Cours n° 5
Polymorphisme
Surdéfinition opérateurs
Bibliothèque standard

Module Info3/M3105C

Sebastien.Kramm@univ-rouen.fr

IUT GEII Rouen

2018-2019

Sommaire

Polymorphisme ?

- Ce terme désigne la capacité qu'à une méthode de s'adapter automatiquement à l'objet manipulé.
- N'a de sens **que** dans un contexte "Héritage".
- Principale application : regroupement d'objets dans des listes hétérogènes.
- Exemple : soit une application de gestion de parc de véhicules.
  - Classe VEHICULE, dérivée en MOTO, AUTO, CAMION ;
  - ces 4 classes ont une méthode `Afficher()` ;
  - On mémorise dans une liste d'objets de type VEHICULE tous les véhicules du parc (liste hétérogène) ;
  - Si on applique la méthode `Afficher()` à chaque objet de la liste, on veut exécuter la méthode associée a l'objet (MOTO, AUTO ou CAMION), et non pas la méthode générique de la classe VEHICULE.

⇒ polymorphisme !

Exemple applicatif

- Soit la classe de base suivante :

```
class VEHICULE
{
  protected:
    int nb_roues;
    int nb_places;
  public:
    VEHICULE();
    void Affiche();
};
```

- On dérive celle-ci en deux classes :

```
class MOTO : public VEHICULE
{
  private:
    GUIDON guid;
  public:
    MOTO();  // constructeur
    void Affiche();
};
```

```
class AUTO : public VEHICULE
{
  private:
    VOLANT volant;
  public:
    AUTO(); // constructeur
};
```

Exemple applicatif

- Les méthodes pourront être implémentées comme suit :

```
void VEHICULE::Affiche()
{
  cout << "je suis un véhicule à " << nb_roues << " roues\n";
}
void MOTO::Affiche()
{
  cout << "je suis une moto\n";
}
```

- On pourra appeler dans la méthode de la **classe dérivée** la méthode de la **classe de base**, via l'opérateur de résolution de portée :

```
void MOTO::Affiche()
{
  cout << "Je suis une moto\n";
  VEHICULE::Affiche(); // appel de "Affiche()" de la classe mère
}
```


Polymorphisme en C++

- En C++, le polymorphisme ne peut s'implémenter que si l'objet est géré via une adresse (pointeur ou référence).
- Pourquoi ?
  - Si on crée une liste (tableau) de type VEHICULE, et qu'on y range une MOTO :

```
VEHICULE v[10]; // tableau de 10 vehicules
MOTO mamoto( "Kawasaki", "XYZRW 1500", Rouge );
v[0] = mamoto;
```

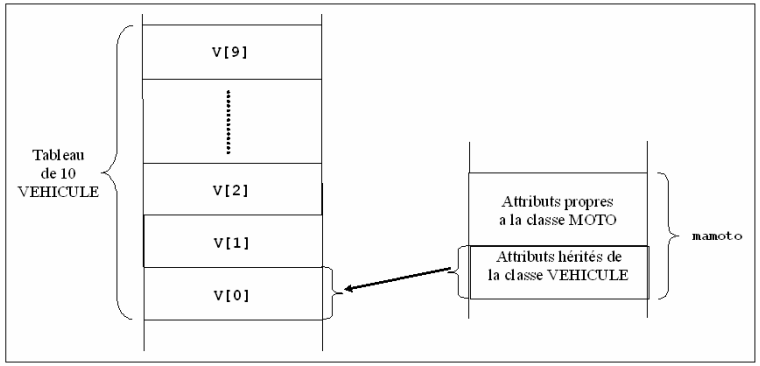
  - Lors de la copie, l'objet est automatiquement **dégradé** dans son type de base (cf. Cours 4).
  - Ici, il perd sa qualité de MOTO, et se transforme en VEHICULE ⇒ on ne peut donc plus disposer de la méthode `Affiche()` de la classe MOTO !

