Analyse et programmation orientée objet (C++)

Séance 1

Bases POO – C++

Abstraction et Encapsulation

Un intérêt de l'encapsulation est que cela permet d'**abstraire** :

En plus du regroupement des données et des traitements relatifs à une entité, l'encapsulation permet en effet de définir deux **niveaux de perception** :

- Le niveau externe : partie « visible » (par les programmeurs-utilisateurs) de l'objet :
  - **prototype** des méthodes et attributs accessibles hors de l'objet
  - ☞ c'est l'**interface** de l'objet avec l'extérieur résultat du processus d'*abstration*
- Le niveau interne : (détails d')**implémentation** de l'objet
  - méthodes et attributs accessibles uniquement depuis l'intérieur de l'objet (ou l'intérieur d'objets similaires)
  - **définition** de l'ensemble des méthodes de l'objet
  - ☞ c'est le **corps** de l'objet

Encapsulation et Interface

Il y a donc deux facettes à l'encapsulation :

1. regroupement de tout ce qui caractérise l'objet : données (attributs) **et** traitements (méthodes)
2. isolement et dissimulation des détails d'implémentation
   Interface = ce que le programmeur-utilisateur (hors de l'objet) peut utiliser
   - ☞ Concentration sur les attributs/méthodes concernant l'objet (*abstraction*)

Exemple : L'interface d'une voiture

- Volant, accélérateur, pédale de freins, etc.
- Tout ce qu'il faut savoir pour la conduire (mais pas la réparer ! ni comprendre comment ça marche)
- L'interface ne change pas, même si l'on change de moteur... ...et même si on change de voiture (dans une certaine mesure) : abstraction de la notion de voiture (en tant qu'« objet à conduire »)

Pourquoi abstraire/encapsuler ?

L'intérêt de regrouper les traitements et les données conceptuellement reliées est de permettre une *meilleure visibilité* et une meilleure cohérence au programme, d'offrir une plus grande modularité.

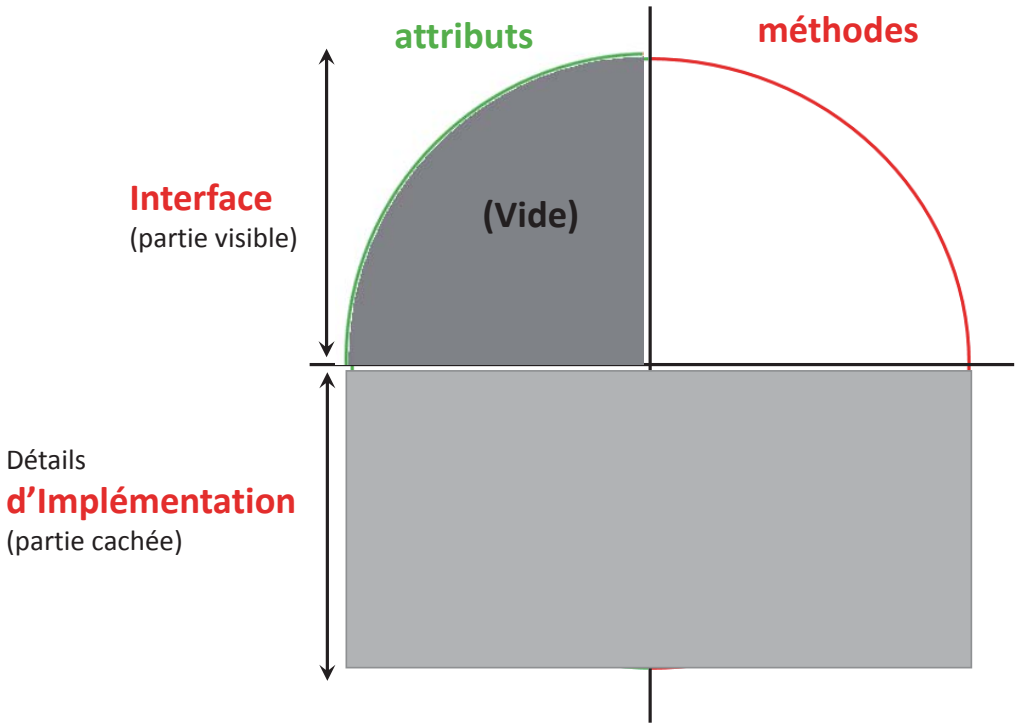
L'intérêt de séparer les niveaux interne et externe est de donner un **cadre** plus **rigoureux** à l'utilisation des objets utilisés dans un programme

Les objets ne peuvent être utilisés qu'au travers de leur interfaces (niveau externe)
et donc les éventuelles **modifications** de la structure interne restent **invisibles** à l'extérieur

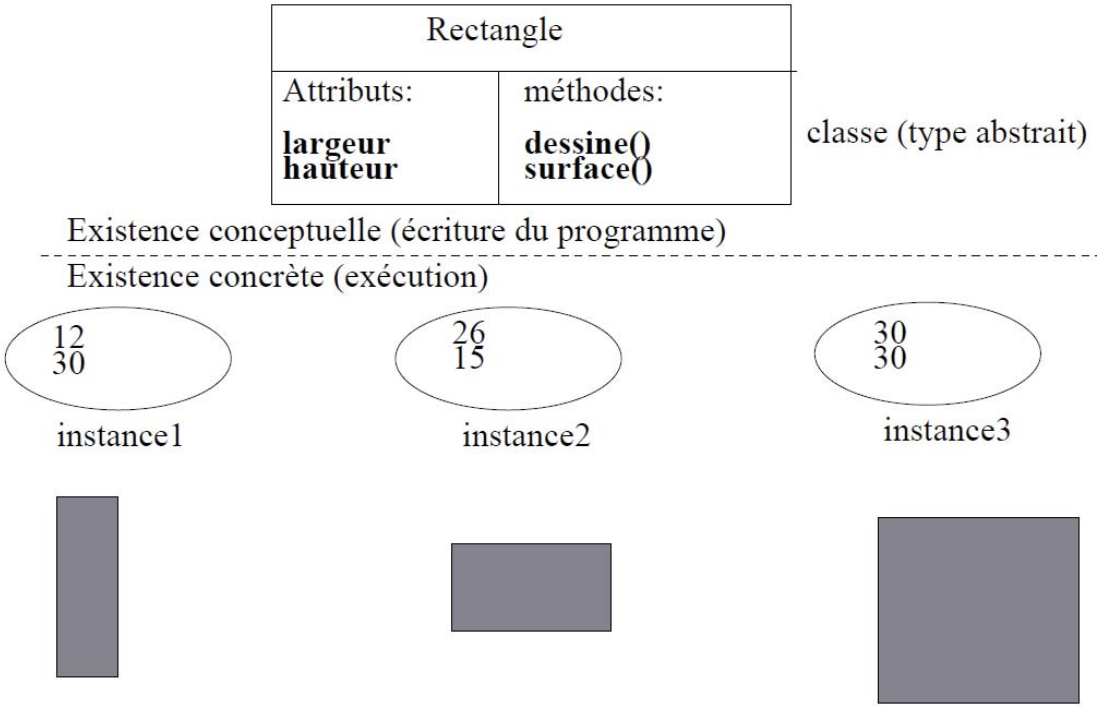
(même idée que la séparation prototype/définition d'une fonction)

Règle : (masquage) les attributs d'un objet ne doivent pas être accessibles depuis l'extérieur, mais uniquement par des méthodes.

Synthèse



Classes v.s. instances



Les classes en C++

La syntaxe de la déclaration des attributs est la même que celle des champs d'une structure.

```
type nom_attribut ;
```

Exemple : les attributs `hauteur` et `largeur`, de type `double`, de la classe `Rectangle` pourront être déclarés par :

```
class Rectangle {
    double hauteur;
    double largeur;
};
```

Déclaration des attributs

En C++ une classe se déclare par le mot-clé `class`.

Exemple :

```
class Rectangle {
    ...
}; // ne pas oublier le ;
```

La déclaration d'une instance d'une classe se fait de la façon similaire à la déclaration d'une variable classique :

```
nom_classe nom_instance ;
```

Exemple :

```
Rectangle rect1;
```

déclare une instance `rect1` de la classe `Rectangle`.

Déclaration des attributs

L'accès aux valeurs des attributs d'une instance de nom `nom_instance` se fait comme pour accéder aux champs d'une structure :

```
nom_instance.nom_attribut
```

Exemple : la valeur de l'attribut `hauteur` d'une instance `rect1` de la classe `Rectangle` sera référencée par l'expression :

```
rect1.hauteur
```

Déclaration des méthodes

La syntaxe de la définition des méthodes d'une classe est la syntaxe normale de définition des fonctions :

```
type_retour nom_methode (type_arg1 nom_arg1, ...) {
  // corps de la méthode
  ...
}
```

sauf qu'on a **pas besoin de passer les attributs** de la classe comme arguments aux méthodes de cette classe

Exemple : une méthode `surface()` de la classe `Rectangle` pourrait être définie par :

```
class Rectangle {
  ...
  double surface() {
    return (hauteur * largeur);
  }
};
```

Déclaration des méthodes

A noter que ce n'est pas parce qu'on n'a pas besoin de passer les valeurs des attributs de la classe comme arguments aux méthodes de cette classe, que les méthodes n'ont jamais d'arguments.

Les méthodes **peuvent très bien avoir des arguments** : ceux qui sont nécessaires (et donc **extérieurs à l'instance**) pour exécuter la méthode en question !
Exemple :

```
class Couleur { ... };

class FigureColoree
{
        //...
        void colorie(Couleur c) { ... }
};

FigureColoree une_figure;
Couleur rouge;
//...
une_figure.colorie(rouge);
//...
```

Actions et Prédicats

En C++ on peut distinguer les méthodes qui modifient l'état de l'objet (« **actions** ») de celles qui ne changent rien à l'objet (« **prédicats** »).

On peut pour cela ajouter le mot `const` **après** la liste des arguments de la méthode :

```
type_retour nom_methode (type_arg1 nom_arg1, ...) const
```

Exemple :

```
class Rectangle {
  ...
  double surface() const {
    return (hauteur * largeur);
  }
  ...
};
```

Si jamais vous déclarez comme prédicat (`const`) une action, vous aurez à la compilation le message d'erreur :

```
assignment of data-member '...' in read-only structure
```

Accès aux méthodes

L'appel aux méthodes définies pour une instance de nom `nom_instance` se fait à l'aide d'expressions de la forme :

```
nom_instance.nom_methode(val_arg1,...)
```

Exemple : la méthode

```
void surface() const;
```

définie pour la classe `Rectangle` peut être appelée pour une instance `rect1` de cette classe par :

```
rect1.surface()
```

Autre exemple :

```
une_figure.colorie(rouge);
```

Portée des attributs

Remarque : Les attributs d'une classe constituent des variables directement accessibles dans toutes les méthodes de la classes (*i.e.* des variables globales à la classe). Il n'est donc **pas nécessaire de les passer comme arguments des méthodes**.

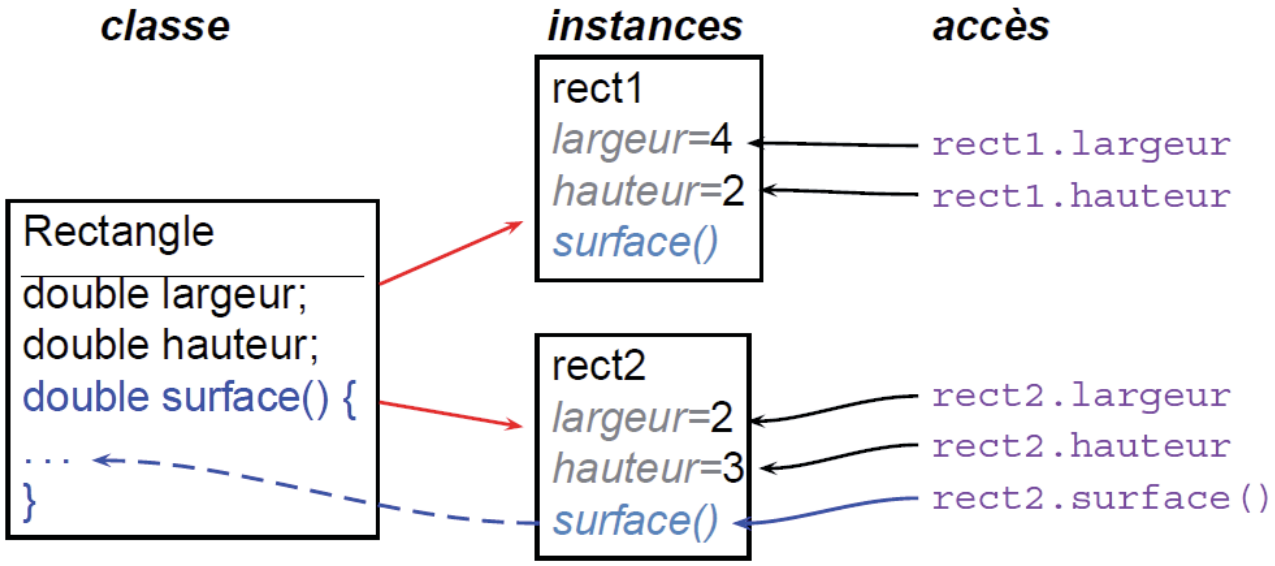
Par exemple, dans toutes les méthodes de la classe `Rectangle`, l'identificateur `hauteur` (resp. `largeur`) fait donc a priori référence à la valeur de l'attribut `hauteur` (resp. `largeur`) de la classe.

Exemple : définition d'une méthode `surface` pour la classe `Rectangle`

```
class Rectangle {
   double surface() const {
     return (hauteur * largeur);
   }
   ...
};
```

Accès aux attributs et méthodes

Chaque instance a ses propres attributs : aucun risque de confusion d'une instance à une autre.



Encapsulation et interface

Tout ce qu'il n'est pas nécessaire de connaître à l'extérieur d'un objet devrait être dans le corps de l'objet et identifié par le mot clé `private` :

```
class Rectangle {
        double surface() const { ... }
private: // ICI
        double hauteur;
        double largeur;
}
```

Attribut d'instance **privée** = inaccessible depuis l'extérieur de la classe. C'est également valable pour les méthodes.

Erreur de compilation si référence à un(e) attribut/méthode d'instance privée :

```
Rectangle.cc:16: 'double Rectangle::hauteur' is private
```

Note : Si aucun droit d'accès n'est précisé, c'est `private` par défaut.

Encapsulation et interface

À l'inverse, l'interface, qui est accessible de l'extérieur, se déclare avec le mot-clé `public:`

```
class Rectangle {
public: // accessible partout
    double surface() const { ... }
private:
...
};
```

Dans la plupart des cas :

- **Privé** :
  - Tous les attributs
  - La plupart des méthodes
- **Publique** :
  - Quelques méthodes bien choisies (interface)

Méthodes «get» et «set» (« accesseurs » et « manipulateurs »)

- Tous les attributs sont privés ?
  - Et si on a besoin de les utiliser depuis l'extérieur de la classe ?!
- Si le programmeur *le juge utile*, il **inclut les méthodes publiques nécessaires** ...
  1. Manipulateurs (« méthodes set ») :
     - Modification
     - Affectation de l'argument à une variable d'instance précise

```
void setHauteur(double h) { hauteur = h;}

void setLargeur(double L) { largeur = L;}
```

  2. Accesseurs (« méthodes get ») :
     - Consultation
     - Retour de la valeur d'une variable d'instance précise

```
double getHauteur() const { return hauteur;}

double getLargeur() const { return largeur;}
```

Masquage (shadowing)

masquage = un identificateur « cache » un autre identificateur

Situation typique : (le nom d')un paramètre cache un (nom d')attribut

```
void setHauteur(double hauteur) {
    hauteur = hauteur;  // Hmm.... pas terrible !
}
```

Masquage

Si, dans une méthode, un attribut est **masqué** alors la valeur de l'attribut peut quand même être référencée à l'aide du mot réservé `this`.

`this` est un **pointeur sur l'instance courante**

`this` ≃ « mon adresse »

Syntaxe pour spécifier un attribut en cas d'ambiguïté :

```
this->nom_attribut
```

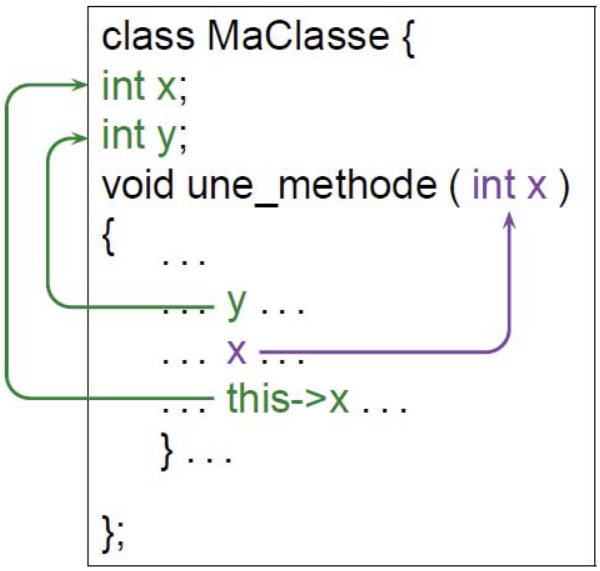
Exemple :

```
void setHauteur(double hauteur) {
    this->hauteur = hauteur;  // fonctionne !
}
```

L'utilisation de `this` est obligatoire dans les situations de **masquage**
(mais évitez ces situations !)

Portée des attributs

La portée des attributs dans la définition des méthodes peut être résumée dans le schéma suivant :



Un exemple complet de classe

```
#include <iostream>
using namespace std;

// definition de la classe
class Rectangle {
public:
  // definition des methodes
  double surface() const { return (hauteur * largeur); }
  double getHauteur() const { return hauteur; }
  double getLargeur() const { return largeur; }
  void setHauteur(double hauteur) { this->hauteur = hauteur; }
  void setLargeur(double largeur) { this->largeur = largeur; }

private:
  // declaration des attributs
  double hauteur;
  double largeur;
};
```

Un exemple complet de classe

```
//utilisation de la classe

int main() {
  Rectangle rect;
  double lu;
  cout << "Quelle hauteur? "; cin >> lu;
  rect.setHauteur(lu);
  cout << "Quelle largeur? "; cin >> lu;
  rect.setLargeur(lu);

  cout << "surface = " << rect.surface() << endl;

  return 0;
}
```

Opérateur ::

Il est possible d'écrire les définitions des méthodes à l'extérieur de la déclaration de la classe

☞ **meilleure lisibilité du code**, **modularisation**

Pour relier la définition d'une méthode à la classe pour laquelle elle est définie, il suffit d'utiliser l'opérateur `::` de résolution de portée :

- La déclaration de la classe contient les *prototypes des méthodes*
- les définitions correspondantes spécifiées à l'extérieur de la déclaration de la classe se font sous la forme :

```
typeRetour NomClasse::nomFonction(arg1, arg2, ...)
{...}
```

Opérateur ::

Par exemple, la déclaration de la classe `Rectangle` pourrait être (dans `rectangle.h`) :

```
class Rectangle
{
public:
        // prototypes des methodes
        double surface() const;

        double hauteur() const;
        double largeur() const;

        void hauteur(double);
        void largeur(double);

        // declaration des attributs
private:
        double hauteur_;
        double largeur_;
};
```

Opérateur ::

Accompagné des définitions « externes » des méthodes (dans `rectangle.cc`) :

```
double Rectangle::surface() const
{
   return hauteur_ * largeur_;
}

double Rectangle::hauteur() const
{
    return hauteur_;
}

... // idem pour largeur

void Rectangle::hauteur(double h)
{
    hauteur_ = h;
}
... // idem pour largeur
```

Objets et Classes en C++

`class MaClasse { ... };` déclare une classe.
`MaClasse obj1;` déclare une instance (objet) de la classe `MaClasse`

Les attributs d'une classe se déclarent comme des champs d'une structure : `class MaClasse { ... type attribut; ... };`

Les méthodes d'une classe se déclarent comme des fonctions, mais dans la classe elle-même :

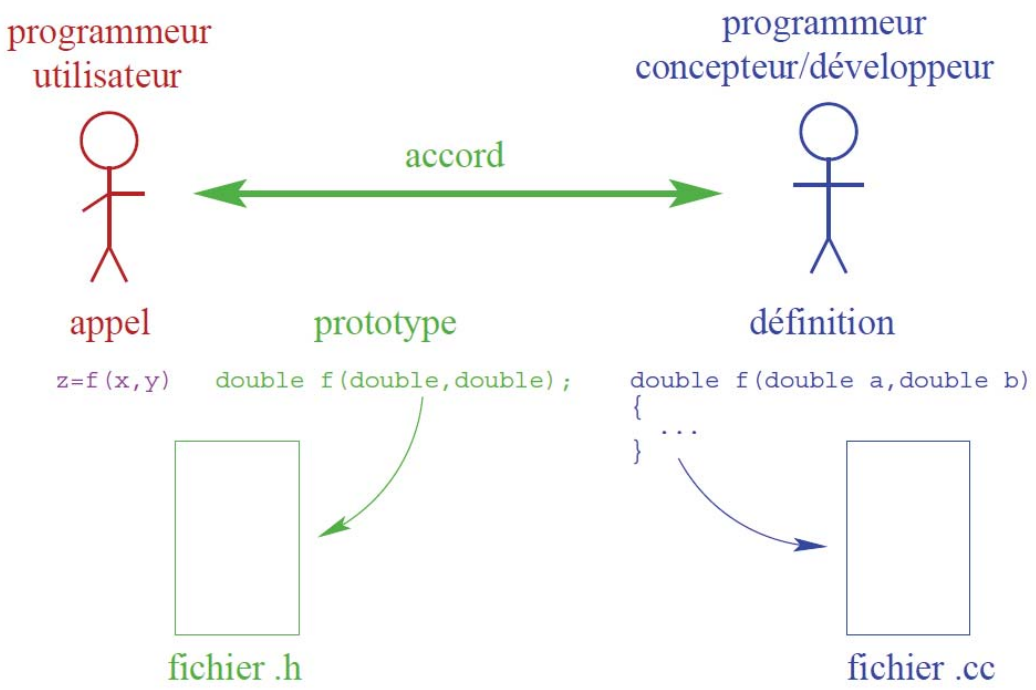
```
class MaClasse { ... type methode(type1 arg1, ...); ... };
```

Encapsulation et interface :

```
class MaClasse {
private: // attributs et methodes privees
...
public: // interface : attributs et methodes publiques
...
};
```

L'attribut particulier `this` est un pointeur sur l'instance courante de la classe. Exemple d'utilisation : `this->monattribut`

Compilation séparée



Compilation séparée

De ce fait, il est nécessaire (en conception modulaire) de séparer ces parties en deux fichiers :

- les fichiers de déclaration (fichiers « *headers* »), avec une extension `.h`).

  Ce sont ces fichiers qu'on inclut en début de programme par la commande `#include`

- les fichiers de définitions (fichiers sources, avec une extension `.cc`)

  Ce sont ces fichiers que l'on compile pour créer du code exécutable.

Analyse et programmation orientée objet (C++)

Séance 2

Constructeur/Destructeur

Recap

Lors du cours précédent, nous avons vu comment déclarer une classe (d'objets) :

On peut par exemple déclarer une instance de la classe `Rectangle` :

```
Rectangle rect;
```

Une fois que l'on a fait ce genre de déclaration, comment peut-on initialiser les attributs de `rect` (i.e. leur attribuer des valeurs) ?

Initialisation des attributs

Première solution : **affecter individuellement une valeur** à chaque attribut

```
Rectangle rect;
double lu;
cout << "Quelle hauteur? "; cin >> lu;
rect.setHauteur(lu);
cout << "Quelle largeur? "; cin >> lu;
rect.setLargeur(lu);
```

Ceci est une *mauvaise solution* dans le cas général :

- si le nombre d'attributs est élevé, ou si le nombre d'objets créés est grand (rectangles `r[0]`, `r[1]`, ..., `r[99]`) elle est fastidieuse et source d'erreurs (oublis)
- elle implique que tous les attributs fassent partie de l'interface (`public`) ou soient assortis d'un manipulateur (`set`)

Initialisation des attributs

Deuxième solution : définir une **méthode dédiée à l'initialisation** des attributs

```
class Rectangle {
private:
        double hauteur;
        double largeur;
public:
        // Methode d'initialisation !
        void init(double h, double L) {
                hauteur = h; largeur = L;}
}
        ...
        };
```

En fait, C++ fait déjà le travail pour vous en fournissant des méthodes particulières appelées **constructeurs**

Un constructeur réalise toutes les opérations requises en « début de vie » de l'objet.

Les constructeurs

Un constructeur est une méthode :

- invoquée *automatiquement* lors de la déclaration d'un objet (instanciation d'une classe)
- assurant *l'initialisation des attributs*.

Syntaxe de base :

```
NomClasse(liste_arguments)
{
 /* initialisation des attributs
    utilisant  liste_arguments */
}
```

Exemple (à améliorer par la suite) :

```
Rectangle(double h, double L) {
   hauteur = h;
   largeur = L;
 }
```

Les constructeurs

Les constructeurs sont donc des méthodes (presque) comme les autres

- que l'on peut surcharger
  Une classe peut donc avoir **plusieurs constructeurs**, pour peu que leur liste d'arguments soient différentes ;
- aux arguments desquels on peut donner des valeurs par défaut
  (exemple dans la suite).

Ce qu'il faut noter de particulier sur la syntaxe des constructeurs :

- pas d'indication de type de retour (pas même `void`)
- doit avoir le même nom que la classe pour laquelle ils sont définis

Initialisation par constructeur

La *déclaration avec initialisation* d'un objet se fait comme pour une variable ordinaire.

Syntaxe :

```
NomClasse instance(valarg1, ..., valargN);
```

où `valarg1, ..., valargN` sont les valeurs des arguments du constructeur.

Exemple :

```
Rectangle r1(18.0, 5.3); // invocation du constructeur
```

Construction des attributs

Que se passe-t-il si les attributs sont eux-mêmes des objets ?

Exemple :

```
class RectangleColore {
private:
        Rectangle rectangle; // OBJET!
        Couleur couleur;
        //...
};
```

Le constructeur d'un `rectangleColore` devrait faire appel au constructeur de `Rectangle`, ce qui n'est pas possible directement. On peut alors imaginer passer par une instance anonyme intermédiaire :

```
RectangleColore(double h, double L, Couleur c) {
        rectangle = Rectangle(h, L); // Aie !
        couleur = c;
}
```

mais c'est une très **mauvaise** solution !

☞ il faut initialiser directement les attributs en faisant appel à leurs constructeurs (propres) !

Appel aux constructeurs des attributs

Un constructeur devrait normalement contenir une section d'appel aux constructeurs des attributs.

Ceci est également valable pour l'initialisation des attributs de type de base.

Il s'agit de la « section deux-points » des constructeurs :

Syntaxe générale :

```
NomClass(liste_arguments)
// section d'initialisation par constructeurs
: attribut1(...), //appel au constructeur de attribut1
  ....
  attributN(...)
{
 // autres opérations
}
```

Exemple :

```
RectangleColore(double h, double L, Couleur c)
        : rectangle(h, L), couleur(c) // Mieux!
{ }
```

Appel aux constructeurs des attributs

Cette section, qui est introduite par `:`, est optionnelle mais **recommandée**.

Par ailleurs :

- les attributs non-initialisés dans cette section
  - prennent une valeur par défaut (on verra comment un peu plus loin) si ce sont des objets ;
  - restent indéfinis s'ils sont de type de base ;
- les attributs initialisés dans cette section peuvent (bien sûr) être changés dans le corps du constructeur.

Exemple :

```
Rectangle(double h, double L)
        : hauteur(h) //initialisation
{
        // largeur a une valeur indefinie jusqu'ici
        largeur = 2.0 * L + h; // par exemple...
    // la valeur de largeur est definie a partir d'ici
}
```

Constructeur par défaut

Un constructeur par défaut est un constructeur qui **n'a pas d'argument** ou dont **tous** les arguments ont des **valeurs par défaut**.

Exemples :

```
// Le constructeur par defaut
Rectangle() : hauteur(1.0), largeur(2.0) {}
// 2eme constructeur
Rectangle(double c) : hauteur(c), largeur(2.0*c) {}
// 3eme constructeur
Rectangle(double h, double L) : hauteur(h), largeur(L) {}
```

Autre façon de faire : regrouper les 2 premiers constructeurs en utilisant les valeurs par défaut des arguments :

```
// 2 constructeurs dont le constructeur par defaut
  Rectangle(double c = 1.0) : hauteur(c), largeur(2.0*c)
 { }
```

Constructeur par défaut par défaut

Si aucun constructeur n'est spécifié, le compilateur *génère automatiquement* une **version minimale du constructeur par défaut**

(c'est donc un constructeur par défaut par défaut)

qui :

- appelle le constructeur par défaut des attributs objets ;
- laisse non initialisés les attributs de type de base.

Dès qu'**au moins un constructeur a été spécifié**, ce constructeur par défaut par défaut *n'est plus fourni*.

Si donc on spécifie un constructeur sans spécifier de constructeur par défaut, on ne peut plus construire d'objet de cette classe sans les initialiser (ce qui est voulu !) puisqu'il n'y a plus de constructeur par défaut.

C++11 ...mais on peut le rajouter si on veut (voir plus loin).

Constructeur par défaut : exemples

Comparons les quatre codes suivants :

A :

```
class Rectangle {
private:
  double h; double L;
  // suite ...
};
```

B :

```
class Rectangle {
private:
  double h; double L;
public:
  Rectangle()
   : h(0.0), L(0.0)
   { }
  // suite ...
};
```

C :

```
class Rectangle {
private:
  double h; double L;
public:
Rectangle(double h=0.0,
          double L=0.0)
   : h(h), L(L)
   { }
  // suite ...
};
```

D :

```
class Rectangle {
private:
  double h; double L;
public:
Rectangle(double h,
          double L)
   : h(h), L(L)
   { }
  // suite ...
};
```

Constructeur par défaut : exemples

	constructeur par défaut	`Rectangle r;`	`Rectangle r(1,2);`
(A)	constructeur par défaut par défaut	? ?	Illicite !

```
class Rectangle {
private:
  double h; double L;
  // suite ...
};
```

Constructeur par défaut : exemples

	constructeur par défaut	`Rectangle r;`	`Rectangle r(1,2);`
(A)	constructeur par défaut par défaut	? ?	Illicite !
(B)	constructeur par défaut explicite-ment déclaré	0 0	Illicite !

```
class Rectangle {
private:
  double h; double L;
public:
  Rectangle()
   : h(0.0), L(0.0)
   { }
  // suite ...
};
```

Constructeur par défaut : exemples

```
class Rectangle {
private:
  double h; double L;
public:
Rectangle(double h=0.0,
          double L=0.0)
   : h(h), L(L)
   { }
  // suite ...
};
```

	constructeur par défaut	`Rectangle r;`	`Rectangle r(1,2);`
(A)	constructeur par défaut par défaut	? ?	Illicite !
(B)	constructeur par défaut explicite-ment déclaré	0 0	Illicite !
(C)	un des **trois** constructeurs est par défaut	0 0	1 2

Constructeur par défaut : exemples

```
class Rectangle {
private:
  double h; double L;
public:
Rectangle(double h,
          double L)
   : h(h), L(L)
   { }
  // suite ...
};
```

	constructeur par défaut	`Rectangle r;`	`Rectangle r(1,2);`
(A)	constructeur par défaut par défaut	? ?	Illicite !
(B)	constructeur par défaut explicite-ment déclaré	0 0	Illicite !
(C)	un des **trois** constructeurs est par défaut	0 0	1 2
(D)	pas de construc-teur par défaut	Illicite !	1 2

C++11 Remettre le constructeur par défaut par défaut

Dès qu'au moins un constructeur a été spécifié, ce constructeur par défaut par défaut n'est plus fourni.
C'est très bien si c'est vraiment ce que l'on veut (i.e. forcer les utilisateurs de la classe à utiliser nos constructeurs).

