Programmation Orientée Objet en C++

Emanuel Aldea <emanuel.aldea@u-psud.fr>
http://hebergement.u-psud.fr/emi

M1-E3A

Deuxième chapitre

Les objets

L'approche objet

La notion de classe

- un type de données parmi d'autres

L'approche objet

La notion de classe

- un type de données parmi d'autres
- mais definition de nouveaux types utilisables comme les types prédéfinis !

L'approche objet

La notion de classe

- un type de données parmi d'autres
- mais definition de nouveaux types utilisables comme les types prédéfinis !
- une instance d'une classe est appelée **objet**

L'approche objet

La notion de classe

- un type de données parmi d'autres
- mais definition de nouveaux types utilisables comme les types prédéfinis !
- une instance d'une classe est appelée **objet**
- déclaration : syntaxe modifiée d'une structure
  - **droits** : public, protected, private
  - **méthodes** : les fonctions de l'objet

L'approche objet

La notion de classe

- un type de données parmi d'autres
- mais definition de nouveaux types utilisables comme les types prédéfinis !
- une instance d'une classe est appelée **objet**
- déclaration : syntaxe modifiée d'une structure
  - **droits** : public, protected, private
  - **méthodes** : les fonctions de l'objet
- méthodes spéciales :
  - **constructeurs** : appelés à la création, sans type de retour, entre zéro et plusieurs paramètres : `type_classe (...)`
  - **destructeur** : unique, appelé à la destruction, sans paramètres, sans retour : `~type_classe ()`

Quelle est la vie d'un objet ?

Création

- allocation de mémoire : `sizeof (type_classe)` octets

Quelle est la vie d'un objet ?

Création

- allocation de mémoire : `sizeof (type_classe)` octets
- appel d'un des constructeurs

Quelle est la vie d'un objet ?

Création

- allocation de mémoire : `sizeof (type_classe)` octets
- appel d'un des constructeurs

Utilisation

Quelle est la vie d'un objet ?

Création

- allocation de mémoire : `sizeof (type_classe)` octets
- appel d'un des constructeurs

Utilisation

- lecture/écriture des attributs accessibles

Quelle est la vie d'un objet ?

Création

- allocation de mémoire : `sizeof (type_classe)` octets
- appel d'un des constructeurs

Utilisation

- lecture/écriture des attributs accessibles
- appel des méthodes accessibles

Quelle est la vie d'un objet ?

Création

- allocation de mémoire : `sizeof (type_classe)` octets
- appel d'un des constructeurs

Utilisation

- lecture/écriture des attributs accessibles
- appel des méthodes accessibles

Destruction

Quelle est la vie d'un objet ?

Création

- allocation de mémoire : `sizeof (type_classe)` octets
- appel d'un des constructeurs

Utilisation

- lecture/écriture des attributs accessibles
- appel des méthodes accessibles

Destruction

- appel de l'unique destructeur

Quelle est la vie d'un objet ?

Création

- allocation de mémoire : `sizeof (type_classe)` octets
- appel d'un des constructeurs

Utilisation

- lecture/écriture des attributs accessibles
- appel des méthodes accessibles

Destruction

- appel de l'unique destructeur
- libération de la mémoire : `sizeof (type_classe)` octets

L'approche objet

- en C++ toute **donnée** est un **objet** et inversement

L'approche objet

- en C++ toute **donnée** est un **objet** et inversement
- au début de la vie : **construction** $\Rightarrow$ appel d'un constructeur (même un qui ne fait rien)

L'approche objet

- en C++ toute **donnée** est un **objet** et inversement
- au début de la vie : **construction** $\Rightarrow$ appel d'un constructeur (même un qui ne fait rien)
- tout objet possède deux constructeurs **par défaut** :
  - un constructeur sans paramètres
  - un constructeur de copie

L'approche objet

- en C++ toute **donnée** est un **objet** et inversement
- au début de la vie : **construction** $\Rightarrow$ appel d'un constructeur (même un qui ne fait rien)
- tout objet possède deux constructeurs **par défaut** :
  - un constructeur sans paramètres
  - un constructeur de copie
- a la fin de la vie : **destruction** $\Rightarrow$ appel du destructeur (même un qui ne fait rien)

L'approche objet

- en C++ toute **donnée** est un **objet** et inversement
- au début de la vie : **construction** $\Rightarrow$ appel d'un constructeur (même un qui ne fait rien)
- tout objet possède deux constructeurs **par défaut** :
  - un constructeur sans paramètres
  - un constructeur de copie
- a la fin de la vie : **destruction** $\Rightarrow$ appel du destructeur (même un qui ne fait rien)
- tous les objets (y compris les objets natifs) possèdent un destructeur par défaut qui ne fait rien

