Langage C++ et programmation orientée objet

Vincent Vidal

Maître de Conférences
Enseignements : IUT Lyon 1 - pôle AP - Licence RESIR - bureau 2ème étage
Recherche : Laboratoire LIRIS - bât. Nautibus - bureau 241
E-mail : vincent.vidal@univ-lyon1.fr
Supports de cours et TPs : `https://clarolineconnect.univ-lyon1.fr/` espace d'activité "M41L02C - Langage C++"
26H prévues ≈ 24H de cours+TDs/TPs, et 2H - examen final
Évaluation : Contrôle continu (TPs) + examen final

Plan

Plan

```
// déclaration "basique" :
struct complex
{
    float im, re;
};
```

Le C++ permet d'associer des **méthodes** (des **fonctions membres**) à une structure.

```
// déclaration "basique" :
struct complex
{
    float im, re;
};
```

Le C++ permet d'associer des **méthodes** (des **fonctions membres**) à une structure.

Déclaration d'une structure généralisée

```
// déclaration qui associe des fonctions membres :
struct complex
{
    // déclaration des données membres (attributs)
    float im, re;
    // déclaration des fonctions membres (méthodes)
    void initialise() ;
    void modifie(float, float) ;
    void affiche() ;
};
```

Les méthodes d'une structure ont accès aux membres données ou **attributs** de la structure. **Les attributs sont donc des arguments implicites des méthodes de leur structure.**

Déclaration d'une structure généralisée

```
// déclaration qui associe des fonctions membres :
struct complex
{
    // déclaration des données membres (attributs)
    float im, re;
    // déclaration des fonctions membres (méthodes)
    void initialise() ;
    void modifie(float, float) ;
    void affiche() ;
};
```

Les méthodes d'une structure ont accès aux membres données ou **attributs** de la structure. Les attributs sont donc des arguments implicites des méthodes de leur structure.

Déclaration d'une structure généralisée

```
// déclaration qui associe des fonctions membres :
struct complex
{
    // déclaration des données membres (attributs)
    float im, re;
    // déclaration des fonctions membres (méthodes)
    void initialise() ;
    void modifie(float, float) ;
    void affiche() ;
};
```

Les méthodes d'une structure ont accès aux membres données ou **attributs** de la structure. **Les attributs sont donc des arguments implicites des méthodes de leur structure.**

Définitions des méthodes d'une structure généralisée

```
void complex::initialise() {
   re = im = 0.f ;
}
void complex::modifie(float im_arg, float re_arg) {
   re = re_arg ;
   im = im_arg ;
}
void complex::affiche() {
   std::cout << re << " + i*" << im << std::endl ;
}
```

:: est l'opérateur **résolution de portée** (le même que pour les espaces de noms).

Que se passe-t-il si on oublie le complex :: *?*

Définitions des méthodes d'une structure généralisée

```
void complex::initialise() {
   re = im = 0.f ;
}
void complex::modifie(float im_arg, float re_arg) {
   re = re_arg ;
   im = im_arg ;
}
void complex::affiche() {
   std::cout << re << " + i*" << im << std::endl ;
}
```

: : est l'opérateur **résolution de portée** (le même que pour les espaces de noms).

Que se passe-t-il si on oublie le complex : : *?*

Quelques précisions sur les définitions des méthodes

- **Les définitions des méthodes nécessitent la déclaration préalable de la structure** : il faut soit les *mettre après la déclaration de la structure* si on reste dans le même fichier .cpp, soit les *mettre dans un fichier .cpp qui inclut le fichier .h* (**#include** "fichier.h") contenant la déclaration de la structure (cf. notion de composant/module logiciel).
- **Pour des définitions très courtes** (1 return, 1 affectation ou l'affichage de la valeur d'1 seul membre...), **la définition directe de la méthode dans la structure est autorisée** : **méthode en ligne**.

Quelques précisions sur les définitions des méthodes

- **Les définitions des méthodes nécessitent la déclaration préalable de la structure** : il faut soit les *mettre après la déclaration de la structure* si on reste dans le même fichier .cpp, soit les *mettre dans un fichier .cpp qui inclut le fichier .h* (**#include** "fichier.h") contenant la déclaration de la structure (cf. notion de composant/module logiciel).
- **Pour des définitions très courtes** (1 return, 1 affectation ou l'affichage de la valeur d'1 seul membre...), **la définition directe de la méthode dans la structure est autorisée** : **méthode en ligne**.

Utilisation d'une structure généralisée

```
#include "complex.h" // on souhaite utiliser le module "complex"
int main()
{
    complex c1, c2; // les variables c1 et c2 disposent des méthodes de la structure complex
    c1.modifie(3.4f, -5.f); c1.affiche(); // c1.modifie(3.4f, -5.f) : ne change la valeur
                                          // des attributs im et re que pour c1
                                          // c1.affiche() : la méthode affiche reçoit la
                                          //  variable c1 en argument effectif
    c2.initialise(); c2.affiche();
    c1.re += 4; // opération autorisée (pas encore l'encapsulation des données, mais
                // on s'en rapproche) : on peut accéder aux attributs depuis l'extérieur!
    return 0;
}
```

Remarques : chaque variable de type *complex* dispose de ses propres exemplaires de données (ici `im` et `re`), mais un seul exemplaire des méthodes est partagé ; `c1.re` a un emplacement mémoire différent de `c2.re` !