Mais si l'on veut quand même avoir le constructeur par défaut par défaut, on peut le re-demander en écrivant :

```
NomClasse() = default;
```

Exemple (modification du cas (D) précédent) :

```
class Rectangle {
public:
        // on remet le constructeur par defaut
        // par defaut, mais bof...
        Rectangle() = default;

        Rectangle(double h, double L) : h(h), L(L) {}
        // suite ...
};
```

C++11 méthodes default et delete

Ce que l'on à fait précédemment (`= default`) pour le constructeur par défaut se généralise :

- à la supression de méthode, via la syntaxe `= delete;`
- à toute méthode (pour laquelle cela est pertinent).

Exemple :

```
class Demo {
public:
  double pas_d_int(double x) { ... }
  double pas_d_int(int) = delete;
};
```

Constructeur de copie

C++ vous fournit également un moyen de créer la copie d'une instance : le *constructeur de copie*

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

Syntaxe :

```
NomClasse(NomClasse const& obj)
:...
{...}
```

Exemple :

```
Rectangle(Rectangle const& obj)
: hauteur(obj.hauteur), largeur(obj.largeur)
{}
```

Constructeur de copie

L'invocation du constructeur de copie se fait par une instruction de la forme :

```
Rectangle r1(12.3, 24.5);
Rectangle r2(r1);
```

`r1` et `r2` sont deux *instances* **distinctes** mais ayant des mêmes valeurs pour leurs attributs (à ce moment là du programme).

Autre exemple de copie (appel au constructeur de copie) :

```
double f(Rectangle r);
//...
x = f(r1);
```

Constructeur de copie

- Un constructeur de copie est *automatiquement généré* par le compilateur s'il n'est pas explicitement défini
- Ce constructeur opère une initialisation *membre à membre* des attributs (si l'attribut est un objet le constructeur de cet objet est invoqué)
  ☞ **copie de surface**
- Cette copie de surface suffit dans la plupart des cas.

Cependant, il est parfois nécessaire de redéfinir le constructeur de copie, en particulier lorsque certains attributs sont des *pointeurs*

Destructeur

SI l'initialisation des attributs d'une instance implique la mobilisation de ressources : *fichiers, périphériques, portions de mémoire (pointeurs), etc.*

☞ il est alors important de **libérer ces ressources** après usage !

Comme pour l'initialisation, l'*invocation explicite* de méthodes de libération n'est pas satisfaisante (fastidieuse, source d'erreur, affaiblissement de l'encapsulation)

☞ C++ fourni une méthode appelée *destructeur* invoquée automatiquement en fin de vie de l'instance.

Destructeur

La syntaxe de déclaration d'un destructeur pour une classe `NomClasse` est :

```
~NomClasse() {
  // opérations (de libération)
}
```

- Le destructeur d'une classe est une méthode *sans arguments*
  ☞ **pas de surcharge possible**
- Son nom est celui de la classe, précédé du signe ~ (tilda).
- Si le destructeur n'est pas défini explicitement par le programmeur, le compilateur en génère automatiquement une version minimale.

Exemple

Supposons que l'on souhaite compter le nombre d'instances d'une classe actives à un moment donné dans un programme.

Utilisons comme compteur une variable globale de type entier :

- le constructeur incrémente le compteur
- le destructeur le décrémente

```
int long compteur(0); // Hmm....

class Rectangle {
  //...
  Rectangle(): hauteur(0.0), largeur(0.0) { //constructeur
    ++compteur; }
  ~Rectangle() { --compteur; } // destructeur
  //...
```

☞ on est obligé ici de **définir explicitement le destructeur**

Exemple

```
int
main ()
{
  //compteur=0
  Rectangle r1;
  //compteur=1
  {
    Rectangle r2;
    //compteur=2
  }
  //compteur=1
  return 0;
} //compteur=0
```

Exemple

Que se passe-il si l'on souhaite utiliser la copie d'objet ?

```
int
main ()
{
  //compteur=0
  Rectangle r1;
  //compteur=1
  {
    Rectangle r2;
    //compteur=2

    Rectangle r3 (r2);
    //compteur=??
  }
  //compteur=0
  return 0;
} // compteur=-1
```

☞ oops... la copie d'un rectangle échappe au compteur d'instance car il n'y a pas de définition explicite du constructeur de copie

Exemple

Il faudrait donc encore ajouter au code précédent, la **définition explicite du constructeur de copie** :

```
Rectangle(Rectangle const& r)
        : hauteur(r.hauteur), largeur(r.largeur)
{ ++compteur; }
```

Analyse et programmation orientée objet (C++)

Séance 3

Surcharge des opérateurs

Opérateur

Rappel : un opérateur est une opération sur un ou entre deux opérande(s) (variable(s)/expression(s)) :

opérateurs arithmétiques (`+`, `-`, `*`, `/`, ...), opérateurs logiques (`&&`, `||`, `!`), opérateurs de comparaison (`==`, `>=`, `<=`, ...), opérateur d'incrément (`++`), ...

En pratique, un appel à un opérateur est similaire à un appel de fonction.
`A Op B` se traduisant par : `A.operatorOp(B)`
et `Op A` (unaire) par : `A.operatorOp()`

Exemples :

`a + b`	est la même chose que	`a.operator+(b)`
`b + a`		`b.operator+(a)`
`a = b`		`a.operator=(b)`
`++a`		`a.operator++()`
`!a`		`a.operator!()`
`not a`		`a.operatornot()`
`cout << a`		`cout.operator<<(a)`

Surcharge ?

Rappel : surcharge de fonction

- ☞ deux fonctions ayant le même nom mais pas les mêmes arguments

Exemple :

```
int max(int, int);
double max(double, double);
```

Presque tous les opérateurs sont surchargeables (sauf, parmi ceux que vous connaissez, `::` et `.`).

Leur surcharge ne pose généralement pas plus de problèmes que la surcharge des fonctions.

La surcharge des opérateurs peut être réalisée soit à l'intérieur, soit à l'extérieur de la classe à laquelle ils s'appliquent.

Intérêt ?

Exemple avec les nombres complexes :

```
class Complexe { ... };
Complexe a, b, c;
```

plutôt que d'avoir à écrire

```
a = b.addition(c);
```

écrire

```
a = b + c;
```

est quand même plus naturel...

Surcharge des opérateurs

Pour surcharger un opérateur `Op` dans une classe `Classe`, il faut **ajouter la définition de la méthode** prédéfinie `operatorOp` dans la classe en question :

```
class Classe {
   ...
   // prototype de l'operateur Op
   type_retour operatorOp(type_argument);
   ...
};


// définition de l'operateur Op
type_retour Classe::operatorOp(type_argument) {
   ...
}
```

Surcharge des opérateurs

```
class Complexe {
  ...
  Complexe operator-(Complexe const& z2) const;
  // exemple : z3 = z1 - z2;

  void operator-=(Complexe const& z2);
  // exemple : z1 -= z2;
};

void Complexe::operator-=(Complexe const& z2) {
  x -= z2.x;
  y -= z2.y;
}

Complexe Complexe::operator-(const Complexe& z2) const {
  Complexe z3(*this); /* utilise le constructeur de copie :
                       *   copie z1 dans z3                */

  z3 -= z2; // utilise l'operateur -= redefini ci-dessus

  return z3;
}
```

Surcharge des opérateurs

```
class Polynome {
  ...
  Polynome operator-(Polynome const& q) const; // ex: r = p-q;
  Polynome& operator-=(Polynome const& q);     // ex: p -= q;
};

// un polynome a pour attribut un vecteur de coefficients p
Polynome& Polynome::operator-=(Polynome const& q) {
  while (degre() < q.degre()) p.push_back(0);
  for (unsigned int i(0); i <= q.degre(); ++i) p[i] -= q.p[i];
  simplifie();
  return *this;
}


Polynome Polynome::operator-(const Polynome& q) const {
  return Polynome(*this) -= q; /* utilise le constructeur de
                                * copie et l'operateur -=     */
}
```

Opérateur d'affectation et constructeur de copie

L'opérateur d'affectation `=` (utilisé par exemple dans `a = b`) est similaire au constructeur de copie,

sauf que le premier s'appelle lors d'une affectation et le second lors d'une initialisation (on peut donc avoir à faire des choses un tout petit peu différentes).

En général, on pourra créer une méthode (privée) unique pour les deux, par exemple `copie()`, qui est appelée à la fois par le constructeur de copie et par l'opérateur d'affectation :

```
Classe& Classe::operator=(Classe const& source) {
    if (&source != this) {
        //  Copie  effective des donnees
        copie(source);
    }
    return *this;
}
```

Surcharge externe

La surcharge **externe** est utile pour des opérateurs concernés par une classe, mais pour lesquels la classe en question n'est **pas** l'opérande de gauche.

Exemples :

1. multiplication d'un polynôme par un double
   ```
   double a;
   Polynome p, q;
   q = a * p;
   ```
   s'écrirait `a.operator*(p);`
   ce qui **n'a pas de sens** (`a` n'est pas un objet mais de type élémentaire `double`).
2. écriture sur `cout` : `cout << p`
   Il s'agit bien ici de `cout.operator<<`, mais on souhaite le surcharger dans la classe de `p` et non pas dans la classe de `cout` (`ostream`).

Surcharge externe

Dans ces cas on utilise des **opérateurs externes**, c'est-à-dire ne faisant pas partie de la classe.

Les opérateurs externes se déclarent avec un argument de plus que les opérateurs internes : la classe doit être ici explicitée.

Déclarations (hors de la classe) :

```
Polynome operator*(double, Polynome const&);
ostream& operator<<(ostream&, Polynome const&);
```

Parfois, il peut être nécessaire d'ajouter, **dans** la classe, ce même prototype, précédé du mot clé `friend` :

```
friend Polynome operator*(double, Polynome const&);
friend ostream& operator<<(ostream&, Polynome const&);
```

Le mot clé `friend` signifie que ces opérateurs, bien que ne faisant **pas** partie de la classe, peuvent avoir accès aux attributs et méthodes privés de la classe.
(mais *préférez passer par les accesseurs* (« méthodes get »))

Définitions : les définitions sont mises **hors** de la classe (et **sans** le mot clé `friend`)

Surcharge externe

```
Complexe operator*(double a, Complexe const& z) {
  return z * a; // utilisation ici de l'operateur interne
}

ostream& operator<<(ostream& out, Complexe const& z) {
  out << '(' << z.x << ", " << z.y << ')';
  return out;
}
```

ou mieux (via les accesseurs) :

```
ostream& operator<<(ostream& out, Complexe const& z) {
        // notez l'utilisation des getters
  out << '(' << z.get_x() << ", " << z.get_y() << ')';
  return out;
}
```

Surcharge externe

```
Polynome operator*(double a, Polynome const& p) {
  return p * a; // utilisation ici de l'operateur interne
}

ostream& operator<<(ostream& out, Polynome const& p) {
  // plus haut degre : pas de signe + devant
  Degre i(p.degre());
  affiche_coef(out, p.coef(i), i, false);

  // degre de N a' 0 : +a*X^i
  for (--i; i >= 0; --i) affiche_coef(out, p.coef(i), i);

  // degre 0 : afficher quand meme le 0 si rien d'autre
  if ((p.degre() == 0) && (p.coef(0) == 0.0))
    out << 0;

  return out;
}
```

Avertissement

Attention de ne pas utiliser la surcharge des opérateurs à **mauvais escient** et à veiller à les écrire avec un soin particulier

Les performances du programme peuvent en être gravement affectées par des opérateurs surchargés mal écrits.

En effet, l'utilisation inconsidérée des opérateurs peut conduire à un grand nombre de copies d'objets,

Utiliser des références quand cela est approprié

Avertissement

Exemple : comparez

```
Polynome Polynome::operator-=(Polynome q) {
   Polynome local;
   local = this;
   while (local.degre() < q.degre()) local.ajoute(0);
   for (unsigned int i(0); i <= q.degre(); ++i)
       local[i] -= q[i];
   local.simplifie();
   return local;
}
```

(plein de copies inutiles !)
et

```
Polynome& Polynome::operator-=(Polynome const& q) {
   while (degre() < q.degre()) p.push_back(0);
   for (unsigned int i(0); i <= q.degre(); ++i)
       p[i] -= q.p[i];
   simplifie();

   return *this;
}
```

Quelle surcharge en pratique ?

Dans votre pratique du C++, vous pouvez, en fonction de votre niveau (et des contraintes de développement de votre code), choisir un degré divers de complexité dans la surcharge des opérateurs :

① ne pas du tout faire de surcharge des opérateurs ;

② surcharger simplement les opérateurs arithmétiques de base (+, -, ...) sans leur version « auto-affectation » (+=, -=, ...) ; libre à vous ici de choisir ou non la surcharge (externe) de l'opérateur d'affichage (<<) ;

③ surcharger les opérateurs en utilisant leur version « auto-affectation », mais sans valeur de retour pour celles-ci :

```
void operator+=(Bidule const&);
```

④ faire la surcharge complète comme présentée précédemment, avec gestion de la valeur de retour des opérateurs d'« auto-affectation » :

```
Bidule& operator+=(Bidule const&);
```

Exemples de surcharges usuelles d'opérateurs

```
bool operator==(Classe const&) const; // ex: p == q
bool operator<(Classe const&) const;  // ex: p < q

Classe& operator=(Classe const&);   // ex: p = q

Classe& operator+=(Classe const&);  // ex: p += q
Classe& operator-=(Classe const&);

Classe& operator++();               // ex: ++p
Classe& operator++(int useless);    // ex: p++

Classe& operator*=(autre_type const); // ex: p *= x;

Classe operator+(Classe const&) const; // r = p + q
Classe operator-(Classe const&) const;

Classe operator-() const; // ex: q = -p;

[friend] ostream& operator<<(ostream&, Classe const&); // ex: cout << p;

[friend] Classe operator*(autre_type, Classe const&); // ex: q = x * p;
```

Liste des opérateurs pouvant être surchargés

NOTES :

1. Plusieurs d'entre eux n'ont pas été et ne seront pas présentés dans ce cours (hors cadre). Le but de cette liste est juste de vous donner tous les symboles possibles pour la surcharge d'opérateurs.
2. Évitez de «trop» changer le sens (/la sémantique) d'un opérateur lorsque vous le surchargez.
3. Évitez absolument de surcharger

```
,
->          ->*
new         new[]
delete      delete[]
```

(à moins de savoir *exactement* ce que vous faites).

Liste des opérateurs pouvant être surchargés

=	+	-	*	/	^	%
==	+=	-=	*=	/=	^=	%=
<	<=	>	>=	<<	>>	<<=
>>=	++	--	&	|	!	&=
|=	!=	,	->	->*	[]	()
~	&&	||	xor	xor_eq	and	and_eq
or	or_eq	not	not_eq	bitand	bitor	compl
new	new[]	delete	delete[]			

Surcharge d'opérateurs

```
class Classe {
   ...
   type_retour operatorOp(type_argument); // prototype de l'operateur Op
   ...
};

// définition de l'operateur Op
type_retour Classe::operatorOp(type_argument) { ... }

// operateur externe
type_retour operatorOp(type_argument, Classe&) { ... }
```

Quelques exemple de prototypes :

```
bool operator==(Classe const&) const;  // ex: p == q
bool operator<(Classe const&) const;   // ex: p < q
Classe& operator=(Classe const&);      // ex: p = q
Classe& operator+=(Classe const&);     // ex: p += q
Classe& operator++();                  // ex: ++p
Classe& operator*=(const autre_type);  // ex: p *= x;
Classe operator-(Classe const&) const; // ex: r = p - q
Classe operator-() const;              // ex: q = -p;

// operateurs externes
ostream& operator<<(ostream&, Classe const&);
Classe operator*(double, Classe const&);
```

Analyse et programmation orientée objet (C++)

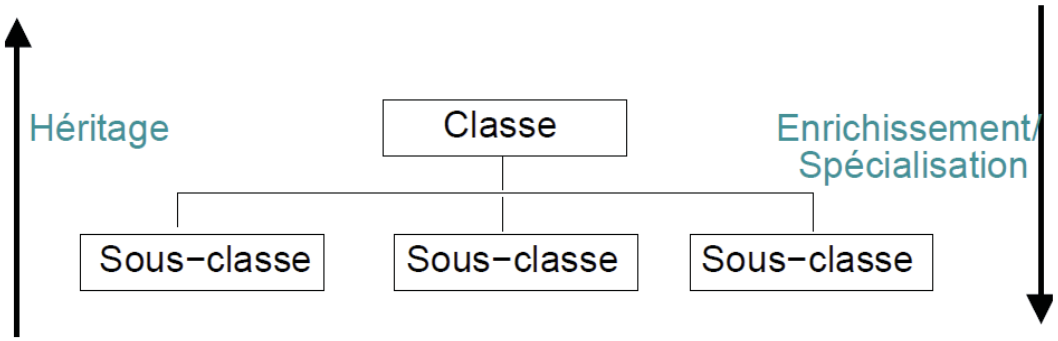
Séance 4

Héritage

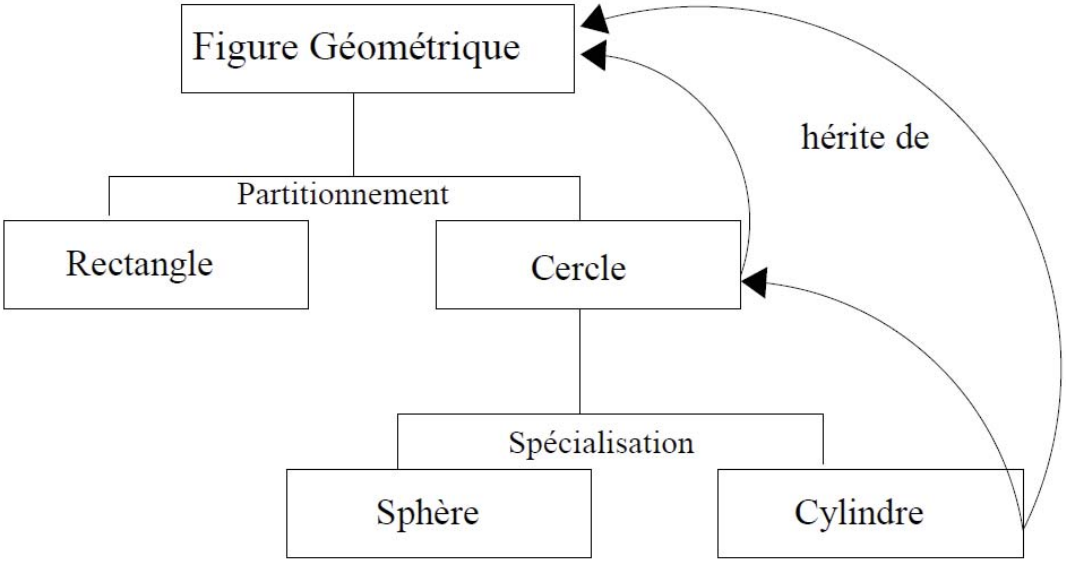
Héritage

- Après les notions **d'encapsulation** et **d'abstraction**, le troisième aspect essentiel des objets est la notion **d'héritage**
- L'héritage est une technique extrêmement efficace pour créer des classes plus spécialisées, appelées **sous-classes**, à partir de classes plus générales déjà existantes, appelées **super-classes**.

  Elle représente la relation «**est-un**»



Hiérarchie de classes un exemple



Héritage (2)

Plus précisément, lorsqu'une sous-classe *C1* est créée à partir d'une classe *C*, *C1* va **hériter** de (i.e. *recevoir*) l'ensemble :

- des attributs de *C* public
- des méthodes de *C* (sauf les constructeurs/destructeurs)
  - Les attributs et méthodes de *C* vont être disponibles pour *C1* sans que l'on ait besoin de les redéfinir explicitement dans *C1*.

De plus :

- le type est aussi hérité : *C1* **est** (aussi) **un** *C* :

  Pour un C x; et un C1 y;, on peut tout à fait faire :

  x = y; (mais bien sûr pas y = x; !!!)

Par ailleurs :

- des attributs et/ou méthodes supplémentaires peuvent être définis par la sous-classe *C1*

☞ ces membres constituent **l'enrichissement** apporté par cette sous-classe.

Héritage (3)

L'héritage permet donc :

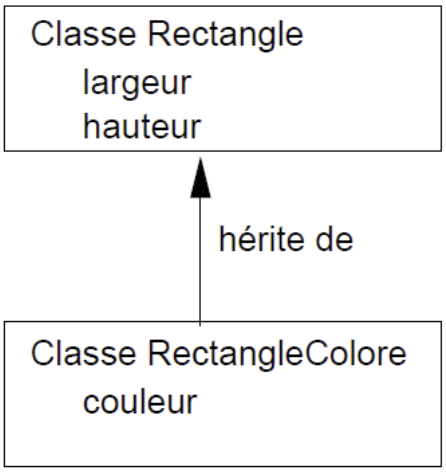
- d'**expliciter des relations** structurelles et sémantiques entre classes

- de **réduire les redondances** de description et de stockage des propriétés

Héritage (4)

- Supposons par exemple que l'on veuille étendre la classe Rectangle précédemment définie en lui ajoutant le nouvel attribut couleur.
- Une façon de procéder est de créer une nouvelle classe, par exemple RectangleColore, définie comme une sous-classe de Rectangle et contenant le nouvel attribut couleur.

☞ on évite ainsi de dupliquer inutilement du code commun aux classes Rectangle et RectangleColore !



Héritage (5)

Par **transitivité**, les instances d'une sous-classe possèdent :

- les attributs et méthodes (hors constructeurs/destructeur) de l'ensemble des classes parentes (classe parente, classe parente de la parente, etc . . .)

La notion d'**enrichissement par héritage** :

- crée un *réseau de dépendances entre classes*,
- ce réseau est organisé en une *structure arborescente* où chacun des nœuds hérite des propriétés de l'ensemble des nœuds du chemin remontant jusqu'à la racine.

- ce réseau de dépendance définit une **hiérarchie de classes**

Exemple

Définition d'une sous-classe en C++ :

Syntaxe :

```
class NomClasseEnfant : public NomClasseParente
{
        /* Déclaration des attributs et méthodes
           spécifiques à la sous-classe */
        //...
};
```

Exemple :

```
class RectangleColore : public Rectangle
{
   Couleur couleur;
   //...
};
```

Accès aux membres d'une sous-classe

Jusqu'à maintenant, l'accès aux membres (attributs et méthodes) d'une classe pouvait être:

- soit **public** : visibilité totale à l'intérieur et à l'extérieur de la classe (mot-clé public)
- soit **privé** : visibilité uniquement à l'intérieur de la classe (mot-clé private ou par défaut)

Un troisième type d'accès régit l'accès aux attributs/méthodes au sein d'une hiérarchie de classes :

- l'accès **protégé** : assure la visibilité des membres d'une classe dans les classes de sa descendance [et uniquement elles, et uniquement dans ce rôle de sous classe, (voir exemple plus loin)]. Le mot clé est «protected».

Définition des niveaux d'accès

L'ordre de définition conseillé est le suivant :

```
// lien d'heritage si necessaire seulement
class NomClasse : public NomClasseParente
{
        // par defaut : private
public:
        // attributs et methodes public
protected:
        // attributs et methodes protected
private:
        // attributs et methodes private
};
```

Mais il peut y avoir plusieurs zones publiques, protégées ou privées dans une même définition de classe.

Accès protégé

Le niveau d'accès protégé correspond à une **extension du niveau privé** aux membres des sous-classes

Exemple :

```
class Rectangle
{
public:
        Rectangle(): largeur(1.0), hauteur(2.0) {}
protected:
        double largeur; double hauteur;
};
class RectangleColore : public Rectangle
{
public:
        // on profite ici de protected
        void carre() { largeur = hauteur; }
protected:
        Couleur couleur;
};
```

Accès protégé (2)

```
class A {
//...
protected: int a;
private: int prive;
};
class B: public A {
public:
                        //...
void f(B autreB, A autreA, int x) {
            a = x;              // OK A::a est protected => acces possible
         // prive = x;          // erreur : A::prive est private
        // a += autreB.prive; // erreur (meme raison)
            a += autreB.a;      // OK : dans la meme classe (B)
       // a += autreA.a;       // INTERDIT ! : this n'est pas de la meme
                                // classe que autreA (role externe)
}  };
```

Le niveau d'accès protégé correspond à une extension du niveau privé aux membres des sous-classes... **mais uniquement dans ce rôle** (de sous-classe) pas dans le rôle de classe extérieure :

Restriction des accès lors de l'héritage

Les niveaux d'accès peuvent être **modifiés lors de l'héritage**

Syntaxe :

```
class ClasseEnfant: [accès] classeParente
{
        /* Déclaration des membres
           spécifiques à la sous-classe */
        //...
};
```

où **accès** est le mot-clé public, protected ou private. Les crochets entourant un élément [ ] indiquent qu'il est optionnel.

Les droits peuvent être conservés ou restreints, mais **jamais relachés** !

Par défaut, l'accès est privé.

Restriction des accès lors de l'héritage (2)

Récapitulatif des changements de niveaux d'accès aux membres hérités, en fonction du niveau initial et du type d'héritage :

		accès initial		
		public	protected	private
héritage	public	public	protected	pas d'accès
	protected	protected	protected	pas d'accès
	private	private	private	pas d'accès

Le type d'héritage constitue une *limite supérieure à la visibilité.*

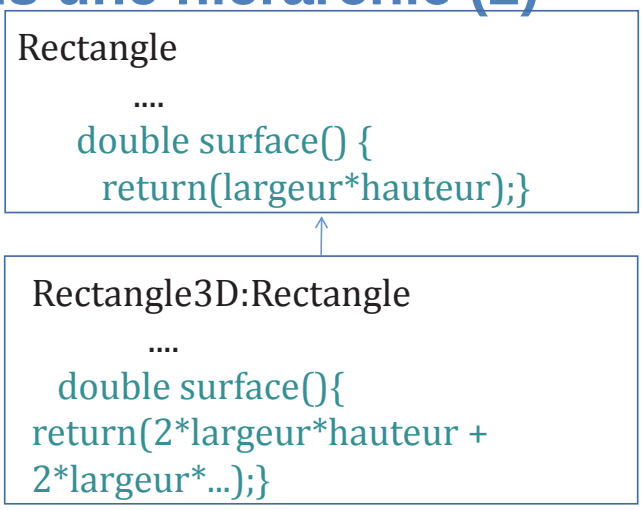
Utilisation des droits d'accès

- Membres *publics* : accessibles pour les **programmeurs utilisateurs** de la classe
- Membres *protégés* : accessibles aux **programmeurs d'extensions** par héritage de la classe
- Membres *privés* : pour le **programmeur de la classe** : structure interne, **modifiable** si nécessaire **sans répercussions** ni sur les utilisateurs ni sur les autres programmeurs.

Masquage dans une hiérarchie

- Masquage : un identificateur qui en cache un autre
- Situation possible dans une hiérarchie :
  - Même nom d'attribut/méthode utilisé sur plusieurs niveaux
  - Peu courant pour les attributs
  - Très courant et **pratique** pour les méthodes
- Exemple :
  - Rectangle3D hérite de Rectangle
  - calcul de la surface pour les Rectangle3D
    2* (largeur*hauteur) + 2*(largeur*profondeur)
    + 2*(hauteur*profondeur)
  - calcul de la surface pour tous les autres Rectangle : (largeur*hauteur)
- Faut-il re-concevoir toute la hiérarchie ?
  - Non, on ajoute simplement une méthode surface spéciale à Rectangle3D

Masquage dans une hiérarchie (2)



La méthode surface de Rectangle3D **masque** celle de Rectangle

- Un objet de type Rectangle3D n'utilisera donc **jamais** la méthode surface de la classe Rectangle
- Vocabulaire OO :
  - Méthode héritée = méthode générale, *méthode par défaut*
  - Méthode qui masque la méthode héritée = *méthode spécialisée*

Masquage dans une hiérarchie (3)

```
class Rectangle {
public:
        // les constructeurs seraient ici...
        double surface() {return (largeur*hauteur);}}
protected:
double largeur; double hauteur;
// le reste de la classe...
};
class Rectangle3D : public Rectangle {
public:
        // les constructeurs seraient ici...
        double surface() { // Masquage !
                return(2.0*(largeur*hauteur) + 2.0*(largeur*profondeur)
                        + 2.0*(hauteur*profondeur));
}
protected:
double profondeur;
// le reste de la classe...
};
```

Accès à une méthode masquée

- Il est parfois souhaitable d'accéder à une méthode/un attribut caché(e)

- Exemple :
  - surface des Rectangle3D ayant une profondeur nulle (largeur*hauteur)
  - ☞ identique au calcul de surface pour les Rectangle
- Code désiré :
  1. Objet non-Rectangle3D :
     - Méthode générale (surface de Rectangle)
  2. Objet Rectangle3D :
     - Méthode spécialisée (surface de Rectangle3D)
  3. Objet Rectangle3D de profondeur nulle :
     - D'abord la méthode spécialisée
     - Ensuite appel à la méthode générale depuis la méthode spécialisée

Accès à une méthode masquée (2)

- Pour accéder aux attributs/méthodes caché(e)s de la super-classe :
  - on utilise **l'opérateur de résolution de portée**
  - Syntaxe : NomClasse::methode ou attribut
  - Exemple : Rectangle::surface()

```
class Rectangle3D : public Rectangle {
        //... constructeurs, attributs comme avant
        double surface () {
                if (profondeur == 0.0)
                                // Acces a la methode masquee
                        return Rectangle::surface();
                else
                        return(2.0*(largeur*hauteur)
                                + 2.0*(largeur*profondeur)
                                + 2.0*(hauteur*profondeur));
        }
};
```

Constructeurs et héritage

Lors de l'instanciation d'une sous-classe, il faut initialiser :

- les attributs *propres à la sous-classe*
- les attributs *hérités des super-classes*

MAIS...

...il ne doit pas être à la charge du concepteur des sous-classes de réaliser lui-même l'*initialisation des attributs hérités*

L'**accès** à ces attributs peut notamment être **interdit** ! (private)

L'initialisation des attributs hérités doit donc se faire au niveau des classes où ils sont explicitement définis.

Solution : l'initialisation des attributs hérités doit se faire en invoquant les *constructeurs des super-classes*.

Constructeurs et héritage (2)

L'invocation du constructeur de la super-classe se fait au **début de la section d'appel aux constructeurs des attributs**.

Syntaxe :

```
SousClasse(liste d'arguments)
        : SuperClasse(Arguments),
                attribut1(valeur1),
                ...
                attributN(valeurN)
{
                // corps du constructeur
}
```

Lorsque la super-classe admet un constructeur par défaut, l'invocation explicite de ce constructeur dans la sous-classe n'est pas obligatoire

☞ le compilateur se charge de réaliser l'invocation du constructeur par défaut

Constructeurs et héritage (3)

Si la classe parente n'admet pas de constructeur par défaut, l'**invocation explicite** d'un de ses constructeurs **est obligatoire** dans les constructeurs de la sous-classe

☞ La sur-classe doit admettre *au moins un constructeur explicite*.

