Le mot-clef `const` en C++

Valeurs constantes

En C, on écrit des lignes du style :

```
#define FACTEUR_AVOGADRO 6.022e+23
```

pour représenter par, exemple, une constante physique. Dans le reste du code, plutôt que d'écrire la valeur numérique, on utilise l'alias $\Longrightarrow$ quand on veut corriger la constante, il suffira de modifier une seule ligne.

En C++ (valable aussi pour des compilateurs C récents), on peut aussi écrire (souvent préférable) :

```
const double FACTEUR_AVOGADRO=6.022e+23;
```

$\Longrightarrow$ meilleur contrôle du type de la constante.

Passage de paramètres : précisions

1. `void f(double x) { ... }`

   Un réel est dupliqué quand on rentre dans `f`. Il s'agit d'une variable de petite taille $\Longrightarrow$ OK

2. `void f(myObject y) { ... }`

   Un objet `myObject` est dupliqué quand on rentre dans `f`. S'il s'agit d'une variable de grande taille $\Longrightarrow$ pas OK

3. `void f(myObject & y) { ... }`

   ou

   `void f(myObject * y) { ... }`

   On passe un pointeur ou une référence vers un objet `myObject` quand on rentre dans `f` $\Longrightarrow$ pas de duplication. L'objet peut être modifié à l'intérieur de la fonction. Pas OK si l'utilisateur ne veut pas récupérer les modifications de l'objet.

4. `void f(const myObject & y) { ... }`

   On passe une référence (ou un pointeur) constant $\Longrightarrow$ pas de duplication et l'utilisateur est assuré que son objet ne sera pas modifié dans la fonction.

Quand on utilise un objet constant (référence ou pointeur), *on ne peut pas le modifier.*

En particulier, on ne peut pas appeler de fonction membre de cet objet qui risque de le modifier.

Les seules fonctions membre qui peuvent être appelées sont des fonctions membres avec l'attribut const.

Exemple :

```
double A::getX() const { return _x; }
void A::setX(double x) { _x = x; }
```

`getX` peut être appelée par un objet constant mais pas `setX`.
Les 2 fonctions peuvent être utilisées par des objets non constants.

Concepts orientés objet en C++ :

Encapsulation

Contexte

Supposons (pour fixer les idées) que :

la classe soit conçue par une personne (”le développeur”) et utilisée par une autre (”l’utilisateur”).

L’opération d’encapsulation revient à décomposer la classe en 2 parties :

- une partie visible, accessible et éventuellement modifiable par l’utilisateur : l’interface,
- une partie contrôlée par le développeur et à laquelle l’utilisateur n’a pas ou peu accès : l’implémentation.

Interface de la classe

- Liste des membres de la classe, avec leur type (entiers, réels, vecteurs, autre classe).
- Liste des prototypes des méthodes de la classe (nom de la méthode, nombre et type des arguments, type du résultat).

Implémentation de la classe

- Code source des méthodes de la classe. L'accès à ce code source est réservé au développeur. L'utilisateur ne voit normalement que le code compilé.

$\Rightarrow$

Possibilité pour le développeur d'une classe de modifier l'implémentation de cette classe en étant certain de ne pas perturber le code des utilisateurs.

Exemple : cas de la classe Vecteur

Interface

```
class Vecteur {
public :
  int n;
  double *x;
  Vecteur(int);
  Vecteur(const Vecteur &);
  ~Vecteur();
  double Norme();
  double & operator[](int);
};
```

Implémentation

```
Vecteur::Vecteur(int m) {
  if (m > 0) then {
    n = m; x = new double[m]; }
  else
    n = 0;
}
Vecteur::~Vecteur() {
  if (n > 0) then delete [] x;
}
double & Vecteur::operator[](int
i) {
  if ((i >= 0) && (i<n))
    then return x[i];
    else exit(-1);
}
...
```

Organisation du code d'une classe

Dès qu'un projet atteint un certaine taille, on conseille la procedure suivante :

L'interface est placée dans un fichier en-tête : fichier avec un suffixe .h (ou .hxx, .hpp, etc).
Ce fichier est inclu dans tous les fichiers utilisant la classe, en ajoutant dans ces fichiers, la ligne :

```
#include "NomDuDichierEntete"
```

L'implémentation est placée dans un fichier source : fichier avec un suffixe .cc (ou .CC, .cxx, .cpp, etc).

Le fichier source doit lui aussi inclure le fichier en-tête.