`v[0].Affiche();`	=>	`"je suis un véhicule à 2 roues"`

Copie par valeur (*downcast*)

- Lors de la copie, on ne copie **que** les attributs communs aux deux classes : ceux propres à la classe MOTO sont perdus.

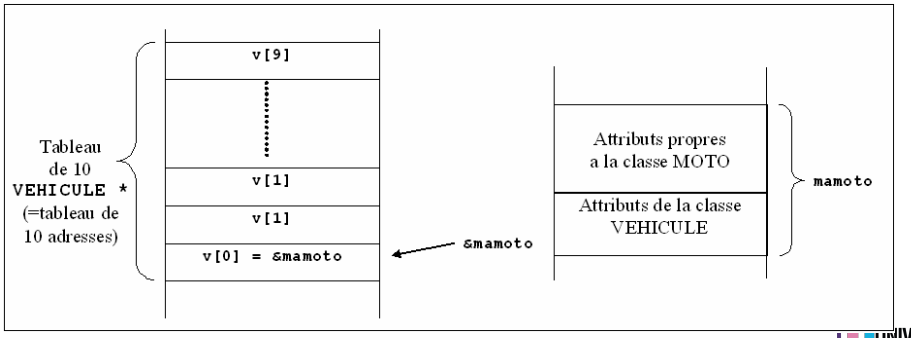
```
v[0] = mamoto;
```



Copie de l'adresse

- Solution : Il faudra utiliser une liste de pointeurs sur des objets de type VEHICULE.

```
VEHICULE* v[10]; // tableau de 10 pointeurs
MOTO mamoto( "Kawasaki", "XYZRW 1500", Rouge );
v[0] = &mamoto;
```



Ligature statique

- Mais pas suffisant, un pointeur est **typé** : si on affecte a un pointeur de type VEHICULE, l'adresse d'un objet de type MOTO :

```
VEHICULE* vehic = 0;
MOTO*     moto  = new MOTO;
vehic = moto;
```

- Et qu'on appelle la méthode `Affiche()` via le pointeur :

```
vehic->Affiche();
```

- Ce sera **encore** la méthode `Affiche()` de la classe VEHICULE qui sera exécutée !
- Les pointeurs sont typés, il y a par défaut **ligature statique** :
  - Le **type** de l'objet pointé est **figé** à la compilation.
  - Même en lui affectant l'adresse d'un objet d'un autre type, la variable `vehic` reste un "pointeur sur un VEHICULE"...
    ...et on ne pourra accéder qu'aux méthodes de la classe VEHICULE

Ligature dynamique

- Pour avoir une **ligature dynamique** (= faire en sorte que la méthode appelée dépende de la nature de l'objet réellement pointé), il faut déclarer la méthode `Affiche()` dans la classe de base comme **polymorphe**.
- Ceci se fait :
  - dans la déclaration de la classe de base,
  - en préfixant la méthode avec le mot clé `virtual`.
- On parle alors de **méthode virtuelle**.

```
class VEHICULE
{
  public:
    virtual void Affiche(); // => la méthode Affiche() pourra être
         redéfinie dans les classes filles
    ...
};
```


Méthodes virtuelles

Définition

Si une méthode d'une classe de base est virtuelle, alors, en cas d'appel de cette méthode sur un pointeur pointant sur un objet d'une classe dérivée,

- **Si** la classe dérivée **a implémenté** la méthode :
  $\Rightarrow$ c'est celle-ci qui sera exécutée.
- **Si** la classe dérivée n'**a pas implémenté** la méthode :
  $\Rightarrow$ c'est la méthode de la classe de base qui sera exécutée.

```
VEHICULE* v[3];
v[0] = new MOTO;
v[1] = new AUTO;
v[2] = new VEHICULE;
v[0]->Affiche(); // Appel de Affiche() de la classe MOTO
v[1]->Affiche(); // Appel de Affiche() de la classe VEHICULE
v[2]->Affiche(); // Appel de Affiche() de la classe VEHICULE
```


Classes abstraites

- Dans certains cas, il arrive qu'une classe n'ait pas vocation à se voir instanciée, mais ne serve qu'a créer d'autres classes par héritage. (créer des objets de type VEHICULE n'a pas beaucoup de sens : il est plus logique de créer des AUTO, MOTO ou CAMION).
- On peut alors matérialiser cette notion en rendant la classe abstraite.