Exemple :

```
class Rectangle {
protected: double largeur; double hauteur;
public:
        Rectangle(double l, double h) : largeur(l), hauteur(h)
    {}
    // le reste de la classe...
};
class Rectangle3D : public Rectangle {
protected: double profondeur;
public:
    Rectangle3D(double l, double h, double p)
        // Appel au constructeur de la super-classe
    : Rectangle(l,h), profondeur(p) {}
// le reste de la classe...
};
```

Constructeurs et héritage (4)

Autre exemple (qui ne fait pas la même chose) :

```
class Rectangle {
protected: double largeur; double hauteur;
public:
// il y a un constructeur par defaut !
    Rectangle() : largeur(0.0), hauteur(0.0)
        {}}
// le reste de la classe...
};
class Rectangle3D : public Rectangle {
protected: double profondeur;
public:
    Rectangle3D(double p)
        : profondeur(p)
        {}
// le reste de la classe...
};
```

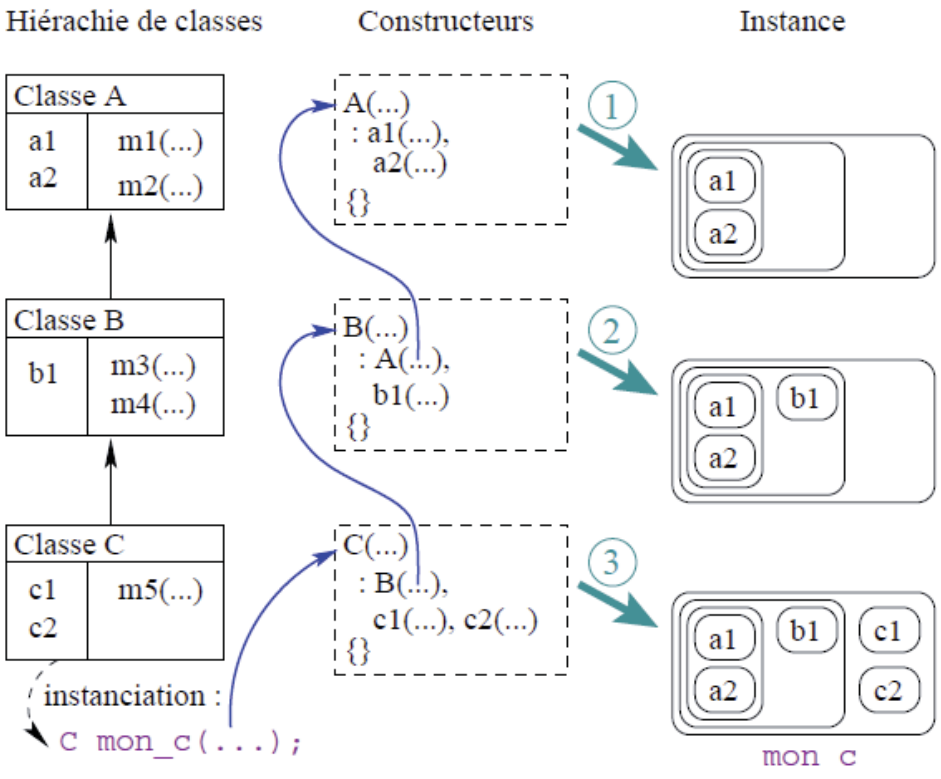
Ici il n'est pas nécessaire d'invoquer explicitement le constructeur de la classe parente puisque celle-ci admet un constructeur par défaut.

Encore un exemple

Il n'est pas nécessaire d'avoir des attributs supplémentaires...

```
class Carre : public Rectangle {
public:
        Carre(double taille) : Rectangle(taille, taille)
        {}
    // et c'est tout ! (sauf s'il y avait des "methodes set")
};
```

Ordre d'appel des constructeurs



Ordre d'appel des destructeurs

Les destructeurs sont toujours appelés dans l'ordre inverse (/symétrique) des constructeurs.

Par exemple dans l'exemple précédent, lors de la destruction d'un C, on aura appel et exécution de :

- C: :~C( )
- B: :~B( )
- A: :~A( )

(et dans cet ordre)
(puisque les constructeurs avaient été appelés dans l'ordre

- A: :A( )
- B: :B( )
- C: :C( )

)

C++11 Héritage des constructeurs

Les constructeurs ne sont, en général, **pas hérités** mais en C++11 on peut *demander leur héritage* en utilisant le mot clé « using ».
On récupère alors **tous** les constructeurs de la super-classe, i.e. on peut construire la sous-classe avec les mêmes arguments, mais...
Attention ! ces constructeurs n'initalisent donc **pas** les attributs spécifiques de la sous-classe.
C'est donc **très risqué**, et je vous conseille de ne l'utiliser que pour des sous-classes n'ayant *pas de nouvel attribut* (et si c'est approprié) !

Exemple :

```
class A {
public:
        A(int);
        A(double, double);
                //...
};
```

```
class B : public A {
        using A::A; // ICI
/* existent alors maintenant
        B::B(int)
        et B::B(double, double) */
};
```

Héritage

Spécifier un *lien d'héritage* :

class Sousclasse **: *[public]* SuperClass** {...}

Droits d'accès : protected accès autorisé au sein de la hiérarchie

Masquage : un attribut/méthode peut être redéfini dans une sous-classe

Accès à un *membre caché* : SuperClasse::membre

Le constructeur d'une sous classe doit faire appel au *constructeur de la super classe* :

```
class SousClasse: SuperClasse
{
        SousClasse(liste de paramètres)
        : SuperClasse(Arguments),
          attribut1(valeur1), ..., attributN(valeurN) {...}
};
```

Petit rappel

Nous avons déjà vu qu'il existe en C++, des méthodes particulières permettant :

- d'initialiser les attributs d'un objet en début de vie : *constructeurs*
- de copier un objet dans un autre objet : *constructeurs de copie*
- de libérer les ressources utilisées par un objet en fin de vie : *Destructeurs*

Une **version par défaut**, minimale, de ces méthodes est **automatiquement générée** si on ne les définit pas explicitement.

Petit rappel (2)

Dans certains cas, les versions minimales par défaut des méthodes constructeurs/destructeurs **ne sont pas adaptées** :
exemple du *comptage des instances*.

Autre exemple :

Le *constructeur de copie par défaut* réalise une copie membre à membre des attributs
→ **copie de surface**

Ceci pose typiquement problème lorsque **certains attributs** de la classe sont des **pointeurs**.

Examinons pourquoi sur un exemple concret...

Exemple

Soit une autre définition (farfelue, mais possible !) de la classe Rectangle :

```
class Rectangle {
private:
        double* largeur; // aie, un pointeur !
        double* hauteur;
public:
        Rectangle(double l, double h)
                : largeur(new double(l)), hauteur(new double(h)) {}
        ~Rectangle() { delete largeur; delete hauteur; }
        double getLargeur() const;
        double getHauteur() const;
        ...
};
```

Que se passe-t-il lorsqu'on invoque la fonction suivante ?

```
void afficher_largeur(Rectangle tmp) {
        cout << "Largeur: " << tmp.getLargeur() << endl;
}
```

Exemple (2)

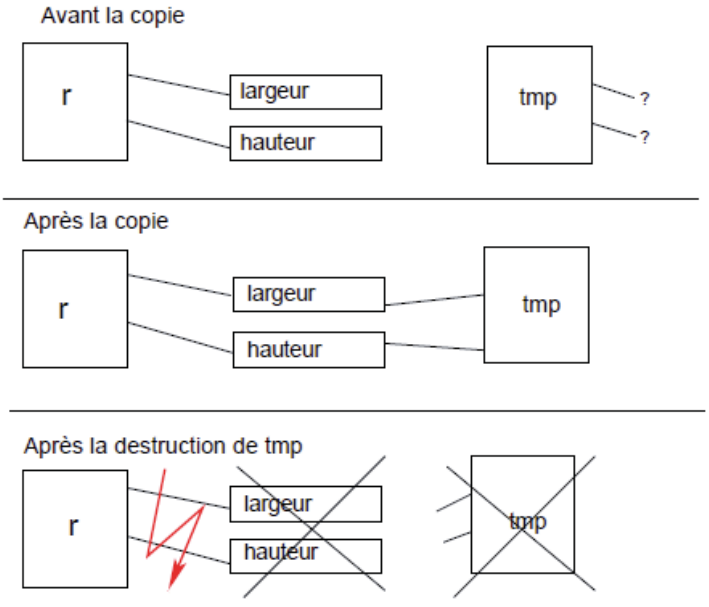
```
Void afficher_largeur(Rectangle tmp) { // une copie !..
        cout << "Largeur: " << tmp.getLargeur() << endl;
} // destruction de tmp...
```

- Lorsque afficher_largeur a fini de s'exécuter, l'objet tmp est automatiquement détruit par le destructeur de la classe Rectangle
- le destructeur va libérer la mémoire pointée par les champs largeur et hauteur de tmp

Attention ! cette portion de mémoire est aussi utilisée par r dans un appel comme afficher_largeur(r) !

Exemple (3)

Voilà ce qui se produit concrètement :



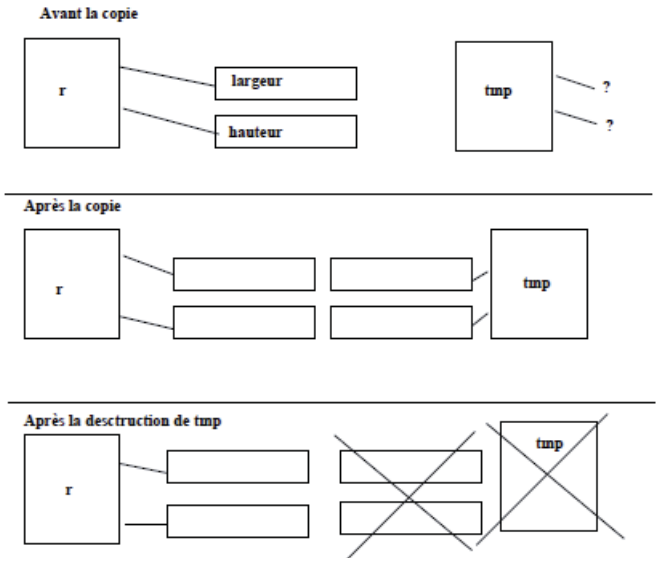
☞ il faut *redéfinir le constructeur de copie* de sorte à ce qu'il duplique véritablement les champs concernés

→ **copie profonde**

Exemple (4)

Une bonne solution consiste alors à **redéfinir le constructeur de copie** :

```
Rectangle(const Rectangle& obj)
        : largeur(new double(*(obj.largeur))),
                hauteur(new double(*(obj.hauteur)))
{ }
```



Définition complète de la classe

```
class Rectangle {
        public:
                Rectangle(double l, double h)
                        : largeur(new double(l)), hauteur(new double(h)) {}
                Rectangle(const Rectangle& obj);
                ~Rectangle();
                // Note: il faudrait aussi redefinir operator= !
private:
        double* largeur; double* hauteur;
};
// constructeur de copie
Rectangle::Rectangle(const Rectangle& obj)
        : largeur(new double(*(obj.largeur))),
                hauteur(new double(*(obj.hauteur)))
        { }
// destructeur
void Rectangle::~Rectangle() {
        delete largeur;
        delete hauteur;
}
```

Il faudra aussi **penser à redéfinir l'opérateur =**

Pour conclure

- Si une classe contient des pointeurs, penser à la copie profonde (**au moins** se poser la question) :
  - constructeur de copie ;
  - surcharge de l'opérateur = ;
  - destructeur.

- Remarque : si l'on redéfinit le constructeur de copie d'une sous-classe, penser à explicitement mettre l'appel au constructeur **de copie** de la super-classe (sinon c'est le constructeur par défaut de la super-classe qui est appelé !!)

Exemple :

```
RectangleColore(const RectangleColore& obj)
        : Rectangle(obj), ...etc...
        { ...etc... }
```

Attributs statiques

Revenons à notre exemple du *comptage des instances* :

```
long compteur(0); // Hmm....

class Rectangle {
        //...
        //constructeur
        Rectangle(): hauteur(0.0), largeur(0.0) {
                ++compteur; }
        // destructeur
        ~{}Rectangle() { --compteur; }
        //...
};
```

Oh horreur !.. Nous avons utilisé une *variable globale* comme compteur pour nos instances.

C'est **très mauvais !** (effets de bord, mauvaise modularisation, contraire au principe d'encapsulation, etc.)

Attributs statiques (2)

La solution à ce problème consiste à utiliser un **attribut statique** :

- un attribut statique est **partagé par toutes les instances** de la même classe (on parle aussi d'« *attribut de classe* »)
- c'est un attribut de la classe qui peut être *privé*, *protégé* ou *public* et dont la déclaration est précédée du mot clé static
- il existe même lorsqu'aucune instance de la classe n'est Déclarée

```
class Rectangle {
private:
        double hauteur, largeur;
        static int compteur; // un membre statique !
//...
};
```

Initialisation des attributs statiques

- un attribut statique doit être initialisé explicitement à l'extérieur de la classe

```
/* initialisation de l'attribut statique dans le fichier de
   definition de la classe (.cc) */

int Rectangle::compteur = 0;

/* Rectangle::compteur existe meme si l'on n'a declare
    aucune instance de la classe Rectangle */
```

Les attributs statiques sont très pratiques lorsque différents objets d'une classe doivent accéder à une même information.

Elle permettent notamment d'éviter que cette information soit dupliquée au niveau de chaque objet !

☞ Concrètement : réserver cet usage à des **constantes** utiles pour toutes les instances de la classe.

Méthodes statiques

- Similairement, si on ajoute static à une méthode :
  - On peut accéder à la méthode à travers un objet mais aussi *sans* objet (à partir du nom de la classe et de l'opérateur : :)

```
class A {
public:
        void methode1 () {
            cout << "Methode 1" << endl;
        }
        static void methode2 () {
            cout << "Methode 2" << endl;
        }
};
int main () {
        A v;
        v.methode1();                   //OK
        v.methode2();                   //OK
        A::methode2();                  //OK, alternative
        A::methode1();                  //Faux
}
```

Restrictions sur les méthodes statiques

- Puisqu'une méthode statique peut être appelée avec ou sans objet :
  - Le compilateur ne peut pas être sûr que l'objet this existe pendant l'exécution de la méthode
  - Il ne peut donc pas admettre l'accès aux variables/méthodes d'instance (car elles dépendent de this)

- Conclusion pour les accès dans la même classe :
  - Une méthode statique peut *seulement* accéder à d'autres méthodes statiques et à des variables statiques

- Le recours à des méthodes statiques ne se justifie que dans des situations très particulières
  - Évitez la prolifération de static !

Conclusion

- On peut réduire la duplication de code et reproduire de bon modèles de la réalité en utilisant des **hiérarchies de classes**
- Une sous classe hérite des membres de ses classes parentes
- Pour implémenter la notion d'héritage il faut avoir recours dans les constructeurs à : et l'appel aux constructeurs des super-classes
- Avec le mécanisme (: :) on peut gérer le masquage d'attributs et de méthodes dans une hiérarchie de classes
- La définition explicite du constructeurs de copie et du destructeurs ainsi que la surcharge de l'opérateur = sont absolument nécessaires lorsque la classe réalise de l'allocation dynamique de mémoire (cas de la copie profonde)
- Un attribut ou méthode statique est commun à toutes les instances d'une classe

Analyse et programmation orientée objet (C++)

Séance 5

Polymorphisme

Objectifs du cours

- *Introduire la notion de* **polymorphisme**
- *Expliquer les techniques de mise en œuvre du polymorphisme en C++ :*
  - *Classes et méthodes virtuelles*
  - *Collections hétérogènes*
  - *Allocation/désallocation dynamique*

Polymorphisme

En programmation, on distingue deux types de polymorphismes :

- *le* **polymorphisme des traitements** *(ou ad hoc)*
  - *mécanisme de* **surcharge** *des fonctions/méthodes : le même identificateur est utilisé pour désigner des séquences d'instructions différentes*
- *le* **polymorphisme des données** *(ou universel)*
  - *le polymorphisme* **d'inclusion** *: le même code peut être appliqué à des données de types différents liés entre eux par une relation de sous-typage*
  - *hiérarchies de classes*
- *le* **polymorphisme paramétrique** *: le même code peut être appliqué à n'importe quel type (généricité)*
  - *Cours sur les templates dans quelques semaines*

Héritage (rappels)

Rappel du dernier cours:

- Dans une hiérarchie de classes, la *sous-classe hérite de la super-classe* :
  - tous les attributs/méthodes (sauf constructeurs et destructeur)
  - le type
- L'héritage est transitif

Héritage du type :

- on peut affecter un objet de type sous-classe à une variable de type super-classe

```
FigureGeometrique fig;
Rectangle r;

fig = r;
```

Héritage : « est-un » : héritage du type

```
class A {
public:
    A(int x = 0)
      : a(x) {}
    int getA() ...
protected: int a;
};
```

```
class B : public A {
public:
   B(int x, int y)
    : A(x), b(y) {}
protected: int b;
};
```

```
void affiche (const A& obj) {
    cout << "Valeur: ";
    cout << obj.getA() << endl;
}

int main() {
  B bb(4, 3);
  A aa; aa=bb;    ou   A(bb);
  affiche(aa);
  affiche(bb);
```

```
Valeur: 4
Valeur: 4
```

Les instances de B sont substituables aux instances de A

- **Compatibilité ascendante des types**

Polymorphisme d'inclusion

En POO, le **polymorphisme d'inclusion** est le fait que :

- les instances d'une sous-classe soient **substituables** aux instances des classes de leur ascendance, *en argument d'une méthode ou lors d'affectations*, tout en **gardant leurs nature/propriétés propre(s).**

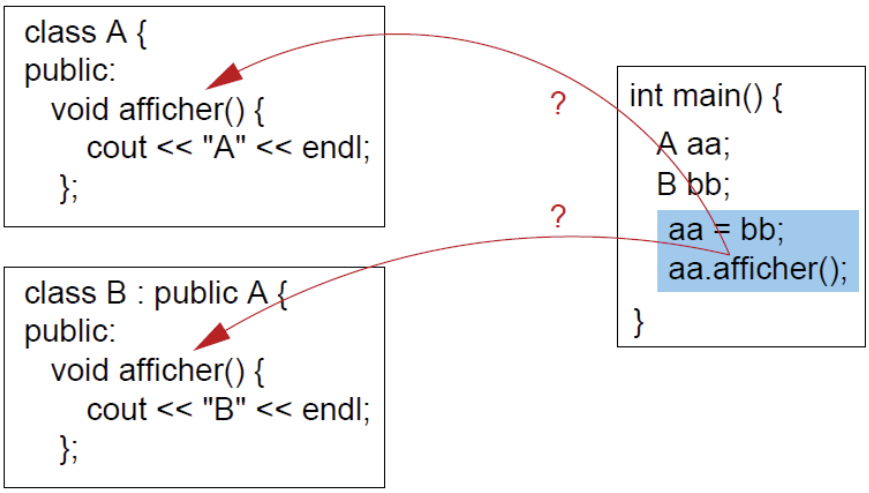
La mise en oeuvre se fait par :

- les mécanismes de **l'héritage** dans les hiérarchies de classes,
- la **résolution dynamique des liens** :
  - le choix des méthodes à invoquer se fait *lors de l'exécution du programme* en fonction de *la nature réelle des instances* concernées

Résolution des liens

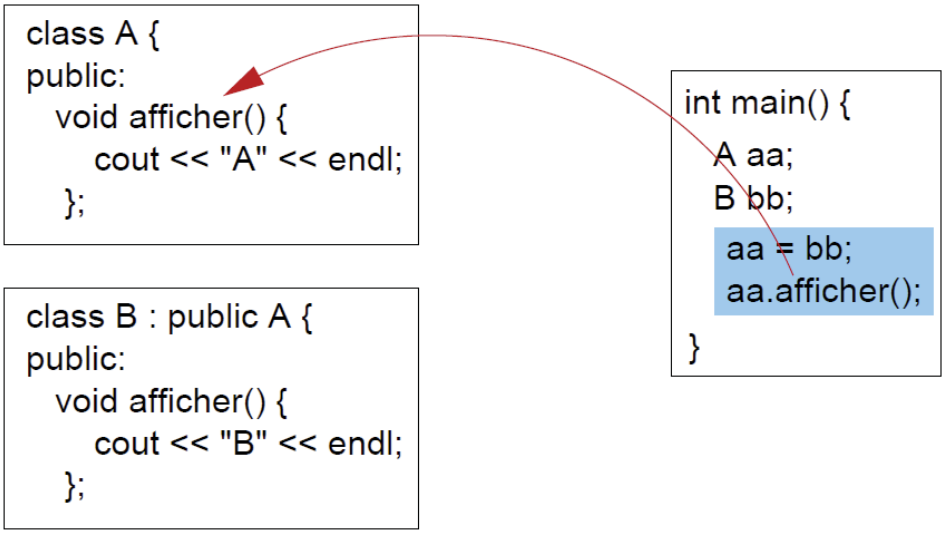
Une instance de sous-classe *B* est substituable à une instance de super-classe *A*.

Que se passe-t-il lorsque *B* redéfinit une méthode de *A* ?



Résolution des liens (2)

En C++, c'est le **type de la variable** qui *détermine la méthode* à exécuter:

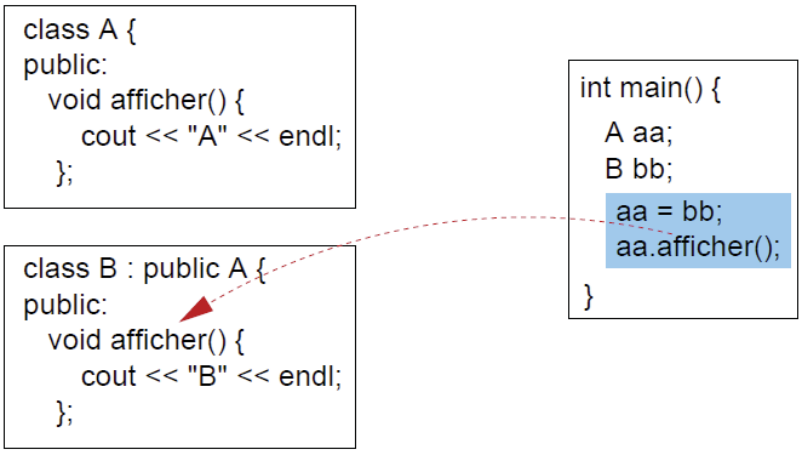


Résolution **statique** *des liens*

Résolution dynamique des liens

Il pourrait dans certains cas sembler plus naturel de choisir la méthode correspondant à la *nature réelle de l'instance* :

- Dans ces cas, il faut permettre la **résolution dynamique des liens** :
- Le *choix de la méthode* à exécuter se fait *à l'exécution*, en fonction de la *nature réelle des instances*



2 ingrédients pour cela :

références/**pointeurs** et **méthodes virtuelles**

Pointeurs – Rappel (1)

En programmation, les pointeurs servent essentiellement à trois choses :

① à permettre à plusieurs portions de code de *partager* des objets (données, fonctions,..) *sans les dupliquer*

« référence »

② à pouvoir *choisir des éléments* non connus *a priori* (au moment de la programmation)

généricité

③ à pouvoir manipuler des objets dont la *durée de vie* dépasse la portée

allocation dynamique

Important : Il faut toujours avoir clairement à l'esprit pour lequel de ces trois objectifs on utilise un pointeur dans un programme !

Pointeurs – Rappel (2)

Un pointeur c'est comme la page d'un carnet d'adresse (sur lesquelles on ne peut écrire qu'une adresse à la fois) :

déclarer un pointeur e.g. int* ptr;	ajouter une page dans le carnet (mais cela ne veut pas dire qu'il y a une adresse écrite dessus !)
affecter un pointeur ptr e.g. ptr = &x;	recopier sur la page ptr l'adresse d'une maison qui existe déja (mais ptr n'est pas la maison, c'est juste la page qui contient l'adresse de cette maison!)
allouer un pointeur ptr e.g. ptr = new int;	aller construire une maison quelque part et noter son adresse sur la page ptr (mais ptr n'est pas la maison!)

Pointeurs – Rappel (3)

Un pointeur c'est comme la page d'un carnet d'adresse (sur lesquelles on ne peut écrire qu'une adresse à la fois) :

« libérer un pointeur » `ptr` (en fait, c'est « libérer l'adresse pointée par le pointeur » `ptr`) e.g. `delete ptr;`	Aller détruire la maison dont l'adresse est écrite en page `ptr`. Cela ne veut pas dire que l'on a effacé l'adresse sur la page `ptr` !! mais juste que cette maison n'existe plus. Cela ne veut pas non plus dire que toutes les pages qui ont la même adresse que celle inscrite sur la page `ptr` n'ont plus rien (mais juste que l'adresse qu'elles contiennent n'est plus valide)

Pointeurs – Rappel (4)

Un pointeur c'est comme la page d'un carnet d'adresse (sur lesquelles on ne peut écrire qu'une adresse à la fois) :

`P1 = p2;`	On recopie à la page `p1` l'adresse écrite sur la page `p2`. Cela ne change rien à la page `p2` et surtout ne touche en rien la maison dont l'adresse se trouvait sur la page `p1` !
`P1 = 0;`	On gomme la page `p1`. Cela ne veut pas dire que cette page n'existe plus (son contenu est juste effacé) ni que la maison dont l'adresse se trouvait sur `p1` (i.e. celle que l'on est en train d'effacer) soit modifiée en quoi que ce soit !! Cette maison est absolument intacte !

Méthodes virtuelles

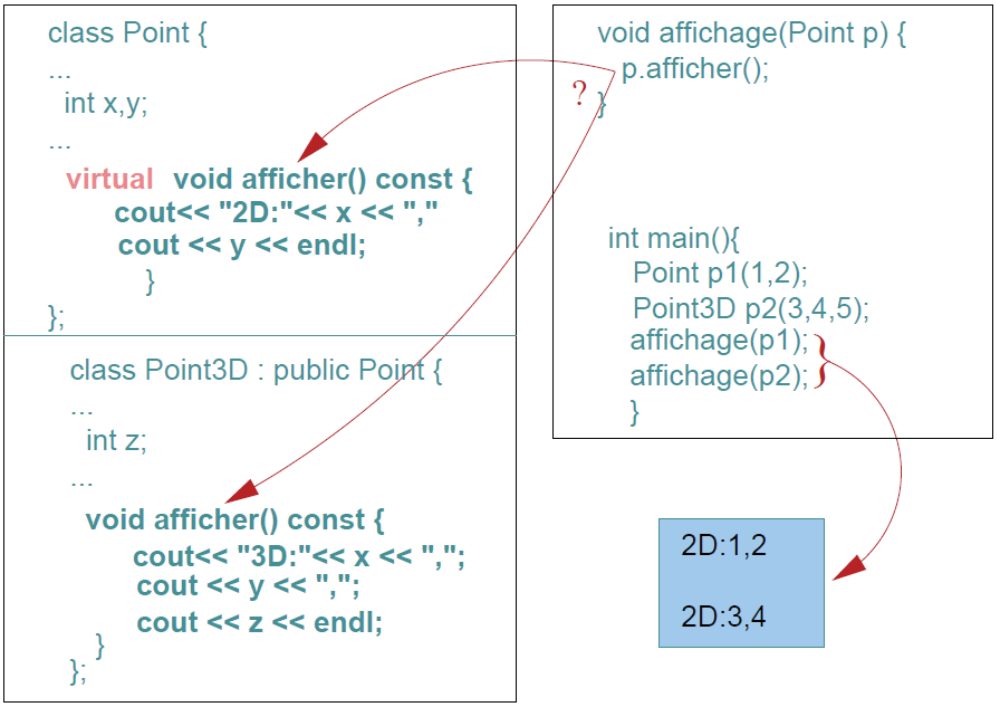
En C++, on indique au compilateur qu'une méthode peut faire l'objet d'une résolution dynamique des liens en la déclarant comme **virtuelle** (mot clé virtual)
cette déclaration doit se faire dans la classe la plus générale qui admet cette méthode (c'est-à-dire lors du *prototypage d'origine*)
les redéfinitions éventuelles dans les sous-classes seront aussi considérées comme *virtuelles par transitivité*.

Syntaxe:
virtual Type nom_fonction(liste d'arguments) [const];

Exemple:

```
class A {
// ajout du modificateur virtual
virtual void afficher() const {
cout << 'A' << endl; }
};
```

Méthodes virtuelles (2)



... il **manque** encore un petit quelque chose pour que ça marche.

Méthodes virtuelles (3)

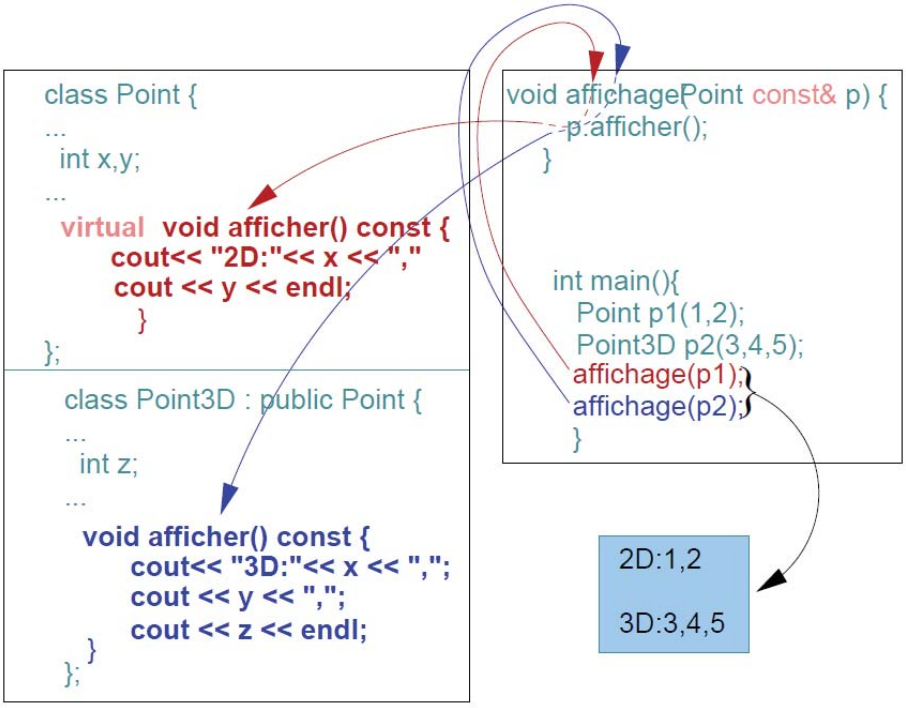
Ici la méthode affichage prend un argument *par valeur*. Elle va donc s'exécuter sur une *copie* de cet argument

```
void affichage(Point p) {
   p.afficher();
}

int main() {
   Point p1(1,2);
   Point3D p2(3,4,5);
   affichage(p1);
   affichage(p2);
}
```

Attention ! Il faut passer p **par référence** pour que la méthode afficher agisse sur l'instance d'origine !

Méthodes virtuelles (4)



Cette fois tout fonctionne comme on voulait !

Méthodes virtuelles (5)

En résumé : Lorsqu'une *méthode virtuelle* est invoquée à partir d'une *référence* ou d'un *pointeur* vers une instance, c'est la méthode associée au type réel de l'instance qui sera exécutée.

Attention !