Les constructeurs

- remplace une méthode de type `init(...)`

Les constructeurs

- remplace une méthode de type `init(...)`
- les constructeurs avec zéro ou plusieurs paramètres : `CObj(...)`

Les constructeurs

- remplace une méthode de type `init(...)`
- les constructeurs avec zéro ou plusieurs paramètres : `CObj(...)`
- le constructeur de copie : `CObj(const CObj& obj)`
  - on crée une instance qui est une copie d'un autre objet
  - appelé lorsqu'un objet est envoyé **par valeur** à une fonction ; la fonction crée une **copie** temporaire
  - appelé aussi lorsqu'une fonction retourne un objet
  - par défaut, il appelle récursivement les constructeurs de copie des membres de la classe
  - nécessaire lorsque la classe contient des ressources dynamiques !
  - pas d'erreur si le compilateur ne le trouve pas ! (pour quoi ?)

Exemple : le type natif `int`

```
void main()
{
  int i;  // appel de int ()
  int j(12); // int j=12, appel de int ( const int & val)
  int k(j);  // int k=j, appel de int ( const int & val)
  int m=k+j; // int m(k+j), appel de int ( const int & val)
  int* pi1=new int (k*j); // appel de int ( const int & val)
  delete pi1; // appel de ~ int ()
}
// quatre appels de ~ int ()
```

- le constructeur `int()` ne fait rien

Exemple : le type natif `int`

```
void main()
{
  int i;  // appel de int ()
  int j(12); // int j=12, appel de int ( const int & val)
  int k(j);  // int k=j, appel de int ( const int & val)
  int m=k+j; // int m(k+j), appel de int ( const int & val)
  int* pi1=new int (k*j); // appel de int ( const int & val)
  delete pi1; // appel de ~ int ()
}
// quatre appels de ~ int ()
```

- le constructeur `int()` ne fait rien
- le constructeur `int(const int & val)` copie `val` dans l'objet

Exemple : le type natif `int`

```
void main()
{
  int i;  // appel de int ()
  int j(12); // int j=12, appel de int ( const int & val)
  int k(j);  // int k=j, appel de int ( const int & val)
  int m=k+j; // int m(k+j), appel de int ( const int & val)
  int* pi1=new int (k*j); // appel de int ( const int & val)
  delete pi1; // appel de ~ int ()
}
// quatre appels de ~ int ()
```

- le constructeur `int()` ne fait rien
- le constructeur `int(const int & val)` copie `val` dans l'objet
- le destructeur `int()` ne fait rien

Les méthodes (fonctions membres)

Une méthode est une fonction qui travaille pour l'objet (l'instance) pour lequel on l'appelle

- son code a un **accès direct** (par nom) à tous les attributs de l'objet et les méthodes de la classe

Les méthodes (fonctions membres)

Une méthode est une fonction qui travaille pour l'objet (l'instance) pour lequel on l'appelle

- son code a un **accès direct** (par nom) à tous les attributs de l'objet et les méthodes de la classe
- elle a toujours un premier paramètre caché appelé **this** qui représente l'adresse de l'objet pour lequel on l'appelle - donc si l'on appelle de l'extérieur il faut préciser l'objet

Les méthodes (fonctions membres)

Une méthode est une fonction qui travaille pour l'objet (l'instance) pour lequel on l'appelle

- son code a un **accès direct** (par nom) à tous les attributs de l'objet et les méthodes de la classe
- elle a toujours un premier paramètre caché appelé **this** qui représente l'adresse de l'objet pour lequel on l'appelle - donc si l'on appelle de l'extérieur il faut préciser l'objet
- appel de l'intérieur d'une autre méthode (accès direct) :

```
NomMethode (parametres)
```

Les méthodes (fonctions membres)

Une méthode est une fonction qui travaille pour l'objet (l'instance) pour lequel on l'appelle

- son code a un **accès direct** (par nom) à tous les attributs de l'objet et les méthodes de la classe
- elle a toujours un premier paramètre caché appelé **this** qui représente l'adresse de l'objet pour lequel on l'appelle - donc si l'on appelle de l'extérieur il faut préciser l'objet
- appel de l'intérieur d'une autre méthode (accès direct) :

```
NomMethode (parametres)
```

- appel de l'extérieur de la classe (accès indirect) :

```
objet.NomMethode (parametres)  // ou
ptrobjet->NomMethode (parametres) // idem a : (*ptrobjet).
```

Définitions des méthodes

Définition inline

- déclaration suivie de définition à l'intérieur de la déclaration de classe

Définition outline

Définitions des méthodes

Définition inline

- déclaration suivie de définition à l'intérieur de la déclaration de classe
- typiquement dans un fichier header

```
class CCercle{
  float rayon;
  //...
  public:
    void SetRay ( float _rayon) {rayon=_rayon;}
  };
```