Utilisation d'une structure généralisée

```
#include "complex.h" // on souhaite utiliser le module "complex"
int main()
{
    complex c1, c2; // les variables c1 et c2 disposent des méthodes de la structure complex
    c1.modifie(3.4f, -5.f); c1.affiche(); // c1.modifie(3.4f, -5.f) : ne change la valeur
                                          // des attributs im et re que pour c1
                                          // c1.affiche() : la méthode affiche reçoit la
                                          //  variable c1 en argument effectif
    c2.initialise(); c2.affiche();
    c1.re += 4; // opération autorisée (pas encore l'encapsulation des données, mais
                // on s'en rapproche) : on peut accéder aux attributs depuis l'extérieur!
    return 0;
}
```

Remarques : chaque variable de type *complex* dispose de ses propres exemplaires de données (ici im et re), mais un seul exemplaire des méthodes est partagé ; c1.re a un emplacement mémoire différent de c2.re !

Notion

Une **classe** est une structure dans laquelle seulement certains membres et/ou certaines méthodes sont accessibles depuis l'extérieur, les autres étant seulement accessibles à l'intérieur.

Une classe permet d'**encapsuler** des données (et des méthodes) en les **déclarant privées** (mot clef `private`) !

La déclaration d'une classe est voisine de celle d'une structure :

- il faut remplacer le mot clef `struct` par `class` ;
- il faut préciser quels membres (données ou fonctions) sont publics ou privés à l'aide des mots clés `public` et `private`. Si tous les membres sont publics on obtient l'équivalent d'une structure !

Notion

Une **classe** est une structure dans laquelle seulement certains membres et/ou certaines méthodes sont accessibles depuis l'extérieur, les autres étant seulement accessibles à l'intérieur.

Une classe permet d'**encapsuler** des données (et des méthodes) en les **déclarant privées** (mot clef **private**) !

La déclaration d'une classe est voisine de celle d'une structure :

- il faut remplacer le mot clef **struct** par **class** ;
- il faut préciser quels membres (données ou fonctions) sont publics ou privés à l'aide des mots clés **public** et **private**. Si tous les membres sont publics on obtient l'équivalent d'une structure !

Notion

Une **classe** est une structure dans laquelle seulement certains membres et/ou certaines méthodes sont accessibles depuis l'extérieur, les autres étant seulement accessibles à l'intérieur.

Une classe permet d'**encapsuler** des données (et des méthodes) en les **déclarant privées** (mot clef **private**) !

La déclaration d'une classe est voisine de celle d'une structure :

- il faut remplacer le mot clef **struct** par **class** ;
- il faut préciser quels membres (données ou fonctions) sont publics ou privés à l'aide des mots clés **public** et **private**. Si tous les membres sont publics on obtient l'équivalent d'une structure !

Notion

Une **classe** est une structure dans laquelle seulement certains membres et/ou certaines méthodes sont accessibles depuis l'extérieur, les autres étant seulement accessibles à l'intérieur.

Une classe permet d'**encapsuler** des données (et des méthodes) en les **déclarant privées** (mot clef **private**) !

La déclaration d'une classe est voisine de celle d'une structure :

- il faut remplacer le mot clef **struct** par **class** ;
- il faut préciser quels membres (données ou fonctions) sont publics ou privés à l'aide des mots clés **public** et **private**. Si tous les membres sont publics on obtient l'équivalent d'une structure !

Déclaration

```
class complex
{
  private :
    float im, re ;    // données privées (encapsulées)
    void sensible() ; // méthode privée
  public :
    // méthodes publiques :
    void initialise() ;
    void modifie(float, float) ;
    void affiche() ;
};
```

Les méthodes d'une classe ont accès à toutes les données de la classes quelles soient publiques ou privées !

Déclaration

```
class complex
{
  private :
    float im, re ;     // données privées (encapsulées)
    void sensible() ; // méthode privée
  public :
    // méthodes publiques :
    void initialise() ;
    void modifie(float, float) ;
    void affiche() ;
};
```

Les méthodes d'une classe ont accès à toutes les données de la classes quelles soient publiques ou privées !

Définitions des méthodes

Identiques au cas d'une structure.

Utilisation d'une classe

```
#include "complex.h"
int main()
{
    complex c1, c2; // les objets c1 et c2 disposent des méthodes de la classe complex
    c1.modifie(3.4f, -5.f);
    c1.affiche();   // les méthodes publiques sont utilisables/accessibles depuis l'extérieur
    c2.initialise(); c2.affiche();
    c1.re += 4;     // opération interdite (sauf si re est public)
    return 0;
}
```

Dans le vocabulaire de la POO, les variables c1 et c2 sont appelées des **objets** ou des **instances** de la classe complex.

c1.re a un emplacement mémoire différent de c2.re !

Utilisation d'une classe

```
#include "complex.h"
int main()
{
    complex c1, c2; // les objets c1 et c2 disposent des méthodes de la classe complex
    c1.modifie(3.4f, -5.f);
    c1.affiche();   // les méthodes publiques sont utilisables/accessibles depuis l'extérieur
    c2.initialise(); c2.affiche();
    c1.re += 4;     // opération interdite (sauf si re est public)
    return 0;
}
```

Dans le vocabulaire de la POO, les variables c1 et c2 sont appelées des **objets** ou des **instances** de la classe complex.

c1.re a un emplacement mémoire différent de c2.re !

Quelques précisions

Sur les méthodes privées :

- elles ne sont appelables que depuis d'autres méthodes de la classe (intérieur de la classe) ;
- elles sont utiles pour découper un problème en sous-problèmes à résoudre sans que l'utilisateur final ne puisse utiliser ces "sous-programmes" ;

Sur les mots clés d'accessibilité :

- les mots clefs **public** et **private** peuvent apparaître plusieurs fois ; par défaut c'est private qui est sélectionné ;
- il existe un autre mot cléf (**protected**) qui désigne un status entre private et public et qui sera abordé avec l'héritage.