- Un constructeur **ne peut pas être virtuel**
- On ne peut pas invoquer de méthode virtuelle dans un constructeur (plus exactement : l'aspect polymorphique est ignoré dans le constructeur, seule l'instance correspondant à la classe courante est appelée)
- Il est conseillé de toujours définir les *destructeurs* comme virtuels

Virtuelle / non virtuelle : un exemple

```
#include <iostream>
using namespace std;

class Mammifere {
public:
  Mammifere() { cout << "Un nouveau mammifere est ne !" << endl; }
  virtual ~Mammifere() {
    cout << "Un mammifere est en train de mourir :(" << endl; }
  void manger() const { cout << "Miam... croumf !" << endl; }
  virtual void avancer() const {
    cout << "Un grand pas pour l'humanite." << endl; }
};

class Dauphin : public Mammifere {
public:
  Dauphin () { cout << "Coui, Couic !" << endl; }
  ~Dauphin() { cout << "Flipper, c'est fini..." << endl; }
  void manger()  const { cout << "Sglups, un poisson." << endl; }
  void avancer() const { cout << "Je nage."           << endl; }
};
```

Virtuelle / non virtuelle : un exemple (2)

que produit le code suivant ?

```
int main() {
        Mammifere* lui(new Dauphin);
        lui->avancer();
        lui->manger();
        delete lui;
        return 0;
}
```

Masquage, substitution et surcharge

Nous avons rencontré trois concepts **différents** :
la surcharge (*overloading*) de fonctions et de *méthodes* ;
le masquage (*shadowing*) (en particulier de *méthodes*) ;
(sans la nommer jusqu'ici) la substitution (ou redéfinition, *overriding*), dans les sous-classes, de nouvelles versions de *méthodes virtuelles*.

Pour les *méthodes virtuelles*, on peut donc avoir les trois !!

qui est quoi exactement ?

Par ailleurs, C++11 introduit trois nouveaux mots clés, optionnels, pour justement aider le programmeur à préciser ses intentions: `override`, `new` et `final`

Masquage, substitution et surcharge

- **surcharge** : même nom mais argument différents.
- **masquage** : objets de mêmes noms mais de portées différentes, masqués par les règles de résolution de portée. Pour les méthodes :
  - non seulement le même nom, mais même « *signature* »
  - pas de résolution dynamique (résolution statique)
  - les deux (ou plus) méthodes coexistent
- substitution/redéfinition : méthodes **virtuelles**
  - la méthode est redéfinie (« substituée »)
  - résolution dynamique : c'est la méthode de l'instance qui est appelée (si pointeur ou référence)

`override`, `new` et `final`

En C++11, le programmeur *peut* (optionnel) indiquer ses intentions lors de la redéclaration d'une méthode :

- avec le qualificatif override pour dire qu'il pense substituer/redéfinir une méthode virtuelle
- avec le mot-clé new pour dire qu'il pense ajouter une nouvelle méthode virtuelle

Il peut également, avec le qualificatif final, empêcher la substitution/redéfinition future d'une méthode virtuelle.

override, new et final

```
class A { ...
 virtual void f1();
 virtual void f2() const;
         void f3();        // non virtuelle (oubli ?)
 virtual void f4();
 virtual void f5() final; // pas de redefinition
... };

class B : public A { ...
 virtual void f1() override; // OK
 virtual void fl() override; // Erreur faute de frappe : 1 <-> l
 virtual void f2() override; // Erreur: a oublie le const
         void f3() override; // Erreur: non virtuelle
 virtual void f4(int) new;   // c'est en effet un nouvel f4
 virtual void f5();          // Erreur : f5 etait final
... };
```

override, new et final

Conseils :

- override : utilisez-le (pour vous prémunir), mais c'est un peu verbeux
- new : utilisez-le à tout prix (pour vous prémunir et clairement indiquer vos intentions : création de nouveauté)
- final : oubliez (pour les méthodes)

Note : le mot clé final peut aussi s'utiliser pour les classes elles-mêmes pour empêcher la dérivation (interdire les sous-classes) :

```
class Sterile final { ... };
class C : public Sterile // INTERDIT !
```

Méthodes virtuelles pures

Au sommet d'une hiérarchie de classe, il n'est pas toujours possible de :

- donner une définition générale de certaines méthodes, *compatibles avec toutes les sous-classes*,
- ...même si l'on sait que toutes ces sous-classes vont effectivement implémenter ces méthodes

Méthodes virtuelles pures (2)

Exemple :

```
class FigureFermee {
        //....
        // difficile a definir a ce niveau !..
        // comment ecrire le corps?
        virtual double surface(...) const {...}
        // ...pourtant la methode suivante en aurait besoin !
        double volumeCylindre const (double hauteur) {
            return (hauteur * surface());
        }
        //...
        };
```

Définir surface de façon arbitraire sachant qu'on va la masquer plus tard n'est pas une bonne solution (source d'erreurs) !

☞ *Solution* : déclarer la méthode surface comme **virtuelle pure**

Méthodes virtuelles pures (3)

Une méthode virtuelle pure, ou *abstraite*, est *incomplètement spécifiée* :

- elle n'a *pas de définition* dans la classe où elle est introduite (pas de corps)
- elle sert à signaler aux sous-classes qu'elles **doivent redéfinir** la méthode virtuelle héritée

Syntaxe :

```
virtual Type nom_methode(liste d'arguments) = 0;
```

Exemple :

```
class FigureFermee {
        public:
                virtual double surface() const = 0;
                virtual double perimetre() const = 0;
//...
};
```

Classes abstraites

Une **classe abstraite** est une classe contenant *au moins une méthode virtuelle pure*.

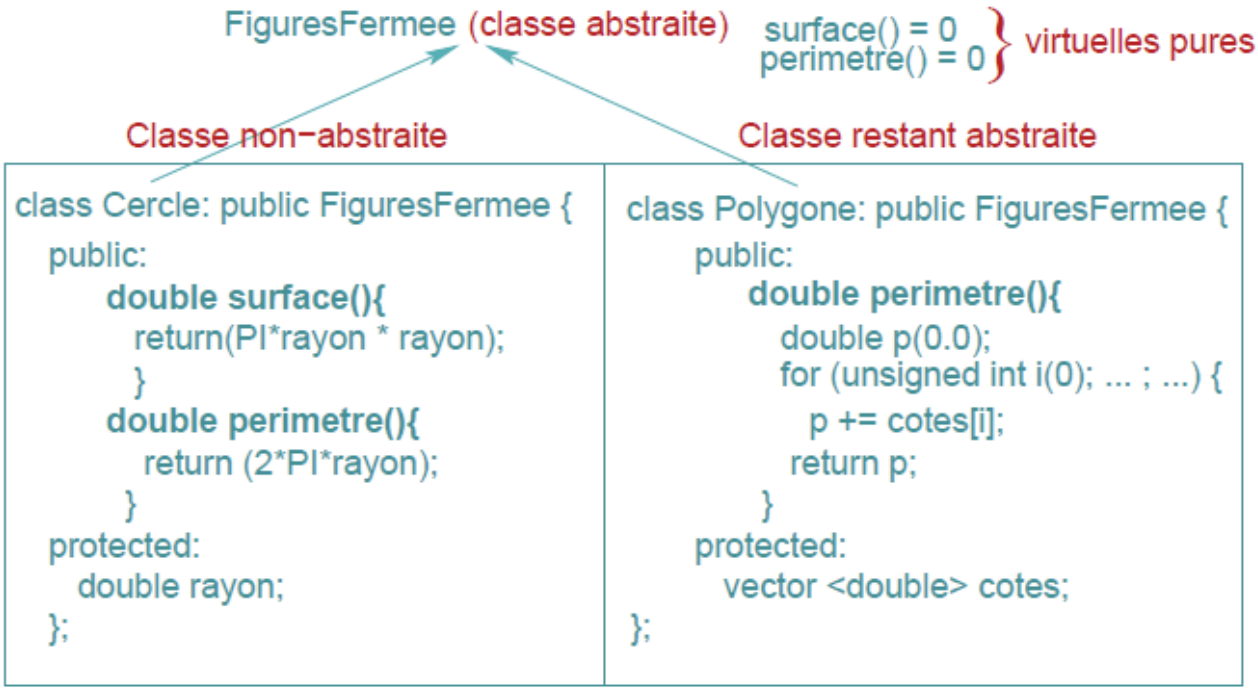
- elle *ne peut être instanciée*
- *ses sous-classes restent abstraites* tant qu'elles ne fournissent pas les définitions de *toutes les méthodes virtuelles pures* dont elles héritent

Note : une classe abstraite peut tout à fait hériter d'une classe non abstraite.

Un exemple « concret »...

Classes abstraites (2)

Exemple :



Collection hétérogène

Nous avons vu jusqu'à maintenant que :

- l'*héritage* et les *méthodes virtuelles* permettent de mettre en oeuvre des **traitements génériques** sur les instances d'une hiérarchie de classes (polymorphisme d'inclusion).

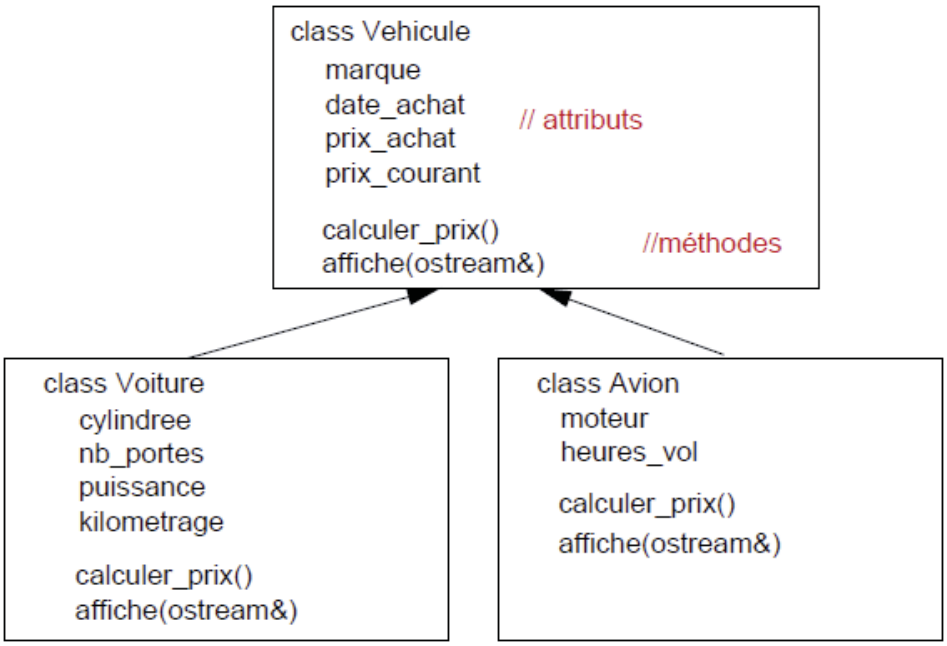
- les fonctions/méthodes génériques doivent utiliser des arguments **passés en référence** pour que le traitement se fasse en fonction de la *nature réelle de l'instance*

Qu'en est-il si tel traitement (générique) doit porter sur un *ensemble* d'instances d'une hiérarchie de classe ?

☞ Collection *hétérogène* (au sens où le comportement spécifique de chaque instance de la collection peut être différent)

Collection hétérogène : exemple

Rappelez-vous de l'exercice sur les véhicules dans la série de la semaine dernière :



Collection hétérogène : exemple (2)

Supposons qu'un aéroport souhaite gérer indifféremment ses avions et ses voitures (en tant que véhicules).

On pourrait définir une classe Aeroport comme suit :

```
class Aeroport {
public:
        void affiche_garage(ostream&);
        void affiche_hangar(ostream&);
        void ajouter_voiture(const Voiture&);
        void ajouter_avion(const Avion&);
        void vider_garage();
        void vider_hangar();
protected:
        vector<Voiture> voitures;
        vector<Avion> avions;
};
```

C'est une solution possible (un point de vue), mais pas nécessairement la seule. On pourrait vouloir regrouper la gestion des avions et voitures (en tant que véhicules) et avoir ainsi une solution plus concise...

Collection hétérogène : exemple (3)

On pourrait par exemple souhaiter plutôt écrire quelque chose comme :

```
class Aeroport {
public:
        void affiche(ostream&) const;
        void ajouter_vehicule(const Vehicule&);
        void vider();
protected:
        vector<Vehicule> vehicules;
};
```

☞ les instances contenues dans l'attribut vehicules font partie d'une *même hiérarchie de classe*, mais sont de nature *hétérogène* (avions ou voitures).

On pourrait par exemple vouloir que, si vehicules[i] est une voiture, la méthode vehicule[i].affiche() soit bien celle de la sous-classe Voiture.

Résolution dynamique des liens

Mais le codage de la classe tel que précédemment ne permet **pas** le comportement polymorphique :

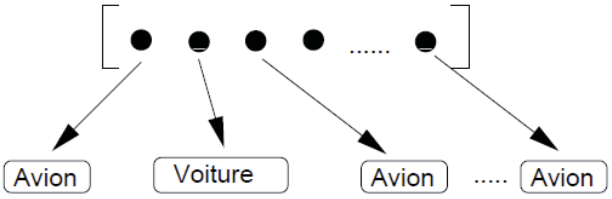
l'attribut vehicules est constitué d'*instances* de type Vehicule et non pas de *références/pointeurs* à ces instances.

☞ on ne peut pas mettre en oeuvre la résolution dynamique des liens.

La solution à ce problème consiste à passer par un vecteur de **pointeurs** :

```
vector<Vehicule*> vehicules;
```

Seuls les pointeurs, c'est-à-dire les *adresses des instances*, sont stockés dans le vecteur, et *non plus les instances* elles-mêmes :



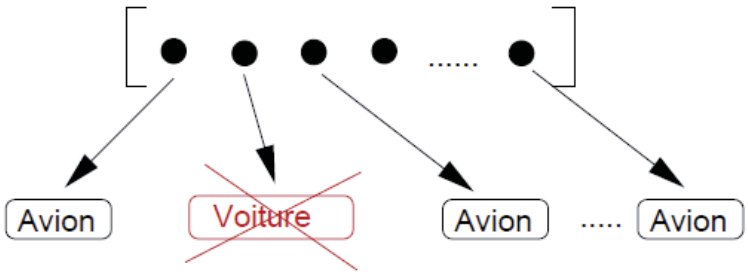
Collection hétérogène – Exemple correct

```
class Aeroport {
public:
        void affiche(ostream&) const;
        void ajouter_vehicule(Vehicule*);
        void vider();
protected:
        vector<Vehicule*> vehicules;
};
void Aeroport::affiche(ostream& affichage) const {
        for (unsigned int i(0); i < vehicules.size(); ++i)
            vehicules[i]->affiche(affichage);
}
void Aeroport::ajouter_vehicule(Vehicule* v) {
        vehicules.push_back(v);
}
```

Pointeurs et intégrité des données

Cette classe Aeroport comporte cependant *un danger potentiel* :

Pour que tout fonctionne bien, il est nécessaire que les éléments pointés *existent aussi longtemps* que leurs pointeurs



Attention ! La co-existence des pointeurs et des éléments pointés n'est cependant pas du tout garantie !

Au programmeur de ne pas faire de bêtises.

Exemple...

Pointeurs et intégrité des données (2)

(Mauvais) Exemple :

```
void creer_avion(Aeroport& a) {
        Avion av("Cessna", 1972, 1230673.90, HELICE, 250);
        a.ajouter_vehicule(&av); //hm...
}
// ...
int main() {
        Aeroport a1;
        creer_avion(a1);
        a1.affiche(cout); // ouille !
        return 0;}
```

La fonction creer_avion ajoute un nouvel avion à l'aéroport a1, mais par le biais d'une *variable locale* (bouh !)

Une fois l'exécution de creer_avion terminée, la **variable locale est détruite** !

Attention ! le pointeur stocké dans le vecteurs vehicules **existe toujours**...

Allocation/désallocation dynamique

La solution à ce problème est que l'utilisateur **alloue dynamiquement** une portion de mémoire *qui sera préservée* après la fin du bloc où l'on crée l'instance.

Exemple :

```
// définition robuste de la fonction creer_avion
void creer_avion(Aeroport& a) {
        a.ajouter_vehicule(new Avion("Cessna",...));
}
```

Grâce à l'utilisation du new, la **mémoire allouée dynamiquement** pour l'avion créé dans créer_avion est préservée à la fin de l'exécution de cette fonction.

Gare aux pointeurs !

mais qui dit «pointeurs», dit aussi « bonne gestion » et « programmation rigoureuse »...

En particulier pensez, si nécessaire, à la *copie profonde* et au *destructeur* pour libérer la mémoire allouée

> Attention ! En cas de copie profonde avec des classes abstraites (typiquement collections hétérogènes), il est nécessaire de définir une méthode de copie comme virtuelle pure au niveau de la classe abstraite.

Et n'oubliez pas la règle d'or :

c'est celui qui a alloué la mémoire (new) **qui est chargé de la libérer** (delete)

(par exemple dans l'exemple précédent, fournir une fonction

```
detruire_avion(Aeroport& ou, Vehicule* qui);
```

ou alors

```
detruire_avion(Aeroport& ou, unsigned int lequel);
```

ou encore (voire les deux) :

```
detruire_tout(Aeroport& ou);
```

)

Collections hétérogènes en pratique

Il y a en fait de nombreuses conceptions possibles pour les collections hétérogènes, principalement suivant deux axes :

① Le contenu de la collection est-il personnel (à la collection) ou partagé ?

☞ **QUI** a la propriété des objets dans la collection : la collection elle-même ou un responsable externe ?

Au niveau de ce cours, je vous conseille (hors exercices courts et faciles) :

- Tant que faire se peut de donner la propriété à la collection
- et dans ce cas d'utiliser des unique_ptr en C++11 ou des pointeurs à la C, mais gérés en interne de la collection
- sinon (si vous ne pouvez pas donner la propriété à la collection), d'utiliser des pointeurs à la C et transférer des adresses d'objets existants plus longtemps que la collection elle-même (il faudra donc le garantir !)

Collections hétérogènes en pratique (2)

Il y a en fait de nombreuses conceptions possibles pour les collections hétérogènes, principalement suivant deux axes :

② **quelle interface** pour les utilisateurs de la classe ?

2.a) Gérer les pointeurs en interne (après tout ce n'est qu'un détail d'implémentation lié au langage) :

```
void Collection::ajoute(un_type const&);
```

voire

```
void Collection::ajoute(un_type&);
```

2.b) ou alors « afficher » les pointeurs et laisser leur gestion en externe :

```
void Collection::ajoute(un_type*);
```

La première (2.a) est plus « propre » (cache les détails d'implémentation), la seconde (2.b) plus directe (et plus simple pour le programmeur de la classe).

Dans le premier cas, il pourrait peut être s'avérer utile d'avoir une copie polymorphique des éléments contenus :

```
virtual un_type* un_type::copie() const;
```

Collections hétérogènes en pratique (3)

Dans le second cas (2.b, gestion externe des pointeurs) :

```
void Collection::ajoute(un_type*);
```

on peut, en tant qu'*utilisateur* de la collection, opter pour une version statique :

```
un_type_possible un_element(...);
...
collection.ajoute(&un_element);
```

(mais attention à la portée des variables : ne pas avoir une collection de plus grande portée que ses éléments ! cf transparent 37)

ou alors dynamique :

```
collection.ajoute(new un_type_possible(...));
```

Dans tous les cas (gestion interne ou externe dynamique), il ne faut pas oublier de gérer correctement la libération de la mémoire **(cf transparent 164).**

Polymorphisme

Résolution dynamique des liens : choix des méthodes à invoquer **lors de l'exécution du programme** en fonction de la **nature réelle des instances**

2 ingrédients :

méthodes virtuelles et **références**/**pointeurs**

Méthode virtuelle :

virtual Type nom_fonction(liste d'arguments)

[const];

Méthode virtuelle *pure* (abstraite) :

virtual Type nom_methode(liste d'arguments)

const **=0**

Classe abstraite : contient *au moins une méthode abstraite*

Collection hétérogène : des pointeurs sur les instances doivent être manipulés, et non pas les instances directement

Conclusion

- Comment mettre en oeuvre le **polymorphisme d'inclusion** en C++ :
  - **méthode virtuelle**
  - **indirection** : références ou pointeurs
- Comment exploiter les possibilités d'abstraction : méthodes virtuelles pures et classes abstraites
- Comment utiliser le polymorphisme pour implémenter des collections hétérogènes

Analyse et programmation orientée objet (C++)

Séance 6

Héritage (suite)

Héritage

- L'héritage (**public**) observé jusqu'ici correspond à la spécialisation conceptuelle
- héritage public
  - autorise le sous/sur-typage, le polymorphisme, la liaison dynamique
- L'héritage peut servir à d'autres choses...

L'héritage comme composition

- une composition « ordinaire »

```
class Moteur {
 public:
        void demarre() { ... };
};

class Voiture {
 private:
        Moteur leMoteur;
 public:
        void demarre() { leMoteur.demarre(); }
};
```

L'héritage comme composition

- la composition obtenue par héritage...

```
class Moteur {
 public:
        void demarre() { ... };
};

class Voiture  : private Moteur {
 public:
        void demarre() { Moteur :: demarre(); }
};
```

L'héritage privé ne définit pas un sous-typage...

```
class Moteur {
};
class Voiture : private Moteur {
};

int main() {
        Voiture v;
        Moteur m;
        m = v; // interdit!
        Moteur *pm;
        pm = &v; // interdit!
        return 0;
}
```

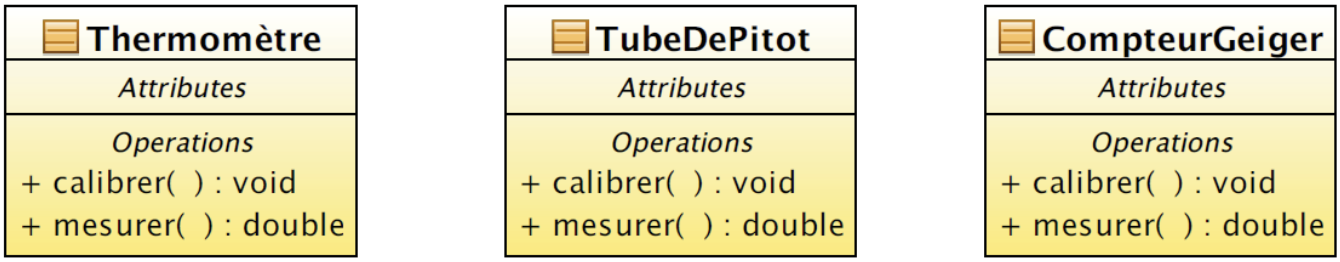
La factorisation conceptuelle

La factorisation conceptuelle conduit à l'apparition d'abstractions, *i.e.* de types abstraits, *i.e.* d'interfaces

- Elle consiste à réunir en une même unité d'encapsulation des actions communes
- La **factorisation conceptuelle** s'appelle la **généralisation**

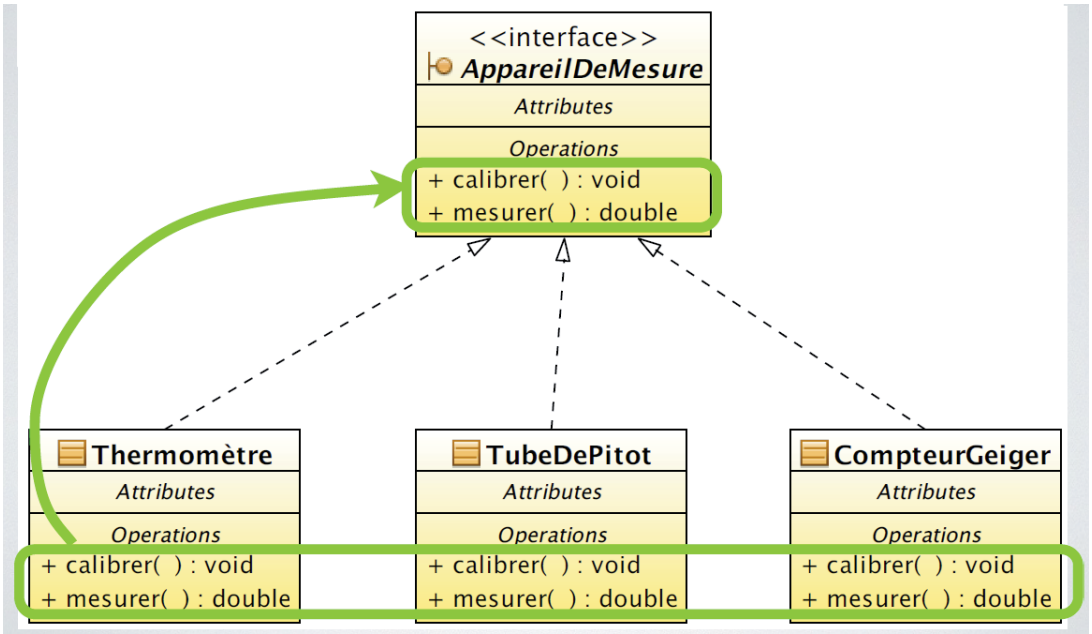
La factorisation conceptuelle

- Des classes concrètes identifiées...



La factorisation conceptuelle

- On « remonte » le facteur commun
- C'est l'interface qui est commune...



Ec C++:

```
class AppareilDeMesure {
  public:
      virtual void calibrer()=0; // équiv. C++ du abstract de Java
      virtual double mesurer()=0;
};
```

```
class CompteurGeiger : public AppareilDeMesure {
  public:
      virtual void calibrer() { /* remettre à zéro */ }
      virtual double mesurer() { /* compter les particules */ }
};
```

```
class TubeDePitot : public AppareilDeMesure {
  public:
      virtual void calibrer() { /* remettre à zéro */ }
      virtual double mesurer() { /* soustraire des pressions */ }
};
```

```
class Thermomètre : public AppareilDeMesure {
  public:
      virtual void calibrer() { /* laisser refroidir */ }
      virtual double mesurer() { /* attendre la stabilisation */ }
};
```

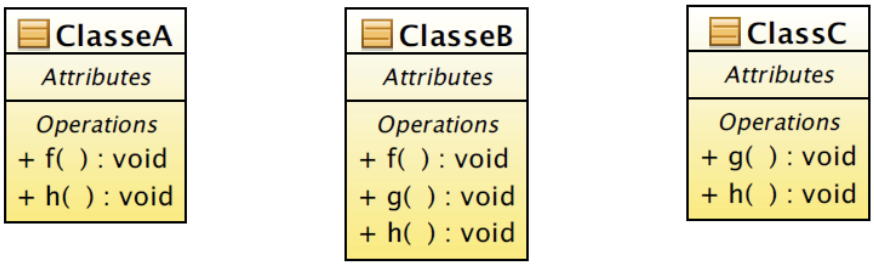
La factorisation conceptuelle

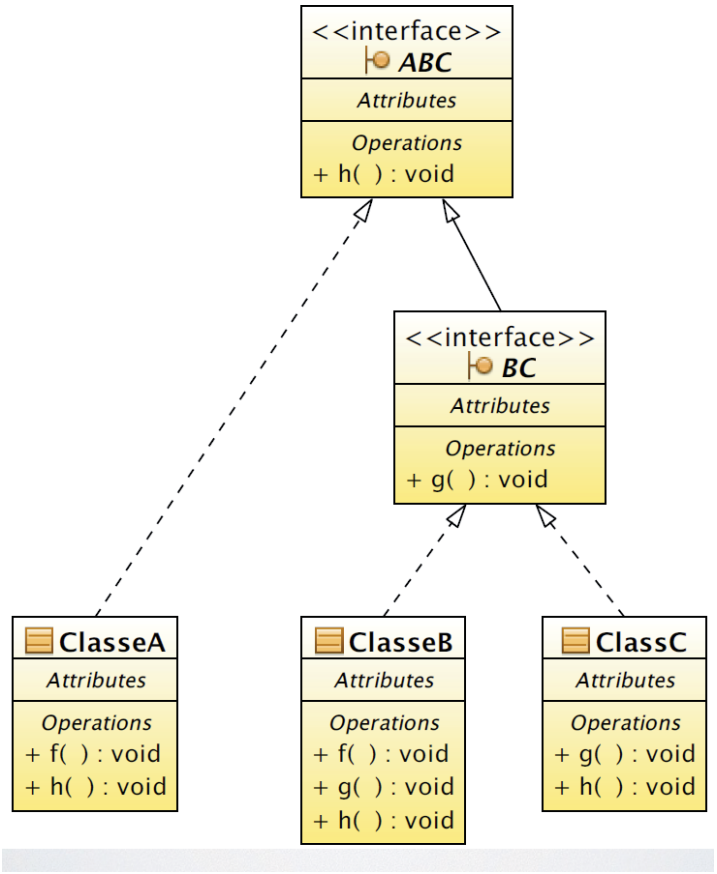
Attention la factorisation n'est pas toujours simple

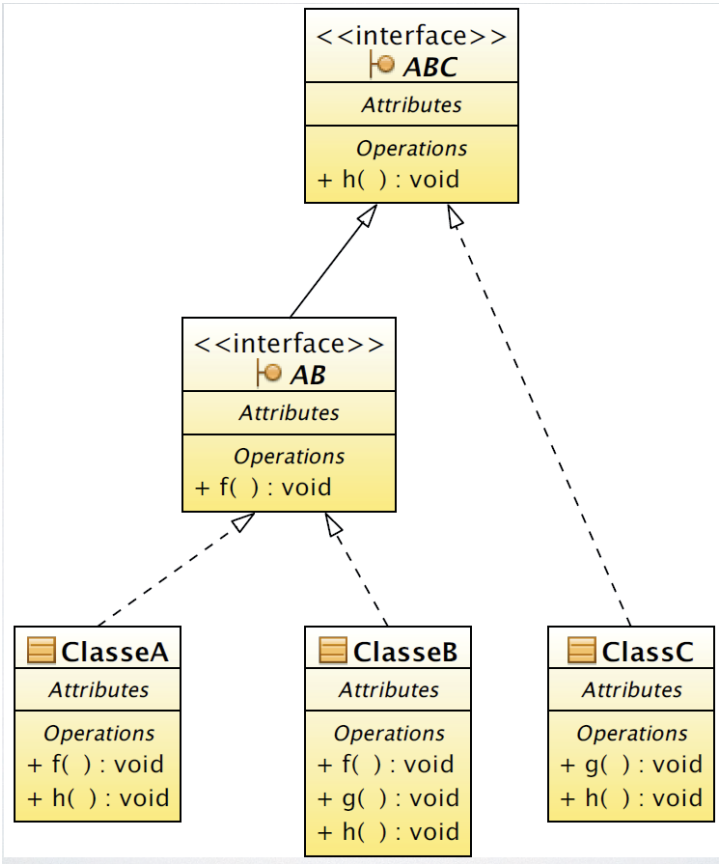
- Elle n'est en général **pas unique**
- Attention à factoriser en conservant du sens, ce qui est loin d'être toujours évident
- Il n'y a **pas une bonne solution**

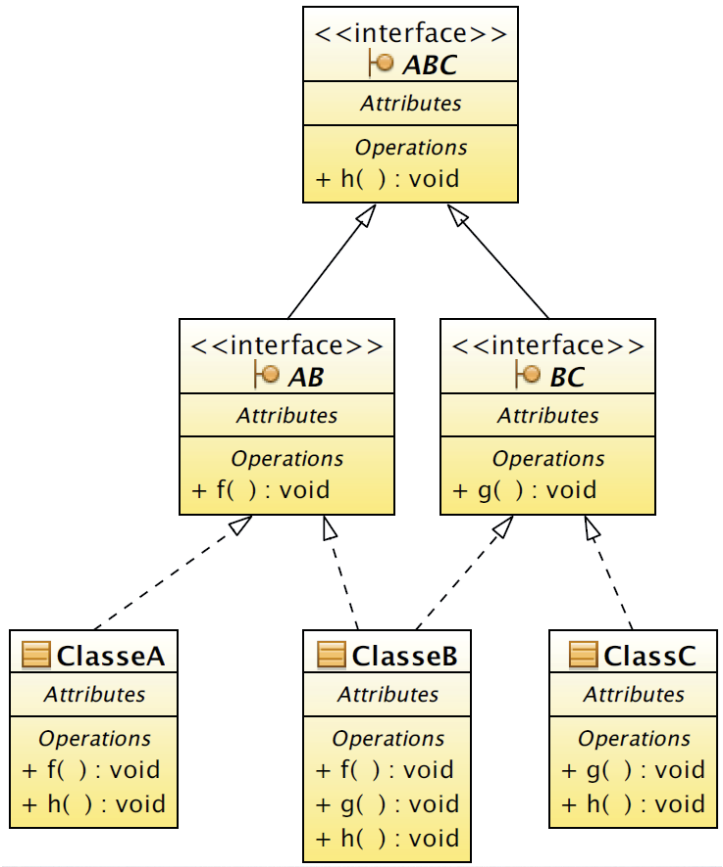
Contrairement à la spécialisation, elle conduit à fabriquer des sur-types (n'est donc intéressante qu'à la conception)