Définition outline

Définitions des méthodes

Définition inline

- déclaration suivie de définition à l'intérieur de la déclaration de classe
- typiquement dans un fichier header

```
class CCercle{
  float rayon;
  //...
  public:
    void SetRay ( float _rayon) {rayon=_rayon;}
  };
```

Définition outline

- déclaration à l'intérieur (fichier header), et définition à l'extérieur de la déclaration de classe (fisier .cpp)

```
class CCercle{ //declaration
  float rayon;
  //...
  public:
  void SetRay ( float _rayon);
};
```

fichier header

```
//definition

void CCercle::SetRay (float _rayon)
{
  rayon=_rayon;
}
```

fichier source

Exemple : CFract

Modéliser une fraction entière :

```
class CFract{
  //private par defaut
  public:
    int a, b;
    void Afficher(){
      printf ("Fraction : (%d/%d) \ n",a,b);
    }
};
```

CFract a déjà deux constructeurs par défaut et un destructeur par défaut. Sans rien demander, le compilateur génère le code suivant (pas inclus dans les sources) :

```
//constructeur sans parametres
CFract () {}

// constructeur de copie
CFract ( const CFract & f) { a=f.a; b=f.b }

//destructeur par defaut
~CFract () {}

//surcharge de l'operateur =
CFract & operator = ( const CFract & f)
{ a=f.a; b=f.b; return * this ; }
```

Toute déclaration explicite d'un constructeur (sauf celui de copie) désactive la génération automatique du constructeur par défaut sans paramètres.

Exemple : CFract

Avec les constructeurs / destructeur par défaut :

- Déclarez la classe puis des variables locales `f1` et `f2` de type `CFract` .
- Faire afficher `f1` et `f2` .
- Modifiez le programme, rajoutez une variable `f3` et utilisez le constructeur de copie pour faire que `f3` ait le même contenu que `f1` .
- Même exercice avec une instance dynamique de la classe `CFract` pointée par `pf4` . Faire que cette variable ait le contenu de `f2`.

Exemple : CFract

Comment initialiser les attributs ?

- De l'extérieur : il faut qu'ils soient accessibles, et c'est contraire à l'approche objet

Exemple : CFract

Comment initialiser les attributs ?

- De l'extérieur : il faut qu'ils soient accessibles, et c'est contraire à l'approche objet
- De l'intérieur : par une méthode publique Init ou par le constructeur de classe :

```
class CFract
{
  int a, b;
  // ...
  public:
    CFract ( int _a, int _b): a(_a), b(_b) {}
};
```

Exemple : CFract

Comment initialiser les attributs ?

- De l'extérieur : il faut qu'ils soient accessibles, et c'est contraire à l'approche objet
- De l'intérieur : par une méthode publique Init ou par le constructeur de classe :

```
class CFract
{
  int a, b;
  // ...
  public:
    CFract ( int _a, int _b): a(_a), b(_b) {}
};
```

- Modifier le programme en rajoutant le constructeur explicite. Est-ce que les deux constructeurs par défaut restent encore valables ?

Exemple : CFract

Comment initialiser les attributs ?

- De l'extérieur : il faut qu'ils soient accessibles, et c'est contraire à l'approche objet
- De l'intérieur : par une méthode publique Init ou par le constructeur de classe :

```
class CFract
{
  int a, b;
  // ...
  public:
    CFract ( int _a, int _b): a(_a), b(_b) {}
};
```

- Modifier le programme en rajoutant le constructeur explicite. Est-ce que les deux constructeurs par défaut restent encore valables ?
- Modifier le programme de test ...

Exemple : CFract

Comment initialiser les attributs ?

- De l'extérieur : il faut qu'ils soient accessibles, et c'est contraire à l'approche objet
- De l'intérieur : par une méthode publique Init ou par le constructeur de classe :

```
class CFract
{
  int a, b;
  // ...
  public:
    CFract ( int _a, int _b): a(_a), b(_b) {}
};
```

- Modifier le programme en rajoutant le constructeur explicite. Est-ce que les deux constructeurs par défaut restent encore valables ?
- Modifier le programme de test ...
- Rajouter un constructeur copie et un destructeur et affichez des messages d'informations dans chacun.

Exemple : CFract

Comment initialiser les attributs ?

- De l'extérieur : il faut qu'ils soient accessibles, et c'est contraire à l'approche objet
- De l'intérieur : par une méthode publique Init ou par le constructeur de classe :

```
class CFract
{
  int a, b;
  // ...
  public:
    CFract ( int _a, int _b): a(_a), b(_b) {}
};
```

- Modifier le programme en rajoutant le constructeur explicite. Est-ce que les deux constructeurs par défaut restent encore valables ?
- Modifier le programme de test ...
- Rajouter un constructeur copie et un destructeur et affichez des messages d'informations dans chacun.
- Déclarez des tableaux locaux de 6 `CFract` et dynamiques de 4 `CFract` .

