

ISEN

ALL IS DIGITAL!

BREST



yncréa

Le langage C++

Didier Le Foll

Édition septembre 2024

Remerciements

Ce cours de C++ est la mise en forme écrite des cours que j'ai donnés verbalement aux étudiants de CIR2 de l'ISEN Brest à partir de septembre 2016. Je remercie les étudiants pour leur participation active, et leur retour qui m'a aidé à améliorer le cours.

Je remercie en particulier Estelle Roué pour sa prise de cours sous forme de fichiers *odt*. Ce sont ces fichiers qu'elle m'a communiqués gracieusement qui ont servi de base au présent cours.

Je suis aussi redevable à Michael Soullignac de l'ISEN Nantes, je lui ai emprunté certaines de ses idées et certains schémas des fichiers *ppt* des cours de C++ qu'il donne aux CIR2 de Nantes.

Table des matières

Ch 1 : Généralités.....	6
Ch 2 : La partie C du C++.....	7
A) Améliorations.....	7
1 - Utilisation de constantes.....	7
2 - Notion de « surcharge » d'une fonction (overloading).....	7
B) Ajouts.....	8
1- Valeurs par défaut des paramètres des fonctions.....	8
2- Nouveaux opérateurs de gestion mémoire.....	9
3- Nouvelle façon d'initialiser une variable.....	10
4- Nouvelle constante d'initialisation pour pointeur.....	10
5- Type bool (booléen).....	11
6- Nouvelle façon de faire les entrées/sorties : iostream.....	11
7- Déclaration auto : inférence de type.....	13
8- « range for » (range based for).....	13
9- Notion de référence.....	14
10- Notion de namespace.....	16
11- Opérateurs de «cast» (conversion de type).....	17
Ch 3 : La couche objet du C++.....	18
1- Présentation.....	18
2- Notion d'objet.....	20
3- Notion d'encapsulation.....	20
Méthodes de l'interface d'accès aux attributs.....	21
4- Notion de base de constructeur.....	22
5- Notion de destructeur.....	22
6- Syntaxes alternatives.....	23
1) Constructeur.....	23
2) Définition des méthodes « in class » ou « in line ».....	23
3) Réduction du nombre de constructeurs à l'aide des paramètres par défaut.....	24
4) Initialisation des attributs directement à leur déclaration.....	24
7- Le constructeur de recopie (copy constructor).....	24
8- Le mot-clé this.....	25
9- Pointeur sur objet.....	26
10- Tableaux d'objets.....	26
1) Tableau de taille fixe.....	26
2) Tableau d'objets alloué dynamiquement.....	26
3) Tableau de pointeurs/objet de taille fixe.....	26
4) Tableau (alloué dynamiquement) de pointeurs/objet alloués dynamiquement.....	27
11- Utilisation du mot-clé const dans un entête de méthode.....	28
12- Notion d'attribut (variable) de classe.....	29
13- Notion de fonction (méthode) de classe.....	30

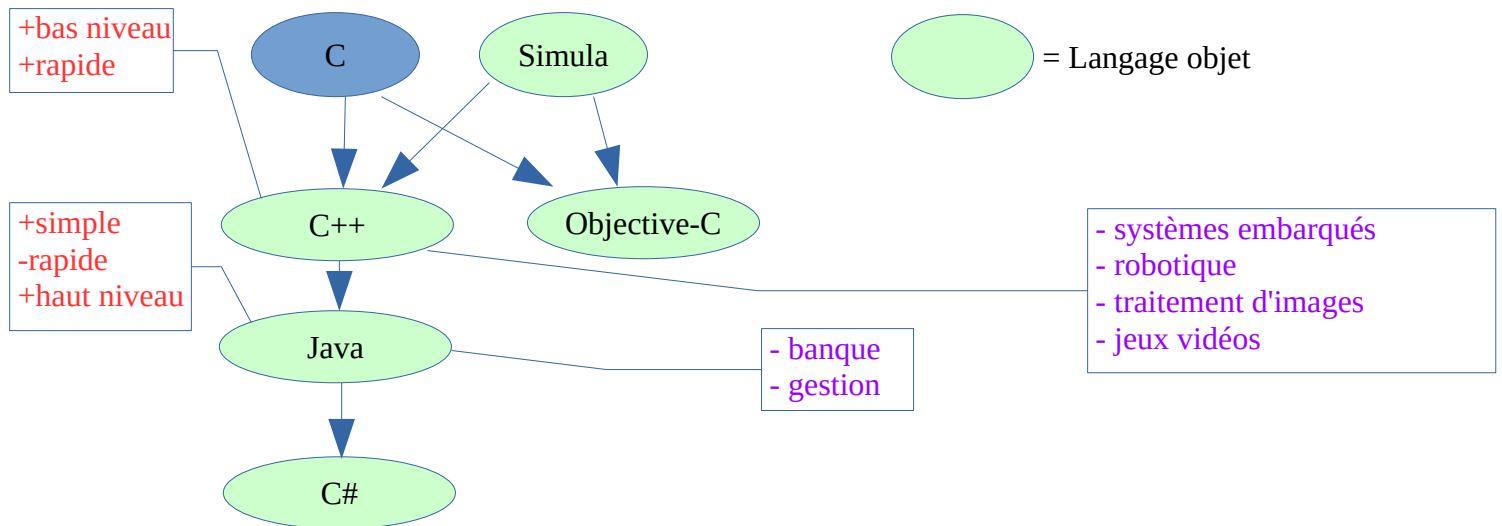
Ch 4 : Composition de classes.....	31
1- Concept de composition.....	31
2- Mise en œuvre.....	31
3- Diagramme UML de classes.....	32
Ch 5 : L'héritage.....	34
1- Généralités.....	34
2- Diagramme UML de classes.....	36
3- Statut des attributs et méthodes hérités.....	37
4- Redéfinition («overriding») de méthode dans une classe Dérivée.....	37
5- Règle d'appel des constructeurs.....	38
6- Règle d'appel des destructeurs.....	39
7- Représentation en mémoire.....	40
8- Le polymorphisme d'héritage (« subtype polymorphism »).....	41
a) Conversion entre objets de classe de Base et de classe Dérivée.....	41
b) Conversion entre pointeurs (ou références) sur objets de classe B et D.....	42
c) Les méthodes (fonctions membres) virtuelles => polymorphisme.....	44
9- Les classes abstraites (fonctions virtuelles pures).....	47
10- Notion de RTTI (Run-Time Type Information).....	48
Ch 6 : Les templates et la librairie standard.....	51
1- Les templates (programmation générique).....	51
a) Les templates de fonction.....	51
b) Les templates de classe.....	53
2- La librairie standard.....	53
a) Les strings.....	54
b) Les flux d'entrée/sortie (I/O streams).....	55
c) Les conteneurs (containers) et leurs « itérateurs » (iterators).....	57
Les conteneurs (containers).....	57
Les itérateurs (iterators).....	60
Les algorithmes de la STL (algorithms).....	62
d) Les « pointeurs intelligents » (smart pointers).....	64
Ch 7 : Gestion des exceptions.....	69
1- La gestion des exceptions.....	69
a) La gestion « traditionnelle » des anomalies.....	69
b) La gestion des exceptions - Généralités.....	70
c) Fonctionnement du « catch ».....	71
d) Exemple : traitement des erreurs liées à une base de données.....	72
2- Notion de RAII.....	74
Ch 8 (hors programme) : Surcharge d'opérateur, foncteurs, fonctions lambda.....	77
1- La surcharge d'opérateur (operator overloading).....	77
a) Généralités.....	77
b) Les opérateurs concernés.....	78
c) Opérateur << (stream insertion operator).....	79
d) Les fonctions amies (friend functions).....	80

2- Les objets-fonctions ou « foncteurs » (functors).....	81
3- Les fonctions lambda (lambda functions).....	82
a) Généralités.....	82
b) Les fermetures (closures).....	83
Références.....	86
Les livres de référence.....	86
Les sites de référence.....	86

Ch 1 : Généralités

C++ : inventeur Bjarne Stroustrup (~1980)

Simula : 1er langage objet (1962)



1980 : C with Classes

1985 : C++ (1ère version)

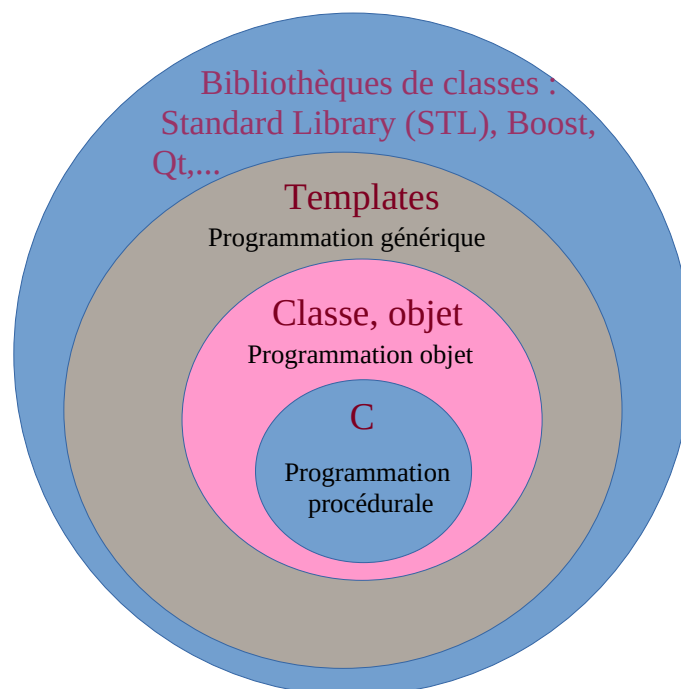
1998 : norme C++98 : C++ « traditionnel »

2011 : norme C++11 : changement majeur (« *modern C++* »)

2014 : norme C++14

2017 : norme C++17

2020 : norme C++20



Ch 2 : La partie C du C++

Plan du chapitre:

A) Améliorations

- 1- Utilisation de constantes
- 2- Notion de « surcharge » d'une fonction (*overloading*)

B) Ajouts

- 1- Valeurs par défaut des paramètres des fonctions
- 2- Nouveaux opérateurs de gestion mémoire
- 3- Nouvelle façon d'initialiser une variable
- 4- Nouvelle constante d'initialisation pour pointeur
- 5- Type *bool* (booléen)
- 6- Nouvelle façon de faire les entrées/sorties : *iostream*
- 7 Déclaration auto : inférence de type
- 8- « *range for* » (*range based for*)
- 9- Notion de référence
- 10- Notion de *namespace*
- 11- Opérateurs de «*cast*» (conversion de type)

A) Améliorations

1 - Utilisation de constantes

En C :

```
#define NB_VALEUR 100
```

En C++ :

```
const int kNbValeur=100; // éviter les majuscules. k : pratique possible (Google C++ style guide)  
// ou bien : const int kNbValeur{100}; // cf. plus loin §B.3
```

Avantage : vérif. de type (ici `int`) par le compilateur à chaque utilisation de la constante

2 - Notion de « surcharge » d'une fonction (*overloading*)

En C :

```
int f(int); //Prototype  
int f(double); //erreur de compil.
```

En C++ :

```
int f(int);  
int f(double); // OK : surcharge de f ( => « Polymorphisme ad-hoc »)
```

En C++, chaque fonction a une « signature » prenant en compte le nom de la fonction et le type des arguments.

=> la signature de `f(int)` et de `f(double)` est différente.



Par contre le type de retour ne fait pas partie de la signature

```
int f(int);  
double f(int); // erreur car, malgré le type de retour différent, pour C++ c'est la même  
// fonction.
```

Note : en arrière-plan, le compilateur (plus spécifiquement le *linker*) crée des noms différents pour deux fonctions de même nom, en construisant un nouveau nom par ajout du type des paramètres. Cette technique s'appelle « ***name mangling*** ». Les règles de ce nommage ne font pas partie de la norme du C++ et les différents compilateurs (GNU, Microsoft,...) ont des règles différentes.

Dans l'exemple ci-dessus (2 fonctions `f(int)` et `f(double)`), les fonctions réellement créées par le compilateur g++ s'appellent `_Z1fi` et `_Z1fd`. Le compilateur GNU (g++) utilise la règle suivante : il rajoute le préfixe `_Z` suivi du nombre de caractères du nom de la fonction (ici 1 pour `f`) suivi de la liste des types des arguments (ex: `i` pour `int`, `d` pour `double`,...).

B) Ajouts

1- Valeurs par défaut des paramètres des fonctions

```
int f(int param = -1){ //ici -1 = valeur du paramètre par défaut  
    printf("%d\n", param);  
}  
int main(){  
    f(12); //affiche 12  
    f(); //affiche -1 (lorsque pas de param => val. par défaut)  
}
```

Cas + compliqué : plusieurs paramètres. Ex :

```
int f(int a = 0, char val = 'x'){ //deux paramètres : un caractère et un entier  
}  
int main(){  
    f(); //OK  
    f(12, 'a'); //OK  
    f(45); //OK parce que c'est le premier paramètre
```



```
f('z'); //NON parce que ce n'est pas le premier
}
```



Il est interdit de définir une valeur par défaut pour un paramètre sans en définir pour le suivant :

```
int f(int a = 0, char val);
int f(int a, char val = 'x'); //OK
```

2- Nouveaux opérateurs de gestion mémoire

	C	C++ (un élément)	C++ (plusieurs éléments)
allocation	malloc()	new	new []
libération	free()	delete	delete[]

En C :

```
char *chaine;
chaine = malloc(10*sizeof(char)); //allocation
strcpy(chaine, "bonjour");
free(chaine); //libération
```

En C++ :

```
char *chaine;
chaine = new char[10]; //allocation
strcpy(chaine, "bonjour");
delete[] chaine; //libération
```

En C :

```
typedef struct{
    int x,y;
} Point;
Point *p = malloc(sizeof(Point));
free(p);
```

En C++ :

```
struct Point{ //typedef Point implicite en C++ (différence avec le C)
    int x,y;
};
```

```
Point *p = new Point;

delete p;
```

3- Nouvelle façon d'initialiser une variable

Le C++11 a introduit une nouvelle syntaxe générique pour initialiser une variable de type quelconque. Elle utilise des accolades (« *brace initialization* ») contenant la(les) valeur(s) initiale(s) ou accolades vides pour initialisation à une valeur par défaut (« *value initialization* »).

En C :

```
int a = 1; //initialisation de l'entier à 1
int b = 0; //initialisation de l'entier à 0
int tab[3] = {1,2,3}; //initialisation des 3 éléments du tableau d'entiers
int autre_tab[3] = {0}; //initialisation à 0 des 3 éléments du tableau d'entiers
char chaine[3] = ""; //initialisation d'une chaîne «vide» (= '\0' au moins en 1er caractère)
int *ptr = NULL; //initialisation du pointeur à NULL
```

En C++ :

```
int a {1}; //initialisation de l'entier à 1
int b {}; //initialisation de l'entier à 0
int tab[3] {1,2,3}; //initialisation des 3 éléments du tableau d'entiers
int autre_tab[3] {}; //initialisation à 0 des 3 éléments du tableau d'entiers
char chaine[3] {}; //initialisation d'une chaîne «vide» ( ici '\0' dans tous les caractères)
int *ptr {}; //initialisation du pointeur à NULL
```

Remarque : la valeur par défaut (accolades vides) est : 0 pour les types numériques, NULL pour les pointeurs, tous les éléments à 0 pour les tableaux, les membres à 0 pour les struct.

Avantage : l'initialisation par accolades est valide pour tous types de variables et apporte donc de la clarté. Elle est conseillée par les guides de bonnes pratiques.

Inconvénient : ce mode d'initialisation n'existe pas en C et les programmeurs habitués au C ne l'utilisent pas souvent en C++. Le C++ autorise par compatibilité tous les modes d'initialisation (=, ={}, {},...) mais le mélange de ces modes dans un même programme doit être évité.



La compilation doit se faire en C++11 ou norme ultérieure (cf. TP1).

4- Nouvelle constante d'initialisation pour pointeur

La constante `nullptr` (seulement depuis le C++11) remplace avantageusement la constante `NULL` du C.

Avantage : `nullptr` ne peut être utilisé que pour des pointeurs, alors que `NULL` (qui est un *define* valant 0) peut être converti implicitement en entier, ce qui peut aboutir à des erreurs.

En C :

```
char *chaine = NULL; //initialisation du pointeur
```

En C++ :

```
char *chaine = nullptr;
```

```
// ou (cf. §précédent) : char *chaine{nullptr}; // ou : char *chaine{};
```



La compilation doit se faire en C++11 ou norme ultérieure (cf. TP1 ci-dessous).

5- Type *bool* (booléen)

Une variable de type `bool` peut valoir soit `true` soit `false` (constantes prédéfinies)

```
bool a { true }; //ou bool a = true; // dans un contexte « entier » => 1
a = false; // => 0
```

TP 1 :

- Créer un tableau d'entiers de taille fixe (taille = constante)
- Faire une fonction `affiche()` qui affiche une valeur entière passée en paramètre
- Faire une seconde fonction `affiche()` (c'est-à-dire de même nom que la première) qui affiche toutes les valeurs du tableau qu'on lui passe en paramètre.
- Appeler successivement ces deux fonctions dans le `main()`.
- Faire une fonction `mult()` à 2 paramètres (2 entiers), qui renvoie `param1*param2`. S'il n'y a qu'un paramètre passé, ne renvoie que celui-la.

Compilation en C++ :

Compilateur GNU : `g++`

Nom des fichiers sources : extension `.cpp` pour le code et `.h` (ou `.hpp`) pour les *headers*. Ex : `truc.cpp` `truc.h`

Commande de compilation simple (si un seul source `.cpp`), ex: `g++ truc.cpp -o truc.exe`

Le compilateur va utiliser par défaut un standard (C++98, C++11, C++17, etc.) différent selon la version de `g++`. Pour connaître ce standard par défaut, vous pouvez taper la commande :

```
man g++ | grep "default for C++ code"
```

Si vous utilisez dans votre code des éléments du langage postérieurs à ce standard par défaut (par exemple en C++20 alors que votre compilateur est en C++17), un *flag* de compilation sera nécessaire. Ex : `g++ -std=c++20 truc.cpp`

6- Nouvelle façon de faire les entrées/sorties : *iostream*

iostream signifie *input/output stream* = flux d'entrée/sortie

iostream est une classe de la librairie standard du C++ (voir chapitre 6)

Caractéristiques principales :

1 flux d'entrée standard → clavier (en C : `stdin`) en C++ : `cin`

flux de sortie standard → écran (en C : `stdout`) en C++ : `cout`

2 deux nouveaux opérateurs :

>> : *get* (entrée) ou « *stream extraction operator* »

<< : *put* (sortie) ou « *stream insertion operator* »

	C	C++
Inclusion de <i>headers</i>	<code>#include <stdio.h></code>	<code>#include <iostream></code> <code>using namespace std;</code>
Entrée standard (clavier)	<code>stdin</code>	<code>cin</code>
Sortie standard (écran)	<code>stdout</code>	<code>cout</code>
Erreur standard (écran)	<code>stderr</code>	<code>cerr</code>
Fonctions ou opérateur de saisie (<i>get</i>)	<code>scanf()</code> , <code>gets()</code> , <code>fgets()</code>	>> (<i>get operator</i>)
Fonctions ou opérateur d'affichage (<i>put</i>)	<code>printf()</code> , <code>puts()</code> , <code>fprintf()</code>	<< (<i>put operator</i>)

Exemple en SORTIE :

En C :

```
#include <stdio.h>
int i=12;
printf("%d\n",i);
```

En C++ :

```
#include <iostream>
using namespace std;// facultatif, cf. plus loin (§10) les namespaces. Sinon, il faut
// préfixer cout (et endl) par std::
int i{12};// ou: int i=12;
cout << i << "\n";
cout << "La valeur de la variable est " << i << "\n"
cout << i << endl; // → endl équivaut à "\n" mais force le vidage du buffer de sortie
```

Exemple en ENTREE (*includes* comme ci-dessus):

En C :

```
int i;
scanf("%d", &i);
```

En C++ :

```
int i;
cin >> i;
```

7- Déclaration auto : inférence de type

Inférence = action de tirer une conclusion à partir d'un fait (déf. du dictionnaire)

En C : type obligatoire à la déclaration

```
int a {12}; // ou int a = 12;
```

En C++ : il est possible de laisser le compilateur déterminer le type à la déclaration (selon la constante d'initialisation).

```
auto a {12}; //auto laisse le compilateur déterminer le type de la variable a
```

NB: cette possibilité a surtout un intérêt pour les types des variables de boucle ou de retour de fonctions.

8- « *range for* » (*range based for*)

Existe depuis la norme C++11

CLASSIQUE

```
float tab[10];  
for ( int i=0; i<10; ++i ){  
    cout << tab[i] << endl;  
}
```

RANGE FOR

```
float tab[10];  
for ( float element : tab ){ //ressemble au foreach du PHP  
    cout << element << endl;  
}  
  
//Variante avec inférence de type pour la variable element (connu par le type du tableau)  
for ( auto element : tab ){  
    cout << element << endl;  
}
```

`element` : variable stockant (par copie) chaque élément successif du tableau `tab` à chaque tour de boucle. La variable `element` est copiée du tableau : si elle est modifiée, la valeur du tableau ne l'est pas.

Avantages : 1) l'indice de boucle n'est plus nécessaire

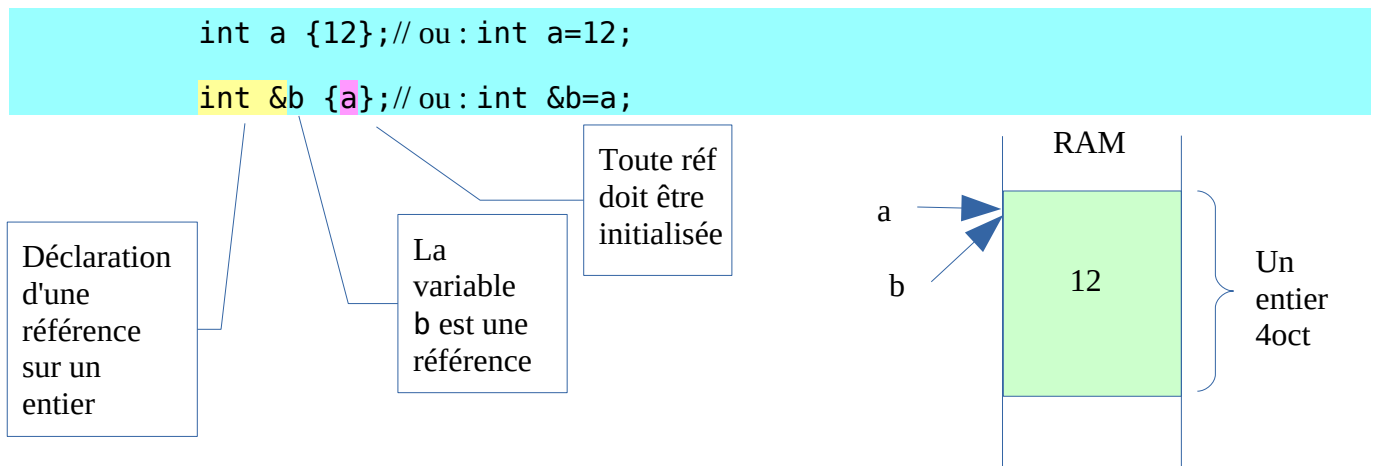
2) fixer le nombre de tours de boucles (ici 10) n'est plus nécessaire (le compilateur connaît la taille de `tab`)

3) *fonctionne avec tous les ensembles séquentiels de la lib standard : vecteur, liste, tableau associatif,... (cf. ch. 6)*

Inconvénient : ne fonctionne pas avec les tableaux alloués dynamiquement (par pointeur), seulement avec les tableaux de taille fixe.