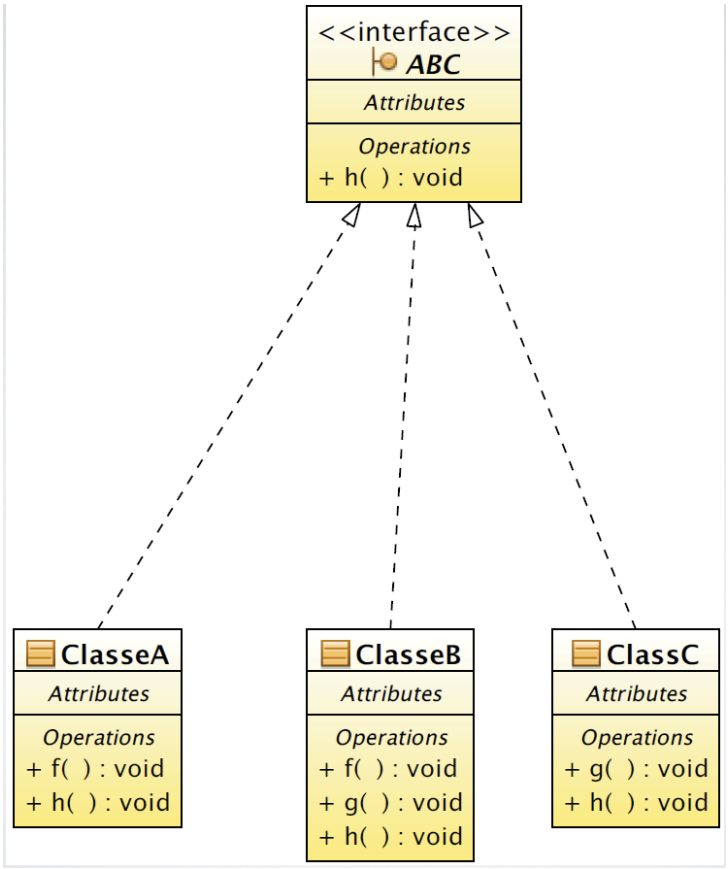
La factorisation conceptuelle





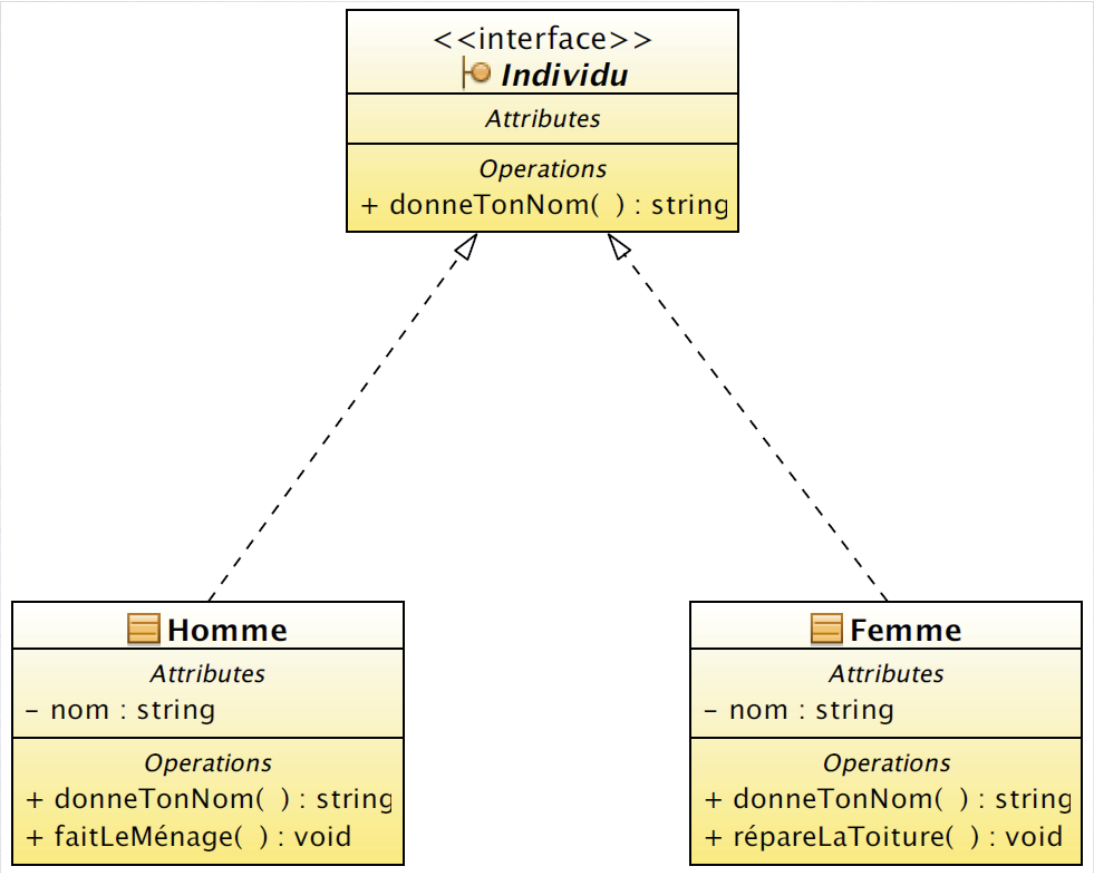

<<interface>>
ABC
Attributes
Operations
+ h( ) : void
<<interface>>
AB
Attributes
Operations
+ f( ) : void
ClasseA
Attributes
Operations
+ f( ) : void
+ h( ) : void
ClasseB
Attributes
Operations
+ f( ) : void
+ g( ) : void
+ h( ) : void
ClassC
Attributes
Operations
+ g( ) : void
+ h( ) : void


<<interface>>
ABC
Attributes
Operations
+ h( ) : void
<<interface>>
AB
Attributes
Operations
+ f( ) : void
<<interface>>
BC
Attributes
Operations
+ g( ) : void
ClasseA
Attributes
Operations
+ f( ) : void
+ h( ) : void
ClasseB
Attributes
Operations
+ f( ) : void
+ g( ) : void
+ h( ) : void
ClassC
Attributes
Operations
+ g( ) : void
+ h( ) : void



La factorisation d'implémentation/de réalisation

- La factorisation d'implémentation/de réalisation conduit à la construction de classes incomplètes, de **réalisations partielles**...
- Elle consiste à réunir en une même unité d'encapsulation des actions (avec une partie de leur implémentation) et des attributs communs
- C'est l'**héritage comme moyen de réutiliser** une implémentation



La factorisation d'implémentation/de réalisation

```
class Individu {
  public:
      virtual string donneTonNom()=0;
};
class _Individu : public Individu {
  private:
      string nom;
  public:
      virtual string donneTonNom() { return nom; }
};
class Homme : public _Individu {
  public:
      void faitLeMénage() { ... }
};
class Femme : public _Individu {
  public:
      void répareLeToit() { ... }
};
```

La factorisation d'implémentation/de réalisation

Attention:

- les classes intermédiaires ne doivent jamais être employées comme type (elles ne sont là que pour faciliter la réalisation des types concrets)...
- inconvénient : pour comprendre comment fonctionne un type concret donné, il faut lire le code de toutes les réalisations partielles faites...
- la conception de telles classes doit être pensée avant l'écriture du code (attribut **`virtual`**)

Desctructeur

- les destructeurs des classes **doivent être** qualifiés de `virtual`
- ceci afin que le bon destructeur soit appelé en cas de polymorphisme

```
class Individu {
  public:
      virtual string donneTonNom()=0;
      ~Individu() { cout << "~Individu()" << endl; };
};
class Femme : public Individu {
  public:
      Femme(string nom) : Individu(nom) {};
      ~Femme() { cout << "~Femme(" << donneTonNom() << ")" << endl; }
};
void libere(Individu *pi) {
      delete pi;
}
int main()
{
      Femme f("Georgette");
      Femme *pf = new Femme("Pascale");
      libere(pf);
      return 0;
}
```

```
class Individu {
  public:
      virtual string donneTonNom()=0;
      virtual ~Individu() { cout << "~Individu()" << endl; };
};
class Femme : public Individu {
  public:
      Femme(string nom) : Individu(nom) {};
      virtual ~Femme() { cout << "~Femme(" << donneTonNom() << ")" << endl; }
};
void libere(Individu *pi) {
      delete pi;
}
int main()
{
      Femme f("Georgette");
      Femme *pf = new Femme("Pascale");
      libere(pf);
      return 0;
}
```

Conversions Polymorphes

- Conversion vers le haut (*upcast*)
  - objet
  - pointeur
  - Référence
- Conversion vers le bas (*downcast*)
  - interdite sur les objets
  - à contrôler sur les pointeurs
  - à contrôler sur les références

Conversions Polymorphes

- L'opérateur `dynamic_cast<`*`type`*`>` permet **si *`type`* est polymorphe** (*i.e.* possède une méthode virtuelle) :
- d'obtenir une conversion vers un sous-type (*downcast*)
- un pointeur nul (0) si la conversion n'est pas correcte
- une exception (`bad_cast`) si la référence n'est pas correcte

```
class A {
  public:
      virtual void f() { cout << "A::f()" << endl; }
};
class B : public A {
  public:
      virtual void f() { cout << "B::f()" << endl; }
};
void f(A *a) {
      B *p = dynamic_cast<B *>(a);
      cout << p << endl;
}

int main() {
      B b;
      f(&b); // ca va marcher : 0xfffac…
      A a;
      f(&a); // ca va rater... : 0
      return 0;
}
```

Analyse et programmation orientée objet (C++)

Séance 7

Héritage multiple

Objectifs

- Introduire la notion d'héritage multiple
- Construction/destruction
- Ambiguïté des attributs/méthodes hérités
- Classes virtuelles

Où en est-on?

Nous connaissons maintenant les aspects essentiels de la POO:

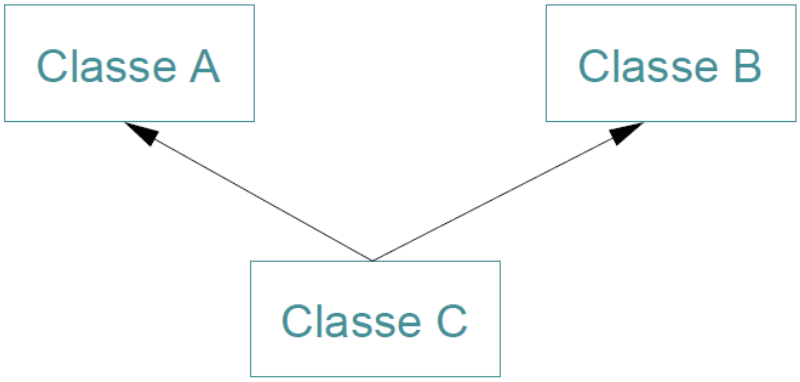
- Encapsulation et abstraction
  - regroupement traitements et données : `class X { ... };`
  - séparation interface et détails d'implémentation : `public:`, `protected:`, `private:`
- Héritage

```
class C1 : public C2 {...};
```

- Polymorphisme
  - pouvoir être vu de plusieurs façons, abstraction, généricité
  - 2 ingrédients nécessaires : pointeurs et méthodes virtuelles
  - classes abstraites et méthodes virtuelles pures
  - collections hétérogènes

Qu'est-ce que l'héritage multiple ?

- En C++, une sous-classe peut hériter de **plusieurs super-classes** :

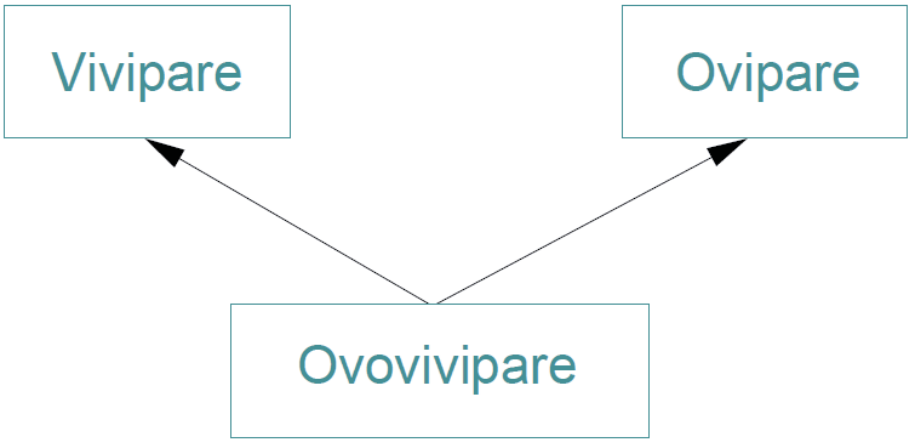


Comme pour l'héritage simple, la sous-classe hérite des super-classes:

- tous leurs *attributs et méthodes* (sauf les constructeurs/destructeurs)
- leur *type*

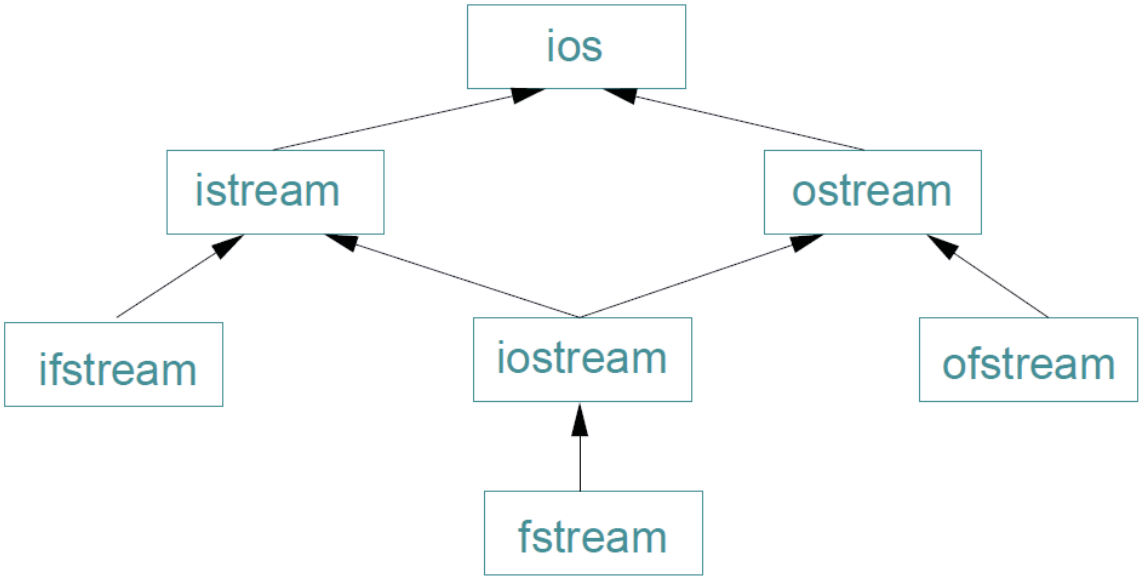
Exemple

Un exemple zoologique:



Exemple (2)

.. et un exemple informatique:



Héritage multiple

Syntaxe :

```
class nomSousClasse: [public] nomSuperClasse1, ...
[public] nomSuperClasseN {
//...
};
```

- Exemple :

```
class Ovovivipare: public Ovipare, public Vivipare {
  public:
    Ovovivipare(unsigned int, unsigned int);
    virtual ~Ovovivipare();
  protected:
    bool espece_rare;
};
```

- il n'y pas de restriction sur le nombre de super-classes dont la sous-classe peut hériter
- l'ordre de déclaration des super-classes est pris en compte lors de l'invocation des constructeurs/destructeurs

Constructeurs/destructeurs

Comme pour l'héritage simple, l'initialisation des attributs hérités doit être faite par invocation des constructeurs des super-classes :

Syntaxe:

```
SousClasse(liste d'arguments)
      : SuperClasse1(arguments1),
      ...
      SuperClasseN(argumentsN),
      attribut1(valeur1),
      ...
      attributK(valeurK)
    {}
```

Lorsque l'une des super-classes admet un constructeur par défaut, il n'est pas nécessaire de l'invoquer explicitement.

Constructeurs/destructeurs (2)

Attention ! L'exécution des constructeurs des super-classes se fait **selon l'ordre de la déclaration d'héritage**, et non selon l'ordre des appels dans le constructeur!

L'ordre des appels des destructeurs de super-classes est **l'inverse** de celui des appels de constructeurs

Constructeurs/destructeurs (3)

Exemple:

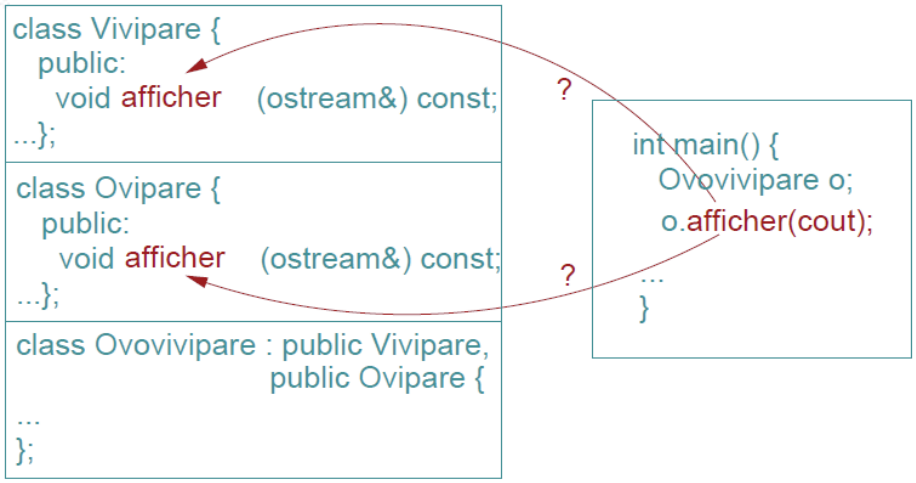
```
class Ovovivipare : public Ovipare, public Vivipare {
  public:
    Ovovivipare(unsigned int, unsigned int);
    virtual ~Ovovivipare();
  protected:
    bool espece_rare;
};
Ovovivipare::Ovovivipare(unsigned int nb_oeufs,
                         unsigned int duree_gestation)
   : Vivipare(duree_gestation),
     Ovipare(nb_oeufs),
     espece_rare(false)
{}}
```

- Ordre d'invocation des constructeurs : `Ovipare()`, puis `Vivipare()` puis le corps de `Ovovivipare()`
- Ordre d'invocation des destructeurs :`~Ovovivipare()`, puis `~Vivipare`, puis `~Ovipare`

Accès direct ambigu

Comme dans le cas de l'héritage simple, une sous-classe peut *accéder directement* aux attributs et méthodes *protégés* de ses super-classes

... et si ces attributs/méthodes *portent le même nom* dans plusieurs super-classes ?



Accès direct ambigu (2)

Attention ! l'accès `o.afficher` provoquera une erreur à la compilation **même** si la méthode `afficher` **n'**avait **pas** les mêmes arguments dans les deux classes `Ovipare` et `Vivipare` !!!

(La raison est qu'en C++ il n'y a surcharge que dans la même portée. Ici ce n'est pas une problème de surcharge, mais un problème de résolution de portée)

Première solution : utiliser *l'opérateur de résolution de portée*.

```
int main(){
Ovovivipare o;
o.Vivipare::afficher(cout);
...
```

mais...

Accès direct ambigu (3)

mais...

L'utilisation de l'opérateur de résolution de portée pour résoudre les ambiguïtés de noms des attributs/méthodes **n'est pas une bonne solution** :

c'est l'utilisateur de la classe `Ovovivipare` qui décide du fonctionnement correct de cette classe, alors que cette responsabilité doit normalement *incomber aux concepteurs de la classe*

Accès direct ambigu – Solution

Une des solutions consiste à lever l'ambiguïté en indiquant explicitement quelle(s) méthode(s) on souhaite invoquer

Il faut ajouter à la liste des déclarations des méthodes/attributs de la sous-classe, une déclaration spéciale indiquant quel(s) méthode(s)/attribut(s) seront invoqué(s) exactement

Syntaxe:

```
using NomSuperClasse::NomAttributOuMethodeAmbigu
```

Exemple:

```
class Ovovivipare : public Ovipare, public Vivipare {
      public:
            using Vivipare::afficher; //Comme ceci
         ...
};
```

Attention ! pas de parenthèse (ni prototype) derrière les noms de méthodes dans un `using` !

Accès direct ambigu – Solution (2)

Une autre bonne solution consiste à incorporer dans la sous-classe *une méthode définissant la bonne interprétation* de l'invocation ambiguë.

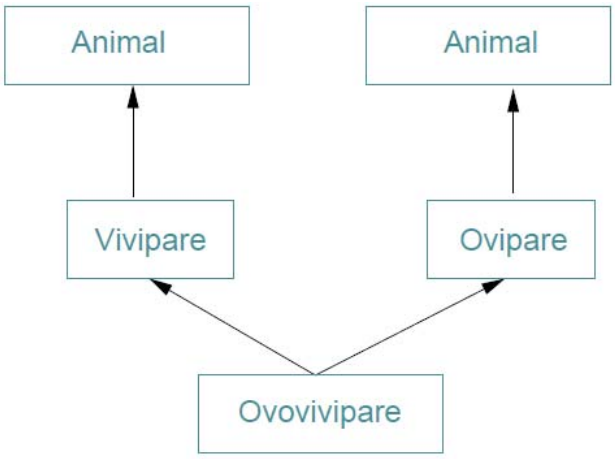
Exemple:

```
class Ovovivipare: public Ovipare, public Vivipare {
      public:
        Ovovivipare(unsigned int, unsigned int);
        virtual ~Ovovivipare();

        // Solution:
        void afficher(ostream& out){
             Ovipare::afficher(out);
             out << " mais aussi ";
             Vivipare::afficher(out);
        }
      protected:
        bool espece_rare;       };
```

Classes virtuelles

Il peut se produire qu'une super-classe soit incluse plusieurs fois dans une hiérarchie à héritage multiple :



Les attributs/méthodes de la super-classe seront inclus plusieurs fois !

- ☞ Chaque objet de la classe `Ovovivipare` possédera **deux** copies des attributs de la classe `Animal`.

Classes virtuelles (2)

Pour *éviter la duplication des attributs* d'une super-classe plusieurs fois incluse lors d'héritages multiples, il faut déclarer son **lien d'héritage** avec *toutes* ses sous-classes comme **virtuel**.

Cette super-classe sera alors dite « **virtuelle** » (à ne pas confondre avec classe abstraite !!)

Syntaxe:

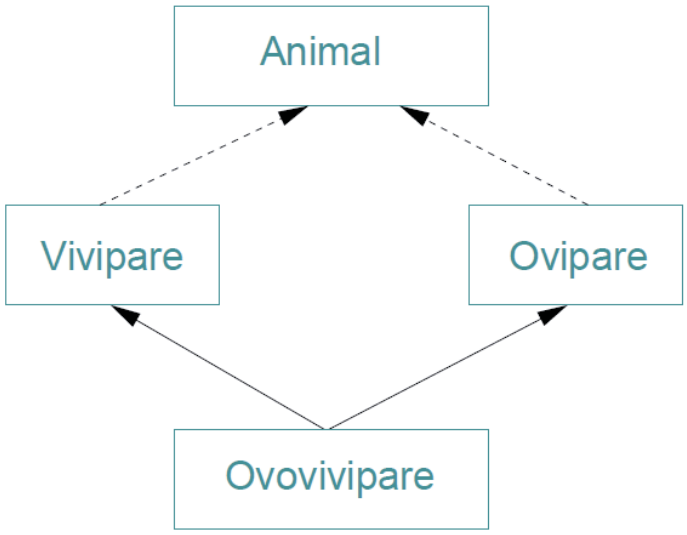
```
class NomSousClasse: [public] virtual NomSuperClasseVirtuelle
```

Exemple:

```
class Ovipare: public virtual Animal {...};
...
class Vivipare: public virtual Animal {...};
```

A noter que c'est la classe pouvant être héritée plusieurs fois qui est virtuelle (i.e. ici la super-super-classe) et non pas directement les classes utilisées dans l'héritage multiple (i.e. les super-classes).

Classes virtuelles (3)



Un seul objet de la super-classe `Animal` est hérité par l'héritage commun des sous-classes `Ovipare` et `Vivipare`.

Constructeurs et classes virtuelles

Dans une dérivation non-virtuelle, le constructeur d'une sous-classe ne fait appel explicitement *qu'aux constructeurs de ses super-classes immédiates* (et ceci récursivement)

Dans une dérivation virtuelle, la super-classe virtuelle est **initialisée par sa classe la plus dérivée**

- ☞ La sous-classe la plus dérivée **doit** donc faire *appel au constructeur de la super-classe virtuelle*

Exemple:

```
Ovovivipare::Ovovivipare(string nom, Habitat habitat,
                         Regime regime,
                         unsigned int nb_oeufs,
                         unsigned int gestation,
                         bool espece_rare)
    : Animal(nom,habitat,regime), //ICI
      Ovipare(nb_oeufs),
      Vivipare(gestation),
      espece_rare(false)
{}
```

Constructeurs et classes virtuelles (2)

Comment sont gérés les appels au constructeur de la super-classe virtuelle ?

Si le constructeur d'un objet de la classe la plus dérivée est invoqué, les appels explicites au constructeur de la super-classe virtuelle dans les classes intermédiaires **sont ignorés**.

Si la super-classe virtuelle a un *constructeur par défaut*, il n'est pas nécessaire de faire appel à ce constructeur explicitement dans sa classe la plus dérivée.

Si l'appel explicite au constructeur de la super-classe virtuelle est omis de la sous-classe la plus dérivée et si cette super-classe virtuelle n'a pas de constructeur par défaut, *la compilation signalera une erreur*.

Exemple...

Constructeurs et classes virtuelles (3)

Exemple:

```
Ovovivipare::Ovovivipare(string nom, Habitat habitat,
                         Regime regime, unsigned int nb_oeufs,
                         unsigned int gestation,
                         bool espece_rare)
: Animal(nom,habitat,regime),
  Ovipare(nb_oeufs),
  Vivipare(gestation),
  espece_rare(espece_rare)
{}
```

- les appels explicites au constructeur de la classe `Animal` dans les constructeurs de `Ovipare` et `Vivipare` *sont ignorés*
- si `Animal` avait un constructeur par défaut, il n'est pas nécessaire de l'invoquer explicitement dans le constructeur de `Ovovivipare`
- si `Ovovivipare` ne fait pas appel explicitement au constructeur de `Animal` et si `Animal` n'a pas de constructeur pas défaut, il y aura une erreur à la compilation.

Ordre des constructeurs/destructeurs

Dans une hiérarchie de classes où il existe des super-classes virtuelles :

- le soin d'initialiser les super-classes virtuelles incombe à la sous-classe la plus dérivée
- les constructeurs des super-classes virtuelles *sont invoqués en premier*
- ceux des classes non-virtuelles le sont ensuite *dans l'ordre de déclaration de l'héritage*
- l'ordre d'appel des *constructeurs de copie* est identique
- l'ordre d'appel des *destructeurs* est *l'inverse* de celui des appels de constructeurs

Héritage multiple : Conclusion

```
class nomSousClasse: [public] nomSuperClasse1, ...
                     [public] nomSuperClasseN
```

Collision de noms d'attributs/méthodes : c'est la sous-classe qui hérite de ces attributs/méthodes qui doit définir *le sens de leur utilisation*

Classe virtuelle : pour éviter qu'une sous-classe *hérite plusieurs fois d'une même super-classe*, il faut déclarer les dérivations concernées comme **virtuelles**

```
NomSousClasse: [public] virtual NomSuperClasseVirtuelle
```

Constructeur :

```
SousClasse(liste de parametres)
   : SuperClasse1(arguments1),
     ...
     SuperClasseN(argumentsN),
     attribut1(valeur1)
     ...
     attributK(valeurK)
{}
```

Analyse et programmation orientée objet (C++)

Séance 8

Fonctions et Classes Génériques

Programmation générique : introduction

- Une cellule une liste chaînée peut se présenter comme suit:

```
// Une cellule de la liste
class Cellule {
public:
  //....
private:
        double donnee; // une donnee de type double
        PtrCell suite;
};
```

Si l'on veut une liste de `int` ?

- **c'est exactement le même code** pour `Liste` et `Cellule` sauf qu'il faut remplacer le type de la données pouvant être stockée dans une `Cellule`
- Duplication de code !!

Programmation générique

- L'idée de base est de **passer les** types de données **comme paramètres** pour décrire des traitements très généraux (« génériques »)

- Il s'agit donc d'un *niveau d'abstraction supplémentaire*.

- De tels modèles de classes/fonctions s'appellent aussi classes/fonctions génériques ou patrons (chablons), ou encore « ***template »***.

- Vous en connaissez déjà sans le savoir. Par exemple la « classe » `vector` n'est en fait pas une classe mais un modèle de classes : c'est le même modèle que l'on stocke des `char` (`vector<char>`), des `int` (`vector<int>`), ou tout autre objet.

Un exemple

Prenons un exemple simple pour commencer :

une fonction échangeant la valeur de 2 variables. Par exemple avec 2 entiers vous écririez une fonction comme:

```
// Echange la valeur de ses arguments
void echange(int& i, int& j) {
        int tmp(i);
        i = j;
        j = tmp;
}
```

Mais vous vous rendez bien compte que vous pourriez faire la même chose (le même **algorithme**) avec deux `double`, ou même deux objets quelconques, pour peu qu'ils aient un constructeur de copie (`Obj tmp(i);`) et un opérateur de copie (`operator=`).

Un exemple (suite)

L'écriture générale serait alors quelque chose comme :

```
// Echange la valeur de ses arguments
void echange(Type& i, Type& j) {
        Type tmp(i);
        i = j;
        j = tmp;
}
```

où `Type` est une représentation **générique** du type des objets à échanger. La façon exacte de le faire en C++ est la suivante :

```
// Echange la valeur de ses arguments
template<typename Type>
void echange(Type& i, Type& j) {
        Type tmp(i);
        i = j;
        j = tmp;
}
```

Un exemple (suite)

On pourra alors utiliser la fonction `echange` avec tout type/classe pour lequel le constructeur de copie et l'opérateur d'affectation (=) sont définis.

Par exemple:

```
int a(2), b(4);
echange(a,b);

double da(2.3), db(4.5);
echange(da,db);

vector<double> va, vb;
echange(va,vb);

string sa("ca marche"), sb("coucou");
echange(sa, sb);
```

Généralisation aux classes

Ce que l'on a fait ici avec une fonction, on peut le généraliser à n'importe quelle classe.

On pourrait par exemple vouloir créer une classe qui réalise une paire d'objets:

```
template<typename T1, typename T2>
class Paire {
public:
    Paire(const T1& un, const T2& deux)
      : premier(un), second(deux) {}
    virtual ~Paire(){}
    T1 get1() const { return premier; }
    T2 get2() const { return second;  }
    void set1(const T1& val) { premier = val; }
    void set2(const T2& val) { second  = val; }
protected:
    T1 premier;
    T2 second;
};
```

Généralisation aux classes (2)

et par exemple créer la classe « paire `string`–`double` » :

```
Paire<string,double>
```

ou encore la classe « paire `char`–`unsigned int` » :

```
Paire<char,unsigned int>
```

Note : un tel modèle de classe existe dans la STL : `pair` (défini dans `<utility>`).