Exemple : CFract

Comment initialiser les attributs ?

- De l'extérieur : il faut qu'ils soient accessibles, et c'est contraire à l'approche objet
- De l'intérieur : par une méthode publique Init ou par le constructeur de classe :

```
class CFract
{
  int a, b;
  // ...
  public:
    CFract ( int _a, int _b): a(_a), b(_b) {}
};
```

- Modifier le programme en rajoutant le constructeur explicite. Est-ce que les deux constructeurs par défaut restent encore valables ?
- Modifier le programme de test ...
- Rajouter un constructeur copie et un destructeur et affichez des messages d'informations dans chacun.
- Déclarez des tableaux locaux de 6 `CFract` et dynamiques de 4 `CFract` .
- Quelle est l'erreur ? Comment retrouver un constructeur sans paramètres ?

Exemple : CFract

Comment initialiser les attributs ?

- De l'extérieur : il faut qu'ils soient accessibles, et c'est contraire à l'approche objet
- De l'intérieur : par une méthode publique Init ou par le constructeur de classe :

```
class CFract
{
  int a, b;
  // ...
  public:
    CFract ( int _a, int _b): a(_a), b(_b) {}
};
```

- Modifier le programme en rajoutant le constructeur explicite. Est-ce que les deux constructeurs par défaut restent encore valables ?
- Modifier le programme de test ...
- Rajouter un constructeur copie et un destructeur et affichez des messages d'informations dans chacun.
- Déclarez des tableaux locaux de 6 `CFract` et dynamiques de 4 `CFract` .
- Quelle est l'erreur ? Comment retrouver un constructeur sans paramètres ?
- Surveiller la vie de tous les objets `CFract` (comment ?) .

Exemple : CFract

Comment initialiser les attributs ?

- De l'extérieur : il faut qu'ils soient accessibles, et c'est contraire à l'approche objet
- De l'intérieur : par une méthode publique Init ou par le constructeur de classe :

```
class CFract
{
  int a, b;
  // ...
  public:
    CFract ( int _a, int _b): a(_a), b(_b) {}
};
```

- Modifier le programme en rajoutant le constructeur explicite. Est-ce que les deux constructeurs par défaut restent encore valables ?
- Modifier le programme de test ...
- Rajouter un constructeur copie et un destructeur et affichez des messages d'informations dans chacun.
- Déclarez des tableaux locaux de 6 `CFract` et dynamiques de 4 `CFract` .
- Quelle est l'erreur ? Comment retrouver un constructeur sans paramètres ?
- Surveiller la vie de tous les objets `CFract` (comment ?) .
- Rajouter une méthode `MultTo` pour multiplier une fraction par une autre.

Exemple : CFract

Comment initialiser les attributs ?

- De l'extérieur : il faut qu'ils soient accessibles, et c'est contraire à l'approche objet
- De l'intérieur : par une méthode publique Init ou par le constructeur de classe :

```
class CFract
{
  int a, b;
  // ...
  public:
    CFract ( int _a, int _b): a(_a), b(_b) {}
};
```

- Modifier le programme en rajoutant le constructeur explicite. Est-ce que les deux constructeurs par défaut restent encore valables ?
- Modifier le programme de test ...
- Rajouter un constructeur copie et un destructeur et affichez des messages d'informations dans chacun.
- Déclarez des tableaux locaux de 6 `CFract` et dynamiques de 4 `CFract` .
- Quelle est l'erreur ? Comment retrouver un constructeur sans paramètres ?
- Surveiller la vie de tous les objets `CFract` (comment ?) .
- Rajouter une méthode `MultTo` pour multiplier une fraction par une autre.
- Rajouter une méthode `AddTo` pour additionner une fraction à une autre.

Exemple : CFract

Comment initialiser les attributs ?

- De l'extérieur : il faut qu'ils soient accessibles, et c'est contraire à l'approche objet
- De l'intérieur : par une méthode publique Init ou par le constructeur de classe :

```
class CFract
{
  int a, b;
  // ...
  public:
    CFract ( int _a, int _b): a(_a), b(_b) {}
};
```

- Modifier le programme en rajoutant le constructeur explicite. Est-ce que les deux constructeurs par défaut restent encore valables ?
- Modifier le programme de test ...
- Rajouter un constructeur copie et un destructeur et affichez des messages d'informations dans chacun.
- Déclarez des tableaux locaux de 6 `CFract` et dynamiques de 4 `CFract` .
- Quelle est l'erreur ? Comment retrouver un constructeur sans paramètres ?
- Surveiller la vie de tous les objets `CFract` (comment ?) .
- Rajouter une méthode `MultTo` pour multiplier une fraction par une autre.
- Rajouter une méthode `AddTo` pour additionner une fraction à une autre.
- Rajouter une méthode `Norm` qui normalise la valeur de la fraction. Utilisez-la dans les endroits nécessaires.