9- Notion de référence

Exemple de démonstration (sans intérêt pratique).



Une référence : un autre nom pour une même variable. Si la valeur stockée par la référence est modifiée, la variable initiale sera aussi modifiée.

Ex (suite du code précédent) :

```
b = 45;
cout << a; //affiche 45
```

Intérêt des références pour le passage de paramètres à une fonction :

Il y a 3 façons de passer des paramètres à une fonction :

1-

```
void increment(int a){ //passage par valeur
    a++;
}
int main(){
    int i=0;
    increment(i);
    cout << i; // → 0
}
```

2-

```
void increment(int *a){ //passage par adresse
    (*a)++;
}
```

```
int main(){
    int i=0;
    increment(&i);
    cout << i; // → 1
}
```

3-

```
void increment(int &a){ //passage par référence
    a++;
}
int main(){
    int i=0;
    increment(i); // a (paramètre formel) devient une référence sur i (paramètre effectif)
    cout << i; // → 1
}
```

Intérêt du passage par référence

- écriture fonction plus simple dans le cas où on veut un paramètre d'E/S (différence entre versions 2 et 3)
- gain de temps à l'exécution (pas de recopie du paramètre effectif vers le paramètre formel, car c'est en fait la même variable en mémoire)

Inconvénient

- un seul petit `&` (différence entre versions 1 et 3) change totalement le comportement du programme (`i` vaudra 0 ou 1 après appel de la fonction).

TP 2 :

Modifier exo1

1- au lieu d'un tableau de taille fixe

→ Allocation dynamique : `int *tab ;`

+ libération à la fin

2- Remplacer les `printf` par l'opérateur `<<`

TP 3 : (exo3.cpp)

PARTIE 1

- une fonction `affiche()` qui affiche un `Point` (struct `Point`, cf. §B.2)
- une fonction `modif_point1()` qui prend en paramètre un `Point` et qui modifie la valeur (x et/ou y) du point
- une fonction `modif_point2()` qui prend en paramètre un `Point *` et - - - - -
- une fonction `modif_point3()` qui prend en paramètre un `Point &` et- - - - -
- dans le `main()` :
 - créer un `Point`
 - appeler `modif_point1()`, `2()`, `3()` successivement pour modifier le `Point`
 - afficher le `Point` après chaque `modif` (fonction `affiche()`)

PARTIE 2

- dans le `main()` :
 - * créer un tableau de `Point` de taille fixe
 - * afficher tous les points du tableau avec un "range for" (avec et sans inférence de type)
 - * puis, à l'intérieur de cette boucle for, appeler `modif_point3()` pour modifier la valeur de chaque point du tableau
 - + ajouter une nouvelle boucle for pour réafficher le tableau et vérifier que les points ont bien été modifiés.

10- Notion de *namespace*

Méthode pour isoler un groupe de fonctions et de définitions (struct, ..).

Cette notion est proche de la notion de module dans d'autres langages (ex: PHP). Cependant le C++20 a introduit le mot-clé « *module* » qui est une notion différente, plutôt à rapprocher de celle de « *header* ».

Syntaxe

```
namespace truc{
    //définitions
    struct Point{int x, y};
    ...
    //fonctions
    void affiche (Point p){..}
    ..
}
```

On encadre les définitions

Utilisation

3 façons de faire :

```
#include "truc.h" //définition du namespace truc
```



```
1- using namespace truc;

int main(){
    Point p{3,5};
    affiche(p);
}
```

Avantage : simple (un seul **truc**)
Inconvénient : par ex si `affiche()` est défini dans plusieurs *namespaces*

```
2- using truc::Point;
using truc::affiche;
int main(){
    //comme1
}
```

Avantage : on précise d'où vient chaque définition utilisée dans le programme
Inconvénients : fastidieux, pb de plusieurs `affiche()` n'est pas réglé

```
3- int main(){
    truc::Point p{3,5};
    truc::affiche(p);
}
```

Inconvénient : très fastidieux (beaucoup de `truc::` à taper...)
Avantage : très précis

11- Opérateurs de «cast» (conversion de type)

- Rappel du problème (ex: pb de la division de deux entiers dont on veut un résultat flottant => *cast* explicite nécessaire du numérateur et/ou du dénominateur) :

```
int a{3}, b{2};
float x = a/b;

std::cout<<x; //affiche 1.0

// ci-dessous, cast explicite du numérateur, selon la seule syntaxe possible en C (également
//possible en C++)

float y = (float)a/b; //1.5
```

- **Nouveautés pour le cast en C++ :**

1) Nouvelle syntaxe possible (cast «fonctionnel»)

```
float y = float(a)/b; //cast explicite, qui ressemble à un appel de fonction
```

2) Nouveaux opérateurs (*static_cast*, *dynamic_cast*, *const_cast*, *reinterpret_cast*)

Dans la plupart des cas, le *cast* explicite habituel du C est un *static_cast*.

```
float y = static_cast<float>(a)/b; //préconisé par les guides de bonnes
// pratiques, préférentiellement aux 2 syntaxes précédentes
```

Intérêt : facilement repérer les *cast* dans un programme

Inconvénient : fastidieux

Cf. Ch. 5 §8.b pour l'utilisation de l'opérateur *dynamic_cast*.

Ch 3 : La couche objet du C++

Plan du chapitre:

1- Présentation

2- Notion d'objet

3- Notion d'encapsulation

Méthodes de l'interface d'accès aux attributs

4- Notion de base de constructeur

5- Notion de destructeur

6- Syntaxes alternatives

- 1) Constructeur
- 2) Définition des méthodes « *in class* » ou « *in line* »
- 3) Réduction du nombre de constructeurs à l'aide des paramètres par défaut

7- Le constructeur de copie (copy constructor)

8- Le mot-clé *this*

9- Pointeur sur objet

10- Tableaux d'objets

- 1) Tableau de taille fixe
- 2) Tableau d'objets alloué dynamiquement
- 3) Tableau de pointeurs/objets de taille fixe
- 4) Tableau (alloué dynamiquement) de pointeurs/objet alloués dynamiquement

11- Utilisation du mot-clé *const* dans un entête de méthode

12- Notion d'attribut (variable) de classe

13- Notion de fonction (méthode) de classe

1- Présentation

En C :

Note : les `struct` existent aussi en C++

```
struct Point{  
    int x;  
    int y;  
};
```

Regroupement de variables liées entre elles (données membres)

En C++ :

On étend cette notion de struct → classe (**class**)

Pour obtenir la notion de classe, on ajoute à la notion de struct :

- des fonctions → **Méthodes**
- séparation public/privé (**public/private**) (quand les membres sont privés → **Encapsulation**)
- notion de «constructeur» (**constructor**) : méthode particulière qui initialise un objet
- notion de «destructeur» (**destructor**) : méthode particulière qui libère l'objet

```
class Point {  
    private :  
        int x; //données membres  
        int y;  
    public :  
        void affiche(); //fonction membre "banale"  
        // Constructeurs  
        Point(); //constructeur sans argument (par défaut)  
        Point(int x,int y); //constructeur à 2 arguments  
        //Destructeur  
        ~Point();  
};
```

Prototype des
fonctions

Fonctions
membres
« spéciales »

- 1- pas de type de retour !
- 2- même nom que la classe

Termes C	Termes C++	Termes «objet» généraux
Membre ou champ (member , field)	Donnée membre (data member)	Attribut ou variable d'instance (attribute , instance variable)
Fonction (function)	Fonction membre (member function)	Méthode (method)
N'existe pas	Fonction constructeur (constructor function)	Constructeur (constructor)
N'existe pas	Fonction destructeur (destructor function)	Destructeur (destructor)
Variable d'un type struct	Objet (object)	Objet ou instance (object , instance)
Déclaration = création d'une variable	Déclaration d'un objet (object declaration) Création - - - - creation	Instanciation (instantiation)
Variable statique	Donnée (ou variable) membre statique (static data member , static member variable)	Attribut (ou variable) de classe (class attribute , class variable)
N'existe pas	Fonction membre statique (static member function)	Méthode de classe (class method)

2- Notion d'objet

Parallèle avec le C :

```
struct Point{int x,y};  
typedef struct Point Point;  
//déclaration d'une variable de type struct Point  
Point le_point;  
le_point.x=12;  
le_point.y=5;
```

En C++ :

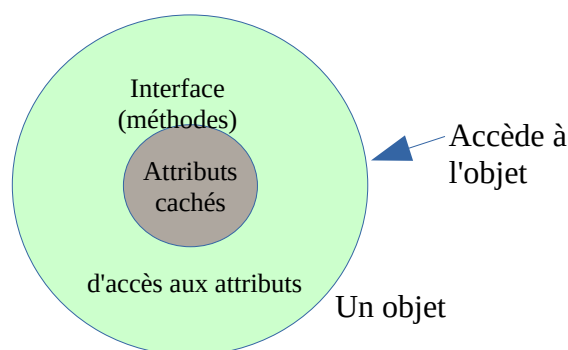
```
//déclaration de la classe Point  
(cf §1 ci-dessus)  
...  
//déclaration d'une variable de classe Point  
Point le_point; //instanciation (du verbe instancier)  
le_point.x=12;  
le_point.y=5; } Interdit car x et y  
                  sont privés  
//Il faut des méthodes d'accès en lecture/modif
```

Résumé :

Un objet est une variable de type classe qu'on instancie (= déclare).

3- Notion d'encapsulation

On considère qu'une "bonne pratique" en développement objet est que les attributs soient privés.



Les méthodes d'accès (en lecture ou écriture) aux attributs (=interface) permettent de mettre un "filtre".

Méthodes de l'interface d'accès aux attributs

→ accesseurs (**getters**)

ex :

```
int getX();
```

```
int getY();
```

→ mutateurs (**setters**)

```
void setX(int x);
```

```
void setY(int y);
```

Ex. de **définition** complète d'une classe (à mettre dans **Point.h**, le corps des fonctions sera dans **Point.cpp**):

```
class Point {
    public :

        // Constructeurs
        Point();
        Point(int x,int y);

        // Destructeur
        ~Point();

        // Accesseurs/mutateurs
        int getX();
        int getY();
        void setX(int x);
        void setY(int y);

        // Autres fonctions
        void affiche();

    private :

        // Attributs
        int x;
        int y;

};
```

Instanciation d'un objet de cette classe :

```
Point le_point; //instanciation d'un objet
```

```
le_point.setX(12);
le_point.setY(5);
```

} Modification de l'objet par les mutateurs

4- Notion de base de constructeur

Il n'est pas obligatoire d'avoir un constructeur dans une classe. Il peut y en avoir plusieurs (surcharge cf. Ch 2 §A.2), avec des paramètres différents (en nombre et/ou en type). Ils portent obligatoirement le même nom que la classe (ex. pour la classe `Point`, la(les) fonction(s) constructeur(s) se nomme(nt) `Point`).

Le constructeur sert essentiellement à initialiser les données membres (=attributs), et aussi éventuellement à acquérir des ressources système (par ex. allocation mémoire, ouverture de fichier,...).

Écriture du **code des constructeurs**, ex. de la classe `Point` (dans fichier `Point.cpp`). On choisit d'initialiser `x` et `y` à 0 par défaut :

//Constructeur par défaut :

```
Point::Point(){  
    x=0;// ou x{0};  
    y=0;// ou y{0};  
}
```

Attributs
privés

Définition de la fonction
`Point()` = **constructeur**
sans argument

//Constructeur à 2 arguments

```
Point::Point(int x_init, int y_init){  
    x = x_init;// ou x{x_init};  
    y = y_init;// ou y{y_init};  
}
```

Utilisation des constructeurs (par ex. dans fichier `main.cpp`)

```
Point le_point; //appel implicite du constructeur par défaut (ou bien : Point le_point{};)  
cout << "x=" << le_point.getX() << ",y=" << le_point.getY() << endl; //affiche  
x=0,y=0
```


```
Point autre_point {12,17}; // appel implicite du constructeur à 2 arguments  
// autre syntaxe avec des parenthèses : Point autre_point (12,17);  
cout << "x=" << autre_point.getX() << ",y=" << autre_point.getY() << endl;  
//affiche x=12,y=17
```

TP : reprise du cours en 3 fichiers

`Point.h`, `Point.cpp`, `main.cpp`

5- Notion de destructeur

Il n'est pas obligatoire d'avoir un destructeur dans une classe. Si vous décidez de l'écrire, il ne peut y en avoir qu'un seul, sans paramètre. Il porte obligatoirement le même nom que la classe précédé par un « tilde » (ex. pour la classe `Point`, la fonction destructeur se nomme `~Point`).

Le destructeur sert essentiellement à libérer les ressources système qui avaient été acquises par le constructeur (par ex. libération mémoire, fermeture de fichier,...). Il est appelé automatiquement lorsque l'objet sort de la portée (bloc ou fonction), mais  pas quand l'objet a été alloué par new.

Écriture du **code du destructeur**, ex. de la classe Point. Ici le destructeur ne fait rien, sinon laisser un message à l'écran (permet de voir quand il est appelé) :

```
//Destructeur :  
Point::~~Point(){  
    cout << "Appel du destructeur\n";  
}
```

} Définition de la fonction
~Point()=destructeur

6- Syntaxes alternatives

1) Constructeur

- Syntaxe « classique »

```
Point::Point(int xinit, int yinit){  
    x=xinit;  
    y=yinit;  
}
```

} Initialisation dans le
corps de la fonction

- Syntaxe « par liste d'initialisation » : préconisée par les guides de bonnes pratiques

```
Point::Point(int xinit, int yinit) : x{xinit}, y{yinit} {}
```

Je mets xinit dans x, et yinit dans y.

.....
Liste d'initialisation

Corps vide

```
Point::Point():x{0},y{0}{} // Cteur sans arg avec liste d'initialisation
```

Remarque importante : avant le C++11, la syntaxe utilisait des **parenthèses**. Cette forme est encore souvent utilisée, mais celle avec les accolades est préconisée par les guides de bonnes pratiques.

```
Point::Point(int xinit, int yinit) : x(xinit), y(yinit) {} // en C++98
```

```
Point::Point():x(0),y(0){} // en C++98
```

2) Définition des méthodes « in class » ou « in line »

Il est fréquent de définir les méthodes dont le code est très court directement dans la définition de la classe (dans le .h), au lieu de le faire dans le .cpp. On appelle cela définition « in line » ou « in class ». On peut mixer : certaines fonctions « in line », d'autres en dehors du bloc de classe (cf. ci-dessous).

```
class Point{  
    public :  
        Point():x{0}, y{0}{}  
        Point(int xinit, int yinit) : x{xinit}, y{yinit}{}  
        int getX(){return x;}  
        int getY(){return y;}  
        void setX(int x);  
}
```

} in line

```

        void setY(int y);
        void affiche();
    private :
        int x,y;
};

```

3) Réduction du nombre de constructeurs à l'aide des paramètres par défaut

Pour la notion de paramètres par défaut, cf. Ch 2 §B.1.

```

Point(int xinit=0, int yinit=0):x{xinit}, y{yinit}{} //Joue le rôle des 2
// constructeurs : sans arg et à 2 args

```

Remarque : les **valeurs par défaut** ne peuvent être définies qu'**en un seul endroit**, soit dans le prototype (dans le bloc de déclaration de la classe dans le *.h*), soit dans le *.cpp* (si la définition de la fonction n'est pas « *in line* »). Il est préconisé des les mettre **plutôt dans le *.h***.

4) Initialisation des attributs directement à leur déclaration

Il est possible, depuis le C++11, de donner des valeurs par défaut aux attributs en dehors d'un constructeur, en les initialisant à leur déclaration. De ce fait le rôle du constructeur par défaut perd son intérêt (qui était justement de donner des valeurs par défaut aux attributs). Cependant, il est généralement nécessaire d'écrire tout de même un constructeur par défaut (sauf s'il n'y a aucun autre constructeur), même s'il ne fait rien.

Ex. (code simplifié, avec les constructeurs écrits « *in line* ») :

```

class Point{
    public :
        Point(){} // Constructeur par défaut « vide »
        // ou bien : Point() = default;
        Point(int xinit, int yinit) : x{xinit}, y{yinit}{}
        . . .
    private :
        int x{0},y{0}; // Initialisation des attributs
};

```

Remarque : les avis sont partagés sur cette pratique. Certains considèrent qu'il est préférable de fixer les valeurs initiales dans le constructeur par défaut, sauf dans le cas où ces initialisations sont communes à plusieurs constructeurs (ce qui évite alors de dupliquer du code).

7- Le constructeur de copie (*copy constructor*)

Le constructeur de copie est un constructeur qui crée un objet en prenant en paramètre un autre objet (de même classe) et en le recopiant.

Il n'est pas obligatoire dans une classe, mais si vous ne l'écrivez pas vous même, le compilateur le crée en arrière-plan. Il fait cette copie en recopiant l'objet initial membre à membre.

Vous devez l'écrire vous-même si vous souhaitez que celui-ci fasse autre chose que cette recopie membre à membre.

Écriture du **code du constructeur de recopie** de la classe Point (dans fichier Point.cpp). Ici il ne fait que la recopie membre à membre, mais en plus laisse un message à l'écran (permet de voir quand il est appelé) :

//Constructeur de recopie :

```
Point::Point( const Point &ptinit): x {ptinit.x}, y{ptinit.y}{  
    cout << "Appel constructeur de recopie\n";  
}
```

Pour le mot-clé const cf. §13

Important! : ce constructeur de recopie est appelé implicitement chaque fois qu'un paramètre de type Point est passé par valeur à une fonction.

8- Le mot-clé *this*

this est un pointeur sur l'« objet courant ». Il est créé automatiquement par le compilateur et est disponible à l'intérieur du corps de toutes les méthodes (sauf les méthodes de classe, cf. §12). Il contient l'adresse de l'instance sur laquelle la méthode est appelée.

En conséquence ***this** représente l'« objet courant » dans le code des méthodes.

Ex. d'utilisation : le code de la fonction *setter* setX() de Point peut s'écrire de deux façons :

1- // Le nom du paramètre doit être différent du nom de l'attribut (ici xparam ≠ x)

```
void setX (int xparam){  
    x = xparam;  
}
```

2- // Avantage : permet d'utiliser un nom de paramètre identique à celui de l'attribut

```
void setX (int x){  
    this->x = x;  
    // ou : (*this).x = x;  
}
```

Résumé sur les appels de constructeur/destructeur

A chaque fois qu'on instancie un objet, il y a un appel automatique d'un constructeur (celui qui convient).

	Quand a lieu l'appel?	Rôle
Constructeur	appelé automatiquement à l'instanciation	<ul style="list-style-type: none">• Initialise les attributs• Acquiert éventuellement (si nécessaire à la classe) des ressources système (mémoire,...)
Destructeur	appelé automatiquement quand l'objet sort de la portée (bloc ou fonction)	Laisser l'environnement dans l'état où il était avant l'instanciation → libérer mémoire éventuellement allouée → fermer fichier éventuellement ouvert, etc.

9- Pointeur sur objet

On parle ici des pointeurs « classiques », comme en C (appelés *raw pointers* ou *dumb pointers*). On verra plus tard (Ch. 6) les pointeurs plus évolués du C++ (dits « *smart pointers* »).

```
#include "Point.h"

int main(){
    Point un_point; //ou bien : Point un_point{}; mais jamais de ( ): Point un_point{};
    Point *un_p_point{nullptr}; // Pointeur sur objet
    un_p_point = new Point; //Allocation mémoire + appel implicite du cteur sans argument
    //On peut écrire aussi : un_p_point = new Point{}; //Différence « subtile » au niveau de
    // l'initialisation si on n'a pas défini de constructeur par défaut, ce qui n'est pas le cas ici
    Point *un_p_point2 = new Point{3,5}; // Appel implicite du cteur à 2 args
    // ou bien new Point{3,5}; ancienne syntaxe C++98
    un_p_point->affiche();
}
```

10- Tableaux d'objets

1) Tableau de taille fixe

```
const int kTaille{100};
Point tab_point[kTaille]; //n'est possible que s'il y a un cteur sans argument (cteur par défaut)
// 100 fois appel implicite du cteur par défaut
tab_point[2].affiche();
```

2) Tableau d'objets alloué dynamiquement (c'est le tableau qui est alloué, pas les objets)

```
Point *tab_point{nullptr};
tab_point = new Point[kTaille];
tab_point[2].affiche();
```

3) Tableau de pointeurs/objet de taille fixe

```
Point *tab_point[kTaille]; //crée 100 pointeurs sauvages (un zoo de pointeurs :-)
//tab_point[0]->affiche(); //L'accès à un élément du tableau va créer une erreur de segmentation
// Deux façons possibles pour initialiser chaque pointeur:
// 1- Avec un for classique
for ( int i = 0; i < kTaille; ++i ){
    tab_point[i] = new Point;
}
```

```

// 2- Avec un range for (ne fonctionne pas sans la référence)
for ( auto &obj : tab_point ){//obj est une référence sur chaque élément du tableau
    obj = new Point;
}
// Affichage de chaque point (possible aussi avec un for classique)
for ( auto obj : tab_point ){//obj est une copie de chaque élément du tableau
    obj->affiche();
}
// ou bien :
for ( const auto &obj : tab_point ){//évite la recopie de chaque élément =>performance ++
    obj->affiche();
}

```

4) Tableau (alloué dynamiquement) de pointeurs/objet alloués dynamiquement

```

Point **tab_point {nullptr};
tab_point = new Point * [kTaille]; //allocation dynamique de 100 pointeurs sauvages
// Mêmes méthodes que ci-dessus pour l'allocation des pointeurs et l'affichage des points :
// 1- Avec un for classique
for ( int i = 0; i < kTaille; ++i ){
    tab_point[i] = new Point;
}
// 2- Avec un range for (ne fonctionne pas sans la référence)
for ( auto &obj : tab_point ){
    obj = new Point;
}
// Affichage de chaque point (possible aussi avec un for classique)
for ( const auto &obj : tab_point ){
    obj->affiche();
}

```

11- Utilisation du mot-clé const dans un entête de méthode

Le mot-clé `const` est un « *qualifier* » (terme anglais qu'on peut traduire par « modificateur »), qui va s'ajouter à un type dans une déclaration. Il peut se trouver à deux positions différentes dans la déclaration et dans l'entête de la définition d'une méthode :

- dans le type d'un paramètre : c'est une utilisation qui existe déjà en C. Elle indique que le paramètre ne doit pas être modifié par la fonction.
- après la liste des paramètres (mais avant le début du corps) : il indique que l'objet courant ne doit pas être modifié par la fonction. Typiquement, c'est le cas des fonctions *getters*. Il est conseillé de mettre `const` systématiquement si la méthode n'a pas comme but de modifier l'objet courant.