Quelques précisions

Sur les méthodes privées :

- elles ne sont appelables que depuis d'autres méthodes de la classe (intérieur de la classe) ;
- elles sont utiles pour découper un problème en sous-problèmes à résoudre sans que l'utilisateur final ne puisse utiliser ces "sous-programmes" ;

Sur les mots clés d'accessibilité :

- les mots clefs `public` et `private` peuvent apparaître plusieurs fois ; par défaut c'est private qui est sélectionné ;
- il existe un autre mot cléf (`protected`) qui désigne un status entre private et public et qui sera abordé avec l'héritage.

Quelques précisions

Sur les méthodes privées :

- elles ne sont appelables que depuis d'autres méthodes de la classe (intérieur de la classe) ;
- elles sont utiles pour découper un problème en sous-problèmes à résoudre sans que l'utilisateur final ne puisse utiliser ces "sous-programmes" ;

Sur les mots clés d'accessibilité :

- les mots clefs **public** et **private** peuvent apparaître plusieurs fois ; par défaut c'est private qui est sélectionné ;
- il existe un autre mot cléf (**protected**) qui désigne un status entre private et public et qui sera abordé avec l'héritage.

Quelques précisions

Sur les méthodes privées :

- elles ne sont appelables que depuis d'autres méthodes de la classe (intérieur de la classe) ;
- elles sont utiles pour découper un problème en sous-problèmes à résoudre sans que l'utilisateur final ne puisse utiliser ces "sous-programmes" ;

Sur les mots clés d'accessibilité :

- les mots clefs **public** et **private** peuvent apparaître plusieurs fois ; par défaut c'est private qui est sélectionné ;
- il existe un autre mot cléf (**protected**) qui désigne un status entre private et public et qui sera abordé avec l'héritage.

Concernant les variables de structures généralisées et les objets

L'**affectation entre 2 variables de type structure est possible**, *même si la structure est généralisée*.

Elle **consiste à recopier les valeurs des attributs 1 à 1** (elle n'agit pas sur les méthodes d'une structure généralisée).

Cette possibilité s'étend aux objets ! Une affectation entre 2 objets affectera les attributs 1 à 1 qu'ils soient **private**, **public** ou **protected**.

```
complex c1, c2 ;
...
c1 = c2 ;
```

Concernant les variables de structures généralisées et les objets

L'**affectation entre 2 variables de type structure est possible**, *même si la structure est généralisée*.
Elle **consiste à recopier les valeurs des attributs 1 à 1** (elle n'agit pas sur les méthodes d'une structure généralisée).

Cette possibilité s'étend aux objets ! Une affectation entre 2 objets affectera les attributs 1 à 1 qu'ils soient **private**, **public** ou **protected**.

```
complex c1, c2 ;
...
c1 = c2 ;
```

Concernant les variables de structures généralisées et les objets

L'**affectation entre 2 variables de type structure est possible**, *même si la structure est généralisée*.
Elle **consiste à recopier les valeurs des attributs 1 à 1** (elle n'agit pas sur les méthodes d'une structure généralisée).

Cette possibilité s'étend aux objets ! Une affectation entre 2 objets affectera les attributs 1 à 1 qu'ils soient **private**, **public** ou **protected**.

```
complex c1, c2 ;
...
c1 = c2 ;
```

Concernant les variables de structures généralisées et les objets

L'**affectation entre 2 variables de type structure est possible**, *même si la structure est généralisée*.
Elle **consiste à recopier les valeurs des attributs 1 à 1** (elle n'agit pas sur les méthodes d'une structure généralisée).

Cette possibilité s'étend aux objets ! Une affectation entre 2 objets affectera les attributs 1 à 1 qu'ils soient **private**, **public** ou **protected**.

```
complex c1, c2;
...
c1 = c2;
```

Plan

Qu'est-ce qu'un objet en POO ? (1/2)

C'est une **notion intimement liée à la notion d'objet dans notre monde physique** (souris, clavier...).

	Objet physique	**Objet POO**
interface extérieure	boutons *On/Off*	méthodes *marche()/arret()*
implantation intérieure	cachée	cachée *encapsulation*
propriétés	physiques (couleur...)	valeurs des attributs
durée de vie	de l'usine à la casse	de la construction à la destruction

TABLE – Le concept d'objet

Qu'est-ce qu'un objet en POO ? (2/2)

Un objet est **une boite noire** : on peut interagir avec lui sans connaître son fonctionnement interne.

Exemple : un chronomètre a 1 bouton marche/arrêt et 1 bouton de remise à zéro. *Quelles sont les méthodes publiques d'un objet de type chronomètre en POO ?*

Qu'est-ce qu'un objet en POO ? (2/2)

Un objet est **une boite noire** : on peut interagir avec lui sans connaître son fonctionnement interne.

Exemple : un chronomètre a 1 bouton marche/arrêt et 1 bouton de remise à zéro. *Quelles sont les méthodes publiques d'un objet de type chronomètre en POO ?*

Qu'est-ce qu'un objet en POO ? (2/2)

Un objet est **une boite noire** : on peut interagir avec lui sans connaître son fonctionnement interne.

Exemple : un chronomètre a 1 bouton marche/arrêt et 1 bouton de remise à zéro. *Quelles sont les méthodes publiques d'un objet de type chronomètre en POO ?* afficher(), arreter(), declencher(), remise_a_zero()

Qu'est-ce qu'un objet en POO ? (2/2)

Un objet est **une boite noire** : on peut interagir avec lui sans connaître son fonctionnement interne.