Exemple : CFract

Comment initialiser les attributs ?

- De l'extérieur : il faut qu'ils soient accessibles, et c'est contraire à l'approche objet
- De l'intérieur : par une méthode publique Init ou par le constructeur de classe :

```
class CFract
{
  int a, b;
  // ...
  public:
    CFract ( int _a, int _b): a(_a), b(_b) {}
};
```

- Modifier le programme en rajoutant le constructeur explicite. Est-ce que les deux constructeurs par défaut restent encore valables ?
- Modifier le programme de test ...
- Rajouter un constructeur copie et un destructeur et affichez des messages d'informations dans chacun.
- Déclarez des tableaux locaux de 6 `CFract` et dynamiques de 4 `CFract` .
- Quelle est l'erreur ? Comment retrouver un constructeur sans paramètres ?
- Surveiller la vie de tous les objets `CFract` (comment ?) .
- Rajouter une méthode `MultTo` pour multiplier une fraction par une autre.
- Rajouter une méthode `AddTo` pour additionner une fraction à une autre.
- Rajouter une méthode `Norm` qui normalise la valeur de la fraction. Utilisez-la dans les endroits nécessaires.
- Rendre explicites les différentes protections à la modification en utilisant `const`

Exemple : CFract

Comment initialiser les attributs ?

- De l'extérieur : il faut qu'ils soient accessibles, et c'est contraire à l'approche objet
- De l'intérieur : par une méthode publique Init ou par le constructeur de classe :

```
class CFract
{
  int a, b;
  // ...
  public:
    CFract ( int _a, int _b): a(_a), b(_b) {}
};
```

- Modifier le programme en rajoutant le constructeur explicite. Est-ce que les deux constructeurs par défaut restent encore valables ?
- Modifier le programme de test ...
- Rajouter un constructeur copie et un destructeur et affichez des messages d'informations dans chacun.
- Déclarez des tableaux locaux de 6 `CFract` et dynamiques de 4 `CFract` .
- Quelle est l'erreur ? Comment retrouver un constructeur sans paramètres ?
- Surveiller la vie de tous les objets `CFract` (comment ?) .
- Rajouter une méthode `MultTo` pour multiplier une fraction par une autre.
- Rajouter une méthode `AddTo` pour additionner une fraction à une autre.
- Rajouter une méthode `Norm` qui normalise la valeur de la fraction. Utilisez-la dans les endroits nécessaires.
- Rendre explicites les différentes protections à la modification en utilisant `const`
- Réfléchir sur l'accessibilité à donner aux méthodes de `CFract`.

Encapsulation

- Notion essentielle dans la programmation objet

Encapsulation

- Notion essentielle dans la programmation objet
- Suppose une séparation entre l'**interface publique** (accessible) d'une classe et son **fonctionnement interne** qui ne doit pas être ni perturbé ni forcement connu par l'extérieur.

Encapsulation

- Notion essentielle dans la programmation objet
- Suppose une séparation entre l'**interface publique** (accessible) d'une classe et son **fonctionnement interne** qui ne doit pas être ni perturbé ni forcement connu par l'extérieur.
- Outils pour la mise en ouvre de l'encapsulation

Encapsulation

- Notion essentielle dans la programmation objet
- Suppose une séparation entre l'**interface publique** (accessible) d'une classe et son **fonctionnement interne** qui ne doit pas être ni perturbé ni forcement connu par l'extérieur.
- Outils pour la mise en ouvre de l'encapsulation
    - d'une architecture pensée et validée par le concepteur

Encapsulation

- Notion essentielle dans la programmation objet
- Suppose une séparation entre l'**interface publique** (accessible) d'une classe et son **fonctionnement interne** qui ne doit pas être ni perturbé ni forcement connu par l'extérieur.
- Outils pour la mise en ouvre de l'encapsulation
    - d'une architecture pensée et validée par le concepteur
    - des droits et d'espaces d'accès

Encapsulation

- Notion essentielle dans la programmation objet
- Suppose une séparation entre l'**interface publique** (accessible) d'une classe et son **fonctionnement interne** qui ne doit pas être ni perturbé ni forcement connu par l'extérieur.
- Outils pour la mise en ouvre de l'encapsulation
  - d'une architecture pensée et validée par le concepteur
  - des droits et d'espaces d'accès
  - renforcement / relâchement des contraintes

Accès	class	struct	union
public	possible	par défaut	par défaut
protected	possible	possible	impossible
private	par défaut	possible	impossible