Les modèles de classes sont donc **un moyen condensé d'écrire plein de classes potentielles à la fois**.

(de même que les modèles de fonctions/méthodes sont un moyen condensé d'écrire **plein** de fonctions/méthodes potentielles à la fois)

Déclaration d'un modèle

Pour déclarer un modèle de classe ou de fonction, il suffit de faire précéder sa déclaration du mot clé `template` suivit de ses arguments (qui sont donc des noms génériques de type) suivant la syntaxe:

```
template<typename nom1, typename nom2, ...>
```

Exemple:

```
template<typename T1, typename T2>
class Paire {
...
```

Les types ainsi déclarés (paramètres du modèle) peuvent alors être utilisés dans la définition qui suit, exactement comme tout autre type.

Note : on peut aussi utiliser le mot `class` à la place de `typename`, par exemple :

```
template<class T1, class T2>
class Paire {
...
```

Déclaration d'un modèle (2)

Il est également possible de définir des types par défaut, avec la même contrainte que pour les paramètres de fonction : les valeurs par défaut doivent être placées en dernier.

Exemple:

```
template<typename T1, typename T2 = unsigned int>
class Paire {
...
```

qui permettrait de déclarer la classe « paire `char`–`unsigned int` » simplement par :

```
Paire<char>
```

Définitions externes des méthodes de modèles de classes

Si les méthodes d'un modèle de classes sont définies en dehors de cette classe, elle devront alors aussi être définies comme modèle et être précédées du mot clé `template`, mais...

...il est **de plus absolument nécessaire** d'**ajouter les paramètres du modèle** (les types génériques) **au nom de la classe**

[pour bien spécifier que dans cette définition c'est la classe qui est en modèle et non la méthode.]

exemple sur le transparent suivant

Définitions externes des méthodes de modèles de classes

Exemple:

```
template<typename T1, typename T2> class Paire {
public:
  Paire(const T1&, const T2&);
...
};

// definition du constructeur
template<typename T1, typename T2>
// le constructeur du modele de classe Paire
// parametr'e par T1 et T2
Paire<T1,T2>::Paire(const T1& un, const T2& deux)
  : premier(un), second(deux) { }
```

Instanciation des modèles

La définition des modèles ne génère en elle-même aucun code : c'est juste une **description de plein de codes potentiels**.

Le code n'est produit que lorsque tous les paramètres du modèle ont pris chacun un type spécifique.

Lors de l'utilisation d'un modèle, il faut donc fournir des valeurs pour tous les paramètres (au moins ceux qui n'ont pas de valeur par défaut). On appelle cette opération une **instanciation** du modèle.

L'instanciation peut être implicite lorsque le contexte permet au compilateur de décider de l'instance de modèle à choisir.

Par exemple, dans le code :

```
double da(2.3), db(4.5);
echange(da,db);
```

il est clair (par le contexte) qu'il s'agit de l'instance `echange<double>` du modèle `template<typename T> void echange(T&,T&);` qu'il faut utiliser.

Instanciation des modèles (2)

Mais dans la plupart des cas, on **explicite l'instanciation** lors de la déclaration d'un objet.

C'est ce qui vous faites lorsque vous déclarez par exemple `vector<double> tableau;`

Il suffit dans ce cas de spécifier le(s) type(s) désiré(s) après le nom du modèle de classe et entre `<>`.

L'instanciation explicite peut aussi être utile dans les cas où le contexte n'est pas suffisamment clair pour choisir.

l'appel `monmax(3.14, 7);` est ambigu. Il faudra alors écrire `monmax<double>(3.14, 7);`

```
template <typename Type>
Type monmax(const Type& x, const Type& y) {
   if (x < y) return y;
   else       return x;
 }
```

Modèles, surcharge et spécialisation

Les modèles de fonctions peuvent très bien être surchargés comme les fonctions usuelles.

Par exemple:

```
template<typename Type>
void affiche(const Type& t) {
  cout << "J'affiche " << t << endl;
}
// surcharge pour les pointeurs : on prefere ici
//  ecrire  le contenu plutot que l'adresse.
template<typename Type> void affiche(Type* t) {
  cout << "J'affiche " << *t << endl;
}
```

Note : on aurait même pu faire mieux en écrivant:

```
template<typename Type> void affiche(Type* t) {
        affiche<Type>(*t);
}
```

Modèles, surcharge et spécialisation (2)

Mais les modèles (y compris les modèles de classes) offrent un mécanisme supplémentaire : la **spécialisation** qui permet de définir une version particulière d'une classe ou d'une fonction pour un choix spécifique des paramètres du modèle.

Par exemple, on pourrait spécialiser le second modèle ci-dessus dans le cas des pointeurs sur des entiers:

```
template<> void affiche<int>(int* t) {
   cout << "J'affiche le contenu d'un entier: ";
        << *t << endl;
}
```

La spécialisation d'un modèle (lorsqu'elle est totale) se fait en :

- ajoutant `template<>` devant la définition
- nommant explicitement la classe/fonction spécifiée C'est le `<int>` après `affiche` dans l'exemple ci-dessus.

Exemple de spécialisation de classe

```
template<typename T1, typename T2>
class Paire {
...
};
// specialisation pour les paires <string,int>
template<> class Paire<string,int> {
public:
  Paire(const string& un, int deux)
    : premier(un), second(deux) {}
  virtual ~Paire() {}
  string get1() const { return premier; }
  int get2()    const { return second;  }
  void set1(const string& val) { premier = val; }
  void set2(int val) { second  = val; }

  // une methode de plus
  void add(int i) { second += i; }

protected:
  string premier;
  int    second;
};
```

Spécialisation : remarques

- **Note 1 :** La spécialisation peut également s'appliquer uniquement à une méthode d'un modèle de classe sans que l'on soit obligé de spécialiser toute la classe. Utilisée de la sorte, la spécialisation peut s'avérer particulièrement utile.

- **Note 2 :** La spécialisation n'est pas une surcharge car il n'y a pas génération de plusieurs fonctions de même nom (de plus que signifie une surcharge dans le cas d'une classe ?) mais bien une **instance spécifique** du modèle.

- **Note 3 :** il existe aussi des spécialisations partielles (de classe ou de fonctions).

Modèles de classes et compilation séparée

Les modèle de classes doivent nécessairement être définis au moment de leur instanciation afin que le compilateur puisse générer le code correspondant.

Ce qui implique, lors de compilation séparée, que les fichiers d'en-tête (`.h`) doivent contenir non seulement la déclaration, mais également la définition complète de ces modèles !!

On ne peut donc pas séparer la déclaration de la définition dans différents fichiers... Ce qui présente plusieurs inconvénients:

- Les mêmes instances de modèles peuvent être compilées plusieurs fois,
- et se retrouvent en de multiples exemplaires dans les fichiers exécutables.
- On ne peut plus cacher leurs définitions (par exemple pour des raisons de confidentialité, protection contre la concurrence, etc...)

Templates

Déclarer un modèle de classe ou de fonction:

```
template<typename nom1, typename nom2, ...>
```

Définition externe des méthodes de modèles de classes:

```
template<typename nom1, typename nom2, ...>
NomClasse<nom1, nom2, ...>::NomMethode(...
```

Instanciation : spécifier simplement les types voulus après le nom de la classe/fonction, entre `<>` (Exemple : `vector<double>`)

Spécialisation (totale) de modèle pour les types *type1*, *type2*... :

```
template<> NomModele<type1,type2,...> ...suite de la declaration...
```

Compilation séparée : pour les templates, il faut tout mettre (déclarations et définitions) dans le fichier d'en-tête (`.h`).

```
class ConteneurGenerique {
  private:
     void *element;
  private:
     void *getElement() { return element; }
     void setElement(void *e) { element = e; }
};
```

```
class ConteneurGenerique {
  private:
     void *element;
  protected:
     void *getElement() { return element; }
     void setElement(void *e) { element = e; }
};
```

```
class ConteneurDeChose : private ConteneurGenerique
{
   public:
     Chose *getElement() {
         return (Chose *) ConteneurGenerique::getElement();
}
   void setElement(Chose  *e) {
        ConteneurGenerique::setElement(e);
   }
};
```

```
template <class T>
class Conteneur : private ConteneurGenerique {
public:
   T *getElement() {
     return (T
*)ConteneurGenerique::getElement();
   }

   void setElement(T *e) {
     ConteneurGenerique::setElement(e);
   }
};
```

Cette fois on bénéficie de l'héritage et donc on réutilise le code existant

```
template <class T> class A {
private:
  T *e;
public:
  A(T *e) { this->e = e; }
  template <class X> friend ostream &operator <<(ostream &os,const A<X> &);
};

template <class X> ostream &operator<<(ostream &os,const A<X> &a) {
      os << *(a.e); return os;
}

class B {};
ostream &operator<<(ostream &os,const B &b) {
      os << « objet B"; return os;
}
int main() {
  int i=12;
  A<int> unA(&i);
  cout << unA << endl;
  B unB;
  A<B> a2(&unB);
  cout << a2 << endl;
}
```

Analyse et programmation orientée objet (C++)

Séance 9

Gestion des exceptions

Gestion des erreurs

Les **exceptions** permettent d'anticiper les erreurs qui pourront potentiellement se produire lors de l'utilisation d'une portion de code.

Exemple : on veut écrire une fonction qui calcule l'inverse d'un nombre réel quand c'est possible :

f
entrée : x sortie : $1/x$
Si $x = 0$ *erreur* ***Sinon*** *retourner 1/x*

mais que faire **concrètement** en cas d'erreur ?

Gestion des erreurs (2)

(1) retourner une valeur choisie à l'avance :

```
double f(double x) {
   if (x != 0.0) return 1.0 / x;
   else
      return numeric_limits<double>().max();
}
```

Mais cela

1. **n'indique pas** à l'utilisateur potentiel qu'il a fait une erreur
2. retourne de toutes façons un **résultat inexact** ...
3. suppose une **convention arbitraire** (la valeur à retourner en cas d'erreur)

Gestion des erreurs (3)

(2) afficher un message d'erreur mais que retourner effectivement en cas d'erreur ?...

on retombe en partie sur le cas précédent

```
double f(double x) {
  if (x != 0.0) return 1.0 / x;
  else {
    cerr << "Erreur d'utilisation de f :"
       << "division par 0"
       << endl;
    return numeric_limits<double>().max();
  }
}
```

De plus, cela est **très mauvais** car cela produit de gros effets de bord : modifie cerr alors que ce n'est pas du tout dans le rôle de f

Pensez par exemple au cas où l'on veut utiliser f dans un programme avec une interface graphique... on ne veut alors plus utiliser cerr (mais plutôt ouvrir une fenêtre d'alerte par exemple)

Gestion des erreurs (4)

(3) retourner un code d'erreur :

```
int f(double x, double& resultat) {
  if (x != 0.0) {
    resultat = 1.0 / x;
    return PAS_D_ERREUR;
  }
  else return ERREUR_DIV_ZERO;
}
// PAS_D_ERREUR, ERREUR_DIV_ZERO :
// constantes definies plus haut
```

Cette solution est déjà **beaucoup mieux** car elle laisse à la fonction qui appelle f le soin de décider quoi faire en cas d'erreur.
Cela présente néanmoins l'inconvénient d'être assez lourd à gérer pour finir :

- cas de l'appel d'appel d'appel.... ...d'appel de fonction,
- mais aussi écriture peu intuitive :

```
if (f(x,y) == PAS_D_ERREUR) //...
```

au lieu de :

```
y=f(x);
```

Gestion des erreurs (5)

```
int d1, d2;
d1 = open("toto",O_RDONLY);
if (d1==-1) {
      switch(errno) {
            case EACCESS:
                  fprintf(stderr,“problème d’accès à glouglou\n“);
                  exit(EXIT_FAILURE);
            case EISDIR:
                  fprint(stderr,“glouglou est un répertoire\n“);
                  return -1;
      }
}
d2 = open("titi",O_WRONLY);
if (d2==-1) {
      switch(errno) {
            …
            break;
```

La vision traditionnelle du déroulement d'un programme tel que projeté dans le monde réel :

```
// programme de préparation du repas
sortir les patates du placard
      si le feu prend dans la maison {
      sortir dehors
      appeler les pompiers
      ...
}
aller chercher l'épluche-légumes
      si le feu prend dans la maison {
      sortir dehors
      appeler les pompiers
      ...
}
éplucher une patate
      si le feu prend dans la maison {
      lâcher patate et épluche-légumes
      sortir dehors
      appeler les pompiers
      ...
      }
...
```

Exceptions

Il existe une solution permettant de généraliser et d'assouplir cette dernière solution : déclencher une **exception**

- mécanisme permettant de prévoir une erreur à un endroit et de **la gérer à un autre endroit**

Principe :

- lorsque qu'une erreur a été détectée à un endroit, on la signale en « lançant » un objet contenant toutes les informations que l'on souhaite donner sur l'erreur (« lancer » = créer un objet disponible pour le reste du programme)
- à l'endroit où l'on souhaite gérer l'erreur (au moins partiellement), on peut « attraper » l'objet « lancé » (« attraper » = utiliser)
- si un objet « lancé » n'est pas attrapé du tout, cela provoque l'arrêt du programme : toute erreur non gérée provoque l'arrêt.

Un tel mécanisme s'appelle une exception.

Exceptions (2)

Avantages de la gestion des exceptions par rapport aux codes d'erreurs retournés par des fonctions :

- écriture plus facile, plus intuitive et plus lisible
- la propagation de l'exception aux niveaux supérieurs d'appel (fonction appelant une fonction appelant ...) est fait **automatiquement**

plus besoin de gérer obligatoirement l'erreur au niveau de la fonction appelante

- une erreur peut donc se produire à n'importe quel niveau d'appel, elle sera toujours reportée par le mécanisme de gestion des exceptions

(**Note :** si une erreur peut être gérée localement, alors il faut le faire localement et ne pas utiliser le mécanisme des exceptions.)

Les exceptions dans le monde réel :

```
// programme de préparation du repas
sortir les patates du placard
aller chercher l'épluche-légumes
  {épluchage}
    éplucher une patate
    s'il reste une patate non épluchée
      alors continuer l'{épluchage}
sortir friteuse
remplir friteuse d'huile de friture
allumer le gaz
couper les patates en frites
attendre que l'huile atteigne 180°
baigner les frites dans l'huile
attendre que les frites soient cuites
sortir les frites
attendre que l'huile atteigne 180°
baigner les frites dans l'huile
attendre que les frites soient grillées
sortir les frites
```

```
Consignes en cas d'incendie

sauter par la fenêtre
Appeler les pompiers
appeler au secours
appeler son assureur
```

Syntaxe de la gestion des exceptions

On cherche à remplir 3 tâches élémentaires :

1. signaler une erreur
2. marquer les endroits réceptifs aux erreurs
3. leur associer (à chaque endroit réceptif aux erreurs) un moyen de gérer leurs erreurs

On a donc 3 mots-clés du langage C++ dédiés à la gestion des exceptions :

throw indique l'erreur (i.e. « lance » l'exception)

try indique un bloc réceptif aux erreurs

catch gère les erreurs associées (i.e. les « attrape »)

Notez bien que :

- L'indication des erreurs (throw) et leur gestion (try/catch) sont le plus souvent à des endroits bien séparés dans le code
- Chaque bloc try possède son/ses catch associé(s)

throw

throw est l'instruction qui **signale l'erreur** au reste du programme.

Syntaxe : throw expression

l'expression peut être de tout type : c'est le résultat de son évaluation qui est « lancé » au reste du programme pour être « attrapé »

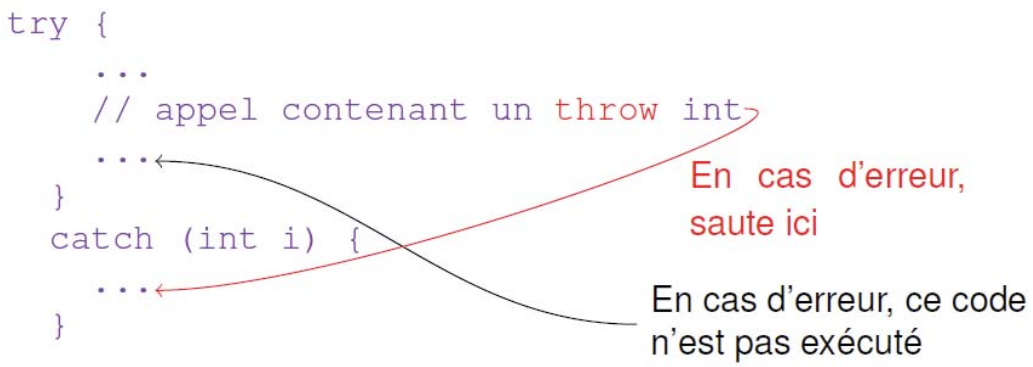
```
throw 21; // "lance" un entier
// "lance" une string:
throw string("quelle erreur !");
struct Erreur {
  int code;
  string message;
};
//...
Erreur faute;
//...
faute.code = 12; faute.message = "Division par 0";
throw faute;
```

throw (2)

throw, en « lançant » une exception, interrompt le cours normal d'exécution et :

- saute au bloc catch du bloc try directement supérieur (dans la pile des appels), si il existe ;
- quitte le programme (« abort ») si l'exécution courante n'était pas dans au moins un bloc try.

Exemple :



try

try introduit un **bloc réceptif aux exceptions** lancées par des instructions, ou des fonctions appelées à l'intérieur de ce bloc (ou même des fonctions appelées par des fonctions appelées par des fonctions... ... ... à l'intérieur de ce bloc)

Exemple 1:

```
try {
  ...
  if (x == 0.0) throw string("valeur nulle");
  //...
}
```

Exemple 2:

```
try {
    //  ...
  y = f(x); // f pouvant lancer une exception
    //  ...
}
```

catch

catch est le mot-clé introduisant un **bloc dédié à la gestion** d'une ou plusieurs **exceptions**.

Tout bloc try doit toujours être suivi d'au moins un bloc catch gérant les exceptions pouvant être lancées dans ce bloc try.

Si une exception est lancée mais n'est pas interceptée par le catch correspondant, le programme s'arrête (« Aborted »).

Syntaxe :

```
catch (type nom) {
  //...
}
```

intercepte toutes les exceptions de type type lancées depuis le bloc try précédent

Exemple d'utilisation de catch

```
try {
  ...
  if (x == 0.0) throw string("valeur nulle");
  ...
    if (j >= 3) throw j;
}
// capture les exceptions lancees sous forme de string
catch(string const& erreur) {
  cerr << "Erreur : " << erreur << endl;
}
// capture les exceptions lancees sous forme d'int
catch(int erreur) {
  cerr << "Avertissement : je n'aurais pas du avoir"
     << " la valeur "
     << erreur
     << endl;
}
```

catch (flot d'exécution 1/3)

Un bloc catch n'est exécuté **que** si une exception de type correspondant a été lancée depuis le bloc try correspondant.

Sinon le bloc catch est simplement ignoré.

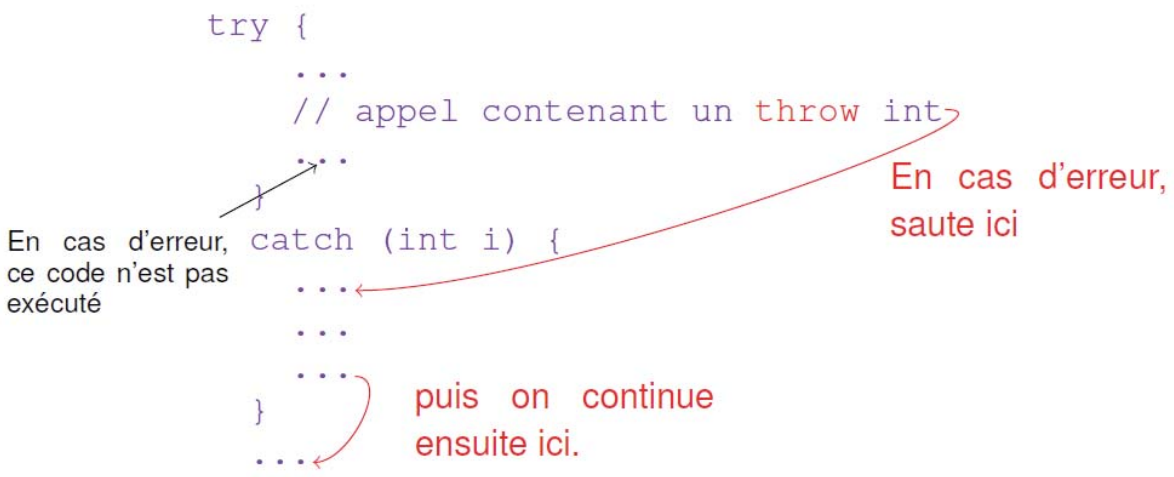
Si un bloc catch est exécuté, le déroulement continue ensuite normalement **après** ce bloc catch (ou après le dernier des blocs catch du même bloc try lorsqu'il y en a plusieurs).

En aucun cas l'exécution ne reprend après le throw !

catch (flot d'exécution 2/3)

Exemple :

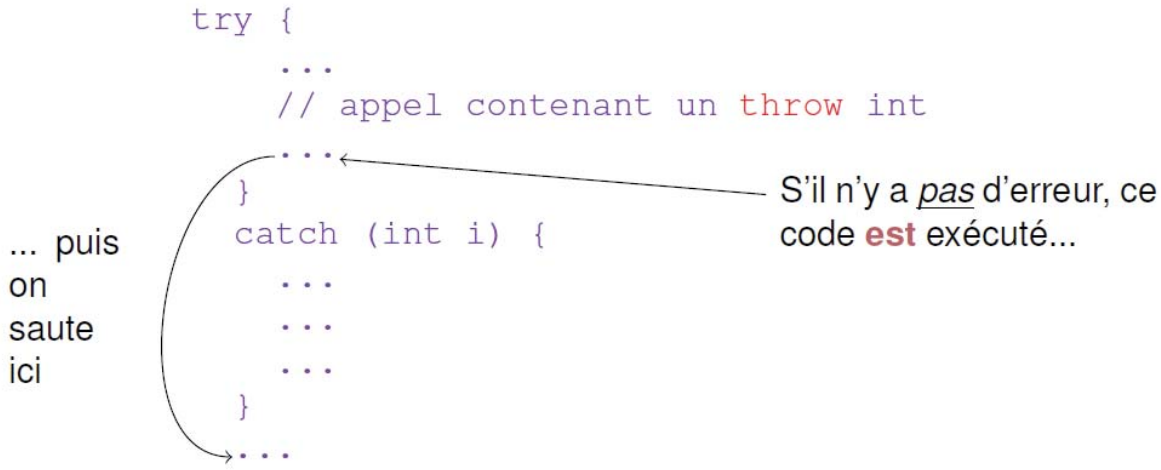
en cas d'erreur (lancement d'une exception) :



catch (flot d'exécution 3/3)

Exemple :

si il n'y a pas d'erreur (**pas** de lancement d'exception) :



catch (Remarques)

Notes :

- « catch(...) » permet d'intercepter n'importe quel type d'exceptions mais, dans le cas où il y a plusieurs catch associés à un même try, « catch(...) » doit être le dernier.
- comme pour les fonctions, on préférera passer les exceptions de type complexe par références constantes :

```
catch (Erreur const& e)
```

Redéclenchement

Une exception peut être **partiellement traitée** par un bloc catch et attendre un traitement plus complet ultérieur (c'est-à-dire à un niveau supérieur).

Il suffit pour cela de **« relancer » l'exception** au niveau du bloc n'effectuant que le traitement partiel. (Il faudra bien sûr pour cela que l'appel à ce bloc catch soit lui-même dans un autre bloc try à un niveau supérieur).

Pour « relancer » une exception, il suffit simplement d'écrire throw (i.e. sans argument)

Exemple :

```
catch (int erreur) {
// traitement partiel :
  cerr << "Hmm... pour l'instant je ne sais pas trop "
     << "quoi faire" << endl
     << "avec l'erreur " << erreur << endl;
// relance l'exception capt'ee:
  throw;
}
```

```
void ouvreFichiers(char *nom) {
  FILE *f = fopen(nom,"r");
  char n[100];
  int lineno=0;
  while ( fgets(n,100,f)!=0 ) {
    try {
      lineno++;
      ouvreFichier(n);
    } catch (ErreurFichier err) {
        err.setLine(lineno);
        throw;
    }
  }
  fclose(f);
}
```

par contre ici on connaît le numéro de ligne

```
void ouvreFichier(char *nom) {
  ...
  if (problème)
    throw ErreurFichier(nom);
  ...
}
```

Ici on ne connaît que le nom

```
void uneFonction() {
  try {
    ouvreFichiers("app.conf");
  } catch(ErreurFichier err) {
    cout << "Impossible d'ouvrir le fichier "
         << err.getName() << " en ligne " << err.getLineNo()
         << " du fichier app.conf" << endl;
  }
}
```

Exemple (1/4)

```
#include <iostream>
#include "mesures.h"
#include "acquisition.h"
#include "plot.h"
using namespace std;

void plot_temp_inverse(Mesures const&);
double inverse(double);

int main() {
  Mesures temperatures;
  acquerir_temp(temperatures);
  plot_temp_inverse(temperatures);
  return 0;
}

void plot_temp_inverse(Mesures const& t) {
  for (unsigned int i(0); i < t.size(); ++i) {
    plot(inverse(t[i]));
  }
}

double inverse(double x) {
  return 1.0/x;
}
```

Exemple (2/4)

```
...
using namespace std;

const int DIVZERO(33);

void plot_temp_inverse(Mesures const&);
double inverse(double);

int main() {
  Mesures temperatures;
  acquerir_temp(temperatures);
  plot_temp_inverse(temperatures);
  return 0;
}

void plot_temp_inverse(Mesures const& t) {
  for (unsigned int i(0); i < t.size(); ++i){
    plot(inverse(t[i]));
  }
}

double inverse(double x) {
  if (x == 0.0) throw DIVZERO;
  return 1.0/x;
}
```

Exemple (3/4)

```
...
int main() {
  Mesures temperatures;
  acquerir_temp(temperatures);
  try {
    plot_temp_inverse(temperatures);
  }
  catch (int i) {
    if (i == DIVZERO) {
      cerr << "Courbe des températures erronée" <<endl;

      /* on fait quelque chose, par exemple refaire
       * les mesures, mais à ce stade le programme
       * n'est pas stoppé.
       */
    }
  }
  return 0;
}
...
```

Exemple (4/4)

```
int main() {
  ...
  try {
    plot_temp_inverse(temperatures);
  }
  catch (int i) {
    if (i == DIVZERO) {
      cerr << "Courbe des températures erronée" <<endl;

      // effectue ici un traitement de plus haut niveau
      ...
    }
...
void plot_temp_inverse(Mesures const& t) {
  for (unsigned int i(0); i < t.size(); ++i) {
    try {
      plot(inverse(t[i]));
    }
    catch (int j) {
      /* Traiter partiellement le problème et relancer l'exception.
       * Cette partie du programme peut par exemple signaler
       * l'indice de la valeur erronée.
       */
      cerr << "problème avec la valeur " << i << endl;
      throw;
    }
  }
}
```

Exemple complet avec reprise (1/3)

```
#include <iostream>
#include "mesures.h"
#include "acquisition.h"
#include "plot.h"
using namespace std;

const int DIVZERO(33);

void plot_temp_inverse(Mesures const&);
double inverse(double);

int main()
{
  Mesures temperatures;
  unsigned int const MAX_ESSAIS(2);
  unsigned int nb_essais(0);
  bool restart(false);

  do {
    ++nb_essais; restart=false;
    acquerir_temp(temperatures);
    try {
      plot_temp_inverse(temperatures);
    }
```

Exemple complet avec reprise (2/3)

```
    catch (int i) {
      if (i == DIVZERO) {
        if (nb_essais < MAX_ESSAIS) {
          cout << "Il faut re-saisir les valeurs" << endl;
          restart = true;
        } else {
          cout << "Il y a déjà eu au moins " << MAX_ESSAIS
               << " essais." << endl;
          cout << " -> abandon" << endl;
        }
      } else {
        cout << "Ne sais pas quoi faire -> abandon" << endl;
      }
    }
  } while (restart);

  return 0;
}

void plot_temp_inverse(Mesures const& t)
{
  for (unsigned int i(0); i < t.size(); ++i) {
```

Exemple complet avec reprise (3/3)

```
        // Exemple de traitement local partiel du problème
        // (ce n'est pas obligatoire).
        try {
          plot(inverse(t[i]));
        }
        catch (int j) {
          cerr << "Erreur : ";
          if (j == DIVZERO) {
            cerr << "la valeur " << i << " est nulle.";
          } else {
            cerr << "problème avec la valeur " << i;
          }
          cerr << endl;
          throw;
        }
      }
    }

    double inverse(double x)
    {
      if (x == 0.0) throw DIVZERO;
      return 1.0/x;
    }
```

Exception lancée par new

new (allocation dynamique de pointeur), retourne une exception de type bad_alloc (défini dans la bibliothèque « new ») si l'allocation dynamique ne se passe pas correctement.

Il est donc conseillé d'écrire par exemple :

```
#include <new>
try {
  ...
  ptr = new ...;
  ...
    }
catch (std::bad_alloc const& e) {
  cerr << "Erreur : plus assez de memoire !" << endl;
  exit 1;
}
```

C++11 Spécification des exceptions

Il est toujours bon en programmation d'**être** le plus **explicite** possible, en particulier sur ce que fait chaque fonction/méthode.

Dans cet esprit, dans un contexte où l'on prévoit d'introduire/de gérer des exception, il est utile d'indiquer les fonctions/méthodes qui ne lancent pas d'exception.

Cela se fait au niveau de leur prototype en ajoutant

```
noexcept
```

derrière le **prototype** de la fonction.

Cela indique simplement que la fonction ne peut pas lancer d'exception (et si elle le fait, le programme se termine immédiatement (par un appel à la fonction terminate()).

Exemple :

```
double f(double) noexcept;
```

Exceptions

throw expression; lance l'exception définie par l'expression

try { ... } introduit un bloc sensible aux exceptions

catch (type& nom) { ... } bloc de gestion de l'exception

Tout bloc try doit toujours être suivit d'un bloc catch gérant les exceptions pouvant être lancées dans ce bloc try.

Si une exception est lancée mais n'est pas interceptée par le catch correspondant, le programme s'arrête (« Aborted »).

le code précédent contient une inconsistance majeure...
Que devient le *handler* de fichier f en cas d'exception ?

```
void ouvreFichiers(char *nom) {
  FILE *f = fopen(nom,“r“);
  char n[100];
  int lineno=0;
  while ( fgets(n,100,f)!=0 ) {
    try {
      lineno++;
      ouvreFichier(n);
    } catch (ErreurFichier err) {
        err.setLine(lineno);
        throw;
    }
  }
  fclose(f);
}
```

bien sûr on peut prendre la précaution de fermer le fichier au bon moment...
Cela nécessite d'être très attentif.

```
void ouvreFichiers(char *nom) {
  FILE *f = fopen(nom,“r“);
  char n[100];
  int lineno=0;
  while ( fgets(n,100,f)!=0 ) {
    try {
      lineno++;
      ouvreFichier(n);
    } catch (ErreurFichier err) {
        err.setLine(lineno);
        fclose(f);
        throw;
    }
  }
  fclose(f);
}
```

- une solution plus sûre est connue sous le nom d'acquisition de ressources par initialisation (*RAII Resource Allocation Is Initialization*)

- cette technique utilise deux caractéristiques essentielles du langage :
  - toute variable locale est détruite à la sortie du bloc qui la déclare
  - toute destruction de variable fait appel au destructeur...