Parties privée et publique de l'interface

Telle qu'elle est écrite ci-avant, l'interface de la classe `Vecteur` est accessible et modifiable par n'importe quel utilisateur de la classe

$\Rightarrow$ on peut, par exemple, modifier la valeur de `n` sans modifier le nombre de coefficients

$\Rightarrow$ perte de cohérence dans les données internes d'un objet de type vecteur.

Dans une interface de classe, il est possible de restreindre l'accès à certains membres et/ou à certaines méthodes

Cette partie est appelée partie privée de la classe, le reste est la partie publique de la classe.

En C++, la séparation de l'interface en partie privée - partie publique se fait comme suit :

```
class C {
public :
  <liste de membres/méthodes publics>
private :
  <liste de membres/méthodes privés>
};
```

Avec les règles d'accès :

Les membres/méthodes publics sont accessibles de n'importe quel endroit du code (interne ou externe à la classe).

Les membres/méthodes privés ne sont (normalement) accessibles que par les autres méthodes de la même classe.

Exemple, soit l'interface :

```
class Vecteur {
   private :
     int n;
     double *x;
   public :
     double Norme();
     double & operator[](int);
   };
```

Dans le programme principal, si `V` est de type `Vecteur` et `b` un réel, on ne pourra pas écrire :

`b = V.x[4];` (`x` est un membre privé de `Vecteur`)

Par contre,

dans le source de la méthode `Norme`,

```
x[i] = 6.0;
```

sera accepté (`Norme` et `x` sont des méthodes/membres de la même classe, avec un statut différent);

de même

```
b = V[4];
```

sera accepté dans le programme principal car `operator[]` est une méthode publique de la classe `Vecteur`.

Concepts orientés objet en C++ :

Héritage

Héritage simple

Relation entre 2 classes (classe de base et classe dérivée). La classe dérivée incorpore les méthodes/membres de la classe de base.

La classe dérivée peut :

- réutiliser telles quelles les méthodes de la classe de base,
- redéfinir les méthodes de la classe de base,
- ajouter ses propres méthodes et membres,
- utiliser les membres de la classe de base

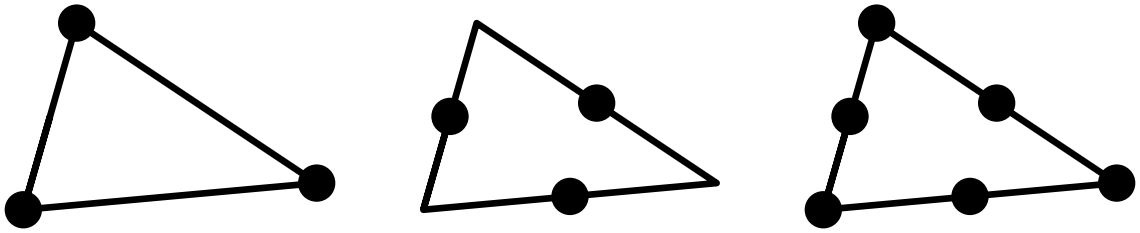
On dit aussi que la classe dérivée est une spécialisation d'une classe de base.

Exemple : famille d'éléments finis.

Caractéristiques d'un élément fini :

- géométrie (triangle, rectangle, tétraèdre,...)
- degrés de liberté (valeur en un point, gradient, ...)
- fonctions de forme (base de fonctions représentables)

Exemple : 3 types d'éléments finis, avec des degrés de liberté et des fonctions de forme différents mais la même géométrie (triangle).



ElemTrP1 **ElemTrP1D** **ElemTrP2**

$\Rightarrow$ ce qui ne dépend que de la géométrie (centre de gravité, aire de l'élément, sommets du triangle, ...) sera reprogrammé à l'identique plusieurs fois.