Encapsulation

- Notion essentielle dans la programmation objet
- Suppose une séparation entre l'**interface publique** (accessible) d'une classe et son **fonctionnement interne** qui ne doit pas être ni perturbé ni forcement connu par l'extérieur.
- Outils pour la mise en ouvre de l'encapsulation
  - d'une architecture pensée et validée par le concepteur
  - des droits et d'espaces d'accès
  - renforcement / relâchement des contraintes

Accès	class	struct	union
public	possible	par défaut	par défaut
protected	possible	possible	impossible
private	par défaut	possible	impossible

Droits d'accès :

Accès	classe elle-même	classe dérivée	exterieur
public	OUI	OUI	OUI
protected	OUI	OUI	NON
private	OUI	NON	NON

Droits d'accès : protection par const

- Méthode/fonction, paramètre précédé par const :
  - seulement pour les références et les pointeurs
  - signifie que l'objet pointé (référencé) ne sera pas modifié par le code de la méthode

Droits d'accès : protection par const

- Méthode/fonction, paramètre précédé par **const** :
  - seulement pour les références et les pointeurs
  - signifie que l'objet pointé (référencé) ne sera pas modifié par le code de la méthode
- Méthode suivie de **const** :
  - l'objet ***this** est protégé par rapport à la méthode

Droits d'accès : protection par const

- Méthode/fonction, paramètre précédé par **const** :
  - seulement pour les références et les pointeurs
  - signifie que l'objet pointé (référencé) ne sera pas modifié par le code de la méthode
- Méthode suivie de **const** :
  - l'objet ***this** est protégé par rapport à la méthode
- Variable (objet) protégée par **const** :
  - on ne peut plus le modifier mais on peut lire ses attributs et appeler ses méthodes constantes
  - une variable entière constante peut servir comme taille de tableau non-dynamique, valeur de case etc.

Droits d'accès : protection par const

- Méthode/fonction, paramètre précédé par **const** :
  - seulement pour les références et les pointeurs
  - signifie que l'objet pointé (référencé) ne sera pas modifié par le code de la méthode
- Méthode suivie de **const** :
  - l'objet ***this** est protégé par rapport à la méthode
- Variable (objet) protégée par **const** :
  - on ne peut plus le modifier mais on peut lire ses attributs et appeler ses méthodes constantes
  - une variable entière constante peut servir comme taille de tableau non-dynamique, valeur de case etc.
- Déclaration de constantes spécifiques pour une classe :
  - eviter **#define** (accessible partout, conflit de noms, **#undef**)
  - utiliser des variables de type **static const**

Encapsulation : architecture

- Comment protéger l'accès à un attribut :

Encapsulation : architecture

- Comment protéger l'accès à un attribut :
  - complètement : par droit d'accès **private**

Encapsulation : architecture

- Comment protéger l'accès à un attribut :
  - complètement : par droit d'accès **private**
  - sauf pour les héritiers : par droit d'accès **protected**

Encapsulation : architecture

- Comment protéger l'accès à un attribut :
  - complètement : par droit d'accès **private**
  - sauf pour les héritiers : par droit d'accès **protected**
  - interdiction d'écriture : **private** et méthode publique **type_attr GetAttribute()**

Encapsulation : architecture

- Comment protéger l'accès à un attribut :
  - complètement : par droit d'accès **private**
  - sauf pour les héritiers : par droit d'accès **protected**
  - interdiction d'écriture : **private** et méthode publique **type_attr GetAttribute()**
  - filtrage d'écriture : **private** et méthode publique **bool SetAttribute(type_attr)**

Encapsulation : architecture

- Comment protéger l'accès à un attribut :
  - complètement : par droit d'accès **private**
  - sauf pour les héritiers : par droit d'accès **protected**
  - interdiction d'écriture : **private** et méthode publique **type_attr GetAttribute()**
  - filtrage d'écriture : **private** et méthode publique **bool SetAttribute(type_attr)**
- les méthodes de type **GetXXX** et **SetXXX** sont appelées des accesseurs (setters et getters)

Encapsulation : architecture

- Comment protéger l'accès à un attribut :
  - complètement : par droit d'accès **private**
  - sauf pour les héritiers : par droit d'accès **protected**
  - interdiction d'écriture : **private** et méthode publique **type_attr GetAttribute()**
  - filtrage d'écriture : **private** et méthode publique **bool SetAttribute(type_attr)**
- les méthodes de type **GetXXX** et **SetXXX** sont appelées des accesseurs (setters et getters)
- pour des raisons d'efficacité, on les déclare **inline**

Encapsulation : exemple

- Comment s'assurer que le dénominateur de la fraction `CFract` ne sera jamais nul ? Et que les deux attributs ne seront jamais négatifs en même temps ?