- on va donc créer une classe intermédiaire

```
void ouvreFichiers(char *nom) {
  FichierOuvert f(nom);
  char n[100];
  int lineno=0;
  while ( fgets(n,100,f)!=0 ) {
    try {
      lineno++;
      ouvreFichier(n);
    } catch (ErreurFichier err) {
        err.setLine(lineno);
        throw;
    }
  }
}
```

```
class FichierOuvert {
  private:
    FILE *file;
  public:
    FichierOuvert(char *nom) {
      file = fopen(nom,"r");
    }

  ~FichierOuvert() {
    if (file != -1)
      fclose(file);
    file=0;
}
  ...

};
```

- initialisations de membres en cas d'exception ?

```
class A {
  public:
    A(int v){
      throw 500;
  }
};
```

```
class B {
  public:
    B(int v) {
      ...
      throw 500;
    }
};
```

```
class C {
  private:
    A a;
    B b;
  public:
    C(int x,int y) : a(x), b(y) { ... }
};
```

- comment, en plus de cela, faire en sorte qu'un code soit exécuté alors que l'initialisation d'un membre a echoué ?

- utiliser un **function-try-block**

- initialisations de membres en cas d'exception ?

```
class C {
  private:
    A a;
    B b;
  public:
    C(int x,int y) try : a(x), b(y) {
      ... // ok : something to do
    } catch(Type err) {
      ... // wrong : something else
    }
};
```

Analyse et programmation orientée objet (C++)

Séance 10

STL et autres outils standards

Objectifs

L'objectif du cours d'aujourd'hui est de vous présenter (sommairement) un certains nombre d'**outils** standards existant en C++

Le but ici n'est pas d'être exhaustif, mais simplement de vous :

- informer de l'existence des **principaux** outils
- faire prendre conscience d'aller **lire/chercher dans la documentation** les éléments qui peuvent vous être utiles

Bibliothèque standard

La bibliothèque standard (d'outils) C++ **facilite la programmation** et permet de la rendre **plus efficace**, si tant est que l'on connaisse bien les outils qu'elle fournit.

Cette bibliothèque est cependant vaste et complexe, mais elle peut dans la plupart des cas s'utiliser de façon très simple, facilitant ainsi la réutilisation des structures de données abstraites et des algorithmes sophistiqués qu'elle contient.

La bibliothèque standard C++11 est formée de 79 « paquets » :

- 33 « classiques » (C++98)
- 20 nouveaux (C++11)
- les 26 bibliothèques C (C99)

Contenu de la bibliothèque standard

La bibliothèque standard C++ contient 33 « paquets » de C++-98 :

<algorithm>	plusieurs algorithmes utiles
<bitset>	gestions d'ensembles de bits
<complex>	les nombres complexes
<deque>	tableaux dynamiques avec push_front
<exception>	diverses fonctions aidant à la gestion des exceptions
<fstream>	manipulation de fichiers
<functional>	objets fonctions
<iomanip>	manipulation de l'état des flots
<ios>	définitions de base des flots
<iosfwd>	anticipation de certaines déclarations de flots
<iostream>	flots standards
<istream>	flots d'entrée
<iterator>	itérateurs
<limits>	diverses bornes concernant les types numériques
<list>	listes doublement chaînées
<locale>	contrôles liés au choix de la langue

Contenu de la bibliothèque standard (2)

<map>	tables associatives clé–valeur ordonnées
<memory>	gestion mémoire pour les containers
<new>	gestion mémoire
<numeric>	fonctions numériques
<ostream>	flots de sortie
<queue>	files d'attente
<set>	ensembles ordonnés
<sstream>	flots dans des chaînes de caractères
<stack>	piles
<stdexcept>	gestion des exceptions
<streambuf>	flots avec tampon (buffer)
<string>	chaînes de caractères
<strstream>	flots dans des chaînes de caractère [en mémoire]
<typeinfo>	information sur les types
<utility>	divers utilitaires
<valarray>	tableaux orientés vers les valeurs
<vector>	tableaux dynamiques

Contenu de la bibliothèque standard (3)

La bibliothèque standard C++ contient 20 nouveaux « paquets » de C++11 :

<array>	tableaux de taille fixe
<atomic>	expression atomique
<chrono>	heures et chronomètres
<codecvt>	conversions d'encodage de caractères
<condition_variable>	concurence (multi-thread)
<forward_list>	listes simplement chaînées
<future>	concurence (multi-thread)
<initializer_list>	listes d'initialisation
<mutex>	concurence (multi-thread)
<random>	nombres aléatoires
<ratio>	constantes rationnelles (Q)
<regex>	expressions régulières
<scoped_allocator>	allocation mémoire
<system_error>	erreurs système
<thread>	concurence (multi-thread)
<tuple>	*n*-uples
<type_traits>	caractéristiques de types

Contenu de la bibliothèque standard (4)

<typeindex>	utiliser les types comme index de containers
<unordered_map>	tables associatives non ordonnées
<unordered_set>	ensembles non ordonnés

Il existe aussi dans les outils standards les 26 « paquets» venant du langage C (C99) :

<cassert>	test d'invariants lors de l'exécution
<ccomplex>	(inutile en C++) = <complex>
<cctype>	diverses informations sur les caractères
<cerrno>	code d'erreurs retournés dans la bibliothèque standard
<cfenv>	manipulation des règles de gestion des nombres en virgule flotante
<cfloat>	diverses informations sur la représentation des réels
<cinttypes>	int de taille fixée (C99)
<ciso646>	(inutile en C++)
<climits>	diverses informations sur la représentation entiers
<clocale>	adaptation à diverses langues
<cmath>	diverses définitions mathématiques
<csetjmp>	branchement non locaux

Contenu de la bibliothèque standard (5)

<csignal>	contrôle des signaux (processus)
<cstdalign>	(inutile en C++)
<cstdarg>	nombre variables d'arguments
<cstdbool>	(inutile en C++)
<cstddef>	diverses définitions utiles (types et macros)
<cstdio>	entrées sorties de base
<cstdint>	sous-partie de cinttypes
<cstdlib>	diverses opérations de base utiles
<cstring>	manipulation des chaînes de caractères à la C
<ctgmath>	<cmath> + <complex>
<ctime>	diverses conversions de date et heures
<cuchar>	char de 16 ou 32 bits
<cwchar>	utilisation des caractères étendus
<cwctype>	classification des codes de caractères étendus

Outils standards

On distingue plusieurs types d'outils. Parmi les principaux :

- les containers de base
- les containers avancés (appelés aussi « adaptateurs »)
- les itérateurs
- les algorithmes
- les outils numériques
- Les traitements d'erreurs
- les chaînes de caractères
- les flots

Outils standards (2)

Les outils les plus utilisés sont :

- les chaînes de caractères (string)
- les flots (stream)
- les tableaux dynamiques (vector) [container]
- les listes chaînées (list) [container avancé]
- les piles (stack) [container avancé]
- les algorithmes de tris (sort)
- les algorithmes de recherche (find)
- les itérateurs (iterators)

String

- Les chaînes de caractères C++ sont définies par le type **string**. (classe)
- Pour utiliser des chaînes de caractères, il faut tout d'abord **importer la bibliothèque:**

```
#include <string>
```

- La déclaration d'une chaîne de caractères se fait alors avec :

```
string identificateur;
```

- Exemple :

```
#include <string>
// declaration
string str1;
// declaration avec initialisation
string str2("Bonjour tout le monde !");
```

String

- Comme pour n'importe quel autre type, toute variable de type string (qui n'a pas été déclarée comme constante) peut être modifiée par une affectation.

- Exemple :

```
string chaine ; // -> chaine vaut ""
string chaine2("test") ; // -> chaine2 vaut "test"
chaine = "test3" ; // -> chaine vaut "test3"
chaine = chaine2 ; // -> chaine vaut "test"
chaine = 'a' ; // -> chaine vaut "a"
```

- ***Remarque 1 :*** dans le cas de l'affectation d'un caractère, la valeur affectée à la chaîne est la chaîne réduite au caractère affecté.
- ***Remarque 2 :*** Toute variable déclarée mais non initialisée est automatiquement initialisée à la valeur correspondant à la chaîne vide ("").

String

- La **concaténation** de chaînes est représentée par l'opérateur +.
- chaine1 + chaine2 correspond à une **nouvelle chaîne** associée à la valeur littérale constituée de la concaténation des valeurs littérales de chaine1 et de chaine2.
- Les combinaisons suivantes sont possibles pour la concaténation de deux chaînes :

```
string + string, string + "...", "..." + string, string + char,    char + string
```

où ***string*** correspond à une variable de type string, ***"..."*** correspond à une valeur littérale et ***char*** à une variable ou une valeur littérale de type char.

String

- Exemples:

```
string nom;
string prenom;
string famille;
...
nom = famille + ' ' + prenom;
```

- Ajout d'un 's' final au pluriel :

```
string reponse("solution");
//...
if (n > 1) {
      reponse = reponse + 's';
}
```

String

- Les **opérateurs relationnels** sont **également définis** pour les chaînes de type string (*ordre alphabétique*).
  - **==** égalité
  - **!=** non-égalité
  - **<** strict. inférieur
  - **<=** inférieur ou égal
  - **>** strict. supérieur
  - **>=** supérieur ou égal

String

- Si ***chaine*** est une ***string***, alors ***chaine[i]*** est le ***(i+1)ème*** caractère de chaine (de type ***char***).

```
string demo("ABCD");
char prems;
char der;
prems = demo[0]; // recoit 'A'
der = demo[3]; // reçoit 'D'
```

String

- Certaines fonctions *propres aux* string sont définies. Elle s'utilisent avec la syntaxe suivante :

```
nom_de_chaine.nom_de_fonction(arg1,arg2,...);
```

- ***chaine.size()*** : renvoie la taille (le nombre de caractères) de ***chaine***.
- ***chaine.insert(position, chaine2)*** : insère, à partir de la position (indice) ***position*** dans la chaîne ***chaine***, la string ***chaine2***
- Exemple :

```
string exemple("abcd"); // exemple vaut "abcd"
exemple.insert(1,"xx"); // exemple vaut "axxbcd"
```

String

- **`chaine.replace(position, n, chaine2)`** : remplace les **`n`** caractères d'indice **`position, position+1, ..., position+n-1`** de **`chaine`** par la string **`chaine2`**.
- **`chaine.find(souschaine)`** : renvoie l'indice dans **`chaine`** du 1er caractère de l'occurrence **`la plus à gauche`** de la string **`souschaine`**.
- **`chaine.rfind(souschaine)`** : renvoie l'indice dans **`chaine`** du 1er caractère de l'occurrence **`la plus à droite`** de la string **`souschaine`**.
- **`chaine.substr(depart, longueur)`** : renvoie la chaîne de **`chaine`**, de longueur **`longueur`** et commençant à la position **`depart`**.
- Dans les cas où les fonctions **`find()`** et **`rfind()`** ne peuvent s'appliquer, elles renvoient la valeur prédéfinie **`string::npos`**

Containers

Comme le nom l'indique, les containers sont des structures de données abstraites (SDA) servant à **contenir** (« collectionner ») **d'autres objets**.

Vous en connaissez déjà plusieurs : les tableaux, les piles et les listes chaînées.

Il en existe plusieurs autres, parmi lesquels, les files d'attentes (queue), les ensembles (set, unordered_set) et les tables associatives (map, unordered_map).

Containers (2)

Les **files d'attente** sont des piles où c'est le premier arrivé (empilé) qui est dépilé le premier... comme dans une file d'attente à un guichet !

(alors que dans une pile « normale », c'est toujours le dernier arrivé qui est dépilé en premier)

Les set permettent de gérer des **ensembles** (finis !) au sens mathématique du terme : collection d'éléments où chaque élément n'est présent qu'une seule fois.

Les **tables associatives** sont une généralisation des tableaux où les index ne sont pas forcément des entiers.

Imaginez par exemple un tableau que l'on pourrait indexer par des chaînes de caractères et écrire par exemple tab["Informatique"]

Containers (3)

Tous les containers contiennent les méthodes suivantes :

bool empty() : le containers est-il vide ?

unsigned int size() : nombre d'éléments contenus dans le container

void clear() : vide le container

iterator erase(*it*) : supprime du container l'élément pointé par *it*. *it* est un itérateur (généralisation de la notion de pointeur, voir quelques transparents plus loin)

Ils possèdent également tous les méthodes begin() et end() que nous verrons avec les *itérateurs*.

Passons maintenant à quelques containers particuliers

Vector

- Nous avons jusqu'ici vu les tableaux de taille fixe (i.e. connue à l'avance).
- Que faire si la taille n'est pas connue ?

- Il existe en C++ ce que l'on appelle des **tableaux dynamiques**, c'est-à-dire dont la taille peut changer au cours du déroulement du programme, par exemple lorsqu'on ajoute ou retire des éléments dans le tableau.

- Les tableaux dynamiques sont définis en C++ par le biais du type vector.

- Pour les utiliser, il faut tout d'abord importer les définitions associées à l'aide d'un include :

```
#include <vector>
```

Vector

- Une variable correspondant à un tableau dynamique se déclare de la façon suivante :

```
vector<type> identificateur;
```

où *identificateur* est le nom du tableau et *type* correspond au type des éléments du tableau.

- Il peut être simple ou composé (on peut donc faire des vector de vector !).

- Exemple :

```
#include <vector>
//...
vector<int> tableau;
```

Vector

- Une taille initiale peut être indiquée si nécessaire.
- La syntaxe de la déclaration/initialisation est alors :

```
vector<type> identificateur(taille);
```

- Exemple : `vector<int> tab(5) ;` correspond à la déclaration d'un tableau initialement constitué de 5 entiers (tous nuls).

Attention ! ce n'est pas la même chose que :

```
int tab[5]; // tableau de taille fixe a la C
```

Vector

- La déclaration d'un tableau avec taille initiale peut en plus être associée à une initialisation explicite des éléments contenus dans le tableau, mais tous à la *même* valeur. Cela s'écrit :

```
vector<type> identificateur(taille, valeur);
```

où *valeur* est la *même* valeur initiale affectée à tous les éléments du tableau

- On peut aussi initialiser un tableau dynamique à l'aide d'une copie d'un autre tableau dynamique:

```
vector<type> identificateur(reference);
```

où ***reference*** est une référence à un tableau de même type de base *type*.

- Exemples :

```
vector<int> tab1(5,1);
vector<int> tab2(tab1);
```

correspondent toutes deux à la déclaration d'un tableau d'entiers dont les 5 éléments de départ sont initialisés à la valeur 1.

Vector

- L'**indexation** des différents éléments d'un tableau dynamique (c'est-à-dire l'accès direct aux éléments du tableau) se fait de la même façon que pour un tableau de taille fixe :
- Si ***tableau*** est un ***vector***, alors ***tableau[i]*** est une référence au (i+1)ème élément de tableau.

Attention ! Il est impératif que cet éléments **existe** effectivement ! Sinon risque de *Segmentation Fault !*

- Exemple (à ne **pas** suivre !) d'erreur classique :

```
vector<double> v;
v[0] = 5.4; // Erreur !! : v[0]
                n'existe pas encore
```

Vector

- Un certain nombre d'opérations sont **directement attachées** au type vector.
- L'utilisation de ces opérations spécifiques se fait avec la syntaxe suivante :

```
nom_de_tableau.nom_de_fonction(arg1, arg2,...);
```

- Exemple :

```
vector<double> tab;
// ...
nombre = tab.size();
```

Vector

- Quelques fonctions disponibles pour un tableau dynamique ***tableau*** de type ***vector<type>*** :
- ***tableau.size()*** : renvoie la taille de tableau (type de retour : ***size_t***)
- ***tableau.front()*** : renvoie une référence au 1er élément
- ***tableau.front()*** est donc équivalent à ***tableau[0]***
- ***tableau.back()*** : renvoie une référence au dernier élément.
- ***tableau.back()*** est donc équivalent à ***tableau[tableau.size()-1]***
- ***tableau.empty()*** : détermine si tableau est vide ou non (***bool***).
- ***tableau.clear()*** : supprime tous les éléments de tableau (tableau vide). Pas de retour.
- ***tableau.pop_back()*** : supprime le dernier élément de tableau. Pas de retour.
- ***tableau.push_back(valeur)*** : ajoute un nouvel élément de valeur ***valeur*** à la fin de tableau. Pas de retour.

Vector

Exemple 1 : les vector

```
#include <iostream>
#include <vector>
using namespace std;

int main() {
  vector<int> v1; // un vecteur d'entier vide

  v1.push_back( 10 ); // ajoute l'entier 10 à la fin
  v1.push_back( 9 ); // ajoute l'entier 9 à la fin
  v1.push_back( 8 ); // ajoute l'entier 8 à la fin
  // enleve le dernier élément
  v1.pop_back(); // supprime l'entier 8

  // utilisation d'un indice pour parcourir le vecteur v1
  cout << "Le vecteur v1 contient " << v1.size() << " entiers : \n";
  for(int i=0;i<v1.size();i++)
      cout << "v1[" << i << "] = " << v1[i] << '\n';
  cout << '\n';

   return 0;
}
```

Vector

Itérer sur un tableau :

- avec une ***itération for*** classique :

```
for (size_t i(0); i < v.size(); ++i)
```

- avec les ***itérations sur ensemble*** de valeurs:
  - Si l'on ne veut pas modifier les éléments du tableau :

```
for (type nom_de_variable : tableau)
```

  - Si l'on veut modifier les éléments du tableau :

```
for (type & nom_de_variable : tableau)
```

où *type* est le type des éléments contenus dans le tableau.

- avec des ***"itérateurs"***

Vector

- Exemples : ***itérations sur ensemble***

```
vector<double> salaires(10);
for(double & salaire : salaires) {
      cout << "Salaire de l'employe suivant ? ";
      cin >> salaire ;
}
cout << " Salaires des employes : " << endl;
for(double salaire : salaires ) {
      cout << " " << salaire << endl;
}
```

Vector : itérateur

- Les **itérateurs** sont une SDA **généralisant** d'une part des accès par ***index*** (SDA séquentielles) et d'autre part les ***pointeurs***, dans le cas de **containers**.

- Ils permettent :
  - de parcourir de façon ***itérative*** les containers
  - d'indiquer (i.e. de pointer sur) un élément d'un container

Vector : itérateur

- Un itérateur associé à un container ***C<type>*** se déclare simplement comme ***C<type>::iterator nom;***
- Exemples :

```
vector<double>::iterator i;
```

- Il peut s'initialiser grâce aux méthodes ***begin()*** ou ***end()*** du container, voire d'autres méthodes spécifiques, comme par exemple ***find*** pour les containers non-séquentiels.
- Exemples :

```
vector<double>::iterator i(vect.begin());
```

- L'élément indiqué par l'itérateur ***i*** est simplement ****i***.

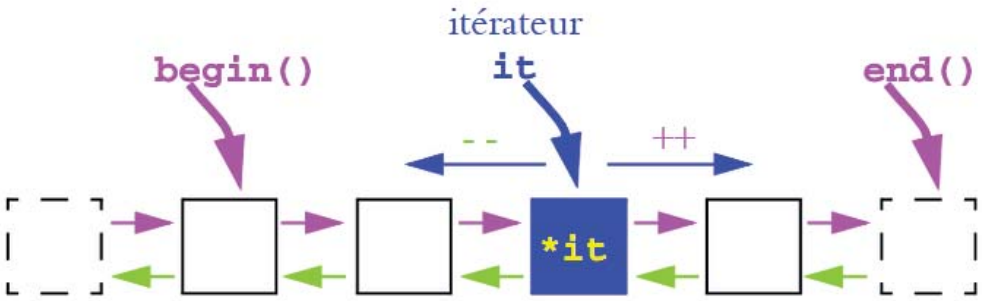
Vector: itérateur

- Exemples : ***itérateurs***

```
vector<double> salaires(10);
    for(vector<double>::iterator i = salaires.begin();
        i != salaires.end(); i++) {

        cout << *i << " " << endl;
    }
```

Vector: itérateur



Vector

- Une fonction peut renvoyer un vecteur.
- Voici une fonction qui recueille toutes les valeurs qui se situent dans un certain intervalle.

```
vector<double> between(vector<double> v, double low,
double high)
{
    vector<double> result;
    for (int i = 0; i < v.size(); i++)
       if (low <= v[i] && v[i] <= high)
          result.push_back(v[i]);
    return result;
}
```

Vector : tableaux multidimensionnels

- Le type de base d'un tableau peut être n'importe quel type, y compris composé. En particulier, le type de base d'un tableau peut être lui même un tableau.
- Les tableaux correspondants sont alors des tableaux de tableaux, c-a-d des **tableaux multidimensionnels**.

- Exemple :

```
vector< vector <int> > tab(5, vector<int>(6));
```

correspond à la déclaration d'un tableau de 5 tableaux de 6 entiers, autrement dit un tableau bidimensionnel à 5 lignes et 6 colonnes.

- Comme pour les tableaux multidimensionnels de taille fixe, `tab[i][j]` permet alors de référencer l'élément de la `(i+1)ème ligne` et de la `(j+1)ème colonne`.

Vector

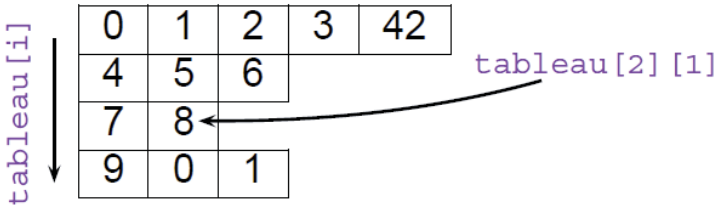
- Notez qu'un ***vector<vector<int>> tab*** n'est pas la même chose qu'un ***int tab[][]*** : dans le premiers cas les lignes de la "***matrice***" n'ont pas forcément toutes la même longueur alors que c'est le cas pour les tableaux de taille fixe.

vector<vector<int> >

- n'est pas une matrice, mais un vecteur de vecteurs d'entiers (pas nécessairement tous de la même taille !).

vector< vector<int> > tableau;

tableau.size() renvoie 4
tableau[0].size() renvoie 5
tableau[2].size() renvoie 2



Tableaux dynamiques : petit complément

Pour accéder directement à un élément d'un tableau dynamique (vector) on utilise l'opérateur [] : tab[i].

Il existe une autre méthode pour cet accès : at(*n*) qui, à la différence de [*n*], lance l'exception out_of_range (de la bibliothèque <stdexcept>) si n n'est pas un index correct.

Exemple :

```
#include <vector>
#include <stdexcept>
...
vector<int> v(5,3); // 3, 3, 3, 3, 3
int n(12);
try {
        cout << v.at(n) << endl;
}
catch (out_of_range) {
        cerr << "Erreur : " << n << " n'est pas correct pour v"
             << endl
             << "qui ne contient que " << v.size()
                  << " elements." << endl;
}
```

Liste (doublement) chaînées

- Les listes chaînées sont, comme les tableaux dynamiques, c'est-à-dire stockant des séquences (ordonnées) d'éléments.
- Par contre dans une liste chaînée, l'accès direct à un élément n'est pas possible, contrairement aux tableaux dynamiques.
- Les listes chaînées sont définies dans la bibliothèque ***list*** et se déclarent de façon similaire à des tableaux dynamiques, par exemple

```
list<int> maliste;
```

Liste (doublement) chaînées

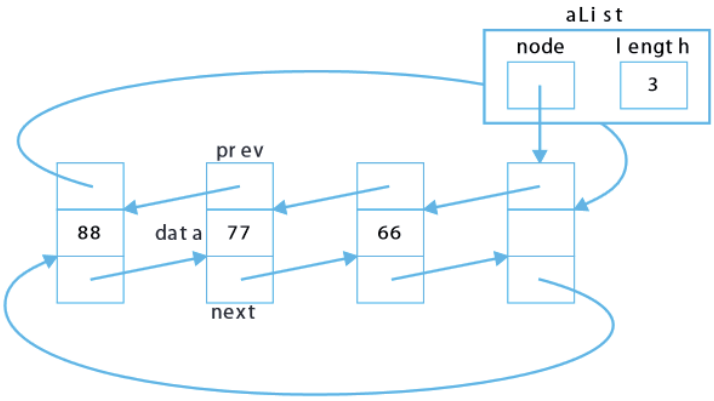
Quelques méthodes des listes chaînées :

- ***Type& front()*** retourne le premier élément de la liste
- ***Type& back()*** retourne le dernier élément de la liste
- ***void push_front(Type)*** ajoute un élément en tête de liste
- ***void push_back(Type)*** ajoute un élément en queue de liste
- ***void pop_front()*** supprime le premier élément
- ***void pop_back()*** supprime le dernier élément
- ***void insert(iterator, Type)*** insertion avant un élément de la liste désigné par un itérateur

Liste (doublement) chaînées

- Exemple:

```
list<int> aList;
aList.push_back(77);
aList.push_back(66);
aList.push_front(88)
```



Entrées/Sorties

- Les interactions d'un programme avec "l'extérieur" sont gérées par des ***instructions d'entrée/sortie*** et ce qu'on appelle des "***flots***".
- Un flot correspond à un canal d'échange de données entre le programme et l'extérieur.
- ***cin*** est le nom de la variable associée par défaut au flot d'entrée. On l'appelle « ***entrée standard*** ».
- ***cout*** est le nom de la variable associée par défaut au flot de sortie. On l'appelle « ***sortie standard*** ».

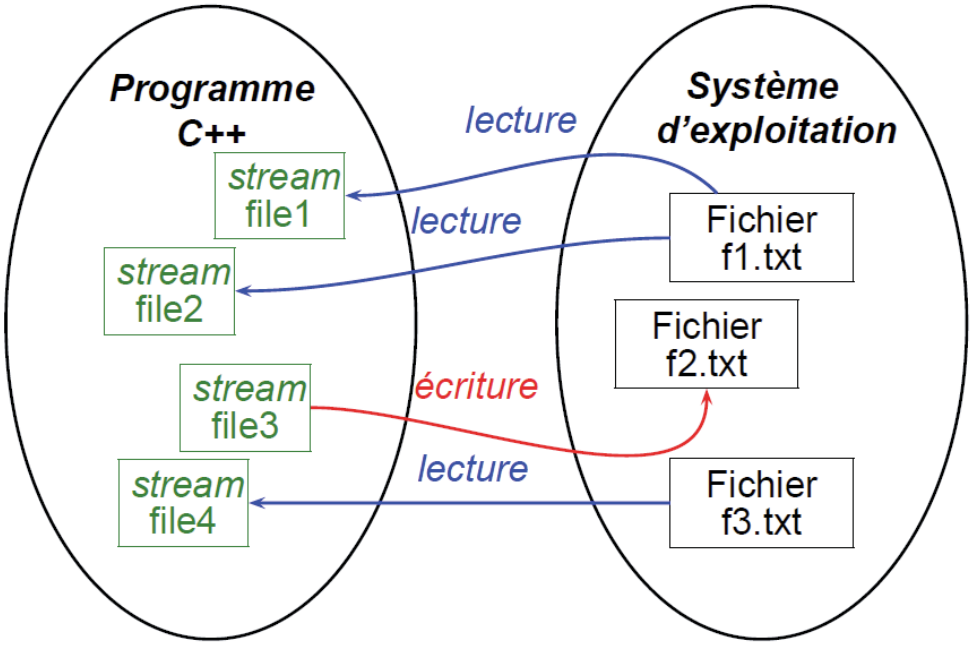
Sortie erreur standard

- En plus de **`cin`** et **`cout`**, il existe une sortie d'erreur standard, **`cerr`**.
- Par défaut **`cerr`** est envoyée sur le terminal, comme **`cout`**. Mais il s'agit bien d'un flot séparé !
- On peut en effet sous plusieurs systèmes d'exploitation séparer la sortie standard de la sortie d'erreur standard : (commandes Unix)
- De plus **`cerr`** n'a pas de mémoire tampon. L'écriture sur **`cerr`** se fait donc directement (on n'a pas besoin de **`flush`**).
  - Conseil : Pour afficher des **`messages d'erreur`** depuis votre programme, préférez **`cerr`** plutôt que **`cout`**.

entrées/sorties avec des fichiers

- Pour **`sauvegarder`** de façon permanente (*i.e.* rendre disponible après la fin de l'exécution du programme) les données produites par un programme, une solution est d'utiliser le **système de fichiers** fourni par le système d'exploitation.
- Il est donc nécessaire d'avoir un objet informatique permettant de gérer les entrées et les sorties du programme vers le système de fichiers.
- Cet objet informatique (appelé un **`flot`**, "stream" en anglais) **représente** en fait ***un lien à un fichier physique*** stocké dans le disque et géré par le système de fichiers.
- L'idée de base des flots est de **séparer l'aspect logique** des entrées-sorties (*i.e. leur intégration dans des programmes*) de leur **aspect physique** (*i.e. leur réalisation par le biais de périphériques particuliers*).

entrées/sorties avec des fichiers



entrées/sorties avec des fichiers

- Pour pouvoir utiliser les flots, il faut tout d'abord inclure les fichiers de définitions correspondant, iostream et fstream :

```
#include <iostream>
#include <fstream>
```

- Deux nouveaux types sont alors disponibles :
  - *ifstream* (pour *input file stream*) qui définit un flot d'**entrée** (similaire à cin)
  - *ofstream* (pour *output file stream*) qui définit un flot de **sortie**. (similaire à cout)

- **Attention !** On ne peut jamais copier de flot (ifstream, ofstream, ...), ni les passer par valeur. Il faut impérativement les passer par référence !

entrées/sorties avec des fichiers

- Mécanisme général pour la mise en oeuvre d'entrées-sorties :
  - **création d'un flot** (d'entrée ou de sortie) par la **déclaration** d'une variable du type correspondant (ifstream ou ofstream).
  - **lien** de la variable déclarée **avec un dispositif** d'entrée-sortie **physique** (fichier)
  - **utilisation** de la variable déclarée et liée pour effectivement réaliser les entrées-sorties
  - **fermeture** du flot
- La plupart de ces actions se font en applicant des fonctions spécifiques définies pour les types ***stream***.
- Ces fonctions sont appelées avec la syntaxe suivante :

```
nom_de_stream.nom_de_fonction(arg1,...);
```

entrées/sorties avec des fichiers

- Pour créer un flot, il suffit de déclarer une variable du type correspondant.

- Exemple :

```
ifstream entree;
ofstream sortie;
```

- déclare deux variables ***entree*** et ***sortie***, respectivement de type ***ifstream*** et ***ofstream***.

entrées/sorties avec des fichiers

- Dans le cas des fichiers textes, l'association d'un flot d'entrée-sortie avec le fichier se fait par le biais de la fonction spécifique open.

- Exemple :

```
ifstream entree;
entree.open("test.txt");
```

associe le ***stream entree*** avec le fichier physique ***test.txt***.

entrées/sorties avec des fichiers

- Dans le cas des fichiers binaires, il faut ajouter un argument supplémentaire :
  ios::in|ios::binary pour la lecture
  ios::out|ios::binary pour l'écriture

- Exemple :

```
ifstream entree;
entree.open("a_lire.zip", ios::in|ios::binary);
ofstream sortie;
sortie.open("a_ecrire.exe", ios::out|ios::binary);
```

entrées/sorties avec des fichiers

- Par défaut, un « *ofstream* » ouvre le fichier en mode "*écrasement*" (c'est-à-dire détruit le contenu du fichier si il existe déjà)
- On souhaite parfois pouvoir ouvrir le fichier en mode "*ajout*" ("append") (c'est-à-dire écrire en fin de fichier)
- Cela se fait aussi en ajoutant un argument supplémentaire à open:

```
ofstream sortie;
sortie.open("a_poursuire.txt", ios::out|ios::app);
```

- Dans le cas de fichiers en binaire :

```
ofstream sortie("a_completer.data",
                ios::out|ios::binary|ios::app);
```

entrées/sorties avec des fichiers

- L'association du stream d'entrée ou de sortie avec un fichier particulier peut aussi se faire directement par une **initialisation** lors de la déclaration du stream :

- Exemple :

```
ifstream entree("test.txt") ;
```

déclare la variable entree et l'associe au fichier texte test.txt.