Ex :

```
class Animal {  
    private:  
        int x, y, energie;  
    public:  
        bool samePosition(const Animal &anim) const {  
            return x == anim.x && y == anim.y;  
        }  
        int getX() const {  
            return x;  
        }  
};
```

indique que le paramètre `anim` ne peut pas être modifié dans la fonction

indique que l'objet courant ne peut pas être modifié dans la fonction

Note : on peut trouver aussi le mot-clé `const` à une 3^{ème} position : en « *qualifier* » du type de retour d'une méthode. C'est une utilisation rare : les cas d'utilisation sont très peu fréquents.

TP ANIMAUX

classe Animal

- 3 attributs : `x`, `y`, `energie`(=points de vie)
- constructeurs : par défaut et à 3 args
- autres méthodes : `getX()`, `getY()`, `getEnergie()`, `setX()`, `setY()`, `setEnergie()`, `affiche()`
- destructeur (corps vide)

Rq : Ajouter tous les `const` possibles dans toutes les déclarations et définitions de fonctions de la classe.

main() :

- instantiation d'un `Animal`, d'un `ptr/Animal`, d'un tableau d'`Animal`, d'un tableau de `ptr/Animal`
- delete d'un `ptr/Animal`
- appel de la méthode `affiche()` sur chaque élément du tableau (peut être codé avec un range for)
- appel des getters et setters

12- Notion d'attribut (variable) de classe

Définition : Donnée de la classe unique pour tous les objets de la classe. Il n'a pas une valeur spécifique pour chaque instance, il a la même valeur pour tous les objets.

N'est pas stocké dans chaque objet, il n'est stocké qu'à un seul emplacement mémoire quel que soit le nombre d'objets (au niveau des segments mémoire, il est stocké dans la zone des globales, statiques et constantes : *data*, *rodata* ou *bss segment*). Il peut être constant (mot-clé `const`) ou variable.

Un attribut de classe constant peut être initialisé à sa déclaration dans le bloc de classe, mais seulement s'il est d'un type entier (`char`, `int`, `long`, ...).

Un attribut de classe variable (ou constant non entier) doit être initialisé à l'extérieur du bloc de classe (**sauf depuis C++17**, où il peut être initialisé directement, en ajoutant le mot-clé `inline`).

Par ex :

- Tous les animaux sont représentés graphiquement par la même lettre 'A' (ou la même image, etc..) = constante identique pour tous
- Je veux stocker le nombre d'objets de type `Animal` instanciés = une seule variable pour la classe

Ecriture d'une variable de classe en C++ : précédée du mot-clé **static**

Animal.h

```
class Animal {
private:
    int x, y, energie;
    static const char lettre{'A'}; //attribut de classe constant
    static int nb_animaux; //attribut de classe variable
    //ou: static inline int nb_animaux{0}; //C++17
public:
    Animal(): x{0}, y{0}, energie{0}{
        nb_animaux++;
    }
    ~Animal(){
        nb_animaux--;
    }
    void affiche() const; // explication de const §13
};
```

indique
attribut de
classe

Animal::lettre

A

Animal::nb_animaux

2

data + bss
segment

lapin

x

y

energie

tortue

x

y

energie

stack

Animal.cpp

```
#include "Animal.h"
```

```
int Animal::nb_animaux{0}; //initialisation obligatoire avant C++17 d'un attribut de classe variable
void Animal::affiche() const{
    ...
    cout << "Nb animaux: " << nb_animaux << endl;
}
```

13- Notion de fonction (méthode) de classe

En C++ : type de retour de la fonction précédé du mot-clé **static**

Méthode de classe = fonction membre de classe = fonction membre statique

(class method) = (class member function) = (static member function))

Définition : Fonction membre d'une classe qui peut être appelée sans objet de la classe. Elle a accès aux membres (données et fonctions) statiques de la classe, même s'ils sont privés.



Une méthode de classe ne peut pas utiliser le pointeur `this` : en effet, elle n'est pas liée à une instance particulière de la classe.

Ex :

```
class Animal{
    ...
    ...
public:
    static int getNbAnimaux(); //retourne la valeur de
nb_animaux
private:
    static int nb_animaux;
};
```

Bloc de déclaration de classe

Utilisation d'une méthode de classe statique : doit être préfixée du nom de la classe (appel sans objet)

```
#include "Animal.h"
int main(){
    Animal un_animal; //instanciation d'un objet de classe Animal
    std::cout << un_animal.getX(); // appel d'une méthode banale (sur un objet)
    std::cout << Animal::getNbAnimaux(); //appel d'une méthode statique (sans objet)
    std::cout << un_animal.getNbAnimaux(); //appel possible aussi sur un objet
}
```

TP animaux :

Ajouter variable `static` et fonction `static` dans le code

Ch 4 : Composition de classes

Plan du chapitre:

1- Concept de composition

2- Mise en œuvre

3- Diagramme UML de classes

1- Concept de composition

Une classe peut avoir des attributs qui sont des objets d'une autre classe. Ceci se traduit par :

Un objet	a	un(des) objet(s) membre(s)
	se compose de	

Le concept de composition de classes (ou d'objets) est donc parfois appelé « **a un** » (« **has a** »). La classe « externe » (contenante) est dite **classe composite** (*composite class*) et la classe de l'objet interne est dite **classe membre** (*member class*).

C'est un concept extrêmement répandu en programmation objet.

Ex : une droite a (se compose de) 2 points

une ligne brisée a (se compose d') un tableau de points

Les méthodes de modélisation objet distinguent la composition (*composition*) de l'agrégation (*aggregation*) : la composition est une agrégation « forte » (cf. §3 UML).

2- Mise en œuvre

Ex. de la droite composée de 2 points.

Droite est la classe composite, Point la classe membre :

```
class Point{
    public :
        Point(int x_init = 0, int y_init = 0): x{x_init}, y{y_init}{}
        int getX() const {return x;}
        int getY() const {return y;}
    private :
        int x,y;
};
```

```

class Droite{
    private :
        Point point_debut, point_fin;
    public :
        Droite(){}
        Droite(const Point &debut, const Point &fin):
            point_debut{debut}, point_fin{fin} {}
        Point getPointDebut() const {return point_debut;}
        Point getPointFin() const {return point_fin;}
        void affiche() const{
            std::cout <<
                "Début:x="<<point_debut.getX()<<",y="<<point_debut.getY()<<
                " Fin: x="<<point_fin.getX()<<",y="<<point_fin.getY()<<"\n";
        }
};

```

Dans le main :

```

int main(){
    Droite une_droite; //appel implicite du constructeur de Droite sans argument
    une_droite.affiche();
    Droite une_autre_droite{Point{5,12}, Point{3,4}}; //appel implicite du
                                                        // constructeur de Droite à 2 args
    une_autre_droite.affiche();
}

```

Remarque : le constructeur sans argument de `Point` est appelé automatiquement (2 fois car 2 points) à l'instanciation d'une `Droite` sans argument.

3- Diagramme UML de classes

UML : Unified Modeling Language

Langage de modélisation créé originellement pour la conception des programmes en langage objet.

Plusieurs types de diagrammes UML, mais le + important est le diagramme de classes. On y représente les classes utilisées par le programme et les liens entre elles.

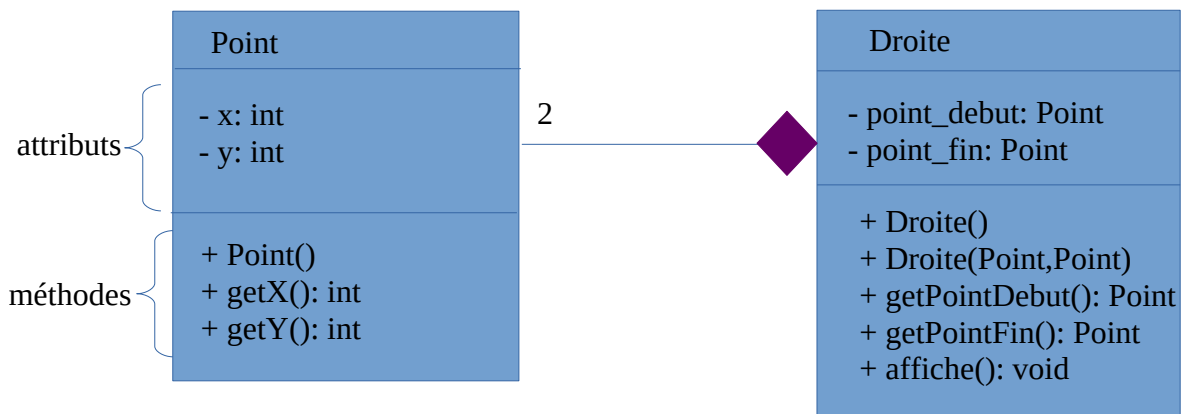
Quelques règles de représentation :

- un bloc par classe avec 3 sous-blocs, de haut en bas : nom de la classe, attributs et méthodes,
- le type de chaque attribut est indiqué, ainsi que le type des paramètres et du retour des méthodes,
- méthodes et attributs précédés par + : *public*, par - : *private*, par # : *protected*

- attributs et méthodes de classe : soulignés
- lien de composition entre classes : représenté par un trait entre les 2 classes, avec un losange plein (ou vide si agrégation seulement) du côté de la classe composite et le nombre d'objets membres (appelé « multiplicité ») noté du côté de la classe membre.

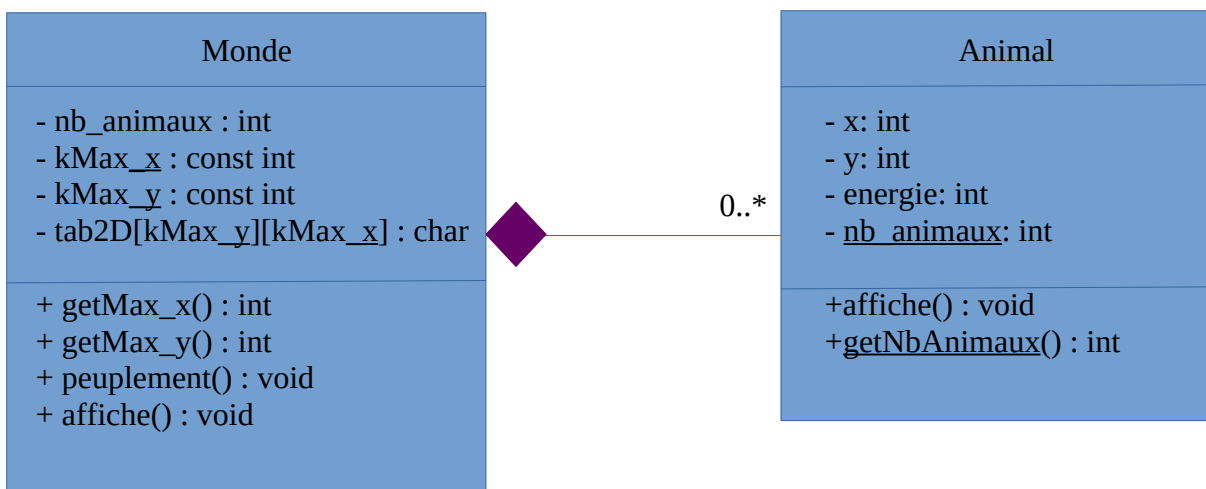
Note : différence entre composition et agrégation : dans la composition, les objets membres sont des parties de l'objet principal (composite), et disparaissent si l'objet composite disparaît. Dans l'agrégation, les objets membres peuvent exister même si l'objet principal a disparu (le lien est moins fort entre les 2 classes).

Ex de la Droite composée de 2 Points :



Remarque : pour une représentation simplifiée, il n'est pas nécessaire de montrer certains éléments évidents, comme les constructeurs, destructeurs, getters/setters (s'ils sont obligatoires pour tous les attributs). On peut aussi ne pas représenter les attributs stockant les liens vers les objets membres dans un objet composite (ex: vecteur d'Animal ci-dessous) : ils sont représentés graphiquement par le losange et les multiplicités.

Ex : TP Animal + Monde (simplifié)



Ch 5 : L'héritage

Plan du chapitre:

1- Généralités

2- Diagramme UML de classes

3- Statut des attributs et méthodes hérités

4- Redéfinition (« *overriding* ») de méthode dans une classe Dérivée

5- Règles d'appel des constructeurs

6- Règles d'appel des destructeurs

7- Représentation en mémoire

8- Le polymorphisme d'héritage (« *subtype polymorphism* »)

- a) Conversion entre objets de classe de Base et de classe Dérivée
- b) Conversion entre pointeurs (ou références) sur objets de classe B et D
- c) Les méthodes (fonctions membres) virtuelles => polymorphisme

9- Les classes abstraites (fonctions virtuelles pures)

10- Notion de RTTI (*Run-Time Type Information*)

1- Généralités

Héritage (*inheritance*) : concept majeur dans les langages objets.

A partir d'une **classe de base B** on crée une **classe dérivée D** par **spécialisation**.

La classe de base est dite **classe mère** (*parent class*), superclasse (*superclass*) ou super-type (*supertype*) et la classe dérivée est dite **classe fille** (*child class*), sous-classe (*subclass*) ou sous-type (*subtype*).

On dit qu'une classe fille D **hérite** (*inherits*) de sa classe mère B. On peut aussi toujours dire qu'un objet de classe D est aussi un objet de classe B => le concept d'héritage de classes est donc parfois appelé « **est un** » (« **is a** »). Ce concept ne doit donc pas être confondu avec celui de la composition de classes (concept « a un », cf. Ch. 4).

Exemple :

B= Animal, D = Lion , D = Gazelle

Les Lions et les Gazelles héritent des caractéristiques de la classe de base Animal. Un Lion est un Animal, une Gazelle est un Animal. Les caractéristiques supplémentaires de Lion et Gazelle (qui ne sont pas dans Animal) sont des caractéristiques spécialisées à ces deux type dérivés.

Ex. : une classe de base `Point` et une classe dérivée `PointColor` : caractéristique supplémentaire (attribut) : couleur.

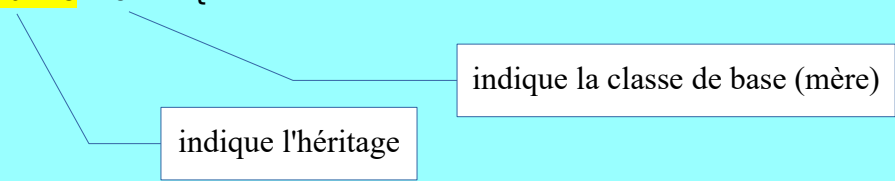
Une méthode `affiche()` est définie dans `Point` et redéfinie dans `PointColor`.

```
class Point{
    private :
        int x,y;
    public :
        // Constructeur
        Point(int x_init = 0, int y_init = 0):x{x_init},y{y_init}{}
        // Destructeur
        ~Point(){}
        // Autres méthodes
        void affiche() const {std::cout << "x=" << x << ",y=" << y << "\n";}
};

//classe dérivée PointColor
class PointColor: public Point {
    private :
        int couleur;
    public :
        // Constructeurs
        PointColor(): couleur{0}{} // appel implicite du constructeur de Point (classe mère)
                                   // sans arguments (cf. §5)
        PointColor(int x_init, int y_init, int col_init)
            : Point{x_init, y_init}, couleur{col_init}{} // appel explicite du
                                                         // constructeur Point à 2 args (cf. §5)

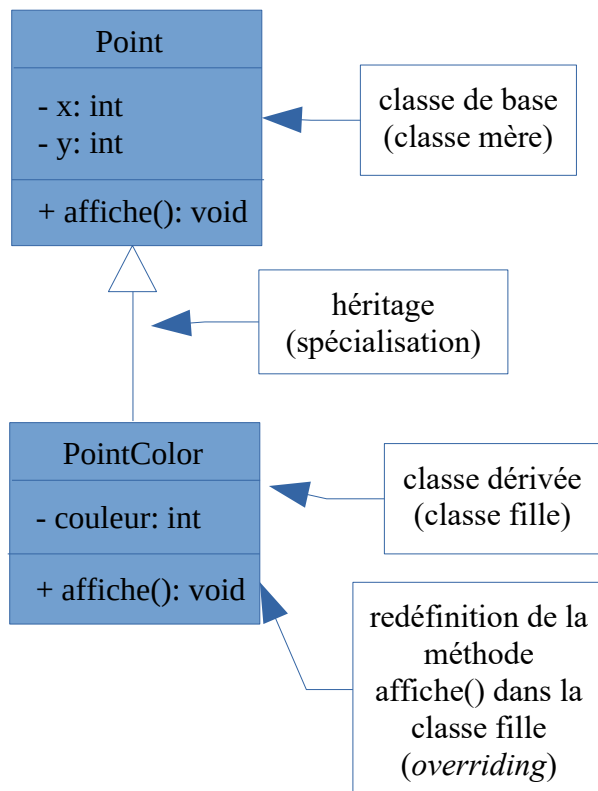
        // Destructeur
        ~PointColor(){}
        // Autres méthodes
        void affiche() const { // Redéfinition de la fonction affiche() dans la classe fille (cf. §4)
                               // => possibilité de « polymorphisme d'héritage » (cf. §8)

            Point::affiche();// Appel explicite de la méthode affiche() de la classe mère
            std::cout << "couleur=" << couleur << "\n";
        }
};
```

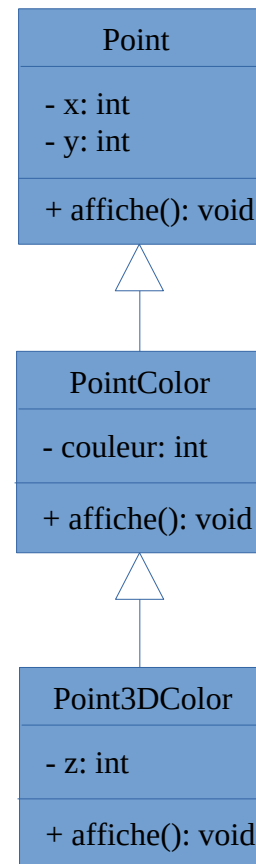


2- Diagramme UML de classes

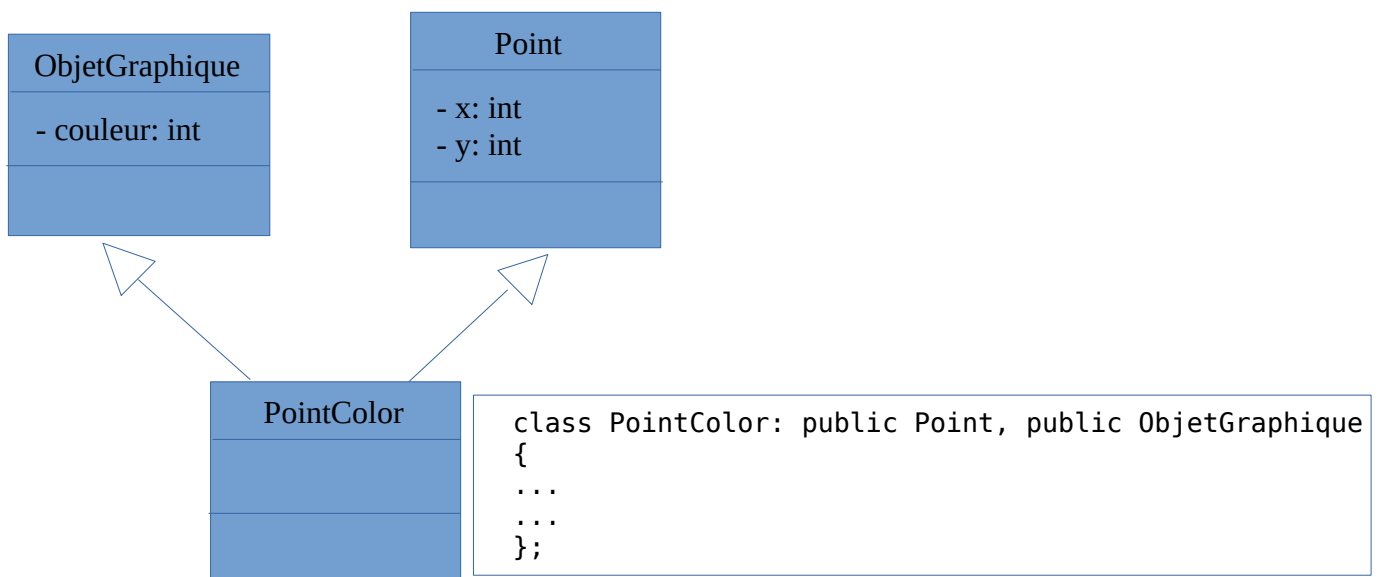
Ex. : héritage de deux classes (2 niveaux)
cf. code §1



Autre ex. : hiérarchie à 3 niveaux (3 classes)
Point3DColor est une classe fille de **PointColor**



En C++ (contrairement à d'autres langages objets comme *Java* et *PHP*) il existe un concept d'**héritage multiple** : une classe fille peut avoir plusieurs classes mères. Ex :



Ici, la classe **PointColor** hérite l'attribut `couleur` de **ObjetGraphique** et les attributs `x`, `y` de la classe **Point**. L'héritage multiple est assez complexe à mettre en oeuvre => il est assez rare.

3- Statut des attributs et méthodes hérités

Une classe dérivée hérite automatiquement (sans avoir à les redéclarer : seuls sont déclarés dans la classe les attributs/méthodes « en propre » = spécifiques) :

- des attributs de la classe de base
- des méthodes publiques de la classe de base

L'héritage est transitif : avec une hiérarchie de classes à plus de deux niveaux, les membres de la classe de plus haut niveau sont hérités par la classe fille et aussi par les classes de plus bas niveau (c'est-à-dire par toutes les classes « descendantes »).

L'héritage est automatique aussi bien pour les attributs privés que publics. Cependant, s'ils sont privés, on ne peut y accéder dans la classe fille qu'à travers des getters/setters publics de la classe mère. Inconvénient : ça alourdit le code.

Ex. dans PointColor :

```
void affiche() const {  
    std::cout << "x=" << getX() << ",y=" << getY() <<  
        ",couleur=" << couleur << endl;  
}
```

getters de la classe de base
(mère) pour les attributs hérités

accès direct pour les
attributs « en propre »

Solution pour éviter cette lourdeur : déclarer les **attributs « *protected* »** dans la classe de base.