Encapsulation : exemple

- Comment s'assurer que le dénominateur de la fraction `CFract` ne sera jamais nul ? Et que les deux attributs ne seront jamais négatifs en même temps ?
- Réorganiser la classe `CFract` pour attendre ces objectifs, sans restreindre l'accès en lecture aux deux attributs de la fraction.

Encapsulation : exemple

- Comment s'assurer que le dénominateur de la fraction `CFract` ne sera jamais nul ? Et que les deux attributs ne seront jamais négatifs en même temps ?
- Réorganiser la classe `CFract` pour attendre ces objectifs, sans restreindre l'accès en lecture aux deux attributs de la fraction.
- Et si l'on rajoute la contrainte suivante : "l'utilisateur ne peut jamais modifier directement les deux attributs (seulement les initialiser à la construction)" ?

Surcharge des opérateurs

A quoi cela sert ?

- écrire `a+b` à la place de `Add(a,b)` ou `a.Add(b)`

Comment ?

Quels opérateurs ?

Surcharge des opérateurs

A quoi cela sert ?

- écrire `a+b` à la place de `Add(a,b)` ou `a.Add(b)`

Comment ?

- par méthode

Quels opérateurs ?

Surcharge des opérateurs

A quoi cela sert ?

- écrire `a+b` à la place de `Add(a,b)` ou `a.Add(b)`

Comment ?

- par méthode
- par fonction

Quels opérateurs ?

Surcharge des opérateurs

A quoi cela sert ?

- écrire `a+b` à la place de `Add(a,b)` ou `a.Add(b)`

Comment ?

- par méthode
- par fonction

Quels opérateurs ?

- par méthode/fonction : + - / * | || & && « » < > ++ −

Surcharge des opérateurs

A quoi cela sert ?

- écrire `a+b` à la place de `Add(a,b)` ou `a.Add(b)`

Comment ?

- par méthode
- par fonction

Quels opérateurs ?

- par méthode/fonction : + - / * | || & && « » < > ++ −
- uniquement par méthode : = -> [] ()

Surcharge des opérateurs

A quoi cela sert ?

- écrire `a+b` à la place de `Add(a,b)` ou `a.Add(b)`

Comment ?

- par méthode
- par fonction

Quels opérateurs ?

- par méthode/fonction : + - / * | || & && « » < > ++ −
- uniquement par méthode : = -> [] ()
- pas de surcharge : . .* : : : :* ? : sizeof

Surcharge - exemple par fonction

```
class complexe
{
  float r, i;
  public:
    complexe(float _r, float _i): r(_r), i(_i) {}
    friend const complexe operator+ (const complexe& c1,
                  const complexe& c2);
};

const complexe operator+ (const complexe& c1,
        const complexe& c2)
{
  return complexe(c1.r+c2.r, c1.i+c2.i);
}
```

Surcharge - exemple par méthode

```
class complexe
{
  float r, i;
  public:
    complexe(float _r, float _i): r(_r), i(_i) {}
    const complexe operator+ (const complexe& c2) const
    {
        return complexe(r+c2.r, i+c2.i);
    }
};
```

Opérateurs pour `CFract`

Doter la classe `CFract` :

- d'un opérateur d'addition suivie d'attribution `+=` et de multiplication suivie d'attribution `*=`

Opérateurs pour `CFract`

Doter la classe `CFract` :

- d'un opérateur d'addition suivie d'attribution `+=` et de multiplication suivie d'attribution `*=`
- d'un opérateur d'addition `+` et de multiplication `*`

Opérateurs pour `CFract`

Doter la classe `CFract` :

- d'un opérateur d'addition suivie d'attribution `+=` et de multiplication suivie d'attribution `*=`
- d'un opérateur d'addition `+` et de multiplication `*`
- de post- et pre-incrementation (`++`), de comparaison (`==`)

Opérateurs pour `CFract`

Doter la classe `CFract` :

- d'un opérateur d'addition suivie d'attribution `+=` et de multiplication suivie d'attribution `*=`
- d'un opérateur d'addition `+` et de multiplication `*`
- de post- et pre-incrementation (`++`), de comparaison (`==`)
- d'un opérateur de conversion emphtypecast vers un double

Opérateurs pour `CFract`

Doter la classe `CFract` :

- d'un opérateur d'addition suivie d'attribution `+=` et de multiplication suivie d'attribution `*=`
- d'un opérateur d'addition `+` et de multiplication `*`
- de post- et pre-incrementation (`++`), de comparaison (`==`)
- d'un opérateur de conversion emphtypecast vers un double
- tester les opérateurs ainsi surchargés

Opérateurs pour `CFract`

Doter la classe `CFract` :

- d'un opérateur d'addition suivie d'attribution `+=` et de multiplication suivie d'attribution `*=`

Opérateurs pour `CFract`

Doter la classe `CFract` :

- d'un opérateur d'addition suivie d'attribution `+=` et de multiplication suivie d'attribution `*=`
- d'un opérateur d'addition `+` et de multiplication `*`