Exemple : un chronomètre a 1 bouton marche/arrêt et 1 bouton de remise à zéro. *Quelles sont les méthodes publiques d'un objet de type chronomètre en POO ?* afficher(), arreter(), declencher(), remise_a_zero()

Remarque : l'utilisateur ne peut pas modifier directement le temps de l'objet chronomètre. Il aurait pu le faire avec une structure généralisée...

Qu'est-ce qu'une classe en POO ?

Une classe en POO est un **type désignant une famille d'objets** partageant des mêmes attributs et opérations (spécifiant les comportements possibles des objets).

Une classe apparaît aussi comme un **moule ou une usine à partir de laquelle il est possible de créer/construire des objets**.

Qu'est-ce qu'une classe en POO ?

Une classe en POO est un **type désignant une famille d'objets** partageant des mêmes attributs et opérations (spécifiant les comportements possibles des objets).

Une classe apparaît aussi comme un **moule ou une usine à partir de laquelle il est possible de créer/construire des objets**.

Introduction et intérêt des constructeurs

A la déclaration d'une variable, *les valeurs par défaut* des attributs d'une classe vont dépendre de sa classe d'allocation. Soit 0 pour la classe d'allocation statique, soit une valeur aléatoire.

Si un utilisateur a besoin d'initialiser son objet avec des valeurs spécifiques afin que son objet fonctionne correctement, il doit faire appel à une fonction membre *initialise*.
Si par mégarde il oublie de faire appel à cette fonction *initialise*, alors son objet ne fonctionnera pas correctement...

Le C++ offre une solution puissante à ce problème : la **notion de constructeur**.

Introduction et intérêt des constructeurs

A la déclaration d'une variable, *les valeurs par défaut* des attributs d'une classe vont dépendre de sa classe d'allocation. Soit 0 pour la classe d'allocation statique, soit une valeur aléatoire.

Si un utilisateur a besoin d'initialiser son objet avec des valeurs spécifiques afin que son objet fonctionne correctement, il doit faire appel à une fonction membre *initialise*.
Si par mégarde il oublie de faire appel à cette fonction *initialise*, alors son objet ne fonctionnera pas correctement...

Le C++ offre une solution puissante à ce problème : la **notion de constructeur**.

Introduction et intérêt des constructeurs

A la déclaration d'une variable, *les valeurs par défaut* des attributs d'une classe vont dépendre de sa classe d'allocation. Soit 0 pour la classe d'allocation statique, soit une valeur aléatoire.

Si un utilisateur a besoin d'initialiser son objet avec des valeurs spécifiques afin que son objet fonctionne correctement, il doit faire appel à une fonction membre *initialise*.

Si par mégarde il oublie de faire appel à cette fonction *initialise*, alors son objet ne fonctionnera pas correctement...

Le C++ offre une solution puissante à ce problème : la **notion de constructeur**.

Introduction et intérêt des constructeurs

A la déclaration d'une variable, *les valeurs par défaut* des attributs d'une classe vont dépendre de sa classe d'allocation. Soit 0 pour la classe d'allocation statique, soit une valeur aléatoire.

Si un utilisateur a besoin d'initialiser son objet avec des valeurs spécifiques afin que son objet fonctionne correctement, il doit faire appel à une fonction membre *initialise*.
Si par mégarde il oublie de faire appel à cette fonction *initialise*, alors son objet ne fonctionnera pas correctement...

Le C++ offre une solution puissante à ce problème : la **notion de constructeur**.

Introduction et intérêt des constructeurs

A la déclaration d'une variable, *les valeurs par défaut* des attributs d'une classe vont dépendre de sa classe d'allocation. Soit 0 pour la classe d'allocation statique, soit une valeur aléatoire.

Si un utilisateur a besoin d'initialiser son objet avec des valeurs spécifiques afin que son objet fonctionne correctement, il doit faire appel à une fonction membre *initialise*.
Si par mégarde il oublie de faire appel à cette fonction *initialise*, alors son objet ne fonctionnera pas correctement...

Le C++ offre une solution puissante à ce problème : la **notion de constructeur**.

Pour initialiser les objets à leur création

- Un constructeur d'une classe porte le **même nom que sa classe** et il n'a **pas de valeur de retour** ;
- Un constructeur est une **fonction appelée automatiquement à la création d'une instance** : elle va pouvoir initialiser les membres données ;
- Un constructeur a généralement une **visibilité publique**, sinon on ne peut pas créer d'instance à l'extérieur de la classe (dans quelques rares cas on le rend privé) ;
- Il y a **plusieurs constructeurs pour une classe** : il existe toujours un *constructeur par copie* et si l'utilisateur ne spécifie aucun constructeur, alors il y aura aussi un *constructeur par défaut* (sans argument) ne faisant rien ; il est généralement recommandé de les réécrire soi-même en particulier s'il faut faire des allocations dynamiques.

Pour initialiser les objets à leur création

- Un constructeur d'une classe porte le **même nom que sa classe** et il n'a **pas de valeur de retour** ;
- Un constructeur est une **fonction appelée automatiquement à la création d'une instance** : elle va pouvoir initialiser les membres données ;
- Un constructeur a généralement une **visibilité publique**, sinon on ne peut pas créer d'instance à l'extérieur de la classe (dans quelques rares cas on le rend privé) ;
- Il y a **plusieurs constructeurs pour une classe** : il existe toujours un *constructeur par copie* et si l'utilisateur ne spécifie aucun constructeur, alors il y aura aussi un *constructeur par défaut* (sans argument) ne faisant rien ; il est généralement recommandé de les réécrire soi-même en particulier s'il faut faire des allocations dynamiques.

