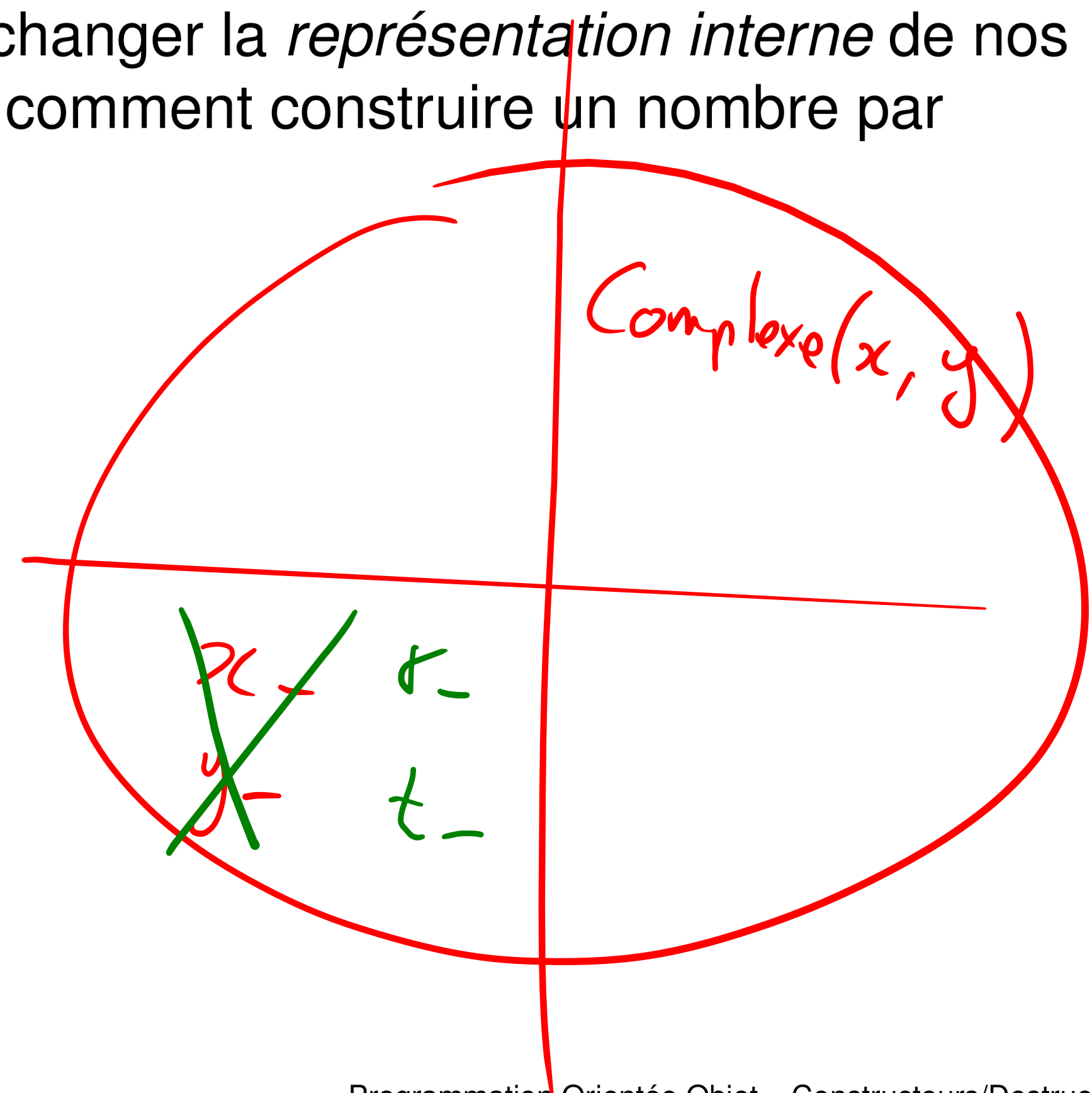
$(A (t + z_0))$

Peut-on construire des complexes comme ceci (pour $z_5 = e^{i\pi}$) :

Complexe `z5(1.0, M_PI);` oui : $z_5 = 1 + \pi i$

Question subsidiaire : si l'on décide de changer la *représentation interne* de nos nombres complexes (p.ex. en polaires), comment construire un nombre par coordonnées cartésiennes ?

Complexe `z3(3.1, 4.6);`



Constructeur en coordonnées polaires ?

Peut-on construire des complexes comme ceci (pour $z_5 = e^{i\pi}$) :

Complexe `z5(1.0, M_PI)`;

Question subsidiaire : si l'on décide de changer la *représentation interne* de nos nombres complexes (p.ex. en polaires), comment construire un nombre par coordonnées cartésiennes ?

Réponse à la 2^e question est : **exactement comme avant !** (*encapsulation*)

Réponse à la 1^{re} question : oui, mais cela ne construit pas $e^{i\pi}$, mais bien $1 + \pi i$!

Constructeur en coordonnées polaires ? (1/2)

Comment faire un constructeur en coordonnées polaires ?

1. Ne pas en faire, mais fournir une méthode :
cf le manipulateur `polaires()` de la semaine passée
☞ Mais cela oblige alors une *construction* en cartésiennes : – (!
Ce **n'**est donc **pas** vraiment une solution !

2. Changer le prototype (la « *signature* ») :
 - version simple : ajouter un argument supplémentaire inutile

```
Complexe(double module, double argument, bool inutile)
: x_(module * cos(argument)), y_(module * sin(argument))
{}
```

$x + y$
 $y + x$

Utilisation :

```
Complexe z5(1.0, M_PI, true);
```

Constructeur en coordonnées polaires ? (2/2)

3. Changer le prototype (la « *signature* ») :

- première version plus explicite : donner du sens à l'argument supplémentaire

```
enum Systeme { cartesiennes, polaires };

Complexe(double prems, double deuz, Systeme mode = cartesiennes)
{
    if (mode == cartesiennes) {
        x_ = prems;
        y_ = deuz;
    } else {
        x_ = prems * cos(deuz); y_ = prems * sin(deuz);
    }
}
```

Utilisation : `Complexe z6(1.0, M_PI, polaires);`

Constructeur en coordonnées polaires ? (2/2)

4. changer le prototype (la « *signature* ») :
- ▶ seconde version plus explicite : créer un nouveau type pour désambiguïser

```
struct Polaires { double r_; double t_; };

class Complexe {
public:
    Complexe(double abscisse = 0.0, double ordonnee = 0.0) // comme avant
    : x_(abscisse), y_(ordonnee)
    {}
    Complexe(Polaires rt) // nouveau
    : x_(rt.r_ * cos(rt.t_)), y_(rt.r_ * sin(rt.t_))
    {}
    // ...
}
```

Utilisation : `Complexe z6(Polaires {1.0, M_PI});`

5.  offrir une « *factory* » : méthode *de classe* construisant un `Complexe` avec les arguments voulus ; p.ex. :

```
static Complexe genere_polaires(double module, double argument)
{ return Complexe(module * cos(argument), module * sin(argument)); }
```