Les attributs *protected* sont accessibles directement (sans *getter/setter*) dans les classes filles (et leur descendantes) mais pas ailleurs, où ils ont un statut équivalent à *private*.

Accès direct des attributs	<i>public</i>	<i>protected</i>	<i>private</i>
Dans fonctions membres de la même classe	✓	✓	✓
Dans fonctions membres des classes dérivées	✓	✓	✗
Dans d'autres fonctions	✓	✗	✗

Dans le cas où une classe a des classes dérivées, déclarer ses attributs *protected* plutôt que *private*, c'est-à-dire accessibles dans la classe elle-même mais aussi dans toutes les classes descendantes, est une pratique courante. Cependant c'est souvent critiqué comme une violation partielle du principe d'encapsulation, et déconseillé par certains guides de bonnes pratiques.

4- Redéfinition («*overriding*») de méthode dans une classe Dérivée

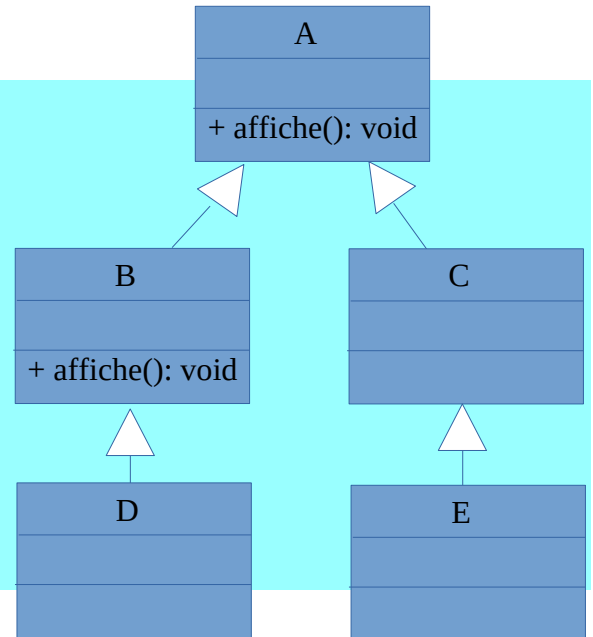
Contrairement aux attributs, pour lesquels ça n'a pas d'intérêt, on peut être amené à redéfinir (« *overriding* » du verbe *override* [sth]: « primer sur, l'emporter sur [qq chose] ») dans une classe Dérivée une méthode déjà définie dans une classe de Base (ex: la méthode `affiche()` des classes `Point` et `PointColor`, cf §1). ⚠ Ne pas confondre avec la surcharge de fonction (« *overloading* »), cf. Ch.2 §A.2.

C'est nécessaire quand une même méthode met en œuvre un comportement différent selon la classe.

Quand une méthode est appelée sur un objet, le compilateur recherche si la méthode existe dans la classe. Si oui, il l'applique, sinon, il va remonter la hiérarchie des classes jusqu'à trouver une méthode de ce nom.

Ex :

```
int main()
{
    B objB;
    D objD;
    E objE;
    objB.affiche(); // appelle affiche() de B
    objD.affiche(); // appelle affiche() de B
                    // OK car D est un B
    objE.affiche(); // appelle affiche() de A
                    // OK car E est un A
}
```



5- Règle d'appel des constructeurs

Règle : Le constructeur de la classe de Base est toujours appelé dans les constructeurs de la classe Dérivée.

Ordre d'appel : quand vous instanciez un objet de classe Dérivée, le constructeur de la classe de Base est appelé d'abord, puis ensuite le constructeur de la classe Dérivée termine l'exécution. Cette règle s'applique récursivement : si vous avez une hiérarchie de classes à plus de deux niveaux, c'est le constructeur de plus haut niveau qui est appelé d'abord, puis celui de sa classe fille, etc. jusqu'à la classe dérivée de plus bas niveau.

Appel implicite/explicite (cf. l'exemple des constructeurs de la classe `PointColor` du §1) :

1. Quand le constructeur par défaut de la classe Dérivée est appelé, le constructeur par défaut de la classe de Base est appelé implicitement avant l'exécution du corps du constructeur.
2. Si vous voulez passer un(des) paramètre(s) au constructeur de la classe de Base, il faut l'appeler explicitement dans la liste d'initialisation, avec les paramètres qui lui sont destinés. Dans ce cas, ce sera le constructeur adapté de la classe de Base (selon le nombre et le type d'arguments) qui sera appelé (mais toujours avant l'exécution du corps du constructeur).

Ex. simplifié :

```
class Base
{
    private:
        int x;
    public:
        // Constructeurs
        Base(){ cout << "Constructeur Base par default\n"; }
        Base(int i): x{i}{ cout << "Constructeur Base à 1 arg\n"; }
};

class Derived : public Base
{
    private:
```

```

    int y;
public:
    // Constructeurs
    Derived(){ cout << "Constructeur Derived par défaut\n"; }
    Derived(int j): Base{j}, y{j}{
        cout << "Constructeur Derived à 1 arg\n";}
};

int main()
{
    Derived d1;
    Derived d2{12};
}

```

va afficher :

```

Constructeur Base par défaut
Constructeur Derived par défaut
Constructeur Base à 1 arg
Constructeur Derived à 1 arg

```

6- Règle d'appel des destructeurs

Règle : Le destructeur de la classe de Base est toujours appelé implicitement dans le destructeur de la classe Dérivée.

Ordre d'appel : à la disparition d'un objet de classe Dérivée (par sortie de la portée ou par le `delete` d'un pointeur), le destructeur de la classe Dérivée est appelé d'abord, puis ensuite le destructeur de la classe de Base termine l'exécution (c'est l'ordre inverse de l'appel des constructeurs). Cette règle s'applique récursivement : si vous avez une hiérarchie de classes à plus de deux niveaux, c'est le destructeur de plus bas niveau qui est appelé d'abord, puis celui de sa classe mère, etc. jusqu'à la classe mère de plus haut niveau.

Ex. simplifié :

```

class Base
{
    private:
        int x;
    public:
        // Destructeur
        ~Base(){ cout << "Destructeur Base\n"; }
};

class Derived : public Base
{
    private:
        int y;
    public:
        // Destructeur
        ~Derived(){ cout << "Destructeur Derived\n"; }
};

int main()
{
    Derived d;
}

```

va afficher :

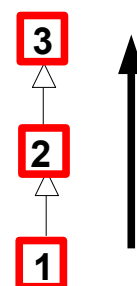
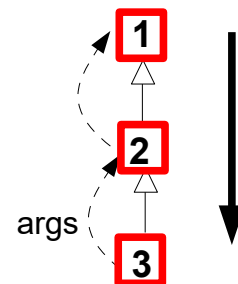
Destructeur Derived
Destructeur Base

Résumé : ordre d'appel des constructeurs/destructeurs dans une hiérarchie de classes

Les constructeurs sont appelés « en descendant »
(de parent à enfant)

Mais les arguments des constructeurs sont transférés d'enfant à parent.

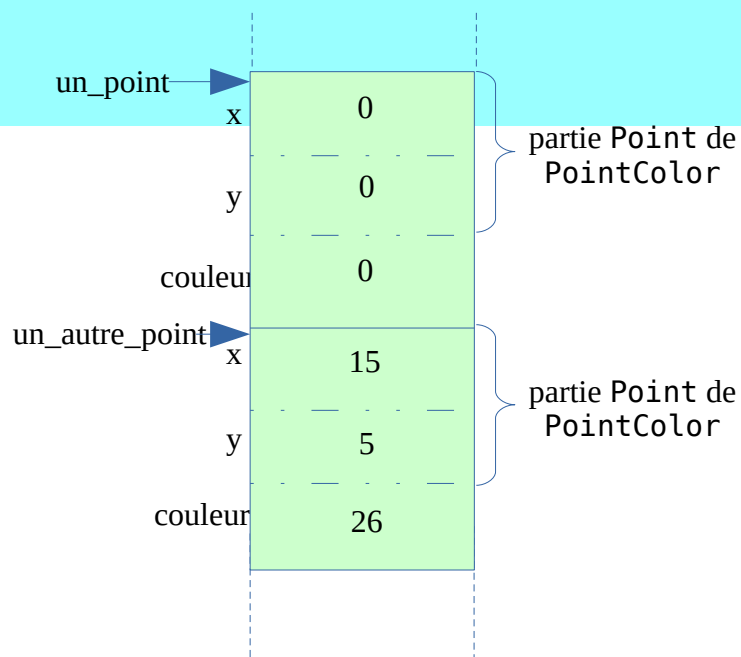
Les destructeurs sont appelés « en remontant »
(d'enfant à parent)



7- Représentation en mémoire (ex. classe PointColor)

Ex : main() basé sur le code des classes du §1.

```
int main(){
    PointColor un_point;
    PointColor un_autre_point {15,5,26};
    un_point.affiche();
    un_autre_point.affiche();
}
```



Le stockage en mémoire de chaque objet regroupe tous les attributs, aussi bien les attributs « en propre » (ici couleur) que les attributs hérités (ici x et y). S'il y a plusieurs niveaux d'héritage, les attributs de tous les niveaux sont regroupés.

8- Le polymorphisme d'héritage (« *subtype polymorphism* »)

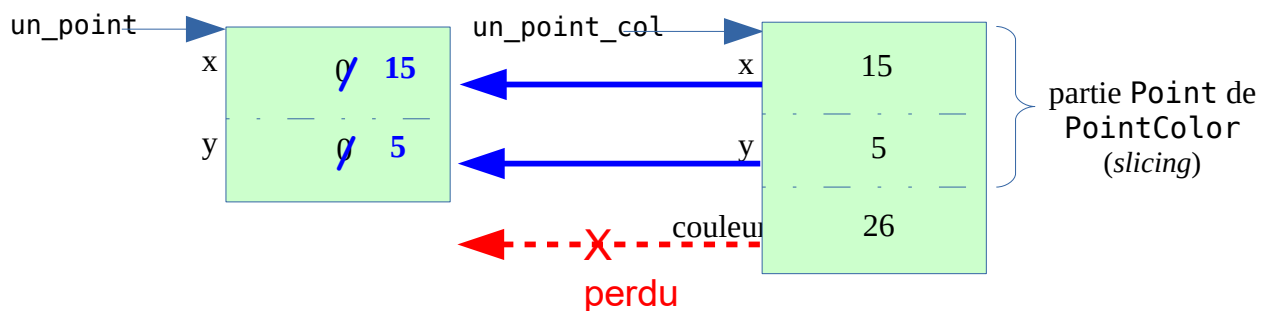
a) Conversion entre objets de classe de Base et de classe Dérivée

Rappel du principe, cf §1 : un objet d'une classe D (dérivée) **est** aussi **un** objet de classe B (Base)

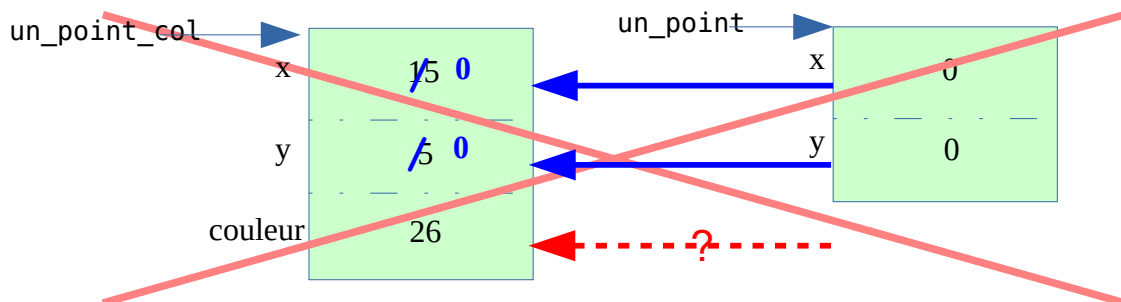
Règle : Un objet d'une classe Dérivée peut être affecté à un objet de la classe de Base. Il y a **conversion implicite**, avec perte des attributs spécifiques de la classe Dérivée (« *slicing* » = « tranchage »).

Ex :

```
int main(){
    Point un_point;
    PointColor un_point_col {15,5,26};
    un_point = un_point_col; // Correct, car un PointColor est un Point, mais perte
                             // d'information
}
```



Remarque importante : le contraire – copier un objet de classe B dans un objet de classe D – est interdit (erreur de compilation). En effet, par ex., ~~un_point_col = un_point;~~ est **interdit**, car un Point n'est pas un PointColor, on ne saurait pas quoi mettre dans l'attribut « couleur ».



Plus généralement, à chaque fois qu'un objet de classe B est attendu, on peut le remplacer par un objet de classe D (« principe de substitution de Liskov »). Ex. : en paramètre d'une fonction.

```
void f( Point p){
    p.affiche();
}
int main(){
    PointColor un_point_col;
    f(un_point_col); // Correct, car un PointColor est un Point
}
```

b) Conversion entre pointeurs (ou références) sur objets de classe B et D

Principe : globalement, c'est le même principe qu'en a) qui prévaut :

- la conversion **pointeur/classe D** → **pointeur/classe B** (**upcasting**) est **implicite** et c'est un comportement « normal », qui suit le principe de substitution de Liskov.
- Par contre, contrairement au cas d'un objet simple (conversion interdite, cf. ci-dessus), la conversion **pointeur/classe B** → **pointeur/classe D** (**downcasting**) est possible, mais seulement à l'aide d'un **cast explicite**, dangereux, et considérée généralement comme une mauvaise pratique.

NB : dans tout ce qui est dit dans ce paragraphe pour les pointeurs est également vrai pour les références.

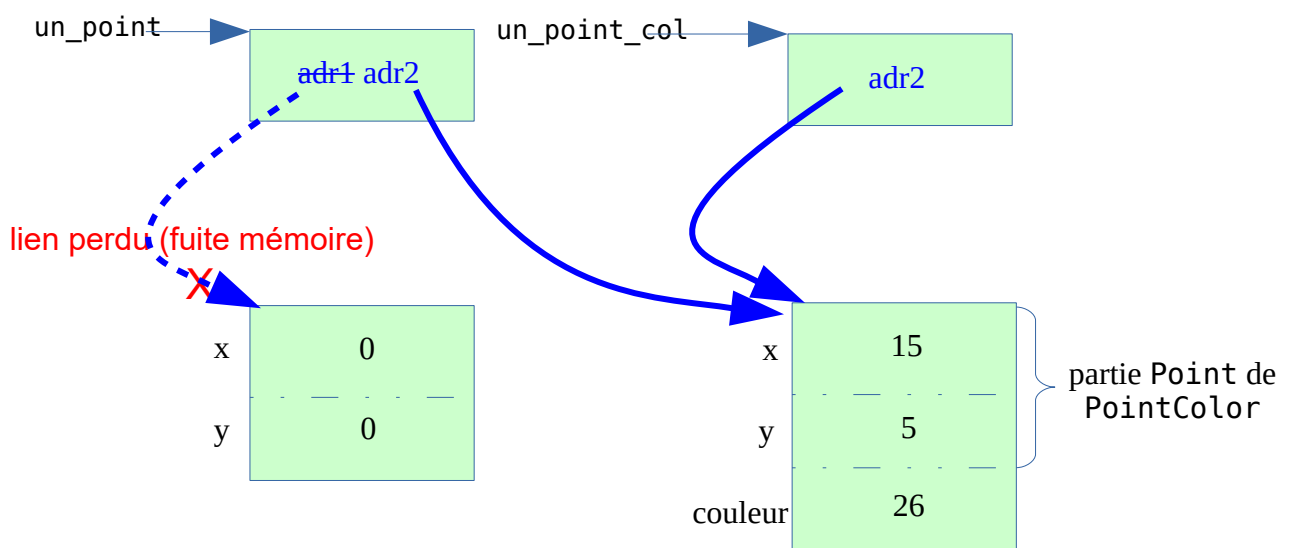
Règle de l'« upcasting » : Un pointeur d'une classe Dérivée peut être affecté à un pointeur de la classe de Base. Il y a conversion implicite. L'affectation de pointeurs a plusieurs avantages :

- plus rapide (juste une copie d'adresse) que l'affectation d'objets (qui implique une recopie attribut par attribut),
- permet d'accéder aux méthodes de la classe Dérivée, même si le pointeur est sur la classe de Base, à travers les **méthodes virtuelles** (cf. § c). C'est le polymorphisme d'héritage.

Inconvénient de l'utilisation des pointeurs : plus délicat car il faut gérer convenablement les allocations et libérations mémoire.

Ex :

```
int main(){
    Point *un_point {new Point}; // adr1 cf. schéma
    PointColor *un_point_col {new PointColor{15,5,26}}; // adr2 cf. schéma
    un_point = un_point_col; // Correct, car upcasting de PointColor vers Point,
                             // (mais cependant dans cet exemple, il y a une fuite mémoire)
}
```



Plus généralement, à chaque fois qu'un pointeur (ou référence) sur classe B est attendu, on peut le remplacer par un pointeur (ou référence) sur classe D.

Le « **downcasting** » :

NB : beaucoup moins important que l'« **upcasting** »

C'est l'opposé, c'est-à-dire convertir un pointeur (ou référence) sur classe de Base vers un pointeur (ou référence) sur classe Dérivée. Ce n'est pas autorisé sans un cast explicite. La raison de cette restriction est que, dans ce sens, ce n'est pas une relation « **est un** ». Une classe Dérivée peut avoir des attributs spécifiques, et les méthodes qui utilisent ces attributs ne s'appliqueraient alors pas à la classe de Base.

Un cast explicite vers la classe Dérivée permet la compilation, mais est dangereux car la conversion n'est pas obligatoirement pertinente (par ex., on veut afficher la couleur d'un Point), et le résultat est aberrant.

Ex : on écrit une fonction générale (pas une méthode) qui traite la couleur d'un PointColor:

```
void f( PointColor *p ){
    std::cout << "couleur: " << p->getCouleur() << std::endl;
}

int main(){
    Point *un_point {new Point{5, 12}};

    f(un_point); // Erreur de compilation (invalid conversion),
                // car un Point n'est pas un PointColor (du moins pas toujours)
    f((PointColor *)un_point); // Compilation OK, mais le résultat qui s'affiche est incorrect:
                // la fonction va tenter d'afficher la couleur du Point, attribut inexistant

    un_point = new PointColor{5, 12, 100}; // on change l'objet pointé

    f(un_point); // Toujours erreur de compilation (invalid conversion), car bien que la variable
                // pointe en réalité vers un PointColor, son type est toujours Point *
    f((PointColor *)un_point); // Compilation OK, et le résultat qui s'affiche est correct:
                // la fonction va afficher 100
}
```

Le passage par le nouvel opérateur de cast (cf. Ch. 2 §B.11) **dynamic_cast** permet de limiter le risque : si l'objet pointé n'est pas de la classe prévue par la conversion, l'opérateur renvoie un pointeur nul.

Ex : on reprend le code ci-dessus en le modifiant à l'aide d'un dynamic_cast:

```
void f( PointColor *p ){
    if ( p == nullptr ){ // ou: if ( !p )
        std::cout << "Paramètre incorrect\n";
    } else {
        std::cout << "couleur: " << p->getCouleur() << std::endl;
    }
}

int main(){
    Point *un_point {new Point{5, 12}};
    f(dynamic_cast<PointColor *>(un_point)); // Affiche:"Paramètre incorrect"
    un_point = new PointColor{5, 12, 100};
    f(dynamic_cast<PointColor *>(un_point)); // Affiche:"couleur: 100"
}
```

c) Les méthodes (fonctions membres) virtuelles => polymorphisme

L'**upcasting implicite** (cf. §b ci-dessus) rend possible pour un pointeur (ou référence) vers une classe de Base, de pointer (dynamiquement, selon les circonstances de l'exécution du programme) :

- soit vers un objet de la classe de Base
- soit vers un objet de la classe Dérivée

C'est ce qui crée le besoin de « **dynamic binding** », (« liaison dynamique » ou « **late binding** ») pour que l'objet pointé utilise les méthodes de sa classe réelle à l'exécution. C'est le rôle des **méthodes virtuelles**.

Le problème ne se pose que si une méthode d'une classe de Base doit être redéfinie dans une classe Dérivée (**overridden method**). Par défaut, la méthode appelée sur l'objet pointé est déterminée à la compilation (cf. §4). C'est ce qu'on appelle le « **static binding** », (« liaison statique ») ou « **early binding** ».

Ex. avec les classes Point et PointColor : on crée un tableau de pointeurs sur Point, mais on y alloue soit des Point, soit des PointColor.

```
int main(){
    Point *tab_point[10]; // tableau de pointeurs sur Point
    tab_point[0] = new Point{5,12};
    tab_point[1] = new PointColor{2,3,10}; // upcasting
    tab_point[0]->affiche(); // x = 5, y=12
    tab_point[1]->affiche(); // x=2, y=3 : appelle affiche() de Point = static binding car
        // bien que tab_point[1] pointe vers un PointColor, son type est toujours Point *
}
```

Pourtant, on voudrait afficher aussi la couleur pour tab_point[1], car c'est un pointeur vers PointColor.

Solution : déclarer virtuelle (mot-clé **virtual**) la fonction affiche() de la classe de Base. En reprenant l'exemple (code simplifié) :

```
class Point{
    protected:
        int x,y;
    public:
        // Constructeurs
        ...
        // Autres méthodes
        virtual void affiche() const{//La fonction et ses descendantes redéfinies sont virtuelles
            std::cout << "x=" << x << ",y=" << y << "\n";
        }
        // Destructeur
        virtual ~Point(){}// Bonne pratique : déclarer le destructeur virtuel, cf. remarque 4 plus loin
};
```

```

class PointColor: public Point {
    private:
        int couleur;
    public:
        // Constructeurs
        ...
        // Destructeur
        ~PointColor(){}
        // Autres méthodes
        void affiche() const { // Implicitement virtuelle par propagation. Cf. remarques ci-dessous
            std::cout << "x=" << x << ",y=" << y;
            std::cout << "couleur=" << couleur << "\n";
        }
};

int main(){
    // Même code que page précédente
    ...
    tab_point[1]->affiche(); // x=2, y=3, couleur=10 : appelle affiche() de PointColor
                           = dynamic binding car affiche() est virtuelle dans la hiérarchie des classes
}

```

En résumé :

Sans `virtual`: lien statique entre objet et fonction → fonction appelée selon le type à la déclaration

Avec `virtual`: lien dynamique entre objet et fonction → à l'exécution on recherche quel est le type réel de l'objet pointé et la fonction est appelée selon ce type => polymorphisme

Remarque 1 : il n'est pas obligatoire de répéter dans les classes Dérivées le mot-clé `virtual` sur les fonctions `virtual` de la classe de Base. Le statut `virtual` d'une fonction se propage implicitement en descendant la hiérarchie des classes.