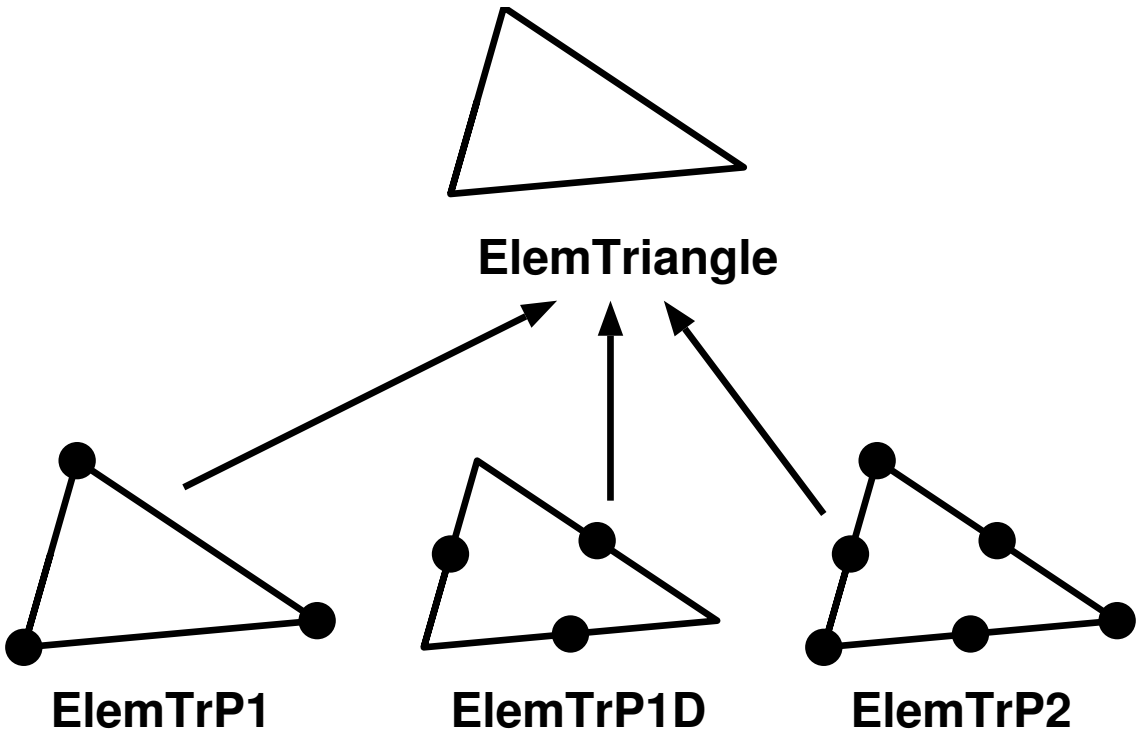
$\Rightarrow$ rassembler les caractéristiques communes des 3 éléments dans une classe "incomplète" `ElemTriangle` : élément fini sans degrés de liberté ni fonctions de forme,

$\Rightarrow$ définir des classes contenant les données spécifiques à chaque type d'élément finis et récupérer les informations communes (i.e. hériter) de la classe `ElemTriangle`.

Interfaces :

```
class ElemTriangle {
...
};
class ElemTrP1 :  public ElemTriangle {
...
};
class ElemTrP1D : public ElemTriangle {
...
};
class ElemTrP2 :  public ElemTriangle {
...
};
```

Résumé par le diagramme :



Comportement des constructeurs/destructeurs

Si une classe `B` dérive d'une classe `A` :

```
class A {
  A();
  ~A();
};
class B : public A {
  B();
  ~B();
};
```

- Quand on définit un objet de type `B`, est appelé :
  - d'abord le constructeur de la classe de base `A`,
  - ensuite le constructeur de la classe dérivée `B`.

- Quand l'objet de type `B` est détruit :
  - d'abord le destructeur de la classe de base `B`,
  - ensuite le destructeur de la classe dérivée `A`.

Par défaut, le constructeur sans paramètre de la classe de base est appelé. Si on veut appeler un constructeur avec paramètre, on doit le spécifier dans l'implémentation du constructeur de la classe dérivée.

```
A(double x, double y)
{ ... };
B(double x, double y, double z)
    :  A(x, y)
{ ... };
```

Accés ”protégé” aux membres/méthodes de la classe de base

Pour chacun de ses membres/méthodes, la classe de base peut choisir qui y aura accès :

- toutes les autres classes et les fonctions hors classe (accès public : public)
- seulement les classes qui dérivent d'elle-même (accès protégé : protected)
- seulement elle-même (accès privé : private)

Les autorisations d'accès sont définis en C++ au niveau des classes : un objet d'une classe accède aux données privées des autres objets de la même classe.

Classes abstraites

Méthode virtuelle pure : on peut dans l'interface d'une classe, déclarer une méthode virtuelle pure comme dans l'exemple :

```
class A {
...
  int f(...);
  virtual g(...)  = 0;
...
};
```

La méthode `f` est "réelle" (son implémentation sera fournie par ailleurs), la méthode `g` est "virtuelle pure" ou "abstraite" (seule l'interface est disponible).

Toute classe dont au moins une des méthodes est virtuelle pure est appelée classes abstraite.