Définition

Une **classe abstraite** est une classe à partir de laquelle on ne peut pas créer d'objets.

- En C++, une classe devient automatiquement abstraite dès lors qu'elle a au moins une **méthode virtuelle pure**.
- Syntaxe :

```
class VEHICULE
{
  public:
    virtual void Affiche() = 0; // méthode virtuelle pure
};
```

Exercice : hierarchie de classes

- Soit une classe : FORME, dérivée en CARRE et CERCLE. Donner aux classes une méthode virtuelle `void Dessine()`

```
class FORME
{
  public:



  private:
    ...
};
```

```
class CARRE
{
  public:


  private:
    ...
};
class CERCLE
{
   public:


   private:
     ...
};
```


Sommaire

Introduction : addition avec des types complexes

- Exemple : soit un type de donnée "nombre complexe" (syntaxe C)

```
typedef struct cplxe {
   float reel;
   float imag;
} COMPLEXE;
```

- En C, avec la déclaration : `COMPLEXE n1, n2, n3;`,
  l'écriture `n3 = n1 + n2;` est **interdite**.
- Il faut écrire :

```
n3.reel = n1.reel + n2.reel;
n3.imag = n1.imag + n2.imag;
```

- C'est lourd...

Introduction : addition avec des types complexes

- En C++, on pourrait créer une classe :

```
class COMPLEXE
{
   private:
     float reel, imag;
   public:
    ...
};
```

- Problème : on ne pourra plus écrire ceci :

```
n3.reel = n1.reel + n2.reel;
n3.imag = n1.imag + n2.imag;
```

  $\Rightarrow$ attributs privés !
- Pas très pratique de toute façon !

Utilisation d'une fonction

- Une solution serait d'utiliser une fonction :

```
COMPLEXE n1, n2, n3;
 ...
n3 = Add( n1, n2 );
```

- Cette fonction aurait pour définition :

```
COMPLEXE Add( COMPLEXE n1, COMPLEXE n2 )
{
   COMPLEXE n;
   n.reel = n1.reel + n2.reel;
   n.imag = n1.imag + n2.imag;
   return n;
}
```

- Problème : ça ne fonctionne pas !
  - La fonction ne peut pas accéder aux attributs de la classe (privés).
  - Il faudrait utiliser accesseurs & mutateurs.
  - Ca devient très lourd à gérer...

Classes amies

- On peut aussi déclarer dans une classe qu'une autre classe est "amie".
- La classe amie pourra accéder sans restrictions aux attributs privés.
- Exemple :

```
class POINT
{
   friend class VECTEUR2D;
   private:
     float x, y;
   public:
    ...
};
```

⇒ Les objets de type VECTEUR2D pourront accéder a tous les attributs des objets de type POINT.

- Attention : pas symétrique !
  (une classe peut en déclarer une autre comme "amie" sans que l'inverse ne soit vrai).

Fonctions amies

- Le C++ propose la notion d'amitié : On peut déclarer dans une classe qu'une fonction est **amie** de la classe (mot clé `friend`).
- Cette fonction peut alors accéder sans restrictions à **tous les attributs** de la classe.
- La déclaration de la fonction se fait alors dans la classe :

```
class COMPLEXE
{
   friend COMPLEXE Add( COMPLEXE n1, COMPLEXE n2 );

   private:
     float reel, imag;
   public:
    ...
};
```

- (La définition de la fonction reste la même.)

Fonctions amies : un piège ?