Remarque 2 : cependant certains guides de bonne pratique recommandaient de répéter le mot-clé `virtual` sur les fonctions des classes Dérivées, pour améliorer la lisibilité du code (pour voir qu'une fonction est virtuelle sans regarder les classes au-dessus). Depuis la **norme C++11**, il existe une meilleure option : on peut rajouter le mot-clé `override` après le nom de la fonction virtuelle redéfinie. Ex:

```

void affiche() const override { // Méthode virtuelle redéfinie dans la classe PointColor
    ...
    // (« explicit overriding »)
}

```

Remarque 3 : les mot-clés `virtual` et `override` ne peuvent être utilisés que dans un bloc de déclaration de classe et pas en dehors (c'est-à-dire en général dans un `.h`, pas dans un `.cpp`).

Remarque 4 : les guides de bonne pratique recommandent, dès qu'il y a une méthode virtuelle dans une classe, de définir aussi un destructeur virtuel pour cette classe. Les constructeurs ne sont jamais virtuels.

Remarque 5 : en UML on représente une fonction virtuelle soit en *italique*, soit précédée de <<virtual>>.

Définition : une classe polymorphe (*polymorphic class*) est une classe contenant au moins une fonction virtuelle ou dont une des classes de base contient une fonction virtuelle.

Conseil des guides de bonne pratique :

Si une fonction doit être redéfinie dans une(des) classe(s) Dérivée(s), il faut la rendre virtuelle dans la classe de Base, c'est-à-dire qu'on doit appliquer systématiquement le « *dynamic binding* », il n'y a pas de raison à priori de rester en « *static binding* » => préférer les hiérarchies de classes polymorphes.

Mécanisme des fonctions virtuelles :

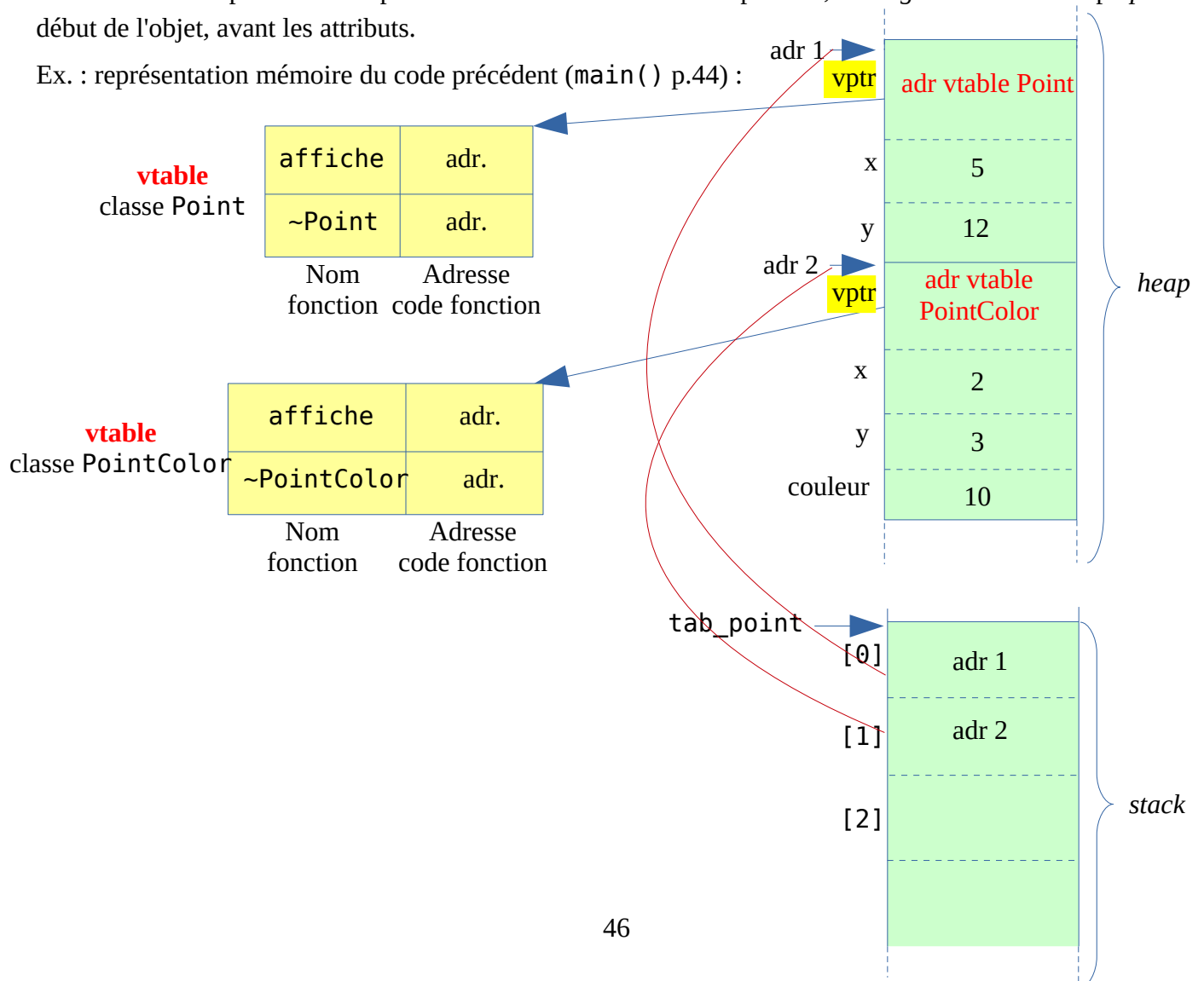
Comment fonctionne le C++ pour déterminer quelle est la fonction à appeler pour le type réel de l'objet?

Cela passe par le mécanisme de « table de fonctions virtuelles » (*virtual function table*), appelée en abrégé **vtable**. A chaque classe contenant une(des) fonction(s) virtuelle(s) est associée une vtable. C'est une table contenant des « slots » de type « clé-valeur », la clé est le nom d'une fonction virtuelle, la valeur est l'adresse du code binaire de la fonction (pointeur de fonction). Cette table est unique pour la classe.

Remarque : au niveau des segments mémoire, le compilateur *g++* stocke les *vtables* dans la zone des constantes (*rodata segment*).

Chaque objet d'une classe contenant au moins une fonction virtuelle possède, en plus des attributs, un champ supplémentaire (caché), le « pointeur de table virtuelle » (*vp_ptr*) qui contient l'adresse de la *vtable* de la classe. L'implémentation peut être différente selon le compilateur, mais *g++* met le champ *vp_ptr* au début de l'objet, avant les attributs.

Ex. : représentation mémoire du code précédent (main () p.44) :



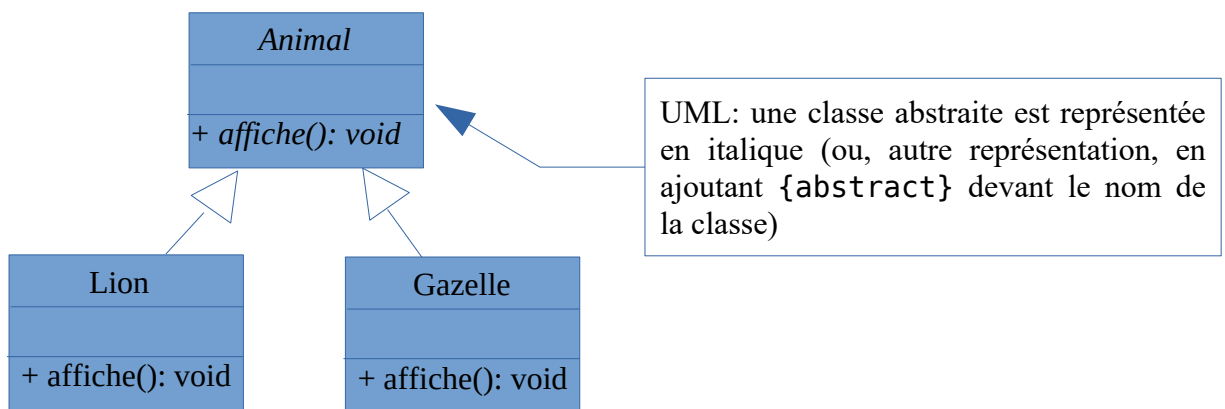
9- Les classes abstraites (fonctions virtuelles pures)

Une **classe abstraite** (*abstract class*) est une classe pour laquelle on ne peut pas instancier d'objet. Elle va donc servir uniquement de modèle de base pour des classes dérivées.

Le contraire d'une classe abstraite est une classe concrète (*concrete class*) : c'était le cas de tous les exemples de classes que l'on a vus depuis le chapitre 3.

Une classe abstraite n'est donc pas « instanciable », mais (en C++) on peut néanmoins créer des pointeurs (ou références) du type de la classe abstraite de Base qui pointeront (ou référenceront) des objets d'une classe Dérivée concrète (*upcasting*, cf. §8).

Ex : une classe abstraite `Animal` peut servir de Base à des classes Dérivées `Lion`, `Gazelle`, etc. On ne pourra donc pas créer d'objets de type `Animal`, mais on pourra créer des pointeurs vers `Animal`.



Syntaxe C++ :

Dans plusieurs langages objets (*Java*, *PHP*, *C#*,...), il suffit de rajouter le mot-clé *abstract* dans la définition de la classe. Ce mot-clé n'existe pas en C++.

En C++ une classe est abstraite si elle contient au moins une **méthode (fonction) virtuelle pure** (*pure virtual function*), c'est-à-dire que son entête commence par **virtual** et se termine par **= 0** (elle n'a en général pas de corps défini, bien que dans la norme C++ ce soit possible).

méthode virtuelle pure <=> classe abstraite

Les classes Dérivées concrètes :

Une fonction virtuelle pure doit être (re)définie (*overriden*) dans les classes Dérivées (sinon ces classes sont elles mêmes abstraites). Ce n'est donc que le prototype de la fonction dans la classe de Base qui a une utilité : il doit être repris à l'identique par les classes Dérivées concrètes.

Ex : la classe de Base `Point` est abstraite, la classe `PointColor` est une classe Dérivée concrète.

```
class Point{
    protected:
        int x,y;
    public:
        // Constructeurs
```

```

...
// Autres méthodes
virtual void affiche() const = 0; // Méthode (fonction) virtuelle pure
// Destructeur
virtual ~Point(){}
};

class PointColor: public Point {
private:
    int couleur;
public:
    // Constructeurs
    ...
    // Destructeur
    ~PointColor(){}
    // Autres méthodes
    void affiche() const override { // Le mot-clé override n'est pas obligatoire, cf. §8
        std::cout << "x=" << x << ",y=" << y;
        std::cout << "couleur=" << couleur << "\n";
    } // Cette définition de la fonction affiche() rend la classe PointColor concrète
};

int main(){
    Point un_point; // Erreur de compilation (cannot declare variable to be of abstract type 'Point')
    PointColor un_point_col; // OK, PointColor est une classe concrète
    Point *tab_point[10]; // OK, on peut déclarer des pointeurs sur Point
    tab_point[0] = new Point; // Erreur de compilation (invalid new of abstract class type 'Point')
    tab_point[1] = new PointColor; // OK (upcasting de PointColor * vers Point *)
}

```

10- Notion de RTTI (Run-Time Type Information)

RTTI est le nom du mécanisme C++ pour déterminer le type (donc la classe) d'un objet pendant l'exécution du programme. Ce mécanisme porte plus généralement, en programmation objet, le nom d'« introspection de type » (*type introspection*). Il s'agit bien du typage dynamique, qui a bien sûr un rapport avec la notion de polymorphisme vue au §8. Le mécanisme RTTI n'a d'intérêt que pour des classes polymorphes, parce que sinon le type des objets est connu à la compilation.

Il y a fondamentalement deux opérations possibles :

- 1) est-ce qu'un objet est d'une classe X (vrai/faux)?

2) de quelle classe est un objet?

Tous les langages objets majeurs ont des fonctions ou des opérateurs pour réaliser ces deux opérations :

- pour l'opération 1) : `instanceof` en *Java*, *JavaScript*, *PHP*, `is` en *C#*, `isinstance` en *Python*
- pour l'opération 2) : `getClass` en *Java*, `get_class` en *PHP*, `GetType` en *C#*, `type` en *Python*

En C++ :

Pour l'**opération 1**) : on utilise l'**opérateur `dynamic_cast`** (cf. §8.b). Cet opérateur ne fonctionne que sur les pointeurs ou les références, pas sur les objets eux-mêmes.

- Avec un pointeur : `dynamic_cast<X *>(p_obj)` : quand la classe de l'objet pointé est X (ou une de ses classes descendantes), `dynamic_cast` renvoie un pointeur valide, sinon il renvoie le pointeur nul. Ex :

```
Point *p_obj {new PointColor{5,12,100}};

if ( dynamic_cast<PointColor *>(p_obj) ){
    std::cout << "L'objet pointé est de classe PointColor\n";
} else {
    std::cout << "L'objet pointé n'est PAS de classe PointColor\n";
}

// Et aussi OK avec la classe de Base (car un PointColor est aussi un Point):
if ( dynamic_cast<Point *>(p_obj) ){
    std::cout << "L'objet pointé est de classe Point\n";
} else {
    std::cout << "L'objet pointé n'est PAS de classe Point\n";
}
```

Va afficher :

```
L'objet pointé est de classe PointColor
L'objet pointé est de classe Point
```

- Avec une référence : `dynamic_cast<X &>(r_obj)` : c'est un peu plus compliqué à tester, car quand `dynamic_cast` ne trouve pas la bonne classe, il ne peut pas renvoyer de référence nulle (ne peut pas exister selon la norme C++). Dans ce cas (=l'objet référencé n'est pas de la classe recherchée ou d'une classe descendante), une exception est levée (`std::bad_cast`), cf. Ch. 7 §1.c.

Pour l'**opération 2**) : on utilise l'**opérateur `typeid`**. Cet opérateur fonctionne aussi bien sur les objets que sur les pointeurs ou les références.

`typeid` renvoie un objet de classe `std::type_info` (défini dans le *header* `<typeinfo>`), dont une des méthodes s'appelle `name()` et renvoie une chaîne de caractères (`const char *`). Malheureusement le nom retourné n'est pas fixé par la norme et est différent selon les compilateurs : certains renvoient le nom effectif de la classe, d'autres (dont `g++`) renvoient un « *mangled name* » (cf. Ch.2 §A.2), par ex : « `10PointColor` » pour un objet de la classe `PointColor`, ou « `P5Point` » pour un pointeur sur la classe `Point` (l'entier indique le nombre de caractères du nom de la classe).

Si la hiérarchie de classe est polymorphe (c'est-à-dire avec des fonctions virtuelles), c'est la classe réelle

(type dynamique) de l'objet pointé/référencé qui est affichée, sinon c'est sa classe déclarée (type statique).

Ex :

```
#include <typeinfo>

...

Point *p_obj {new PointColor{5,12,100}}; // upcasting

PointColor obj{5, 12, 100};
Point &r_obj = obj; // upcasting


std::cout << "p_obj a le type: " << typeid(p_obj).name() << '\n';
std::cout << "*p_obj a le type: " << typeid(*p_obj).name() << '\n';

std::cout << "obj a le type: " << typeid(obj).name() << '\n';
std::cout << "r_obj a le type: " << typeid(r_obj).name() << '\n';
```

Va afficher :

```
p_obj a le type: P5Point
*p_obj a le type: 10PointColor
obj a le type: 10PointColor
r_obj a le type: 10PointColor
```

Remarque : Le mécanisme RTTI fonctionne généralement à l'aide des vtables : un des « slots » de la vtable (non représenté sur le schéma mémoire du §8 c) est le « type_info » de la classe.

 Le mécanisme RTTI devrait être utilisé uniquement pour afficher des informations (en général en *debug*). Ne pas utiliser RTTI pour déterminer dans quelle branche doit passer le programme (par ex. si l'objet est de classe X faire ceci, si il est de classe Y faire cela,...) : ce style de programmation indique en général un défaut de conception, car le mécanisme des liens dynamiques (fonctions virtuelles) est fait pour ça et trouve automatiquement les fonctions adaptées au contexte.

TP héritage

Reprendre le TP Monde/Animal.

1) Créer deux classes dérivées d'Animal : Lion et Gazelle (par ex.). Dans le peuplement du tableau d'animaux du monde instancier une fois sur deux un Lion, l'autre fois une Gazelle. Afficher chaque animal différemment selon sa classe.

2) Rendre la classe de base Animal abstraite.

Ch 6 : Les *templates* et la librairie standard

Plan du chapitre:

1- Les *templates* (programmation générique)

a) Les *templates* de fonction

b) Les *templates* de classe

2- La librairie standard

a) Les *strings*

b) Les flux d'entrée/sortie (*I/O streams*)

c) Les conteneurs (*containers*) et leurs « itérateurs » (*iterators*)

1- Les conteneurs (*containers*)

2 - Les itérateurs (*iterators*)

3 - Les algorithmes

d) Les « pointeurs intelligents » (*smart pointers*)

1- Les *templates* (programmation générique)

Sujet : créer des **fonctions** « **génériques** », c'est-à-dire des fonctions qui résolvent un problème avec un algorithme identique, quel que soit le type des arguments qui leur sont passés.

Ex : créer une fonction `max(arg1, arg2)` qui renvoie le maximum des 2 valeurs `arg1` et `arg2`, que ceux-ci soient des entiers, des doubles, des chaînes de caractères, etc.

Une solution imparfaite existe en **C**, à l'aide des **macros**. Le problème des macros est qu'elles ne permettent pas au compilateur de vérifier la pertinence des types de données passés, car le préprocesseur fait juste une simple substitution « en aveugle ».

Ex :

```
#define MAX(a,b) ((a)>(b)?(a):(b))  
  
int val = MAX(12,"bonjour"); // n'a aucun sens, mais compile en C ! (pas de vérif de type)
```

Le C++ offre une meilleure solution : les « **templates de fonction** » (*function templates*), ou « patrons de fonction » (on utilise plutôt le terme anglais *template*).

a) Les *templates* de fonction

Un *template* de fonction se comporte comme une fonction, excepté le fait qu'elle peut accepter des paramètres de différents types. En fait un *template* de fonction représente une famille de fonctions. Le prototype est de la forme (mots-clé en gras) :

```
template <typename nom_type> déclaration_fonction;
```

Ex: définition du *template* d'une fonction `max()` :

Signifie : ce qui suit est un patron de fonction paramétrisé par le type T

```
template <typename T> T max (const T a, const T b){  
    return a>b? a : b;  
}
```

Type retour

Type param

Nom fonction

Nom param

Fonction
générique

Ici, le type a été appelé simplement **T** (c'est un nom qui peut être choisi librement).

Utilisation de ce « *template* »

Quand une fonction de ce nom est appelée, le compilateur va générer le code d'une fonction (parmi toutes celles possibles) en remplaçant T par le type déduit des paramètres effectifs (int, double,...). Le type T peut être exprimé explicitement, mais ce n'est pas nécessaire, sauf ambiguïté.

- Pour des entiers :

```
int val = max (12, 5); // ou explicitement max<int>(12, 5);
```

La fonction générée par le compilateur sera :

```
int max (const int a, const int b){  
    return a>b? a : b;  
}
```

- Pour des doubles :

```
double val = max(12.2, 0.5); // ou explicitement max<double>(12.2, 0.5);
```

La fonction générée par le compilateur sera :

```
double max (const double a, const double b){  
    return a>b? a : b;  
}
```

- Pour des chaînes de caractères

```
std::string chaine;  
chaine = max("bonjour", "au revoir"); // ou explicitement max<std::string>  
etc...
```

- Il peut y avoir des ambiguïtés, ex. ici un des paramètres est int, le second double :

```
double val = max(5, 12.7); // Erreur de compil (template argument deduction failed)  
double val = max<double>(5, 12.7); // Paramétrisation explicite du type => OK
```

Ce type de généricité dans l'écriture des fonctions est appelé « **polymorphisme paramétrique** » (*parametric polymorphism*) et, plus généralement, on parle de « programmation générique »

Définition : la programmation générique (*generic programming*) est un style de programmation dans lequel les algorithmes sont écrits en termes de « types à spécifier plus tard », qui seront alors instanciés, selon les besoins, pour des types spécifiques fournis en paramètres.

b) Les templates de classe

Un *template* de classe (*class template*), ou « patron de classe », reprend le même concept que le *template* de fonction. Il s'agit d'une classe générique, paramétrisée par un (ou plusieurs) type(s).

Ex: on veut écrire une classe TabDyn, qui stocke dans un tableau alloué dynamiquement des éléments de type quelconque. Voici, très simplifiée, la définition d'une telle classe :

```
template <typename T> class TabDyn {
    private :
        T *tab;
        int taille;
    public :
        TabDyn(int nb_elem){ // Constructeur à un argument (nb éléments à allouer)
            tab = new T[nb_elem];
            taille = nb_elem;
        }
        ~TabDyn() { // Destructeur (libération)
            delete[] tab;
        }
};

int main(){
    TabDyn<int> t1{10}; // tableau de 10 entiers
    TabDyn<std::string> t2{50}; // tableau de 50 chaînes de caractères
}
```

Contrairement au cas des *templates* de fonctions, il est ici nécessaire de déclarer explicitement le type (comme ci-dessus TabDyn<int> dans main()). Ce n'est plus obligatoire (depuis le C++17) dans le cas où le compilateur peut déduire lui-même le type selon les valeurs fournies à l'initialisation.

Les *templates* de classe sont à la base d'une grande partie de la librairie standard.

2- La librairie standard

La « bibliothèque standard » (appelée le plus souvent « **librairie standard** » par traduction abusive de l'anglais « *standard library* ») est une partie importante de la norme du C++ (environ les 2/3 de la norme). Son but est de faciliter la programmation en rajoutant des composants « pratiques » (classes, fonctions, etc.) au C++ de base.

Classification succincte et non exhaustive de ses composants :

- les « *strings* »,
- le support des expressions régulières,
- les entrées/sorties (« *iostream* »),
- les conteneurs,
- le support du calcul mathématique (classe *complex*, génération de nombres aléatoires,...)
- le support du *multi-threading*,
- les « *smart pointers* ».

Tous les composants de la librairie standard font partie du **namespace std** (cf. Ch.2 §B.10 pour la notion de *namespace*). Si on n'utilise pas l'instruction « `using namespace std;` » il faut préfixer tous les noms provenant de cette librairie (classes, fonctions, constantes,...) par **std::**.

a) Les strings

La classe *string* est faite pour masquer la complexité de la gestion des chaînes de caractères du C (*char**), qui reste cependant toujours possible et qui est de toute façon contenue en arrière-plan dans l'implémentation des *strings*.

En réalité, *string* est un *typedef* pour le *template* `basic_string<char>`. Un objet de type *string* contient donc des caractères de type *char*, c'est-à-dire sur un seul octet.