C'est une classe dont l'implémentation est incomplète.
Il ne sera pas possible de construire un objet instance de cette classe.

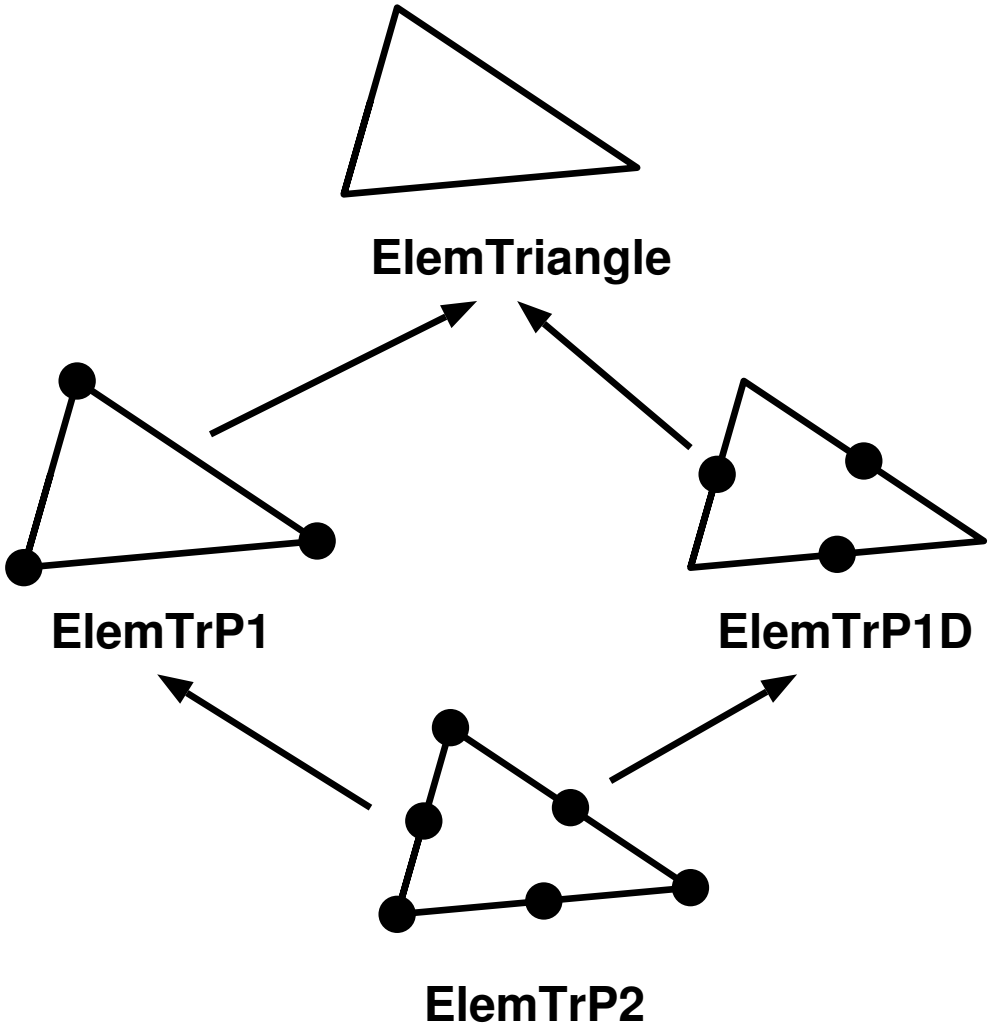
Une classe abstraite sert essentiellement de modèle pour d'autres classes qui en dérivent.
C'est un moyen d'obliger, celui qui veut développer une nouvelle classe dérivée, de bien fournir une implémentation complète de cette classe.

Héritage multiple.

Une classe dérivée peut posséder plusieurs classes de base

```
class ElemTriangle { ... };
class ElemTrP1 : public ElemTriangle { ... };
class ElemTrP1D : public ElemTriangle {...};
class ElemTrP2 :
  public ElemTrP1, public ElemTrP1D {...};
```

Diagramme d’héritage :



En général, on essaie d'éviter l'héritage multiple.

Exemple :

Une classe A définit une méthode f et que A1 et A2, deux classes dérivées de A redéfinissent la méthode f.

Si une classe B dérive à la fois de A1 et de A2, il y a une ambiguïté quand on écrit

```
B b;
b.f();
```

(laquelle des 3 fonctions f utilise-t'on ?)

Pour préciser, on écrira `b.A1::f()` ou `b.A2::f()` suivant la version de `f` demandée.