- La notion d'amitié ne respecte pas le principe d'encapsulation.
  - Approche procédurale de la programmation.
  - Néanmoins bien pratique dans certains cas de figure (pour alléger le code, améliorer les performances, ...)
  - A utiliser avec modération !
- Remarque : la déclaration d'amitié d'une classe envers une autre vaut "déclaration d'existence" (cf. cours 4)

```
// truc.h
class TRUC
{
   friend class BIDULE;
   private:
     ...
   public:
    void une_methode( BIDULE& b );
     ...
};
```

Amitié et héritage

- Attention : l'amitié ne s'hérite pas !
- Exemple : soit les classes suivantes :

```
// cbase.h
class CBASE
{
   friend class BIDULE;
   friend int ma_fonc();
   private:
     ...
   public:
     ...
};
```

```
// cderive.h
class CDERIVE: public CBASE
{
   private:
     ...
   public:
     ...
};
```

⇒ Les objets du type BIDULE pourront accéder aux attributs des objets de type CBASE, mais **pas** aux attributs des objets de type CDERIVE.
⇒ `ma_fonc()` pourra accéder à tous les attributs des objets de type CBASE, mais **pas** aux attributs des objets de type CDERIVE.

Exemple : surdéfinition de '+'

- On souhaite surdéfinir l'opérateur '+' de la classe POINT

```
class POINT
{
  private:
    float x,y;
  public:
    ...
};
```

- De façon à pouvoir écrire :

```
#include "point.h"
int main()
{
  POINT pt1(0,0), pt2(2,4), pt3;
  POINT milieu = (pt1 + pt2) / 2; // (1,2)
}
```


Encore mieux : surdéfinition d'opérateurs

- Le C++ permet de **définir** le sens des opérateurs courants au sein de la classe, et on pourra par exemple écrire de façon plus intuitive :

```
COMPLEXE n1, n2, n3;
 ...
n3 = n1 + n2;
```

Surdéfinition d'opérateurs

- Deux possibilités :
  - sous la forme d'une **fonction amie**,
  - sous la forme d'une **méthode** (fonction membre de la classe).
- Le mot clé : `operator`
- Cette surdéfinition est propre à la classe : l'opérateur **n'est pas** hérité par les classes dérivées.

1 - Surdéfinition par fonction amie

- Il faut :
  1. définir une fonction réalisant l'opération,
  2. déclarer cette fonction comme "amie" dans la déclaration de la classe.
- Cette fonction aura pour prototype (= *signature*) :

```
POINT operator + ( POINT p1, POINT p2 );
```

- Ce qui signifie : "*fonction implémentant l'opérateur '+', ayant 2 arguments de type POINT, et renvoyant un POINT*"
- Dans un programme, l'expression : `pt = pa + pb;` sera traduite pas :
  - exécuter la fonction '+', en copiant les valeurs de `pa` et `pb` dans `p1` et `p2`,
  - copier le résultat (valeur de retour) dans `pt`.

1 - Surdéfinition par fonction amie

- La définition de la **fonction** s'écrira (fichier `point.cpp`) :

```
POINT operator + ( POINT p1, POINT p2 )
{

  POINT pt;

  pt.x = p1.x + p2.x; // Addition
  pt.y = p1.y + p2.y;

  return pt;        // Copie du point en retour
}
```


2 - Surdéfinition par une méthode de classe

- Dans ce cas, il n'y a plus 2 arguments, mais un seul : comme avec n'importe quelle méthode, il y a **transmission automatique** de l'objet auquel la méthode s'applique.
- L'expression : `pt = pa + pb;` sera comprise comme :
  - appliquer l'opérateur '+' à l'objet `pa`,
  - transmettre `pb` comme argument,
  - stocker le résultat (valeur de retour) dans `pt`.
- La déclaration de la classe s'écrira :

```
class POINT
{
  private:
    float x,y;
  public:
    POINT operator + (POINT p);
    ...
};
```