Intérêt :

- l'allocation/libération mémoire est réalisée automatiquement en fonction du nombre de caractères de la chaîne (pas besoin de `new` ou de `delete`).
- on peut utiliser les opérateurs habituels (affectation, comparaison, addition) de la même façon que pour des types prédéfinis (entiers, etc.) => on n'est pas obligé de passer par les fonctions de type `strcpy`, `strcmp`, etc.
- la classe `basic_string` est dotée de beaucoup de méthodes : pour connaître la taille de la chaîne (`size`), pour la recherche de sous-chaînes ou de caractères (`find`, `rfind`,...), pour l'extraction ou le remplacement de sous-chaînes (`substr`, `replace`), etc.

Rq 1 : une méthode appelée `c_str()` retourne le pointeur (*char**) de début de la chaîne contenue à l'intérieur de l'objet *string* => permet de rentrer dans une fonction n'acceptant qu'une chaîne C classique.

Rq 2 : si on écrit `"xxx"s` on crée une chaîne constante de type *string*, pas de type *char** (depuis C++14). Mais obligation d'ajouter dans le fichier source: `using namespace std::string_literals;`

Ex :

```
std::string chaine1, chaine2, chaine3;
chaine1 = "bonjour";// ou : chaine1 = "bonjour"s; // depuis le C++14
std::cout << chaine1.size(); // 7
std::cout << strlen(chaine1.c_str()); // longueur de la chaîne C contenue : 7
chaine2 = " tout le monde";// ou : chaine1 = " tout le monde"s;
chaine1 += chaine2; // concaténation de chaine2 à la suite de chaine1
chaine3 = chaine1 + chaine2; // concaténation de chaine1 et chaine2 => chaine3
if ( chaine1 > chaine2 ){ // comparaison lexicographique
    chaine2 = chaine1; // affectation
}
int pos = chaine2.find ("tout");// ou : chaine2.find ("tout"s);
std::cout << chaine2.substr(pos) << "\n"; // tout le monde
```

Remarque : l'écriture de chaînes de caractères Unicode (ne tenant pas sur un seul octet) pose des problèmes de portabilité :

- Sous *Linux*, le type *string* peut stocker des chaînes en UTF8 (⚠ dans ce cas le nombre de caractères – méthode `size()` – devient inutilisable).
- Sous *Windows*, on peut utiliser le type *wstring* (*wstring* est un *typedef* pour `basic_string<wchar_t>`), mais alors le code n'est plus portable, car le type *wchar_t* est de 4 octets sous *Linux* et de 2 octets sous *Windows*.

b) Les flux d'entrée/sortie (I/O streams)

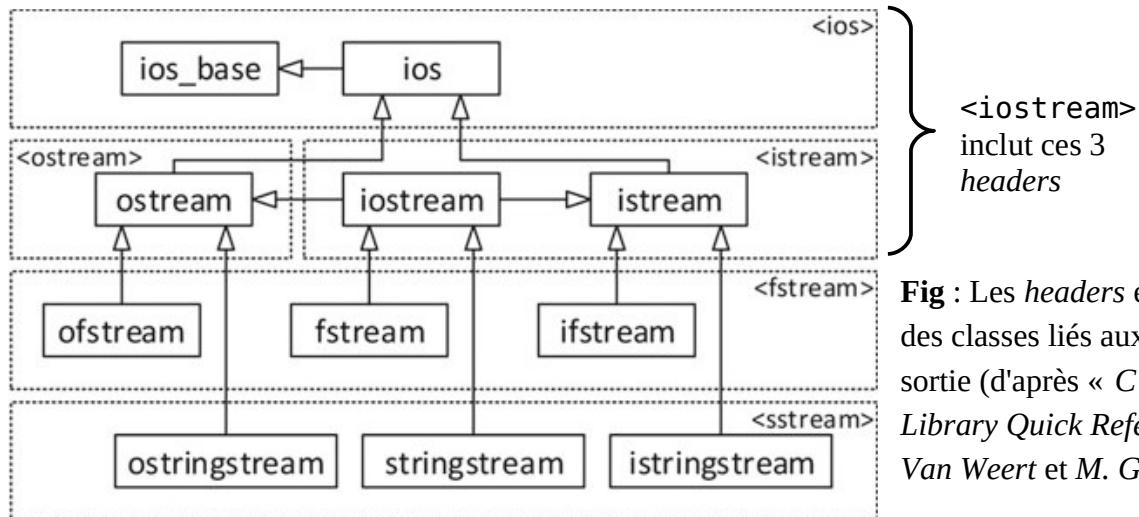


Fig : Les *headers* et la hiérarchie des classes liés aux flux d'entrée/sortie (d'après « C++ Standard Library Quick Reference » de P. Van Weert et M. Gregoire)

En réalité, tous ces noms de classe sont des *typedef* pour des *templates* paramétrisés par des `char` (ex: `istream` est un *typedef* pour `basic_istream<char>`).

Le Ch. 2 §B.6 a présenté rapidement les flux d'entrée/sortie standard (clavier/écran). Ci-dessous, quelques ajouts sur l'état d'un flux d'entrée et le formatage d'un flux de sortie, et une présentation succincte des entrées/sorties fichier.

Etat d'un flux (I/O state)

Un flux a toujours un état (ex: *good*, *bad*, *fail*, *eof*) qu'on peut examiner pour savoir si une opération a réussi ou échoué. On peut notamment lire cet état pour vérifier la conformité des données lues (flux d'entrée standard `cin`) avec ce que le programme attend, et modifier l'état du flux en conséquence. Remarque : `std::cin` est un objet global de classe `istream`.

Ex : on attend une saisie d'entier, et on veut vérifier que ce sont bien des caractères numériques que l'utilisateur a tapé (NB: ci-dessous, pour la clarté du code, on n'a pas écrit les préfixes `std::`)

```
int i;
cin >> i;
if ( cin.fail() ) { // Vrai si cin est dans un état fail ou bad. Équivalent à : if ( !cin )
    cout << "Saisie incorrecte\n";
    cin.clear(); // Remet cin dans un état good
    cin.ignore(numeric_limits<streamsize>::max(), '\n'); // Vide buffer cin ou bien :
    //while (cin.get() != '\n'); // Vide buffer cin ou bien :
    //string line; getline(cin,line); cout << line; // Vide buffer cin
    // et permet d'afficher le contenu erroné
} else {
    cout << "i= "<<i<<'\n';
}
```

Il y a 3 façons de vider le *buffer* d'entrée (les 3 vident tous les caractères restants y compris `\n`):

- en utilisant la méthode prévue à cet effet `cin.ignore()`
- en vidant lettre par lettre (dans un `while`) par la méthode `cin.get()`
- en vidant tout le *buffer* dans une chaîne de type `string` par la méthode `getline()`

Formatage d'un flux de sortie (*output stream formatting*)

La façon la plus simple de contrôler un flux est d'insérer des « manipulateurs » (*manipulators*) dans le flux. Les manipulateurs les plus courants (la plupart sont « rémanents » : ils restent actifs sur le flux tant qu'un autre manipulateur ne donne pas une consigne différente) sont :

- `hex` : pour passer en affichage hexadécimal,
- `dec` : pour passer en affichage décimal (mode par défaut),
- `setw(n)` : (n'est pas rémanent) l'affichage sera sur `n` caractères, en ajoutant des espaces avant,
- `setfill(c)` : complète `setw()` en changeant les espaces par des caractères `c`,
- `setprecision(n)` : l'affichage (entiers ou flottants) sera sur `n` digits (défaut 6), en comptant les digits avant et après le point décimal (sauf si le manipulateur `fixed` est actif, cf. ci-dessous),
- `fixed` : l'affichage (flottants) sera en virgule fixe, le nombre de digits après le point décimal est de 6 par défaut (peut être modifié par le manipulateur `setprecision()`, cf. ci-dessus).

Remarque : pour les manipulateurs prenant un argument (`set...()`), il faut inclure `<iomanip>`, pour les autres `<iostream>` suffit.

Ex (NB: ci-dessous, pour la clarté du code, on n'a pas écrit les préfixes `std::`) :

```
#include <iomanip>
#include <cmath> // pour M_PI
...
cout << "val hex=" << hex << 12 << '\n';
cout << "val dec=" << dec << setw(4) << setfill('0') << 12 << '\n';
cout << "Pi=" << M_PI << '\n';
cout << "Pi (precis 4)=" << setprecision(4) << M_PI << '\n';
cout << "Pi (fixe, precis 4)=" << fixed << M_PI << '\n';
```

va afficher :

```
val hex=c
val dec=0012
Pi=3.14159
Pi (precis 4)=3.142
Pi (fixe, precis 4)=3.1416
```

Lecture de fichier texte

Se fait grâce à la classe standard `ifstream` (*input file stream*). Son constructeur prend en paramètre le nom du fichier (chaîne de caractères de type `string` depuis le C++11, `const char *` auparavant). On peut aussi, au lieu de donner le nom directement au constructeur, utiliser la fonction `open()`. Ex :

```
std::ifstream fentree;
fentree.open(nomfic);
```

Les erreurs d'ouverture (fichier non existant, pas de droits,...) sont gérées ici de façon « traditionnelle », on verra au Ch. 7 un autre mode de gestion des erreurs, par exception.

Ex :

```
#include <fstream> // file stream
const char *nomfic{"test.txt"}; // ou (en C++11) std::string nomfic{"test.txt"};
std::ifstream fentree{nomfic}; // instantiation et ouverture fichier en lecture (if = input file)
```



```

if(!fentree){ // si erreur
    std::cout << "Erreur d'ouverture fichier " << nomfic << "\n";
    exit(1);
}
std::string ligne;
while(std::getline(fentree, ligne)){ //lecture de la ligne
    std::cout << ligne << std::endl;
}
fentree.close(); // fermeture (pas strictement nécessaire : automatique quand fentree est détruit)

```

Écriture de fichier texte

Se fait grâce à la classe standard `ofstream` (*output file stream*).

Ex. (suite du code précédent) :

```

std::ofstream fsortie{nomfic}; // instantiation et ouverture fichier en écriture (of = output file)
if(!fsortie){ // erreur
    std::cout << "Erreur d'ouverture fichier " << nomfic << "\n";
    exit(1);
}
std::string texte{"bonjour"};
fsortie << texte << std::endl; //écriture d'une ligne dans le fichier
fsortie.close(); // fermeture fichier (même remarque que pour fichier de lecture)

```

Lecture/écriture de fichier binaire

Les mêmes classes peuvent être utilisées, mais les fonctions de lecture/écriture sont `read()/write()`. Ils fonctionnent globalement de la même façon que les fonction `fread()` et `fwrite()` du C. Ex :

```

int tab[5]{1,3,5,7,9};
fsortie.write ( reinterpret_cast<char*>(tab), 5*sizeof(int) );

```

c) Les conteneurs (*containers*) et leurs « itérateurs » (*iterators*)

Cette partie de la librairie standard est communément appelée « STL » (*Standard Template Library*), car cette partie dérive d'une librairie plus ancienne de ce nom. Elle fournit :

- des *templates* de classe pour des structures de données abstraites (tableau, liste, tableau associatif, file, etc...) pouvant stocker tout type de données (et les méthodes primitives pour chacune),
- des objets (similaires aux pointeurs) pour se déplacer dans les conteneurs : les « itérateurs »,
- des algorithmes génériques pour rechercher, trier, remplacer, etc. des éléments de ces conteneurs.

1. Les conteneurs (*containers*)

On distingue globalement deux catégories : les **conteneurs séquentiels**, où on accède aux éléments de manière séquentielle, et les **conteneurs associatifs**, qui permettent une recherche rapide des éléments par leur clé, ces clés étant soit triées (conteneurs « ordonnés ») soit hachées (conteneurs « non ordonnés »).

Catégorie de conteneur	Nom du <i>template</i> de classe	Description	Catégorie d'itérateur
Conteneurs séquentiels	<code>vector</code>	tableau dynamique contigu	<i>random</i>
	<code>deque</code>	file à double entrée (<i>double ended queue</i>)	<i>random</i>
	<code>list</code>	liste doublement chaînée	<i>bidir</i>
	<code>array</code> (depuis C++11)	tableau de taille fixe contigu	<i>random</i>
	<code>forward_list</code> (depuis C++11)	liste simplement chaînée	<i>forward</i>
Conteneurs associatifs	<code>set</code>	collection de clés uniques, triées par les clés	<i>bidir</i>
	<code>map</code>	collection de paires clé-valeur, triées par les clés (uniques) = tableau associatif	<i>bidir</i>
	<code>multiset</code>	collection de clés, triées par les clés	<i>bidir</i>
	<code>multimap</code>	collection de paires clé-valeur, triées par les clés	<i>bidir</i>
Conteneurs associatifs non-ordonnés (hachés) (depuis C++11)	<code>unordered_set</code>	collection de clés uniques, hachées par les clés	<i>forward</i>
	<code>unordered_map</code>	collection de paires clé-valeur, hachées par les clés (uniques) = table de hachage	<i>forward</i>
	<code>unordered_multiset</code>	collection de clés, hachées par les clés	<i>forward</i>
	<code>unordered_multimap</code>	collection de paires clé-valeur, hachées par les clés	<i>forward</i>

De plus, il existe 3 autres conteneurs, qui ne sont que des adaptateurs basés sur des conteneurs présentés ci-dessus (en général, il sont implémentés en tant que sur-couche du conteneur `deque` ou `vector`) :

- `stack` : pile (structure LIFO),
- `queue` : file (structure FIFO),
- `priority_queue` : file à priorité.

Dès qu'on veut utiliser un type de conteneur, il faut inclure un *header* du même nom, ex `<vector>`, `<list>`, etc.

- **Exemple de conteneur séquentiel : le conteneur « vecteur » (*vector*)**

C'est le conteneur de la STL le plus utilisé. Il permet de remplacer avantageusement les tableaux alloués dynamiquement (par `new`). Comparaison du code entre version avec/sans vecteur :

```
// Façon « traditionnelle » (sans vector)
int *tab {new int[20]}; // allocation mémoire explicite
// Ajout de valeurs
int i = 0;
tab[i++] = 12;
tab[i++] = 5;
int nb_valeurs = i;
// Lire (impossible avec range for)
for( i=0 ; i < nb_valeurs ; ++i){
    std::cout << tab[i] << "\n";
}
delete[] tab; // libération mémoire explicite
```

```

#include <vector>
vector<int> tab; // gestion allocation/réallocation mémoire implicite => size 0
// ou : vector<int> tab(20); // crée 20 entiers initialisés à leur valeur par défaut (0) => size 20
// tab.reserve(10); // pré-alloue de la place (ici 10 entiers) pour éviter les réallocs implicites => size 0
// Ajout d'une valeur : avec push_back, qui est une méthode de la classe vector
tab.push_back(12); // => size 1
tab.push_back(5); // => size 2
// Lire
for (int i=0; i < tab.size(); ++i){ // size est une méthode commune à tous les conteneurs
    std::cout << tab[i] << "\n";
}
// ou avec range for :
for(auto val : tab){ // ou : for(const auto &val : tab)
    std::cout << val << "\n";
}

```

Remarque : initialisation d'un « vecteur » : un vecteur peut être initialisé à la déclaration,

1. soit en listant les éléments « en dur », comme pour un tableau de taille fixe (*list initialization*),
2. soit en lui fixant une taille et une valeur initiale identique pour tous les éléments (par défaut 0).

Cela pose un problème au compilateur : en effet {10,5} signifie-t-il 10 éléments valant 5 ou deux éléments valant 10 et 5 ? La librairie standard a été obligée de faire un choix, qui est de réserver les accolades à l'initialisation par liste (1) et de mettre des parenthèses pour une initialisation par taille (2).

```

vector<int> tab{1,3,5,7,9}; // initialisation par liste (ou: vector<int> tab={1,3,5,7,9};)
for(auto val : tab) { std::cout << val << " "; } //Affiche 1 3 5 7 9

vector<int> tab2(5,3); // initialisation par taille et valeur
for(auto val : tab) { std::cout << val << " "; } //Affiche 3 3 3 3 3

vector<int> tab3(5); // initialisation par taille et valeur par défaut (0)
for(auto val : tab) { std::cout << val << " "; } //Affiche 0 0 0 0 0

```

- **Exemple de conteneur associatif : « unordered_map »** (tableau associatif par table de hachage)

unordered_map stocke des couples <clé-valeur> uniques, avec un temps de recherche d'une clé très rapide. Par contre, l'opération d'affichage de tous les couples triés par clé est moins rapide que pour un conteneur map. Ex : un répertoire téléphonique.

```

#include <unordered_map>
unordered_map<string,int> repTel; // clé: chaîne de caractères, valeur : entier
//Ajout de paires clé-valeur avec opérateur [ ] (si la clé existe déjà, la valeur est écrasée par la nouvelle)
repTel["Paul"] = 298010203;
repTel["Virginie"] = 177102030;

```

```

//Ajout de paire clé-valeur (type std::pair<K,V>) avec insert (si la clé existe déjà rien n'est inséré)
repTel.insert( {"Kevin",644112233} );

// Obtenir le numéro de téléphone d'une personne (aussi avec l'opérateur [])
int num = repTel["Paul"];

// Affichage (en ajoutant le 0 devant)
cout << "Paul: " << "0"+to_string(num) << "\n";

//Problème de l'opérateur []: si la clé n'existe pas dans le conteneur, elle va être créée (avec une valeur 0)
num = repTel["Pierre"]; // la clé « Pierre » est ajoutée, son numéro est 0

// Pour éviter ce problème, on teste d'abord si la clé existe, avant d'essayer d'obtenir la valeur
if ( repTel.count("Pierre") ){ // la méthode count retourne 1 si la clé existe, 0 sinon
    num = repTel["Pierre"];
}

// On peut aussi utiliser la méthode find, qui retourne un itérateur (cf. § suivant), mais ne crée pas la clé

```

2- Les itérateurs (*iterators*)

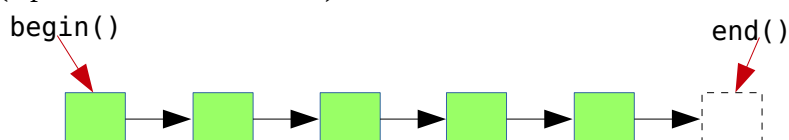
Définition : un itérateur (du verbe *itérer*: faire une seconde fois, une troisième, etc.) est une abstraction du concept de pointeur. L'itérateur fournit un moyen simple et générique de parcourir tout type de conteneur de la même façon qu'un pointeur permet de parcourir un tableau classique.

Un itérateur obéit au minimum à **4 règles** : (dans l'ex. suivant : variable `it` de type itérateur)

- `*it` est la valeur de l'élément pointé,
- `++it` (qui est préférable à `it++`) permet de passer à l'élément suivant,
- si l'élément pointé est une *struct* ou une *class*, `it->m` donne la valeur du membre `m`,
- on peut **comparer 2 itérateurs** entre eux par les opérateurs `!=` et `==`.

Remarque 1 : il y a autant de types d'itérateurs que de types de conteneurs. Par exemple, une variable `it` itérant sur un vecteur de strings sera déclarée `std::vector<std::string>::iterator it`; Cette complexité dans la déclaration conduit souvent à utiliser la déclaration `auto` pour les itérateurs.

Remarque 2 : tous les conteneurs de la STL fournissent les méthodes `begin()` et `end()`, qui renvoient un itérateur vers, respectivement, le premier élément du conteneur et « un élément après la fin » (« *past-the-end iterator* ») du conteneur. Si le conteneur est vide, `begin() == end()`.

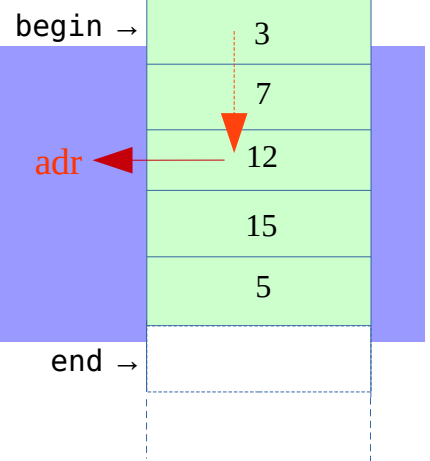


Remarque 3 : les itérateurs sont classés en 3 catégories : à accès aléatoire (*random-access*), bidirectionnels (*bidirectional*), « vers l'avant » (*forward*). Ces catégories sont inclusives, c'est-à-dire qu'un itérateur « *random-access* » est automatiquement bidirectionnel, un bidirectionnel est automatiquement « *forward* ». Cf. le tableau §1 ci-dessus pour voir la catégorie selon le type de conteneur. Les 4 règles ci-dessus s'appliquent aux itérateurs « *forward* », les bidirectionnels acceptent en plus l'opérateur `--`, et sur les « *random-access* » on peut appliquer des décalages quelconques.

Implémentation d'un itérateur : explication de principe : un pointeur est un itérateur

Ex : on écrit une fonction `find()` qui cherche une valeur *value* (ex: 12) entre 2 adresses (paramètres *begin* et *end*), et qui retourne l'adresse de la valeur trouvée :

```
int * find( int *begin, int *end, int value){
    while (begin != end && *begin != value){
        ++begin;
    }
    return begin;
}
```



Cette fonction est utilisable uniquement sur des tableaux d'entiers, mais montre le principe des itérateurs (ici des simples pointeurs *begin* et *end*).

Ci-dessous, en utilisant les itérateurs et un conteneur « vecteur » de la STL :

```
#include <vector>
vector<int> tab {3, 7, 12, 15, 5};
//Cherche la première occurrence de la valeur 12 dans tab
vector<int>::iterator it; // type itérateur / vecteur d'entiers
it = find ( tab.begin(), tab.end(), 12 );
```

La fonction `find()` (algorithme standard de la STL, cf. §3) est générique, ex. ci-dessous : elle fonctionne aussi pour une liste de caractères.

```
#include <list>
list<char> liste {'m', 'a', 'n', 'x', 'y'};
//Cherche la première occurrence de la valeur 'x' dans liste
list<char>::iterator it2; // type itérateur /liste de caractères
it2 = find ( liste.begin(), liste.end(), 'x' );
```

Parcours de conteneur (*container traversal*) à l'aide d'un itérateur

Tous les conteneurs de la STL peuvent être parcourus soit par un « *range for* » (cf. §1 pour un vecteur), soit à l'aide d'un algorithme de type `for_each` (cf. plus loin §3), soit dans une boucle « *for* » sur un itérateur. Ex :

```
for (auto it = cont.begin(); it != cont.end(); ++it ) { // cont est un conteneur qq
    cout << *it << "\n"; // Rq : *it n'est affichable directement que si c'est un type simple
}
```




Si le conteneur doit être remanié en le parcourant (suppression ou insertion d'un ou plusieurs éléments), ce type de boucle (ni le « *range for* » non plus) n'est pas adapté, car l'itérateur devient invalide suite à la suppression (ou insertion). L'intervention de certains algorithmes de la STL (ex: `remove()`, ou `remove_if()`, cf. §3 ci-dessous) va faciliter ce type de modification.