Pour initialiser les objets à leur création

- Un constructeur d'une classe porte le **même nom que sa classe** et il n'a **pas de valeur de retour** ;
- Un constructeur est une **fonction appelée automatiquement à la création d'une instance** : elle va pouvoir initialiser les membres données ;
- Un constructeur a généralement une **visibilité publique**, sinon on ne peut pas créer d'instance à l'extérieur de la classe (dans quelques rares cas on le rend privé) ;
- Il y a **plusieurs constructeurs pour une classe** : il existe toujours un *constructeur par copie* et si l'utilisateur ne spécifie aucun constructeur, alors il y aura aussi un *constructeur par défaut* (sans argument) ne faisant rien ; il est généralement recommandé de les réécrire soi-même en particulier s'il faut faire des allocations dynamiques.

Pour initialiser les objets à leur création

- Un constructeur d'une classe porte le **même nom que sa classe** et il n'a **pas de valeur de retour** ;
- Un constructeur est une **fonction appelée automatiquement à la création d'une instance** : elle va pouvoir initialiser les membres données ;
- Un constructeur a généralement une **visibilité publique**, sinon on ne peut pas créer d'instance à l'extérieur de la classe (dans quelques rares cas on le rend privé) ;
- Il y a **plusieurs constructeurs pour une classe** : il existe toujours un *constructeur par copie* et si l'utilisateur ne spécifie aucun constructeur, alors il y aura aussi un *constructeur par défaut* (sans argument) ne faisant rien ; il est généralement recommandé de les réécrire soi-même en particulier s'il faut faire des allocations dynamiques.

Pour initialiser les objets à leur création

- Un constructeur d'une classe porte le **même nom que sa classe** et il n'a **pas de valeur de retour** ;
- Un constructeur est une **fonction appelée automatiquement à la création d'une instance** : elle va pouvoir initialiser les membres données ;
- Un constructeur a généralement une **visibilité publique**, sinon on ne peut pas créer d'instance à l'extérieur de la classe (dans quelques rares cas on le rend privé) ;
- Il y a **plusieurs constructeurs pour une classe** : il existe toujours un *constructeur par copie* et si l'utilisateur ne spécifie aucun constructeur, alors il y aura aussi un *constructeur par défaut* (sans argument) ne faisant rien ; il est généralement recommandé de les réécrire soi-même en particulier s'il faut faire des allocations dynamiques.

Exemple de déclaration d'une classe avec ses constructeurs

```
class complex
{
  private :
    float im, re;                  // données privées (encapsulées)
  public :
    // méthodes publiques
    complex() ;                    // constructeur sans argument
    complex(float) ;               // autre constructeur
    complex(float, float) ;        // autre constructeur
    complex(complex&) ;            // constructeur par copie

    void modifieIm(float) ;        // modifieur (pour modifier un attribut privé)
    void modifieRe(float) ;        // modifieur (pour modifier un attribut privé)
    float Im();                    // accesseur (pour accéder à un attribut privé)
    float Re();                    // accesseur (pour accéder à un attribut privé)

    void affiche() ;
};
```

Remarque : on peut avoir des arguments par défaut pour les constructeurs.

Exemple de définition des méthodes de la classe

```
complex::complex(){ re = im = 0.f; }
complex::complex(float re_arg){ re = re_arg; im = 0.f; }
complex::complex(float im_arg, float re_arg){ re = re_arg; im = im_arg; }
complex::complex(complex& c){ re = c.Re(); im = c.Im(); }

void complex::modifieIm(float im_arg){ im = im_arg; }
void complex::modifieRe(float re_arg){ re = re_arg; }
float complex::Im(){ return im; }
float complex::Re(){ return re; }
void complex::affiche() { std::cout << re << " + i*" << im << std::endl ; }
```

Exemple d'appels automatiques aux constructeurs

```
complex c;                    // appel au constructeur sans argument
complex c2(1.2f);             // appel au constructeur complex(float re_arg)
complex c3(c2);               // appel au constructeur par recopie
complex c4, c5(4.2f, -8.1f); // appels à deux constructeurs différents

complex c6 = 5.4f;            // appel au constructeur complex(float re_arg) :
                              // ce constructeur permet une conversion de float
                              // vers complex!
complex c7 = c6;              // appel au constructeur par recopie

complex c8();                 // MAUVAIS : on vient de déclarer une fonction C++ sans
                              // argument qui retourne un complex
```

Pour libérer la mémoire des données membres à la destruction d'un objet

- **LE** destructeur d'une classe porte le **même nom que sa classe précédé de** ~, il n'a **pas de valeur de retour** et il n'a **pas d'argument** ;
- Un destructeur est **LA fonction appelée automatiquement** à la destruction d'une instance ;
- Un destructeur a généralement une **visibilité publique**, sinon on ne peut pas supprimer d'instance ;
- **Il existe toujours un destructeur par défaut** ;
- Il est indispensable de définir le destructeur dès qu'il y a de la libération dynamique de mémoire à effectuer ;
- L'opérateur `delete` sur une adresse de type `CType*` fait un appel explicite au destructeur de `CType`.