2 - Surdéfinition par une méthode de classe

- Implémentation :

```
// point.cpp

POINT POINT::operator + ( POINT p )
{
  POINT pt;
  pt.x = x + p.x;
  pt.y = y + p.y;
  return pt;
}
```

- `x`, `y` : attributs de l'objet auquel on applique la méthode
- `p.x`, `p.y` : attributs de l'argument ('p')

Opérateurs de comparaison

- On peut aussi (et souvent on doit) surdéfinir les opérateurs de comparaison : (`==`, `<`, `!=`)
- Ces opérateurs ont tous 2 arguments, et renvoient un `bool`.
- Par exemple : `if( a == b )` sera compris comme : *comparer 'a' et 'b' et renvoyer* `true` *s'ils sont égaux,* `false` *sinon.*
- Implémentation par une méthode de classe : (auto-transmission du premier argument)

```
// point.h

class POINT
{
  private:
    float x, y;
  public:
    bool operator == (POINT a);
    ...
};
```

```
// point.cpp

bool POINT::operator == (POINT a)
{
  if( x == a.x && y == a.y )
    return true;
  else
    return false;
}
```

Opérateur de comparaison 'différent de'

- L'opérateur `!=` renvoie le contraire de l'opérateur `==` .
  ⇒ Renvoie `false` si les objets sont égaux.
- Astuce : en C/C++ : `(!a)` renvoie `false` si a est `true` (et inversement...)
- On a donc équivalence entre : `(a != b)` et `!(a == b)`
  ⇒ On pourra donc utiliser l'opérateur `==` pour écrire l'opérateur `!=`
- Question : comment récupérer l'objet *courant* dans la méthode ? (pour le passer en argument)
- Réponse : pointeur `this`.

mot-clé `this`

Dans chaque méthode d'une classe, un pointeur `this` est **automatiquement** déclaré, et pointe sur l'objet **courant**.

Exemple pour une classe quelconque

```
bool LAMBDA::operator != (LAMBDA a)
{
    return !( *this == a ); // appel de l'operateur == de la classe
}
```

- **Mais** :
  - L'appel de cet opérateur réalise un **clonage** de l'objet transmis ("passage par valeur" = copie de tous les attributs sur la pile).
  - Si l'objet est volumineux, ça peut être long : il est plus judicieux de faire un "passage par référence".

```
bool LAMBDA::operator != ( LAMBDA& a )
{
    return !( *this == a );
}
```


Optimisation (pour les pros...)

- Cet opérateur n'a pas à modifier l'objet, donc on déclare l'argument comme `const` (constant).

```
bool LAMBDA::operator != ( const LAMBDA& a )
{
    return !( *this == a );
}
```

- `const` permet de garantir à l'utilisateur de la classe que cet opérateur ne va pas modifier l'argument (ce qui est à droite du symbole ' !=').
- Rappel : cette modification serait possible du fait que la référence est en fait un alias sur la variable manipulée.

Exercice

- Soit une classe PIXEL. Définir l'opérateur == par une méthode de classe, sans tenir compte du numéro de série (unique).

```
class PIXEL
{
  private:
    int x, y;
    COULEUR coul;
    int NumSerie;
  public:




};
```

Sommaire

Bibliothèque standard du C++

- Des composants généraux prédéfinis (`complex`, `string`, ...),
- des solutions performantes pour manipuler des ensembles d'objets quel que soit leur type (conteneurs génériques),
- des algorithmes indépendant du type des données (programmation générique),
- plus toute la bibliothèque standard C.

Programmation générique

- Les tâches effectuées par les programmes sont souvent les mêmes.
- Exemples :
  - gérer une liste d'étudiants, une liste de livres, une liste de voitures, ...
    $\Rightarrow$ gérer une liste.
  - Compter dans une liste les éléments dont 'tel' attribut vaut 'tant' :
    $\Rightarrow$ algorithme identique que soit le type des objets.
  - Effacer le 'i'ème élément d'une liste :
    $\Rightarrow$ méthode identique quel que soit le type d'objets.
  - etc.
- Seul la nature des données change !
- Afin d'éviter de reprogrammer tout à chaque fois, la "*standard library*" du C++ fournit des outils facilitant l'intégration des données et des algorithmes à y appliquer $\Rightarrow$ "**Programmation générique**".