3- Les algorithmes de la STL (*algorithms*)

Définition : un algorithme (dans le cadre de la STL) est un *template* de fonction qui opère sur des séquences d'éléments, le plus souvent tout ou partie des éléments d'un conteneur. Ils ne fonctionnent qu'avec des paramètres de type « itérateur » et sont totalement indépendants du type de conteneur (en fait rien ne leur indique de quel type de conteneur proviennent les éléments qu'ils traitent).

Chaque fois qu'on utilise un algorithme standard, il faut inclure le *header* `<algorithm>`.

 Certains algorithmes de la STL modifient les valeurs des éléments qui leur sont passés, mais les algorithmes ne peuvent ni supprimer ni insérer d'éléments du fait qu'ils n'ont pas connaissance de la structure sous-jacente. Les fonctions comme `remove()` ne suppriment pas d'éléments du conteneur, elles ne font que les déplacer à la fin.

Il existe **plus d'une centaine d'algorithmes** standards, des plus simples (ex: `std::max()`, qui renvoie le max de deux valeurs) aux plus complexes (ex: `std::transform_reduce()`, qui exécute un algorithme parallèle de type « *map-reduce* »).

La majorité des algorithmes prend en paramètres d'abord un itérateur *begin*, puis un itérateur *end* et, si besoin, des paramètres supplémentaires.

Quelques exemples importants :

- **find()** : montré dans le §2 ci-dessus, il renvoie un itérateur sur la première occurrence d'une valeur : `p = find(b, e, x);` // `p` est le premier itérateur entre `b` et `e` tel que `*p == x`
- **find_if()** : cherche selon une condition (définie par une fonction) : il renvoie un itérateur sur la première occurrence qui valide la condition : `p = find(b, e, f);` // `f` est une fonction, `p` est le premier itérateur entre `b` et `e` tel que `f(*p) == true`

```
bool est_pair ( const int val ){
    return val%2 == 0;
}
vector<int> tab {3, 7, 12, 15, 5};
auto it = find_if ( tab.begin(), tab.end(), est_pair );
// ou, avec fonction lambda (cf. p.82): auto it= std::find_if ( tab.begin(), tab.end(),
    [](auto val){return val%2 == 0;} );
std::cout << *it; // affiche 12
```

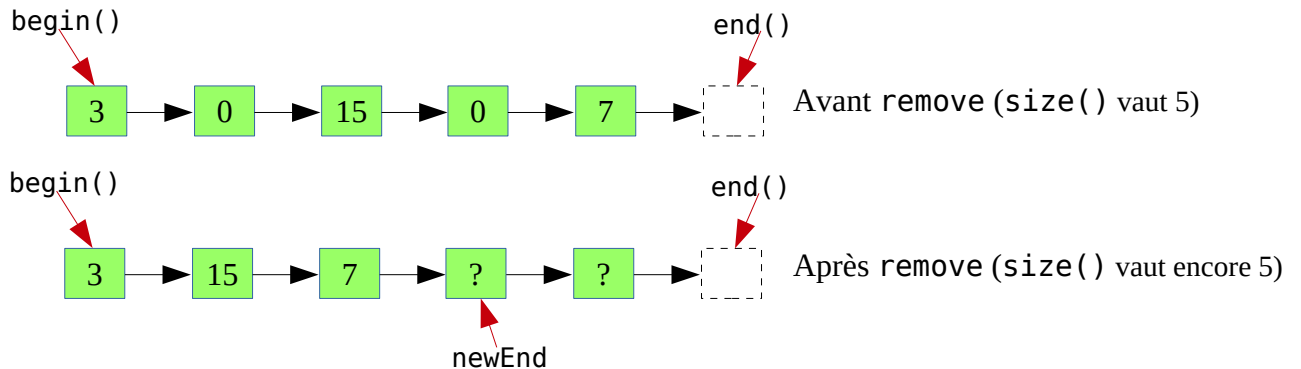
- **for_each()** : algorithme « à tout faire » : il exécute une fonction sur chaque élément de la séquence : `for_each(b, e, f);` // applique `f()` sur chaque itérateur (déréférencé) entre `b` et `e`
Ex : on ajoute 10 à tous les éléments du vecteur `tab`

```
void add10 ( int &val ){ val += 10; }
vector<int> tab {3, 7, 12, 15, 5};
for_each(tab.begin(), tab.end(), add10); // tab vaut maintenant {13, 17, 22, 25, 15}
```

- **sort()** : trie les éléments : soit `sort(b, e);` // trie entre `b` et `e` (utilise la comparaison `<`) soit `sort(b, e, f);` // trie entre `b` et `e` (utilise la fonction de comparaison `f`)

- **remove()** : « enlève » de la séquence les éléments correspondant à une condition (en fait déplace ces éléments vers la fin), et renvoie un itérateur sur la fin de la séquence « restante ». Ex :

```
vector<int> tab {3, 0, 15, 0, 7};
// « enlève » les éléments de valeur 0 dans tab (retourne un itérateur sur la « nouvelle fin »)
auto newEnd = remove ( tab.begin(), tab.end(), 0 );
```



Remarque 1 : pour supprimer vraiment les éléments déplacés, il faut ensuite appliquer la méthode **erase()** du conteneur concerné, à partir de l'itérateur retourné par **remove()**, jusqu'à la fin. On appelle cette technique « *erase remove idiom* ». Ex (suite du code précédent) :

```
tab.erase ( newEnd, tab.end() ); // Après erase(), size() vaut maintenant 3
// ou bien, en regroupant les deux instructions remove() puis erase(),
// sans itérateur intermédiaire :
tab.erase ( remove(tab.begin(), tab.end(), 0), tab.end() );
```

Remarque 2 : si on veut enlever de la séquence les éléments répondant à une condition plus complexe que l'égalité, il faut, au lieu de **remove()**, utiliser **remove_if()** qui prend en 3^{ème} paramètre une fonction retournant un booléen. Ex., on veut enlever toutes les valeurs négatives :

```
bool est_negatif ( const int val ){
    return val < 0;
}
tab.erase ( remove_if(tab.begin(), tab.end(), est_negatif), tab.end());
```

Remarque 3 : la norme **C++20** a ajouté les algorithmes **erase()** et **erase_if()** génériques qui fonctionnent sur (presque) tout type de conteneur. Ces algorithmes évitent de recourir au « *erase remove idiom* ». Ex (réécriture du code précédent) :

```
bool est_negatif ( const int val ){
    return val < 0;
}
erase_if ( tab, est_negatif);
```

Résumé sur les conteneurs

On peut trouver à l'adresse <https://en.cppreference.com/w/cpp/container> :

- un tableau récapitulatif des méthodes disponibles pour tous les types de conteneurs,
- un tableau synthétique de la validité des itérateurs après les opérations d'insertion ou de suppression d'éléments, selon le type de conteneur.

d) Les « pointeurs intelligents » (*smart pointers*)

Les « **smart pointers** » sont des *templates* de classe qui sont des « emballages » (*wrappers*) autour des pointeurs classiques (*raw pointers*). L'apport principal par rapport à un pointeur classique est la libération automatique de mémoire. Quand un « *smart pointer* » est détruit (par ex. quand il sort de la portée), l'objet qu'il pointe est automatiquement libéré (si aucun autre pointeur ne pointe dessus).

Notion de propriété (*ownership*) : une entité qui a la charge de la libération d'une ressource est « propriétaire » (*owner*) de cette ressource. En particulier, une portion de code (fonction, classe, etc.) qui a la charge de la libération (par `delete`) d'un objet alloué (dans le tas) par `new` est propriétaire de cet objet. Le propriétaire détient une variable de type pointeur (ou qui contient un champ de type pointeur) qui pointe vers cet objet. La « propriété » peut être transférée d'une portion de code à une autre (typiquement lors d'un appel de fonction).

Les pointeurs classiques (« bruts ») gèrent mal la propriété : plusieurs pointeurs pouvant pointer sur le même objet, il n'est pas facile de savoir qui a la charge de la libération, ni quand on peut autoriser cette libération. Les « *smart pointers* » ont été conçus pour gérer de façon claire la propriété. Il existe (depuis le C++11) deux types de « *smart pointers* » (définis dans le header `<memory>`) :

- **unique_ptr** : il est l'unique propriétaire de l'objet vers lequel il détient le pointeur. L'objet possédé sera libéré automatiquement quand le `unique_ptr` lui-même sera détruit.
Caractéristiques :
 - **on ne peut pas copier un `unique_ptr` ou faire d'affectation sur un `unique_ptr`,**
 - il est « *exception safe* » (cf. Ch. 7),
 - il permet de passer à une fonction la propriété d'un objet alloué dynamiquement,
 - il permet de retourner la propriété d'un objet alloué dynamiquement depuis une fonction,
 - on peut stocker des `unique_ptr` dans des conteneurs.
- **shared_ptr** : il est utilisé pour représenter une « propriété partagée », par ex. quand deux portions de code ont besoin de l'accès à un objet (contenant une donnée) mais aucun n'a la propriété exclusive (dans le sens de la responsabilité de la libération de l'objet). Le `shared_ptr` compte les références vers l'objet et le libère quand le compteur vaut 0.
- il existe aussi le type `auto_ptr`, qui était le « *smart pointer* » de la norme C++98. Il est obsolète (*deprecated*) depuis le C++17, mais on peut encore le trouver dans du code ancien et peut en général être remplacé par `unique_ptr`.

Les guides de bonnes pratiques **recommandent d'utiliser le type `unique_ptr`** à la place des pointeurs classiques. Le type `shared_ptr` doit être réservé à des situations particulières plus rares.

Remarque 1 : on crée (instancie) généralement un `unique_ptr` à partir d'un pointeur classique, en le passant en paramètre au constructeur. Ex :

```
Point *rawptr {new Point};
unique_ptr<Point> uptr{rawptr}; // Initialisation « directe » (direct initialization)
    // ou bien, sans variable intermédiaire :
unique_ptr<Point> uptr{new Point}; // Aussi initialisation « directe »
// par contre, l'initialisation « par copie » (copy initialization), c'est-à-dire avec =, est interdite :
unique_ptr<Point> uptr = rawptr; // Erreur compilation
```


Remarque 2 : syntaxiquement un `unique_ptr` se comporte en grande partie comme un pointeur classique. Ex : si `uptr` est un `unique_ptr` :

- on peut le déréférencer en écrivant : `*uptr`,
- si l'élément pointé est une *struct* ou une *class*, `uptr->m` donne la valeur du membre `m`,
- on peut tester si le « *smart pointer* » est « propriétaire » d'un objet en écrivant : `if (uptr)` qui est équivalent à : `if (uptr != nullptr)`

Par contre, on ne peut pas faire d'incrémentation ou de décrémentation d'un « *smart pointer* ».

Remarque 3 : on peut lire la valeur du pointeur classique encapsulé dans un `unique_ptr` à l'aide de la méthode `get()`. Ex :

```
std::unique_ptr<Point> uptr{new Point};  
Point* ptr = uptr.get(); // Accès direct au Point possible par le pointeur ptr
```



Après `get()`, c'est toujours le « *smart pointer* » `uptr` qui est propriétaire du `Point` alloué, mais on a un second accès au `Point` à l'aide du pointeur classique `ptr` => danger, car on pourrait faire ensuite `delete ptr`; => catastrophe ! Il faut laisser la libération au propriétaire.

Remarque 4 : on peut détruire l'objet pointé à l'aide de la méthode `reset()`, ou en affectant directement `nullptr`. Cependant c'est rarement nécessaire : la destruction est automatique en sortant du bloc. Ex :

```
std::unique_ptr<Point> uptr{new Point};  
uptr.reset(); // la mémoire allouée pour le Point est libérée (appel implicite de delete)  
// ou : uptr = nullptr; // c'est la seule affectation autorisée sur un unique_ptr
```

Comparaison *raw pointer*/*smart pointer* : exemple de principe

Ex : on écrit une fonction `f()` qui a besoin d'allouer dynamiquement un `Point`, de faire des opérations dessus (ex: appel d'une fonction de prototype `int g(Point *)`), puis de le libérer. En utilisant un pointeur « brut » (*raw*) :

```
void f(){  
    Point *pt {new Point{12, 5}}; // Allocation d'un Point dans le tas  
    pt->affiche();  
    int result = g(pt); // g() modifie le Point (mais pourrait aussi le détruire!)  
    if ( !result ){ return; } // Problème : ici on a oublié de libérer le Point alloué !  
    pt->affiche();  
    delete pt; // Libération explicite du Point alloué  
}
```

Même code, mais en utilisant un `unique_ptr` :

```
#include <memory>  
void f(){  
    std::unique_ptr<Point> pt{new Point{12, 5}}; // Allocation d'un Point dans le tas  
    pt->affiche();  
    int result = g(pt.get()); // g() modifie le Point (mais pourrait aussi le détruire!b cf. )  
    if ( !result ){ return; } // Ici libération automatique du Point alloué  
    pt->affiche();  
    // ici, pas de delete, car libération automatique du Point alloué  
}
```

Problème potentiel : il est dû au passage d'un pointeur brut à la fonction `g()` : le `unique_ptr` de la fonction `f()` perd la propriété exclusive du Point alloué. En effet, `g()` obtient un accès direct à la ressource par le pointeur, et pourrait faire un `delete` (\Rightarrow plantage quand le `unique_ptr` voudra faire la libération, à la fin de `f()`).

Solution : si c'est possible, modifier la fonction `g()` pour qu'elle prenne un `unique_ptr` en paramètre. Pour ce faire, il faut bien comprendre la notion de transfert de propriété (ou pas) par appel de fonction :

Passage de paramètre de type `unique_ptr` à une fonction

Rappel : on ne peut pas copier un `unique_ptr`. Par contre on peut lui appliquer un « déplacement », cf. encadré p. suivante sur la « sémantique de déplacement » (ou de « mouvement ») (*move semantics*). Deux cas de figure :

- on veut passer à la fonction un `unique_ptr` pointant sur un objet qui peut être lu et/ou modifié, mais qui ne sera ni détruit ni réalloué en mémoire : on le passe par référence constante (`const unique_ptr<...> &`). L'objet reste la propriété de la fonction appelante.
- on veut passer à la fonction un `unique_ptr` pointant sur un objet qui peut être lu, modifié, mais aussi détruit ou réalloué en mémoire : on doit le passer par « déplacement » (à l'aide de `std::move()`). La propriété exclusive de l'objet est transférée à la fonction appelée.

1) Passage par référence constante

```
void g ( const unique_ptr<Point> & uptr ) { // Passage par référence const
    uptr->setX(8); // OK on peut modifier l'objet Point pointé par le unique_ptr
    uptr->setY(7); //
    uptr.reset(); // mais on ne peut pas le détruire : g() n'est pas propriétaire ( $\Rightarrow$  erreur compil)
}
void f(){
    unique_ptr<Point> pt{new Point{12, 5}}; // Allocation d'un Point dans le tas
    int result = g(pt); // OK, passage par référence du unique_ptr et modif du Point
    pt->affiche(); // OK, x vaut 8 et y vaut 7
}
```

2) Passage par « *move* »

```
void g ( unique_ptr<Point> uptr ) { // Passage par valeur
    ... // Code identique ci-dessus, mais g() est propriétaire du Point
    // le Point sera détruit (explicitement, cf. reset ci-dessus, ou implicitement en fin de fonction)
}
void f(){
    unique_ptr<Point> pt{new Point{12, 5}}; // Allocation d'un Point dans le tas
    //int result = g(pt); // Erreur compil car on ne peut pas copier un unique_ptr
    int result = g(std::move(pt)); // Passage par move du unique_ptr (transfert propriété)
    pt->affiche(); // NON! f() a perdu la propriété du Point et ne peut plus du tout accéder au
    // Point (il y a toujours destruction dans g())  $\Rightarrow$  Erreur de segmentation
}
```

Une fonction qui prend un paramètre de type `unique_ptr` par *move* devient propriétaire exclusif de l'objet pointé => il ne peut plus être utilisé par l'appelant (le pointeur brut sous-jacent, une fois transféré au paramètre de la fonction, est mis à NULL). Une fonction à laquelle on passe une ressource de cette façon est appelée un « **puits** » (*sink*), car la ressource passée est perdue pour l'appelant. La propriété est transférée « en descendant » les appels, elle ne peut pas « remonter » (sauf par `return`, cf. ci-dessous).

Retour d'une valeur de type `unique_ptr` depuis une fonction

Une fonction qui retourne un paramètre de type `unique_ptr` transfère **toujours** la propriété exclusive de l'objet pointé à la fonction appelante. Une fonction qui renvoie une ressource de cette façon est appelée une « **source** » (*source*), car la ressource « sort » vers l'appelant. La propriété est transférée par `return` « en remontant » les appels. L'implémentation réelle (*move* ou RVO) dépend du compilateur.

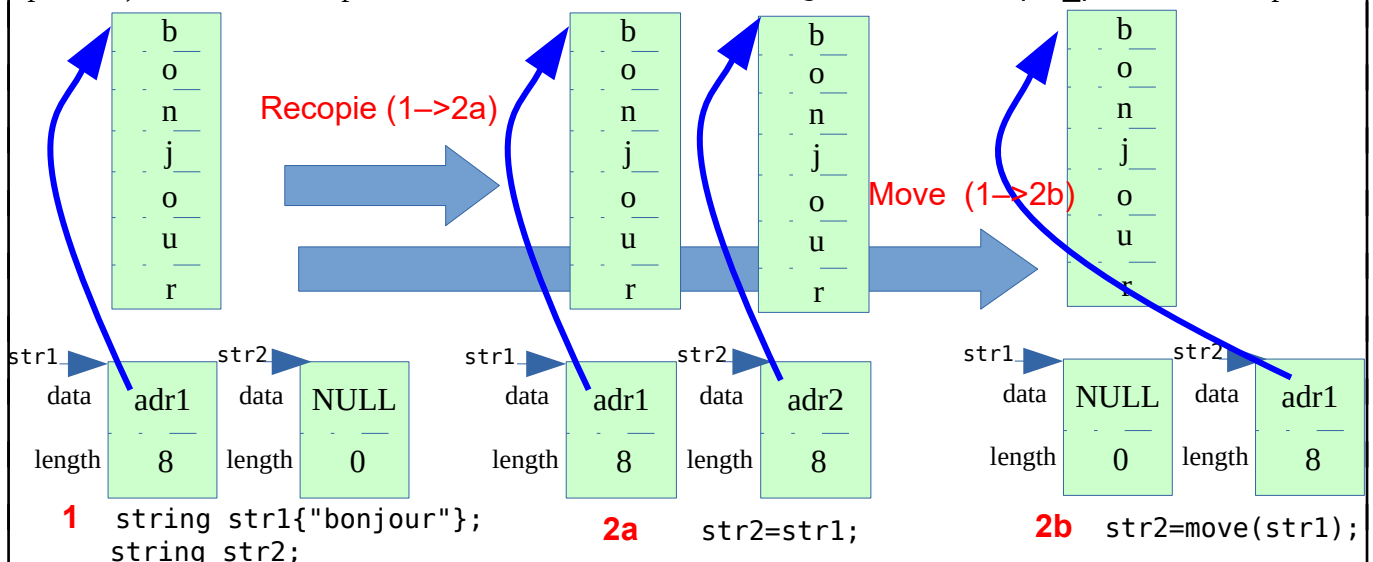
```
unique_ptr<Point> g ( void ){// Retour par valeur : converti en move ou optimisé par
                               // le compilateur : RVO (Return Value Optimization)

    unique_ptr<Point> uptr{new Point{12, 5}};// Allocation d'un Point dans le tas
    ...
    return uptr;// Retour sans recopie (par move ou NRVO) du unique_ptr (transfert propriété)
}

void f(){
    unique_ptr<Point> pt;// Le pointeur brut sous-jacent est initialisé à NULL par défaut
    pt = g();// Retour du unique_ptr venant de g() (transfert propriété)
    pt->affiche();// OK, x vaut 12 et y vaut 5
    // Le Point est détruit ici implicitement (f() en est propriétaire)
}
```

La sémantique de déplacement (*move semantics*) : elle permet à un objet, sous certaines conditions, de se faire transférer la propriété des ressources d'un autre objet. Son principal but est d'éviter les coûteuses opérations de recopie : les opérations de *move* sont plus rapides car elles transfèrent une ressource existante vers un nouveau propriétaire, tandis que la recopie requiert d'allouer une nouvelle ressource pour réceptionner les données copiées. Le « *move* » n'existe que depuis le C++11.

Pour que les opérations de *move* soient autorisées pour une classe, il faut qu'elle possède un constructeur par déplacement (*move constructor*) et un opérateur d'affectation par déplacement (*move assignment operator*). C'est le cas de plusieurs classes de la STL : `string`, `vector`, `unique_ptr`,... Ex. simple :



Remarque : un `unique_ptr` peut, comme un pointeur classique, pointer vers un tableau alloué dynamiquement. La syntaxe est un peu différente. Ex. avec l'allocation d'un tableau de 100 entiers :

```
std::unique_ptr<int[]> tab{new int[100]};  
tab[0] = 5; // Accès à un élément du tableau (ici celui d'indice 0)
```

⚠ Pour créer un tableau alloué dynamiquement il est cependant préférable d'utiliser un objet `vector`, il est fait pour ça !

Fonctions de création de *smart pointers*

Depuis le C++14, il existe des fonctions (en fait des *templates* de fonctions) quiinstancient des `unique_ptr` et des `shared_ptr` directement, sans devoir passer par un `new()` : les fonctions **`make_unique()`** et **`make_shared()`**. Les guides de bonnes pratique préconisent de les utiliser, sous le conseil plus général d'éviter au maximum d'utiliser `new()`. Ex. avec `make_unique()` :

```
// Allocation d'un Point / smart pointer, classique en passant par new() et appel du constructeur à 2 args:  
std::unique_ptr<Point> uptr{new Point{12, 5}}; // Pas de possibilité de déclaration auto  
  
// Allocation d'un Point / smart pointer, en passant par make_unique() auquel on passe directement les  
// arguments du constructeur:  
std::unique_ptr<Point> uptr = std::make_unique<Point>(12, 5);  
  
// ou, plus léger (déclaration auto):  
auto uptr = std::make_unique<Point>(12, 5);  
  
// Création d'un tableau dynamique, cf. remarque en haut de cette page (ex. tableau de 100 int):  
auto tab = std::make_unique<int[]>(100);
```

Exercice :

1) transformer le code suivant de telle sorte qu'il n'y ait plus ni pointeurs bruts, ni `new()` (utiliser un vecteur et des *smart pointers*)

```
#include "Animal.h"  
  
int main(){  
    int nb_animaux = 100;  
    Animal **tab_animal {new Animal *[nb_animaux]}; // Tableau dynamique de  
                                                    // pointeurs/Animal  
  
    //Peuplement  
    for( int i = 0; i < nb_animaux ; ++i){  
        tab_animal[i] = new Animal;  
    }  
  
    //Affichage  
    for( int i = 0; i < nb_animaux ; ++i){  
        tab_animal[i]->affiche();  
    }  
}
```

2) à la fin du `main()`, ajouter du code pour détruire un `Animal` sur 2 (faire en sorte que la taille du vecteur soit effectivement 50 et que la mémoire soit libérée correctement).