Pour libérer la mémoire des données membres à la destruction d'un objet

- **LE** destructeur d'une classe porte le **même nom que sa classe précédé de** ~, il n'a **pas de valeur de retour** et il n'a **pas d'argument** ;
- Un destructeur est **LA fonction appelée automatiquement** à la destruction d'une instance ;
- Un destructeur a généralement une **visibilité publique**, sinon on ne peut pas supprimer d'instance ;
- **Il existe toujours un destructeur par défaut** ;
- Il est indispensable de définir le destructeur dès qu'il y a de la libération dynamique de mémoire à effectuer ;
- L'opérateur `delete` sur une adresse de type `CType*` fait un appel explicite au destructeur de `CType`.

Pour libérer la mémoire des données membres à la destruction d'un objet

- **LE** destructeur d'une classe porte le **même nom que sa classe précédé de** ~, il n'a **pas de valeur de retour** et il n'a **pas d'argument** ;
- Un destructeur est **LA fonction appelée automatiquement** à la destruction d'une instance ;
- Un destructeur a généralement une **visibilité publique**, sinon on ne peut pas supprimer d'instance ;
- **Il existe toujours un destructeur par défaut** ;
- Il est indispensable de définir le destructeur dès qu'il y a de la libération dynamique de mémoire à effectuer ;
- L'opérateur `delete` sur une adresse de type `CType*` fait un appel explicite au destructeur de `CType`.

Pour libérer la mémoire des données membres à la destruction d'un objet

- **LE** destructeur d'une classe porte le **même nom que sa classe précédé de** ~, il n'a **pas de valeur de retour** et il n'a **pas d'argument** ;
- Un destructeur est **LA fonction appelée automatiquement** à la destruction d'une instance ;
- Un destructeur a généralement une **visibilité publique**, sinon on ne peut pas supprimer d'instance ;
- **Il existe toujours un destructeur par défaut** ;
- Il est indispensable de définir le destructeur dès qu'il y a de la libération dynamique de mémoire à effectuer ;
- L'opérateur `delete` sur une adresse de type `CType*` fait un appel explicite au destructeur de `CType`.

Pour libérer la mémoire des données membres à la destruction d'un objet

- **LE** destructeur d'une classe porte le **même nom que sa classe précédé de** ~, il n'a **pas de valeur de retour** et il n'a **pas d'argument** ;
- Un destructeur est **LA fonction appelée automatiquement** à la destruction d'une instance ;
- Un destructeur a généralement une **visibilité publique**, sinon on ne peut pas supprimer d'instance ;
- **Il existe toujours un destructeur par défaut** ;
- Il est indispensable de définir le destructeur dès qu'il y a de la libération dynamique de mémoire à effectuer ;
- L'opérateur `delete` sur une adresse de type `CType*` fait un appel explicite au destructeur de `CType`.

Pour libérer la mémoire des données membres à la destruction d'un objet

- **LE** destructeur d'une classe porte le **même nom que sa classe précédé de** ~, il n'a **pas de valeur de retour** et il n'a **pas d'argument** ;
- Un destructeur est **LA fonction appelée automatiquement** à la destruction d'une instance ;
- Un destructeur a généralement une **visibilité publique**, sinon on ne peut pas supprimer d'instance ;
- **Il existe toujours un destructeur par défaut** ;
- Il est indispensable de définir le destructeur dès qu'il y a de la libération dynamique de mémoire à effectuer ;
- L'opérateur **delete** sur une adresse de type CType* fait un appel explicite au destructeur de CType.

Quand la destruction d'un objet a-t-elle lieu ?

- **Objet de la classe dynamique** (allocation avec l'opérateur **new**) : la destruction a lieu au moment de l'utilisation de l'opérateur **delete** sur l'adresse retournée par **new**.
- **Objet de la classe automatique** (variable locale usuelle) : le destructeur est appelé automatiquement lorsqu'on sort du bloc (bloc imbriqué ou bloc de fonction) dans lequel l'objet a été instancié.
- **Objet de la classe statique** (variables locales statiques ou variables globales) : le destructeur est appelé automatiquement à la fin du programme.

Quand la destruction d'un objet a-t-elle lieu ?

- **Objet de la classe dynamique** (allocation avec l'opérateur **new**) : la destruction a lieu au moment de l'utilisation de l'opérateur **delete** sur l'adresse retournée par **new**.
- **Objet de la classe automatique** (variable locale usuelle) : le destructeur est appelé automatiquement lorsqu'on sort du bloc (bloc imbriqué ou bloc de fonction) dans lequel l'objet a été instancié.
- **Objet de la classe statique** (variables locales statiques ou variables globales) : le destructeur est appelé automatiquement à la fin du programme.

Quand la destruction d'un objet a-t-elle lieu ?

- **Objet de la classe dynamique** (allocation avec l'opérateur **new**) : la destruction a lieu au moment de l'utilisation de l'opérateur **delete** sur l'adresse retournée par **new**.
- **Objet de la classe automatique** (variable locale usuelle) : le destructeur est appelé automatiquement lorsqu'on sort du bloc (bloc imbriqué ou bloc de fonction) dans lequel l'objet a été instancié.
- **Objet de la classe statique** (variables locales statiques ou variables globales) : le destructeur est appelé automatiquement à la fin du programme.

Exemple d'une classe avec un destructeur

```
class tableau
{
  private :
    int m_nbe ;
    double * m_valeurs ;
  public:
    tableau(int nb_case=10) ; // constructeur
    ~tableau() ;              // destructeur

    void affiche() ;
};

tableau::tableau(int nb_case){
   m_nbe = nb_case ;
   m_valeurs = new double[m_nbe] ; // on ne traite pas le cas où new plante avec une exception
}
tableau::~tableau() { delete m_valeurs ; }
void tableau::affiche(){
   for(int i=0; i<m_nbe; ++i) std::cout << m_valeurs[i] << " ";
   std::cout << std::endl;
}
```

Comment partager une information commune entre tous les objets d'une même classe ?