Trois concepts liés

1 - Conteneurs

- Classes qui permettent d'encapsuler les données dans une collection (objet qui contient des objets).
- deux types :
  - Conteneurs séquentiels : `list` et `vector`.
  - Conteneurs associatifs (= liste de paires).
- Gestion et allocation mémoire automatisée.

2 - Itérateurs

- Classes qui permettent d'accéder à **un** des éléments d'une collection.
  $\Rightarrow$ généralisation du concept de pointeur.

3 - Algorithmes

- Traitements réalisé sur une collection (recherche, enumération, tri,...)
- Implémentés sous forme de **fonctions**.

1 - Conteneurs séquentiels : `list` et `vector`

- Contiennent une série d'objets de nature identique.
- La plupart des méthodes applicables (accès et algorithmes) s'utilisent de façon identiques pour les deux conteneurs.
- Création :

```
#include "point.h"

#include <vector>
#include <list>
using namespace std;

int main()
{
  vector<POINT> vec; // Crée un "vector" de POINTs
  list<POINT> li;  // Crée une " list " de POINTs
}
```


`vector` : exemples

- Création d'un vecteur de points (vide) : `vector<POINT> v;`
- Création d'un vecteur de 10 points : `vector<POINT> v(10);`
  note : appel du constructeur sans arguments, qui doit exister
- Création d'un vecteur de 20 points, tous initialisés à une valeur :

```
POINT pt0(3,4);
vector<POINT> v(20, pt0);
```

- Accès à des valeurs existantes (lecture ou écriture) : 2 notations possibles :
  - Ecriture :

```
v.at(i) = POINT(3,4);
```

```
v[i] = POINT(3,4);
```

  - Lecture :

```
POINT pt = v.at(i);
```

```
POINT pt = v[i];
```

- La première méthode (`at()`) est plus sûre : arrêt "propre" en cas d'erreur d'indice.

`vector` : exemples

- Nombre d'éléments : méthode `size()` :

```
cout << "Nbe d'éléments = " << v.size() << endl;
```

- Ajout d'un élément en dernière position :

```
POINT pt;
v.push_back( pt );
```

- Supprimer tous les élements :

```
v.clear();
```

- Modifier le nombre d'éléments :

```
v.resize( 25 );
```


`list` et `vector` : lequel choisir ?

- L'implémentation sous-jacente est différente :
  - `vector` : bloc mémoire contigu,
  - `list` : liste doublement chainée.
- Différence principale pour l'utilisateur :
  - `vector` : accès direct à un élément, via la méthode at().
  - `list` : liste chainée, **pas d'accès direct**
    ⇒ on doit passer par un **itérateur**.
- Le choix du conteneur à une incidence sur le temps d'accès aux données.
  - Insertion d'un élément au milieu des autres :
    - `list` : temps constant,
    - `vector` : temps proportionnel au nombre d'éléments.
  - Accès à un élément arbitraire :
    - `list` : temps proportionnel au nombre d'éléments (parcours de tous les éléments un par un),
    - `vector` : accès direct ⇒ temps constant.