- Dans le cas de fichiers en binaire :

```
ifstream entree("a_lire.zip", ios::in|ios::binary);
ofstream sortie("a_ecrire.exe", ios::out|ios::binary);
```

- Dans le cas de fichiers en écriture en mode "ajout" :

```
ofstream sortie("a_ajouter.txt", ios::out|ios::app);
```

entrées/sorties avec des fichiers

- L'utilisation directe d'une variable de type string pour donner un nom de fichier **n'est malheureusement pas possible**.
- Le nom du fichier doit être une chaîne de caractères littérale ("...").
- Les variables de type string doivent être utilisées par le biais de la fonction ***c_str()*** spécifique aux variables de type string.

- Exemple :

```
string chaine1("un_fichier");

// ofstream sortie1(chaine1); // NE COMPILE PAS !!
// Bonne solution :
ofstream sortie1(chaine1.c_str());
```

entrées/sorties avec des fichiers

- L'utilisation des variables de type ***ifstream*** ou ***ofstream*** dans les programmes pour réaliser les entrées-sorties se fait de la même façon que pour les flots particuliers ***cin*** et ***cout***, à l'aide des opérateurs **<<** et **>>.**

- Exemple :

```
ifstream entree("fichier_entree");
ofstream sortie("fichier_sortie");
string mot;
...
entree >> mot; // lit un mot dans le fichier
               //"fichier_entree"
sortie << mot; // ecrit ce mot dans le fichier
               //"fichier_sortie"
```

entrées/sorties avec des fichiers

- Une fonction utile pour tester si la lecture d'un fichier (associé à un ***ifstream***) est terminée est la fonction ***eof***.

- Exemple :

```
while ( ! entree.eof() )
    entree >> mot;
```

entrées/sorties avec des fichiers

- La fermeture du stream se fait par la fonction ***close***

- Exemple:

```
ifstream entree("fichier_entree");
...
entree.close();
```

- **NE PAS** oublier de fermer tout fichier ouvert ! En particulier en écriture.

entrées/sorties avec des fichiers

- Il est **important** de **vérifier** que le lien du flot avec le fichier **s'est bien passé** (e.g le fichier existe, il est bien lisible, ...).
- Ceci est fait en testant l'état du flot, par exemple après l'appel à la fonction ***open().***
- En cas de problème, la fonction ***fail()*** du flot retourne la valeur ***true*** si le flot n'est pas dans un état correct pour la prochaine opération (lecture ou écriture). Elle renvoie ***false*** si le flot est prêt.

entrées/sorties avec des fichiers

Exemple de programme de lecture d'un fichier texte de nom « test » :

```
int main() {
  string nom_fichier("test");
  ifstream entree(nom_fichier.c_str());

  if (entree.fail()) {
      cerr << "Erreur : impossible de lire le fichier "
           << nom_fichier << endl;
  } else {
      string mot;

      while (!entree.eof()) {
          entree >> mot ;
          cout << mot << endl;
      }
      entree.close() ;
  }
}
```

entrées/sorties avec des fichiers

Exemple de programme d'écriture dans un fichier texte :

```
main() {
  string nom_fichier;

  cout << "Dans quel fichier voulez vous ecrire ? "
       << flush;
  cin >> ws;
  getline(cin, nom_fichier);

  ofstream sortie(nom_fichier.c_str());
  if (sortie.fail()) {
      cerr << "Erreur : impossible d'ecrire dans le fichier "
           << nom_fichier << endl;
  } else {
      string phrase;

      cout << "Entrez une phrase : " << flush;
      cin >> ws;
      getline(cin, phrase);
      sortie << phrase << endl;
      sortie.close();
  }
}
```

C++11 Liste chaînées

Les listes chaînées sont, comme les tableaux dynamiques, des SDA **séquentielles**, c'est-à-dire stockant des séquences (ordonnées) d'éléments.

Par contre dans une liste chaînée, l'accès direct à un élément n'est pas possible, contrairement aux tableaux dynamiques.

Les listes simplement chaînées sont définies dans la bibliothèque forward_list et se déclarent de façon similaire à des tableaux dynamiques, par exemple

```
forward_list<int> maliste;
```

(quelques) méthodes des listes chaînées :

Type& front()	retourne le premier élément de la liste
void push_front(*Type*)	ajoute un élément en tête de liste
void pop_front()	supprime le premier élément
void insert(*iterator*, *Type*)	insertion avant un élément de la liste désigné par un itérateur

Ensembles – Exemple

Les ensembles (au sens mathématique) sont implémentés dans la bibliothèque <set>. Ils ne peuvent cependant contenir que des éléments du même type, lesquels sont ordonnés par operator<.

(Pour des éléments de même type mais non ordonnés, i.e. *sans* operator<, on utilisera un unordered_set.)

On déclare un ensemble comme les autres containers, en spécifiant le type de ses éléments, par exemple :

```
set<char> monensemble;
```

Les ensembles n'étant pas des SDA séquentielles, l'accès direct à un élément n'est pas possible.

(quelques) méthodes des ensembles :

insert(*Type*)	insère un élément s'il n'y est pas déjà
erase(*Type*)	supprime l'élément (s'il y est)
find(*Type*)	retourne un itérateur indiquant l'élément a recherché

À noter que la bibliothèque <algorithm> fournit des fonctions pour faire la réunion, l'intersection et la différence d'ensembles.

Ensembles – Exemple

```
#include <set>

...
set<char> voyelles;
voyelles.insert('a');
voyelles.insert('b');
voyelles.insert('e');
voyelles.insert('i');
voyelles.erase('b');
voyelles.insert('e'); /* n'insere pas 'e' car *
                       * il y est deja */
```

Comment parcourir cet ensemble ?

```
for (unsigned int i(0); i < voyelles.size(); ++i)
        cout << voyelles[i] << endl;
```

ne fonctionne pas car c'est une SDA non-indexé (et même non-séquentielle).

Ensembles – parcours

Comment parcourir cet ensemble ?

En C++11 c'est facile :

```
for (auto const v : voyelles)
        cout << v << endl;
```

Il y a aussi un autre moyen, plus avancé :

☞ utilisation d'**itérateurs**

Itérateurs

Les **itérateurs** sont une SDA **généralisant** d'une part des accès par index (SDA séquentielles) et d'autre part les pointeurs, dans le cas de **containers**.

Ils permettent :

- de parcourir de façon itérative les containers
- d'indiquer (i.e. de pointer sur) un élément d'un container

Il existe en fait 7 sortes d'itérateurs, mais nous ne parlons ici que de la plus générale, qui permet de tout faire : lecture et écriture du containers, aller en avant ou en arrière (accès quelconque en fait).

Itérateurs (2)

Un itérateur associé à un container *C<type>* se déclare simplement comme *C<type>*::iterator *nom*;

Exemples :

```
vector<double>::iterator i;
set<char>::iterator j;
```

Il peut s'initialiser grâce aux méthodes begin() ou end() du container, voire d'autres méthodes spécifiques, comme par exemple find pour les containers non-séquentiels.

Exemples :

```
vector<double>::iterator i(monvect.begin());
set<char>::iterator j(monset.find(monelement));
```

L'élément indiqué par l'itérateur i est simplement *i, comme pour les pointeurs.

Retour sur l'exemple des ensembles

Pour parcourir notre ensemble précédent, nous devons donc faire :

```
for (set<char>::iterator i(voyelles.begin());
          i != voyelles.end(); ++i)
        cout << *i << endl;
```

Exemple d'utilisation de find :

```
set<char>::iterator i(voyelles.find('c'));

if (i == voyelles.end())
        cout << *i << "n'est pas dans l'ensemble" << endl;
else
        cout << *i << "est dans l'ensemble" << endl;
```

Code complet de l'exemple

```
#include <set>
#include <iterator>
#include <iostream>
using namespace std;
int main() {
        set<char> voyelles;
        voyelles.insert('a');
        voyelles.insert('b');
        voyelles.insert('e');
        voyelles.insert('i');
        voyelles.insert('a'); // ne fait rien car 'a' y est deja
        voyelles.erase('b'); // supprime 'b'
// parcours l'ensemble
for (set<char>::iterator i(voyelles.begin()); i!=voyelles.end(); ++i)
        cout << *i << endl;
// recherche d'un element
set<char>::iterator element(voyelles.find('c'));
if (element == voyelles.end())
        cout << "l'element n'est pas dans l'ensemble" << endl;
else
        cout << *element << " est dans l'ensemble" << endl;
return 0;
}
```

Suppression d'un élément d'un container

On a vu que tout container possédait une méthode

```
iterator erase(it)
```

permettant de supprimer un élément, mais...

Attention ! on **ne** peut **pas** continuer à utiliser l'itérateur *it* sans autre !

(plus exactement : erase rend invalide tout itérateur et référence situé(e) au delà du premier point de suppression)

Exemple d'**erreur** classique :

```
vector<double> v;
...
for (vector<double>::iterator i(v.begin()); i != v.end(); ++i)
        if (cond(*i)) v.erase(i);
```

(avec `bool cond(double);`)

n'est **pas** correct («Segmentation fault»)

pas plus que :

```
for (vector<double>::iterator i(v.begin()); i != v.end(); ++i)
        if (cond(*i)) i = v.erase(i);
```

Suppression d'un élément d'un container (2)

Ce qu'il faut faire c'est :

```
vector<double>::iterator next;
for (vector<double>::iterator i(v.begin()); i != v.end();
                        i = next) {
        if (cond(*i)) { next = v.erase(i); }
        else { next = ++i; }
}
```

ou mieux en utilisant remove_if (ou remove) de <algorithm> :

```
v.erase(remove_if(v.begin(), v.end(), cond), v.end());
```

mais qui sont de toutes façons «**coûteux**» (O(v.size()^2)) (voir transparent suivant)

Suppression d'un élément d'un container (3)

En effet, un tableau dynamique **n'est pas la bonne SDA** si l'on veut détruire un élément au milieu **et** garder l'ordre (utiliser plutôt des *listes chaînées* pour cela)

Note : si l'on ne tient pas à garder l'ordre, on peut toujours faire :

```
for (unsigned int i(0); i < v.size(); ++i)
        if (cond(v[i])) {
                v[i] = v[v.size()-1];
                v.pop_back();
                --i;
}
```

Tables associatives

Les **tables associatives** sont une généralisation des tableaux où les index ne sont pas forcément des entiers.

Imaginez par exemple un tableau que l'on pourrait indexer par des chaînes de caractères et écrire par exemple tab["Informatique"]

On parle d'« associations clé–valeur »

Les tables associatives sont définies dans la bibliothèque <map>.

Elles nécessitent deux types pour leur déclaration : le type des « clés » (les index) et le type des éléments indexé.

Par exemple, pour indexer des nombres réels par des chaînes de caractères on déclarera :

```
map<string,double> une_variable;
```

Si l'ordre (operator<) des clés n'importe pas, on utilisera une unordered_map.

Tables associatives – exemple

```
#include <map>
#include <string>
#include <iostream>
using namespace std;
int main()
{
map<string,double>
moyenne;
moyenne["Informatique"] =
5.5;
moyenne["Physique"] = 4.5;
moyenne["Histoire des
maths"] = 2.5;
moyenne["Analyse"] = 4.0;
moyenne["Algebre"] = 5.5;

// parcours de tous les elements
for (map<string,double>::iterator
i(moyenne.begin());
i != moyenne.end(); ++i)
cout << "En " << i->first << ", j'ai " << i-
>second
<< " de moyenne." << endl ;
// recherche
cout << "Ma moyenne en Informatique
est de ";
cout << moyenne.find("Informatique")-
>second << endl;
return 0;  }
```

Piles et files

Pour utiliser les piles de la STL : #include <stack>

Les **files d'attente** sont des piles où c'est le premier arrivé (empilé) qui est dépilé le premier. Elles sont définies dans la bibliothèque <queue>.

Une pile de type *type* se déclare par stack<*type*> et une file d'attente par queue<*type*>. par exemple :

stack<double> une_pile;

queue<char> attente;

méthodes :

Type top()	accède au premier élément (sans l'enlever)
void push(*Type*)	empile/ajoute
void pop()	dépile/supprime
bool empty()	teste si la pile/file est vide

Piles - exemple

```
#include <stack>
using namespace std;
...
bool check(string s) {
        stack<char> p;
        for (unsigned int i(0); i < s.size(); ++i) {
                if ((s[i] == '(') || (s[i] == '['))
                        p.push(s[i]);
                else if (s[i] == ')') {
                        if ((!p.empty()) && (p.top() == '('))
                                p.pop();
                        else
                                return false;
                } else if (s[i] == ']') {
                        if ((!p.empty()) && (p.top() == '['))
                                p.pop();
                        else
                                return false;
                        }
        }
        return p.empty();                       }
```

Conclusion

Il existe beaucoup d'outils prédéfinis dans la bibliothèque standard de C++

Le but n'est évidemment pas les connaître tous par cœur, mais de **savoir qu'ils existent** pour penser aller chercher dans la documentation les informations complémentaires.

Analyse et programmation orientée objet (C++)

Séance 11

C++11: Smart Pointers

le code précédent contient une inconsistance majeure...
Que devient le *handler* de fichier f en cas d'exception ?

```
void ouvreFichiers(char *nom) {
  FILE *f = fopen(nom,“r“);
  char n[100];
  int lineno=0;
  while ( fgets(n,100,f)!=0 ) {
    try {
      lineno++;
      ouvreFichier(n);
    } catch (ErreurFichier err) {
        err.setLine(lineno);
        throw;
    }
  }
  fclose(f);
}
```

bien sûr on peut prendre la précaution de fermer le fichier au bon moment...
Cela nécessite d'être très attentif.

```
void ouvreFichiers(char *nom) {
  FILE *f = fopen(nom,“r“);
  char n[100];
  int lineno=0;
  while ( fgets(n,100,f)!=0 ) {
    try {
      lineno++;
      ouvreFichier(n);
    } catch (ErreurFichier err) {
        err.setLine(lineno);
        fclose(f);
        throw;
    }
  }
  fclose(f);
}
```

- une solution plus sûre est connue sous le nom d'acquisition de ressources par initialisation (*RAII Resource Allocation Is Initialization*)

- cette technique utilise deux caractéristiques essentielles du langage :
  - toute variable locale est détruite à la sortie du bloc qui la déclare
  - toute destruction de variable fait appel au destructeur...

- on va donc créer une classe intermédiaire

```
void ouvreFichiers(char *nom) {
  FichierOuvert f(nom);
  char n[100];
  int lineno=0;
  while ( fgets(n,100,f)!=0 ) {
    try {
      lineno++;
      ouvreFichier(n);
    } catch (ErreurFichier err) {
        err.setLine(lineno);
        throw;
    }
  }
}
```

```
class FichierOuvert {
  private:
    FILE *file;
  public:
    FichierOuvert(char *nom) {
      file = fopen(nom,"r");
    }

  ~FichierOuvert() {
    if (file != -1)
      fclose(file);
    file=0;
}
  ...

};
```

Principe

Facilite la gestion de la mémoire en évitant principalement :

- Evite les fuites mémoires
- Evite de libérer prématurément des ressources
- Evite de libérer plusieurs fois une ressource (provoque des erreurs)

Solution :

- Utilisation des pointeurs intelligents (« *smart pointeur* »)
- Destruction automatique de la ressource pointée selon certains critères
- Gestion générale plus sûre

Principe

Objet encapsulant le pointeur brute (« *wrapper* »)

- Permet d'éviter la manipulation directe du pointeur
- Permet d'effectuer des contrôles supplémentaires (verification pointeur non nul ...)

Definir une politique d'acces

- Contrôles supplémentaires
- Casts possible / interdit

Désallocation du pointeur brute

- A la destruction du *wrapper* (et sous condition)

Consequences

- Duree de vie du pointeur intelligent liée au bloc !

Ingrédients

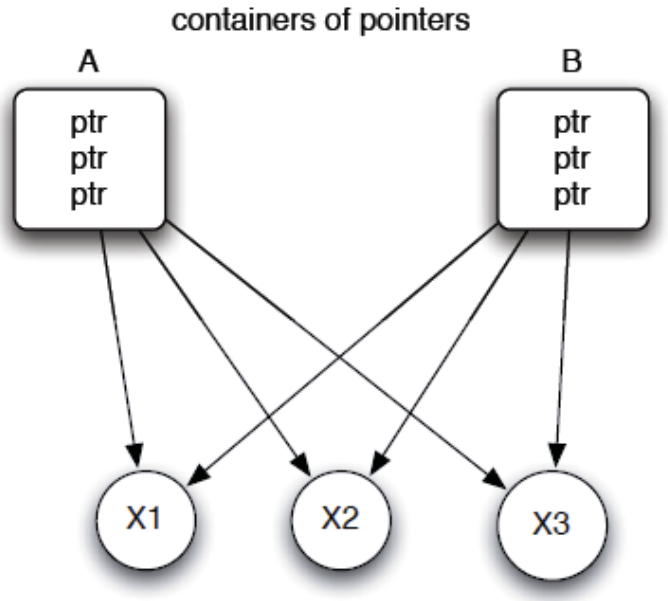
But : Manipulation comme un pointeur traditionnel : *transparence*

- Construction
  - Par défaut : pointeur NULL
  - Autre : pointeur brute
- Constructeur de copie : dépendant de la politique choisie
  - Transfert de propriété
  - Ajout de référence
  - ...
- Opérateur d'affectation : identique au constructeur de copie

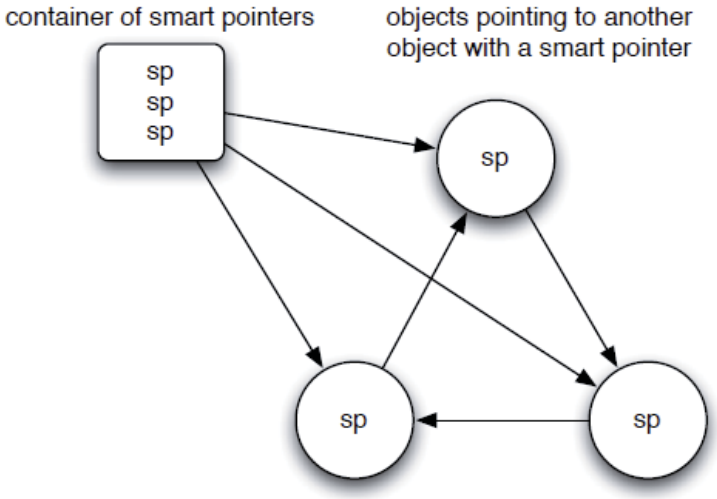
Shared pointeur

- Encapsulation d'un pointeur brute et d'un compteur de référence
- Possibilité de partager une ressource pointée entre différents pointeurs
- Chaque copie ajoute une référence (+ + compteur)
- Chaque destruction supprime une référence (- - compteur)
- Désallocation lorsque le nombre de références est 0

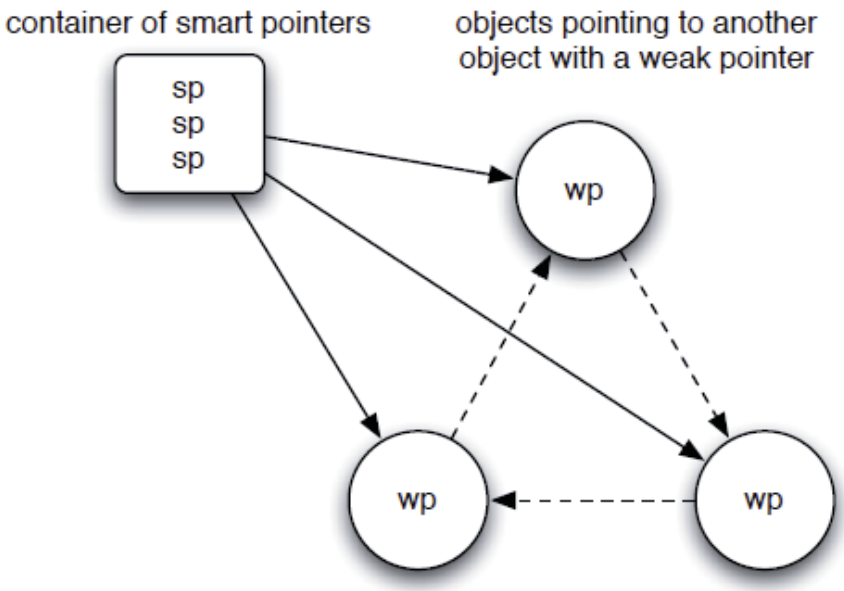
Shared pointeur



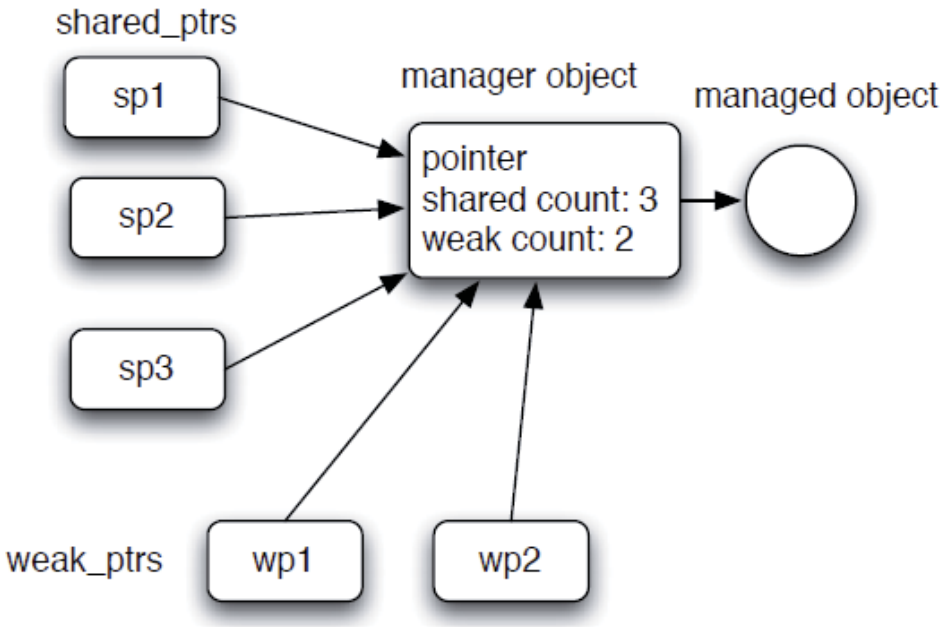
Shared pointeur



Shared pointeur & weak pointeur



Shared pointeur & weak pointeur : fonctionnement



Remarques

- Vous ne pouvez utiliser ces pointeurs intelligents que pour faire référence à des objets alloués avec *new* (suppression avec *delete*).
- Vous devez vous assurer qu'il n'y a qu'un seul objet gestionnaire pour chaque objet géré. Pour ce faire, l'écriture votre code de sorte que quand un objet est créé, il est immédiatement remis à un shared_ptr pour le gérer, et tout autre shared_ptr ou weak_ptr qui sont nécessaires pour pointer vers cet objet sont tous directement ou indirectement, copiés à partir du premier shared_ptr. La façon habituelle d'y parvenir est d'écrire la nouvelle expression de l'objet comme argument pour un constructeur de shared_ptr, ou utiliser le make_shared (voir plus loin).
- Si vous voulez obtenir le plein bénéfice des pointeurs intelligents, votre code devrait éviter d'utiliser des pointeurs brutes pour se référer aux mêmes objets

Shared pointeur : Utilisation

```
class Thing {
public:
    void defrangulate();
};
ostream& operator<< (ostream&, const Thing&);
...
// a function can return a shared_ptr
shared_ptr<Thing> find_some_thing();
// a function can take a shared_ptr parameter by value;
shared_ptr<Thing> do_something_with(shared_ptr<Thing> p);

...
void foo()
{
    // the new is in the shared_ptr constructor expression:
    shared_ptr<Thing> p1(new Thing);
    ...
    shared_ptr<Thing> p2 = p1; // p1 and p2 now share ownership of the Thing
    ...
    shared_ptr<Thing> p3(new Thing); // another Thing

    p1 = find_some_thing();    // p1 may no longer point to first Thing
    do_something_with(p2);
    p3->defrangulate();     // call a member function like built-in pointer
    cout << *p2 << endl;    // dereference like built-in pointer
    // reset with a member function or assignment to nullptr:
    p1.reset();  // decrement count, delete if last
    p2 = nullptr; // convert nullptr to an empty shared_ptr, and decrement count;
}
// p1, p2, p3 go out of scope, decrementing count, delete the Things if last
```

Shared pointeur : Utilisation

```
Thing * bad_idea()
{
    shared_ptr<Thing> sp; // an empty pointer
    Thing * raw_ptr = new Thing;
    sp = raw_ptr; // disallowed - compiler error !!!
    ...
    return raw_ptr;  // danger!!! - caller could make a mess with this!
}
```

```
shared_ptr<Thing> better_idea()
{
    shared_ptr<Thing> sp(new Thing);
    ...
    return sp;
}
```

Shared pointeur : Utilisation

```
Thing * another_bad_idea()
{
    shared_ptr<Thing> sp(new Thing);
    Thing * raw_ptr  = sp;  // disallowed! Compiler error!

    Thing * raw_ptr = sp.get();    // you must want it, but why?
    ...
    return raw_ptr;  // danger!!! - caller could make a mess with this!
}
```

Shared pointeur et l'héritage

```
class Base {};
class Derived : public Base {};
...
Derived * dp1 = new Derived;
Base * bp1 = dp1;
Base * bp2(dp1);
Base * bp3 = new Derived;
```

```
class Base {};
class Derived : public Base {};
...
shared_ptr<Derived> dp1(new Derived);
shared_ptr<Base> bp1 = dp1;
shared_ptr<Base> bp2(dp1);
shared_ptr<Base> bp3(new Derived);
```

Shared pointeur : Cast

```
shared_ptr<Base> base_ptr (new Base);
shared_ptr<Derived> derived_ptr;
// if static_cast<Derived *>(base_ptr.get()) is valid, then the following is valid:
derived_ptr = static_pointer_cast<Derived>(base_ptr);
```

Shared pointeur : Meilleure performance

```
shared_ptr<Thing> p(new Thing); // ouch - two allocations

shared_ptr<Thing> p(make_shared<Thing>()); // only one allocation!

shared_ptr<Base> bp(make_shared<Derived1>());

shared_ptr<Thing> p (make_shared<Thing>(42, "I'm a Thing!"));
```

weak pointeur : Utilisation

```
shared_ptr<Thing> sp(new Thing);

weak_ptr<Thing> wp1(sp);   // construct wp1 from a shared_ptr
weak_ptr<Thing> wp2;       // an empty weak_ptr - points to nothing
wp2 = sp;                  // wp2 now points to the new Thing
weak_ptr<Thing> wp3 (wp2); // construct wp3 from a weak_ptr
weak_ptr<Thing> wp4
wp4 = wp2;                 // wp4 now points to the new Thing.
```

```
shared_ptr<Thing> sp2 = wp2.lock();  // get shared_ptr from weak_ptr
```

weak pointeur : Utilisation

```
void do_it(weak_ptr<Thing> wp){
    shared_ptr<Thing> sp = wp.lock(); // get shared_ptr from weak_ptr
    if(sp)
        sp->defrangulate(); // tell the Thing to do something
    else
        cout << "The Thing is gone!" << endl;
}
```

weak pointeur : Utilisation

```
bool is_it_there(weak_ptr<Thing> wp) {
    if(wp.expired()) {
        cout << "The Thing is gone!" << endl;
        return false;
        }
    return true;
}
```

weak pointeur : Utilisation

```
void do_it(weak_ptr<Thing> wp){
    shared_ptr<Thing> sp(wp);  // construct shared_ptr from weak_ptr
    // exception thrown if wp is expired, so if here, sp is good to go
    sp->defrangulate(); // tell the Thing to do something
}
...
try {
    do_it(wpx);
}
catch(bad_weak_ptr&)
{
    cout << "A Thing (or something else) has disappeared!" << endl;
}
```

Cas particulier: Obtenir un shared_ptr pour l'objet "this"

```
class Thing {
public:
    void foo();
    void defrangulate();
};

void transmogrify(Thing *);
int main()
{
    Thing * t1 = new Thing;
    t1->foo();
    ...
    delete t1;    // done with the object
}
...
```

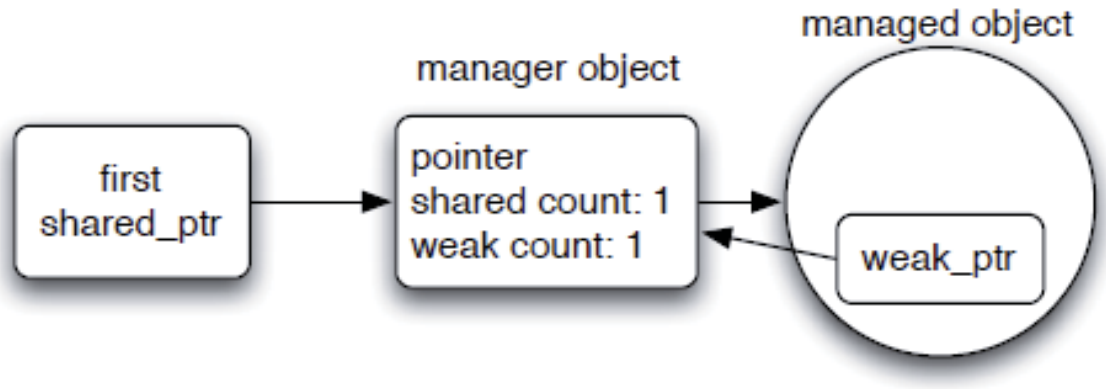
```
void Thing::foo()
{
    // we need to transmogrify this object
    transmogrify(this);
}
...
void transmogrify(Thing * ptr)
{
    ptr->defrangulate();
    /* etc. */
}
```

Cas particulier: Obtenir un shared_ptr pour l'objet "this"

```
class Thing {
public:
    void foo();
    void defrangulate();
};
void transmogrify(shared_ptr<Thing>);
int main()
{
    shared_ptr<Thing> t1(new Thing); // start a manager object for the Thing
    t1->foo();
    ...
    // Thing is supposed to get deleted when t1 goes out of scope
}
...
void Thing::foo()
{
    // we need to transmogrify this object
    shared_ptr<Thing> sp_for_this(this); // danger! a second manager object!
    transmogrify(sp_for_this);
}

...
void transmogrify(shared_ptr<Thing> ptr)
{
    ptr->defrangulate();
    /* etc. */}
```

Cas particulier: Obtenir un shared_ptr pour l'objet "this"



Cas particulier: Obtenir un shared_ptr pour l'objet "this"

```
class Thing : public enable_shared_from_this<Thing> {
public:
    void foo();
    void defrangulate();
};

int main()
{
    // The following starts a manager object for the Thing and also
    // initializes the weak_ptr member that is now part of the Thing.
    shared_ptr<Thing> t1(new Thing);
    t1->foo();
    ...
}
...
void Thing::foo()
{
    // we need to transmogrify this object
    // get a shared_ptr from the weak_ptr in this object
    shared_ptr<Thing> sp_this = shared_from_this();
    transmogrify(sp_this);
}
...

void transmogrify(shared_ptr<Thing> ptr)
{
    ptr->defrangulate();
    /* etc. */
}
```