Dans tous les exemples précédents de classe, *chaque objet est propriétaire de ses propres copies des attributs de la classe*.

Pour **partager une donnée membre entre tous les objets d'une classe**, il faut précéder sa déclaration du mot clef **static**.

Attention : l'initialisation d'un membre statique doit être réalisée UNE SEULE FOIS, elle ne peut donc être faite par le constructeur de la classe. On peut par exemple réaliser cette initialisation au début du fichier *maclasse.cpp* (cf. comp. logi.).
Un membre *statique et constant* peut lui être directement initialisé dans la déclaration de la classe.

Comment partager une information commune entre tous les objets d'une même classe ?

Dans tous les exemples précédents de classe, *chaque objet est propriétaire de ses propres copies des attributs de la classe*.

Pour **partager une donnée membre entre tous les objets d'une classe**, il faut précéder sa déclaration du mot clef **static**.

Attention : l'initialisation d'un membre statique doit être réalisée UNE SEULE FOIS, elle ne peut donc être faite par le constructeur de la classe. On peut par exemple réaliser cette initialisation au début du fichier *maclasse.cpp* (cf. comp. logi.).
Un membre *statique et constant* peut lui être directement initialisé dans la déclaration de la classe.

Comment partager une information commune entre tous les objets d'une même classe ?

Dans tous les exemples précédents de classe, *chaque objet est propriétaire de ses propres copies des attributs de la classe*.

Pour **partager une donnée membre entre tous les objets d'une classe**, il faut précéder sa déclaration du mot clef **static**.

Attention : l'initialisation d'un membre statique doit être réalisée UNE SEULE FOIS, elle ne peut donc être faite par le constructeur de la classe. On peut par exemple réaliser cette initialisation au début du fichier *maclasse.cpp* (cf. comp. logi.).

Un membre *statique et constant* peut lui être directement initialisé dans la déclaration de la classe.

Comment partager une information commune entre tous les objets d'une même classe ?

Dans tous les exemples précédents de classe, *chaque objet est propriétaire de ses propres copies des attributs de la classe*.

Pour **partager une donnée membre entre tous les objets d'une classe**, il faut précéder sa déclaration du mot clef **static**.

Attention : l'initialisation d'un membre statique doit être réalisée UNE SEULE FOIS, elle ne peut donc être faite par le constructeur de la classe. On peut par exemple réaliser cette initialisation au début du fichier *maclasse.cpp* (cf. comp. logi.).
Un membre *statique et constant* peut lui être directement initialisé dans la déclaration de la classe.

Exemple d'un compteur des instances d'une classe

```
class compte_obj
{
  private :
    static int m_cpt ; // compteur d'objets crées
  public:
    compte_obj() ;     // constructeur
    ~compte_obj() ;    // destructeur
    void affiche() ;
};

int compte_obj::m_cpt = 0; // initialisation globale du compteur
compte_obj::compte_obj() { m_cpt++; }
compte_obj::~compte_obj() { m_cpt--; }
void compte_obj::affiche() {
   std::cout << " Il y a actuellement " << m_cpt << " instances de la classe." << std::endl;
}
```

Plan

La classe comme composant logiciel (ou module)

Découpler l'implantation d'une classe de son utilisation : on l'utilise dans le fichier source que l'on désire. Les étapes :

- **isoler les déclarations de la classe** dans un fichier en-tête (.h ou .hpp), par exemple *maclasse.h* ;
- **fabriquer un module objet (.o)** contenant les *définitions des méthodes* et les *initialisations des membres statiques* de la classe : pour cela il faut compiler un fichier .cpp (*maclasse.cpp*) contenant les définitions et initialisations ;
- `#include "maclasse.h"` au début du fichier *maclasse.cpp* et dans tous les fichiers .h ou .cpp qui font référence à la classe déclarée dans *maclasse.h*.

La classe comme composant logiciel (ou module)

Découpler l'implantation d'une classe de son utilisation : on l'utilise dans le fichier source que l'on désire. Les étapes :

- **isoler les déclarations de la classe** dans un fichier en-tête (.h ou .hpp), par exemple *maclasse.h* ;
- **fabriquer un module objet (.o)** contenant les *définitions des méthodes* et les *initialisations des membres statiques* de la classe : pour cela il faut compiler un fichier .cpp (*maclasse.cpp*) contenant les définitions et initialisations ;
- `#include "maclasse.h"` au début du fichier *maclasse.cpp* et dans tous les fichiers .h ou .cpp qui font référence à la classe déclarée dans *maclasse.h*.

La classe comme composant logiciel (ou module)

Découpler l'implantation d'une classe de son utilisation : on l'utilise dans le fichier source que l'on désire. Les étapes :

- **isoler les déclarations de la classe** dans un fichier en-tête (.h ou .hpp), par exemple *maclasse.h* ;
- **fabriquer un module objet (.o)** contenant les *définitions des méthodes* et les *initialisations des membres statiques* de la classe : pour cela il faut compiler un fichier .cpp (*maclasse.cpp*) contenant les définitions et initialisations ;
- `#include "maclasse.h"` au début du fichier *maclasse.cpp* et dans tous les fichiers .h ou .cpp qui font référence à la classe déclarée dans *maclasse.h*.