2 - Itérateurs : introduction

- Un itérateur est un objet qui permet de parcourir un conteneur.
- Est homogène à un pointeur :
  - Pointe sur une des valeurs (un des éléments du conteneur).
  - Peut être incrémenté (`it++`).
  - Peut être **déréférencé** :
    si `it` désigne un itérateur sur une liste de points, `*it` désignera un point.
- Création et initialisation d'un itérateur :

```
list<POINT> li; // creation liste de points
list<POINT>::iterator it; // création de l'itérateur 'it'
it = li.begin();  // initialisation itérateur sur le
                  // premier élément de la liste
```

- Nombre d'élements entre deux itérateurs :

```
int n = std::distance( it1, it2 );
```


Itérateur : utilisation

- Parcours d'une liste :

```
list<POINTS> li;          // création de la liste 'li'
list<POINTS>::iterator it; // création de l'itérateur 'it'

POINT pt;
for( it = li.begin(); it != li.end(); it++ )
{
   pt = *it;    // copie de l'élément pointé par 'it' dans 'pt'
   pt.Affiche(); // Affichage du point 'pt'
}
```

- On peut se passer de la variable intermédiaire `pt` en écrivant :

```
for( it = li.begin(); it != li.end(); it++ )
   it->Affiche();
```


3 - Algorithmes

- Implémentés sous forme de **fonctions** qui sont déclarées dans le fichier d'en-tête `<algorithm>`
- Ces algorithmes vont éventuellement **modifier, comparer, ...** les élements contenus.
  ⇒ Il sera **nécessaire** que les opérateurs correspondant (==, =, <) de la classe manipulée soit correctement définis.
- Le compilateur génère une implémentation par défaut pour l'opérateur de copie (=)
  ⇒ effectue une copie octet par octet (suffisant pour les classes triviales).
- pour les opérateurs de comparaison (== et <), le compilateur n'en génère pas par défaut, il faudra le définir explicitement.

Algorithmes : exemples

- Comptage d'éléments ayant une certaine valeur :

```
list<POINTS> li;
 ... // ici, on remplit la liste avec des points
POINT pt0(10,10);
// renvoie le nb de points égaux à pt0 dans la liste
int n = std::count( li.begin(), li.end(), pt0 );
```

  (fonctionne de façon similaire avec un `vector`)
- Tri d'un conteneur a (de type `list` ou `vector`) :

```
std::sort( a.begin(), a.end() );
```

- Recherche du plus petit élément d'une liste de POINT `li` :

```
list<POINT>::iterator pos_min;
pos_min = std::min_element( li.begin(), li.end() );
POINT pt_min = *pos_min;
```

  ou en une ligne (en se passant de l'itérateur) :

```
POINT pt_min = *std::min_element( li.begin(), li.end() );
```

Algorithmes : exemples 2

- Remplissage d'un `vector` vec avec une valeur v0 :

```
std::fill( vec.begin(), vec.end(), v0 );
```

- Remarque : en spécifiant correctement les itérateurs, on peut agir sur une partie du conteneur
- Exemple : on veut remplir la 1ère moitié d'un vecteur d'entier avec une nouvelle valeur :

```
// creation d'un vector de 20 entiers , initialisés à 10
std::vector<int> vv( 20, 10 );
// création d'un itérateur qui pointe au milieu
std::vector<int>::iterator it_milieu = vv.begin()+vv.size()/2;
// remplissage de la première moitié avec la valeur 5
std::fill( vv.begin(), it_milieu, 5 );
```


Algorithmes : conclusion

- De part leur **généricité** très forte, ces fonctions sont parfois délicates à mettre en oeuvre. Elles donnent cependant une très grande expressivité au code.
- Pour une référence complète :
  - `http://en.cppreference.com/w/cpp/algorithm`
  - `http://www.cplusplus.com/reference/algorithm/`

Fin du cours

- On a pas tout vu :
  Ce cours n'a été qu'une **introduction** à la POO avec C++.
- C++ : reste à voir :
  - mécanisme des exceptions,
  - patrons de classe (*template*),
  - héritage multiple,
  - STL : utilisation avancée, conteneurs associatifs,
  - espaces de noms,
  - constructeur de recopie,
  - qualificatif `const`,
  - foncteurs (classes définissant l'opérateur `()`)
  - etc.

  $\Rightarrow$ Un langage, riche, puissant, ... mais difficile !
- Pour une référence sur la syntaxe :
  `http://www.cppreference.com`
- POO : U.M.L. !