Opérateurs pour `CFract`

Doter la classe `CFract` :

- d'un opérateur d'addition suivie d'attribution `+=` et de multiplication suivie d'attribution `*=`
- d'un opérateur d'addition `+` et de multiplication `*`
- de post- et pre-incrementation (`++`), de comparaison (`==`)

Opérateurs pour `CFract`

Doter la classe `CFract` :

- d'un opérateur d'addition suivie d'attribution `+=` et de multiplication suivie d'attribution `*=`
- d'un opérateur d'addition `+` et de multiplication `*`
- de post- et pre-incrementation (`++`), de comparaison (`==`)
- d'un opérateur de conversion emphtypecast vers un double

Opérateurs pour `CFract`

Doter la classe `CFract` :

- d'un opérateur d'addition suivie d'attribution `+=` et de multiplication suivie d'attribution `*=`
- d'un opérateur d'addition `+` et de multiplication `*`
- de post- et pre-incrementation (`++`), de comparaison (`==`)
- d'un opérateur de conversion emphtypecast vers un double
- tester les opérateurs ainsi surchargés

La forme canonique de Coplien (FCC)

```
class CFract
{
  int a, b;
  // ...
  public:
    CFract  ();                                  //constructeur par defaut
    CFract  (const CFract &);                    //constructeur de copie
    CFract& operator= (const CFract &);          //operateur d'affectation
    ~CFract ();                                  // destructeur
};
```

Les classes vérifiant la FCC sont utilisables sans effets indésirables dans les cas suivants

- création d'une instance de la classe

La forme canonique de Coplien (FCC)

```
class CFract
{
  int a, b;
  // ...
  public:
    CFract  ();                                  //constructeur par defaut
    CFract  (const CFract &);                    //constructeur de copie
    CFract& operator= (const CFract &);          //operateur d'affectation
    ~CFract ();                                  // destructeur
};
```

Les classes vérifiant la FCC sont utilisables sans effets indésirables dans les cas suivants

- création d'une instance de la classe
- utilisation de la classe comme valeur de retour d'une fonction

La forme canonique de Coplien (FCC)

```
class CFract
{
  int a, b;
  // ...
  public:
    CFract  ();                                  //constructeur par defaut
    CFract  (const CFract &);                    //constructeur de copie
    CFract& operator= (const CFract &);          //operateur d'affectation
    ~CFract ();                                  // destructeur
};
```

Les classes vérifiant la FCC sont utilisables sans effets indésirables dans les cas suivants

- création d'une instance de la classe
- utilisation de la classe comme valeur de retour d'une fonction
- création d'un tableau d'instances de la classe

La forme canonique de Coplien (FCC)

```
class CFract
{
  int a, b;
  // ...
  public:
    CFract  ();                                  //constructeur par defaut
    CFract  (const CFract &);                    //constructeur de copie
    CFract& operator= (const CFract &);          //operateur d'affectation
    ~CFract ();                                  // destructeur
};
```

Les classes vérifiant la FCC sont utilisables sans effets indésirables dans les cas suivants

- création d'une instance de la classe
- utilisation de la classe comme valeur de retour d'une fonction
- création d'un tableau d'instances de la classe

L'opérateur d'affectation est responsable de recopier tous les champs et des allocations mémoire si nécessaire (comme un constructeur de copie valide).

La forme canonique de Coplien (FCC)

Exemple implémentation opérateur d'affectation

```
CFract& CFract::operator= (const CFract & obj){
  if(this != &obj){
    this->a = obj.a;
    this->b = obj.b;
  }
  return *this; //renvoie l'instance recopiee
}
```

Passage des objets en paramètre

Les options pour passer des paramètres :

- par valeur : **lent** car ils sont copiés

Passage des objets en paramètre

Les options pour passer des paramètres :

- par valeur : **lent** car ils sont copiés
- par adresse : possible mais déconseillé

Passage des objets en paramètre

Les options pour passer des paramètres :

- par valeur : **lent** car ils sont copiés
- par adresse : possible mais déconseillé
- par référence : efficace mais on peut modifier l'instance passée en paramètre

Passage des objets en paramètre

Les options pour passer des paramètres :

- par valeur : **lent** car ils sont copiés
- par adresse : possible mais déconseillé
- par référence : efficace mais on peut modifier l'instance passée en paramètre
- par référence `const` : efficace et safe

Passage des objets en paramètre

Les options pour passer des paramètres :

- par valeur : **lent** car ils sont copiés
- par adresse : possible mais déconseillé
- par référence : efficace mais on peut modifier l'instance passée en paramètre
- par référence `const` : efficace et safe

Quand il ne faut pas utiliser des références pour les paramètres :

- pour des tableaux
- pour des types de base