La classe comme composant logiciel (ou module)

Découpler l'implantation d'une classe de son utilisation : on l'utilise dans le fichier source que l'on désire. Les étapes :

- **isoler les déclarations de la classe** dans un fichier en-tête (.h ou .hpp), par exemple *maclasse.h* ;
- **fabriquer un module objet (.o)** contenant les *définitions des méthodes* et les *initialisations des membres statiques* de la classe : pour cela il faut compiler un fichier .cpp (*maclasse.cpp*) contenant les définitions et initialisations ;
- **#include** "maclasse.h" au début du fichier *maclasse.cpp* et dans tous les fichiers .h ou .cpp qui font référence à la classe déclarée dans *maclasse.h*.

La classe comme composant logiciel (ou module)

Quelques précisions :

- ça sera uniquement pendant l'*édition de liens* que le compilateur vérifiera si les méthodes appelées dans le programme sont bien définies (i.e. implantées) ;
- la majorité des EDIs possèdent la notion de *projet* contenant plusieurs fichiers : les dépendances entre les différents composants logiciels sont déterminées automatiquement via les includes, et la génération des fichiers objets (.o) de chaque composant est "séparée" ;
  après chaque modification du code, seules les compilations nécessaires sont effectuées, c'est ce qu'on appelle la **compilation séparée**.

La classe comme composant logiciel (ou module)

Quelques précisions :

- ça sera uniquement pendant l'*édition de liens* que le compilateur vérifiera si les méthodes appelées dans le programme sont bien définies (i.e. implantées) ;
- la majorité des EDIs possèdent la notion de *projet* contenant plusieurs fichiers : les dépendances entre les différents composants logiciels sont déterminées automatiquement via les includes, et la génération des fichiers objets (.o) de chaque composant est "séparée" ;

après chaque modification du code, seules les compilations nécessaires sont effectuées, c'est ce qu'on appelle la **compilation séparée**.

La classe comme composant logiciel (ou module)

Quelques précisions :

- ça sera uniquement pendant l'*édition de liens* que le compilateur vérifiera si les méthodes appelées dans le programme sont bien définies (i.e. implantées) ;
- la majorité des EDIs possèdent la notion de *projet* contenant plusieurs fichiers : les dépendances entre les différents composants logiciels sont déterminées automatiquement via les includes, et la génération des fichiers objets (.o) de chaque composant est "séparée" ;
  après chaque modification du code, seules les compilations nécessaires sont effectuées, c'est ce qu'on appelle la **compilation séparée**.

Utilisation d'une technique de compilation conditionnelle

```
// fichier maclasse.h

#ifndef _MACLASSE_H_
#define _MACLASSE_H_

... // ici le code de déclaration de ma classe

#endif
```

Alternative aux include guards : #pragma once

Introduction

Comment compiler (e.g. sous Linux) si on a 3 fichiers : main.cpp, maclasse.h et maclasse.cpp, et si main.cpp et maclasse.cpp incluent maclasse.h ?

- g++ -Wall -pedantic -c main.cpp *compilation*
- g++ -Wall -pedantic -c maclasse.cpp *compilation*
- g++ main.o maclasse.o -o monExec *édition de liens*

Introduction

Comment compiler (e.g. sous Linux) si on a 3 fichiers : main.cpp, maclasse.h et maclasse.cpp, et si main.cpp et maclasse.cpp incluent maclasse.h ?

- g++ -Wall -pedantic -c main.cpp *compilation*
- g++ -Wall -pedantic -c maclasse.cpp *compilation*
- g++ main.o maclasse.o -o monExec *édition de liens*

Introduction

Comment compiler (e.g. sous Linux) si on a 3 fichiers : main.cpp, maclasse.h et maclasse.cpp, et si main.cpp et maclasse.cpp incluent maclasse.h ?

- g++ -Wall -pedantic -c main.cpp *compilation*
- g++ -Wall -pedantic -c maclasse.cpp *compilation*
- g++ main.o maclasse.o -o monExec *édition de liens*

Introduction

Comment compiler (e.g. sous Linux) si on a 3 fichiers : main.cpp, maclasse.h et maclasse.cpp, et si main.cpp et maclasse.cpp incluent maclasse.h ?

- g++ -Wall -pedantic -c main.cpp *compilation*
- g++ -Wall -pedantic -c maclasse.cpp *compilation*
- g++ main.o maclasse.o -o monExec *édition de liens*

Introduction

Comment automatiser la compilation sous Linux ?

Fichier *Makefile*.

cible : liste des dépendances
recette pour construire cible (précédée d'une tabulation ! !)

Ensuite il suffit de taper `make` ou `make` *monExec*.

Introduction

Comment automatiser la compilation sous Linux ?
Fichier *Makefile*.

cible : liste des dépendances
recette pour construire cible (précédée d'une tabulation ! !)

Ensuite il suffit de taper `make` ou `make` *monExec*.

Exemple

Fichier *Makefile* :

```
monExec : main.o maclasse.o
        g++ main.o maclasse.o -o monExec

maclasse.o : maclasse.cpp maclasse.h
        g++ -Wall -pedantic -c maclasse.cpp

main.o : main.cpp maclasse.h
        g++ -Wall -pedantic -c main.cpp

clean :
        rm *.o *~
```

Exemple

Fichier *Makefile* :

```
monExec : main.o maclasse.o
	g++ main.o maclasse.o -o monExec

maclasse.o : maclasse.cpp maclasse.h
	g++ -Wall -pedantic -c maclasse.cpp

main.o : main.cpp maclasse.h
	g++ -Wall -pedantic -c main.cpp

clean :
	rm *.o *~
```