Ch 7 : Gestion des exceptions

Plan du chapitre:

1) La gestion des exceptions

a) La gestion « traditionnelle » des anomalies

b) La gestion des exceptions – Généralités

c) Fonctionnement du « catch »

d) Exemple : traitement des erreurs liées à une base de données

2) Notion de RAI

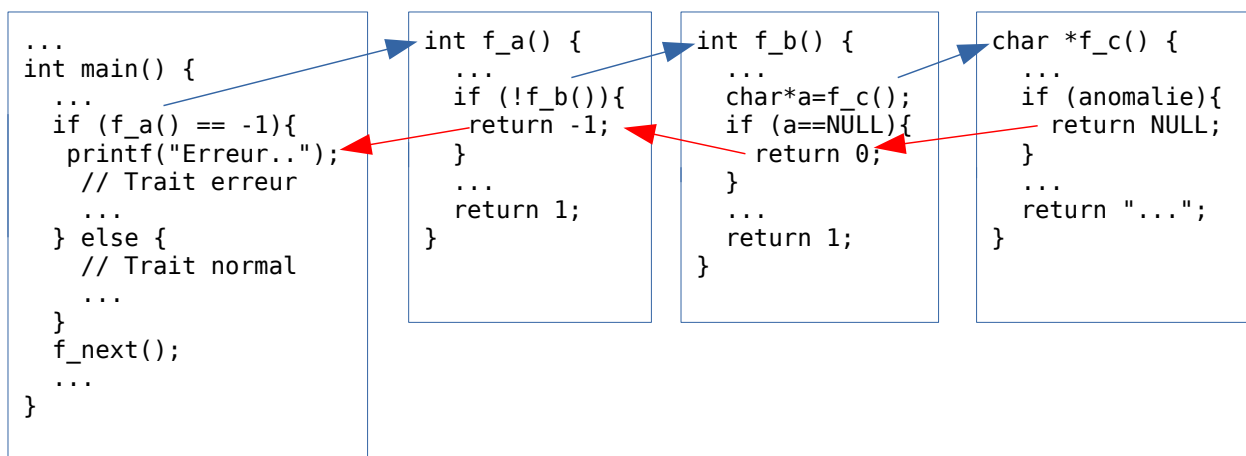
1- La gestion des exceptions

a) La gestion « traditionnelle » des anomalies

En C (ou en C++), cette gestion « traditionnelle » peut se faire de deux façons. Si le programme détecte une anomalie qu'il ne peut traiter « localement » (dans la fonction où elle apparaît), il peut :

- **s'arrêter** : appel à `exit()` ou `abort()`. La fonction `exit()` prend un paramètre entier (=code de retour du programme) et ferme « proprement » le programme. `abort()` arrête le programme brutalement. Cette stratégie doit être réservée à des situations totalement irrécupérables.
- faire **retourner une valeur d'erreur** à la fonction qui a détecté l'anomalie. Stratégie très fréquente en C, mais inconvénients :
 - 1) il faut tester systématiquement tous les retours de fonctions si on veut être sûr de bien traiter toutes les anomalies. Ceci peut facilement doubler la taille d'un programme.
 - 2) il faut « remonter » la chaîne d'appel de fonctions (c'est-à-dire dépiler la pile d'exécution – *call stack* – fonction par fonction) jusqu'à arriver au bon niveau pour traiter l'anomalie.

Ex. : `main()` appelle `f_a()` qui appelle `f_b()` qui appelle `f_c()`. Une anomalie est détectée dans `f_c()` et celle-ci ne peut être traitée qu'au niveau du `main()`.



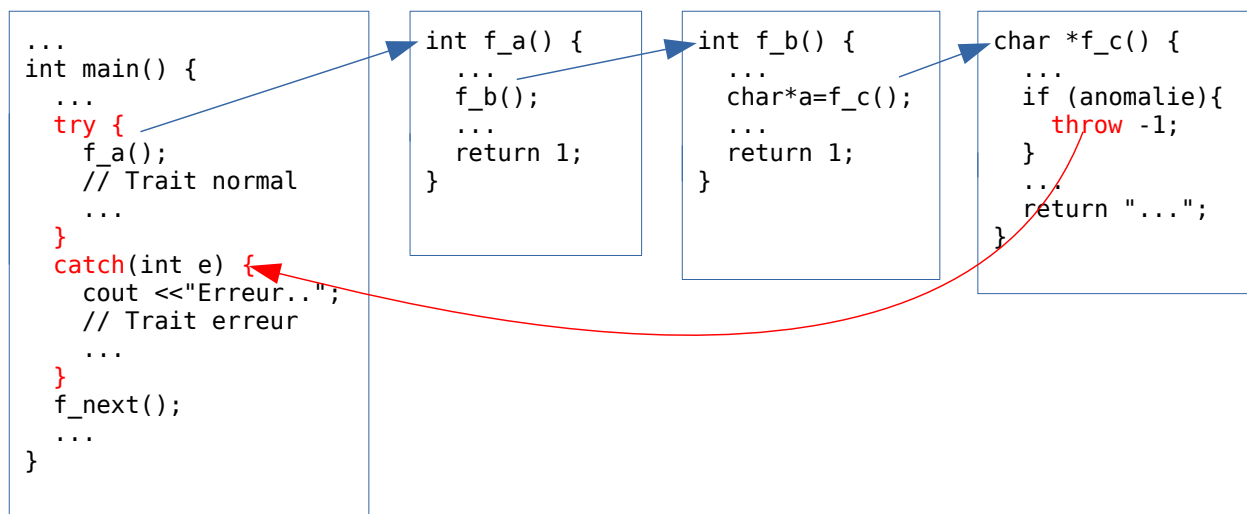
b) La gestion des exceptions - Généralités

Le C++ permet de simplifier le traitement des anomalies à l'aide du mécanisme de « **gestion des exceptions** » :

Définition : la gestion des exceptions (*exception handling*) est le mécanisme mis en place pour répondre à la survenue, pendant l'exécution d'un programme, d'« exceptions » – anomalies ou conditions exceptionnelles – qui obligent en général à interrompre le cours normal du programme. Exceptions courantes : erreur d'accès mémoire, indice de tableau hors limites, fichier absent, valeur incorrecte d'un paramètre,...

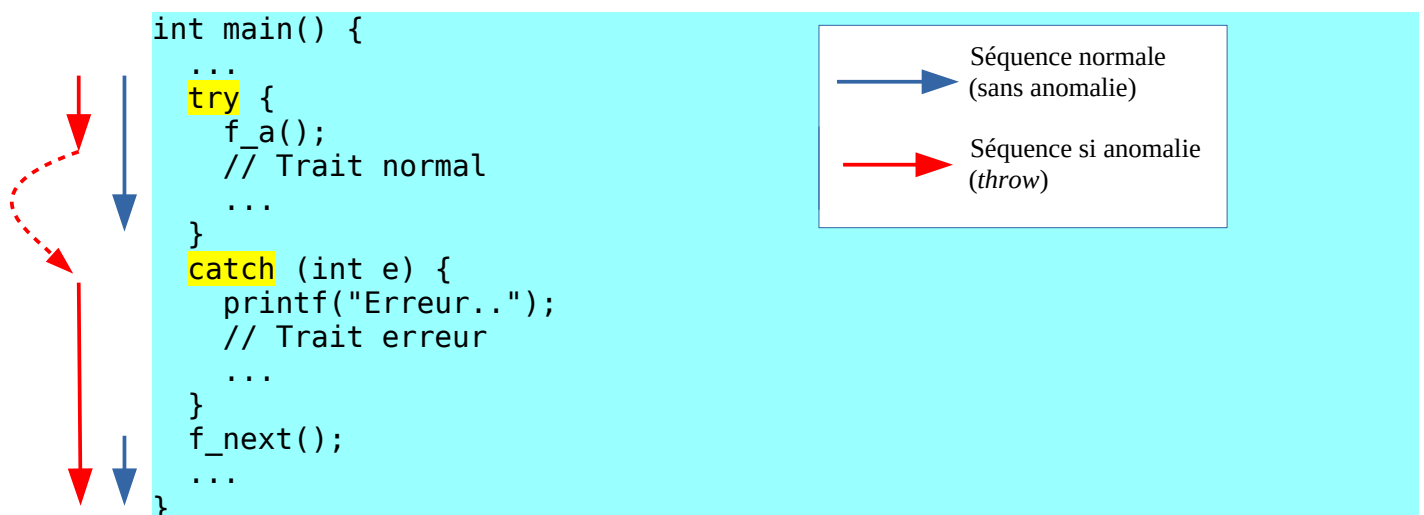
Pour ce faire, le C++ a créé trois nouveaux mots-clés :

throw **try** **catch**
(jette) (essaye) (attrape)



La fonction qui a détecté l'anomalie « jette » ou « lève » (*throws*) une exception. Si l'appel de cette fonction s'est fait (directement ou indirectement) dans un « bloc *try* », l'exception est « capturée » par le « bloc *catch* » approprié (cf. §c ci-dessous : comment savoir si un bloc *catch* convient?). La valeur passée au *throw* est récupérée en paramètre du *catch*.

Déroulement du programme dans le `main()` en reprenant l'ex. précédent :



Si une exception est levée, il y a rupture de la séquence normale du programme : l'exécution « saute » de l'instruction *throw* directement au début du bloc *catch*, sans repasser par les fonctions appelantes. Ceci induit un « *stack unwinding* » (« dévider la pile »), où la fonction qui lève l'exception et les éventuelles fonctions intermédiaires sont dépilées brusquement de la pile d'exécution, sans pouvoir se terminer.

Avantages du mécanisme de gestion des exceptions :

- code des fonctions plus simple : pas de test systématique de code d'erreur sur les retours de fonctions,
- permet une séparation nette entre le code qui détecte l'erreur et le code qui traite l'erreur,
- permet de centraliser le traitement des erreurs, en général à un niveau haut dans l'arbre des appels de fonction (dans le `main()` pour l'exemple ci-dessus).

Inconvénient :

- la rupture de séquentialité et le « *stack unwinding* » induits lorsqu'une exception est levée rendent beaucoup plus délicate la libération des ressources dans les fonctions. Par ex. si on a fait un `new` et qu'un `throw` est déclenché avant le `delete` correspondant, il y a une fuite mémoire. La prise en compte de ce problème sera l'objet du RAII (cf. §2).

c) Fonctionnement du « *catch* »

Un « bloc *catch* » – appelé aussi « **gestionnaire d'exception** » (*exception handler*) – prend un paramètre unique (à la façon d'un entête de fonction). Le type du paramètre correspond au type de l'exception capturée. Un *catch* capture les exceptions levées dans le bloc *try* précédent si son paramètre correspond au type utilisé par le *throw*. Ex: si *throw* a levé une expression de type T, les *catch* suivants conviennent :

```
catch (T e)
catch (T & e)
catch (const T e)
catch (const T & e)
```

La dernière forme (par référence const) est recommandée par les guides de bonne pratique. Par contre, il est recommandé de passer au *throw* une valeur, pas une référence ni un pointeur.

Le *catch* capture aussi les exceptions d'un type dérivé du type de base de son paramètre. Ex:

```
class D : public B{
    ...
};
D exc;
throw exc;
...
try {
    ...
}
catch (const B & e) { // OK capture l'exception exc
    ...
}
```

En principe, n'importe quel type peut convenir, `int` (ex: code d'erreur entier), `std::string` (ex: texte d'erreur explicite), etc. mais la librairie standard fournit une classe de base `std::exception`, et des classes dérivées spécialisées (les principales sont incluses dans le *header* `<stdexcept>`).

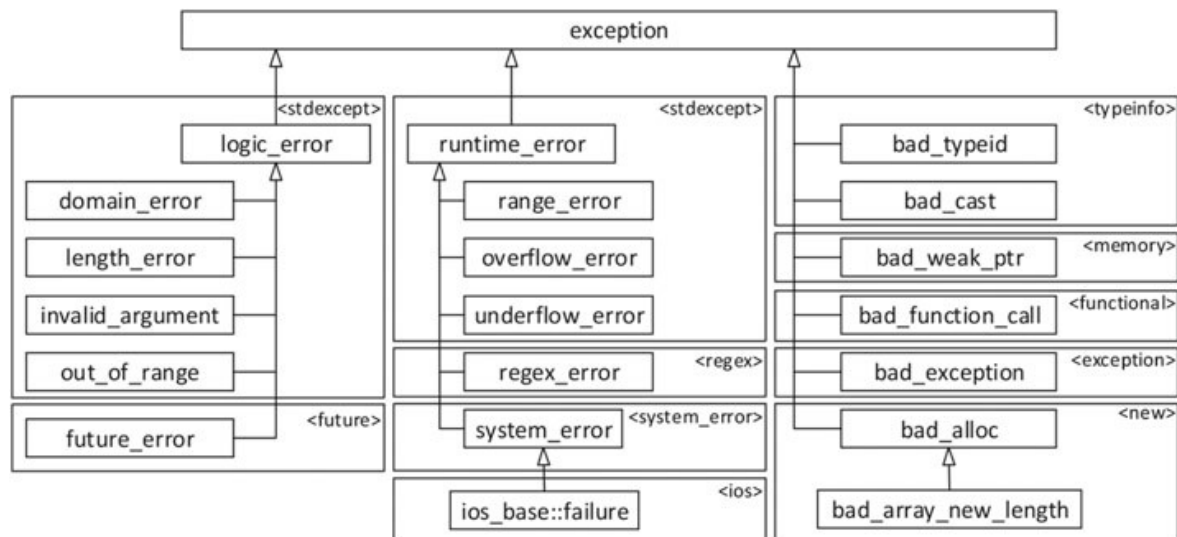


Fig : Les *headers* et la hiérarchie (polymorphe) des classes d'exceptions (d'après « C++ Standard Library Quick Reference » de P. Van Weert et M. Gregoire)

Les guides de bonne pratique recommandent de lever des exceptions instances de ces classes ou de dériver sa propre classe à partir de `std::exception`. Beaucoup de bibliothèques tierces fournissent leurs propres classes d'exceptions dérivées de `std::exception` (ex. §d ci-dessous).

La classe `std::exception` fournit la méthode (virtuelle) `what()` qui doit être redéfinie (*overriden*) dans les classes dérivées. Elle retourne une chaîne de caractères (`const char *`) qui a pour but de décrire la cause de l'erreur.

Ordre de recherche du bloc *catch* approprié

Plusieurs blocs *catch* peuvent suivre un bloc *try*, ils sont évalués séquentiellement. Dès qu'un bloc *catch* possède un type approprié (cf. ci-dessus) le programme rentre dans ce bloc et y continue son exécution, sans rechercher les blocs *catch* suivants. Si aucun bloc *catch* n'est approprié, le programme cherchera s'il y a des éventuels blocs *try/catch* dans la fonction appelante et ainsi de suite jusqu'à remonter au `main()`.

Si, à la suite de ce cheminement, il ne trouve aucun bloc *catch* qui correspond à l'exception levée, le programme appelle la fonction standard `std::terminate()`, qui appelle elle-même `abort()` et le programme s'arrête brutalement (« Abandon (core dumped) ») sans avoir traité l'exception.

Pour éviter de s'arrêter de cette façon, on peut ajouter après les blocs *catch* de la fonction de plus haut niveau, un *catch final* « attrape tout », qui capturera toutes les exceptions non capturées par les *catch* précédents. Il s'écrit `catch(...)` et ne récupère aucune information provenant du `throw`.

d) Exemple : traitement des erreurs liées à une base de données

En exemple, la librairie MySQL « *c++ connector* » (*libmysqlcppconn-dev*) fournit une classe `SQLException`, dérivée de `std::exception`, qui « remonte » les erreurs du SGBD MySQL (erreurs lors de la connexion, de l'exécution d'une requête, etc.). Les différentes méthodes de la librairie (`connect`, `executeQuery`,...) peuvent générer des *throws*.

Dans le code qui suit, une fonction `lecbase()` se connecte à une base de données MySQL, passe une requête SQL pour lire des noms dans une table et renvoie le tableau (vecteur) de noms à l'appelant (le `main()`). On gère toutes les erreurs au niveau du `main()` : connexion impossible, erreur dans la requête SQL, problème avec le vecteur de noms,...


```

#include <iostream>
#include <stdexcept>
#include <vector>
#include <cppconn/driver.h>
#include <cppconn/statement.h>

std::vector<std::string> lecbase(){
    std::vector<std::string> tab;
    sql::Driver *driver {nullptr}; // Voir ci-dessous pour la gestion de ce pointeur
    sql::Connection *conn {nullptr};

    /* Ouverture d'une connexion MySQL à la database "test" */
    driver = get_driver_instance(); // La doc de la librairie indique de ne pas libérer
    // ce pointeur : « do not explicitly free driver, the library takes care of freeing that »
    conn = driver->connect("localhost/test", "root", "pwd");
    std::cout<< "Connecté à la base test\n";

    /* Création et exécution d'une requête SQL à la database "test" */
    sql::Statement *stmt {nullptr};
    sql::ResultSet *res {nullptr};
    stmt = conn->createStatement();
    res = stmt->executeQuery("SELECT nom FROM employe");

    while (res->next()) {
        tab.push_back(res->getString(1)); // Chargement du vecteur
    }
    // Libération requête
    delete res;
    delete stmt;
    // Libération connexion
    delete conn;

    return tab;
}

int main() {

    std::vector<std::string> tab;
    try {
        tab = lecbase();
    }
    catch (const sql::SQLException &e) { // Pour les erreurs remontées par MySQL
        std::cout << "Erreur MySQL: " << e.what() << std::endl;
    }
    catch (const std::exception &e) { // Pour les erreurs éventuelles liées à la STL (vecteur)
        std::cout<< "std::exception:" << e.what() << std::endl;
    }
}

```

throw des
erreurs de
connexion

throw des
erreurs de
requête

Ces delete ne sont pas
faits si un throw a lieu
avant dans lecbase()

```

catch(...){ // Pour d'autres erreurs éventuelles non récupérées par les catch précédents
    std::cout<< "Exception inconnue" << std::endl;
}
// Affichage vecteur
for ( const auto &nom : tab ){
    std::cout << nom << std::endl;
}
return 0;
}

```

Ce code fonctionne et traite les erreurs correctement mais a un **problème potentiel de libération mémoire** si des exceptions sont levées par les méthodes appelées dans la fonction `lecbase()` : dans ce cas, le `throw` fait sortir de la fonction pour sauter directement aux `catch` du `main()` (*stack unwinding*). Donc les libérations des objets pointés par `conn`, `smt`, `res` (à la fin de `lecbase()`) ne se font pas et il y a une fuite mémoire. Le § suivant explique la méthode générale (RAII) pour éviter ce type de problème.

2- Notion de RAII

Définition : la RAII (*Resource Acquisition Is Initialization*) est la technique C++ pour éviter les problèmes de non-libération de ressources dues à la survenue d'exceptions. L'idée est de réaliser l'acquisition de ressource uniquement lors de l'initialisation d'une classe dédiée. On lie cette ressource à la durée de vie d'un objet de cette classe : la ressource est acquise durant la construction de l'objet et elle est libérée au moment de la destruction de l'objet, cette destruction étant garantie même en cas d'erreur.

Remarque 1 : l'acquisition de la ressource n'est pas faite obligatoirement dans le constructeur, elle peut se faire dans une autre méthode de la classe, par contre la libération est obligatoirement dans le destructeur (pour bénéficier de son exécution automatique en fin de portée). C'est pourquoi certains préfèrent appeler cette technique « *Resource Release Is Finalization* ».

Remarque 2 : la libération de la ressource est garantie en cas d'erreur car lors du « *stack unwinding* » (cf. §1.b) toutes les variables locales des fonctions dépilées sont automatiquement détruites.

En reprenant l'exemple du §1.d (fonction `lecbase()`), on constate qu'il faut libérer (`delete`) les pointeurs `conn` (connexion), `stmt` et `res` (requête). On pourrait regrouper ces pointeurs dans une même classe (connexion+requête), mais il paraît plus judicieux de faire une classe (appelée BDD) pour la connexion (qui gère le pointeur `conn`) et une autre (appelée Requete) pour les requêtes d'interrogation (qui gère les pointeurs `stmt` et `res`). C'est ce qui est présenté dans le code qui suit.

```

#include <iostream>
#include <stdexcept>
#include <vector>
#include <cppconn/driver.h>
#include <cppconn/statement.h>

class Requete{
private :
    sql::Statement *stmt {nullptr};
    sql::ResultSet *res {nullptr};
public :
    // Constructeur

```

```

Requete(sql::Connection * conn ){
    // Création d'une requête SQL
    stmt = conn->createStatement();
}
// Méthode qui exécute la requête
std::vector<std::string> exec( const std::string &chaine_req){
    std::vector<std::string> tab;
    // Exécution de la requête
    res = stmt->executeQuery(chaine_req);
    while (res->next()) {
        tab.push_back(res->getString(1));
    }
    return tab;
}
// Destructeur
~Requete(){
    // Libération requête
    delete res;
    delete stmt;
}
};

class BDD{
private:
    sql::Connection *conn {nullptr};
public :
    // Constructeur
    BDD(){
        /* Ouverture d'une connexion MySQL à la database "test" */
        sql::Driver *driver;
        driver = get_driver_instance();
        conn = driver->connect("localhost/test", "root", "pwd");
        std::cout<< "Connecté à la base test\n";
    }
    // Méthode qui crée la requête et lance son exécution
    std::vector<std::string> interrogation(const std::string &ch_req){
        std::vector<std::string> tab;
        Requete req(conn);
        tab = req.exec(ch_req);
        return tab;
    }
    // Destructeur
    ~BDD(){
        // Libération connexion
        delete conn;
    }
};

```

Ces *delete* seront toujours faits même si un *throw* a lieu avant

Ce *delete* sera toujours fait même si un *throw* a lieu avant

```
// Fonction lecbase(), appelée par main() de la même façon qu'en §1.d

std::vector<std::string> lecbase(){
    std::vector<std::string> tab;

    BDD base; // Instanciation d'un objet de classe BDD
    tab = base.interrogation("SELECT nom FROM employe");
    return tab;
}
```

Remarque : La librairie standard a érigé en principe général la RAI et l'applique dans toutes ses classes. De plus, elle offre les « *smart pointers* » pour aider à la mise en œuvre de la RAI sans passer obligatoirement par des classes dédiées.

On aurait eu le même comportement que ci-dessus en transformant les pointeurs bruts (*raw pointers*) en `unique_ptr` (cf. Ch.6 §2.d), et ceci sans créer les classes BDD et Requete. Ex. dans la fonction `lecbase()` :

```
#include <iostream>
#include <stdexcept>
#include <vector>
#include <memory>
#include <cppconn/driver.h>
#include <cppconn/statement.h>

std::vector<std::string> lecbase(){
    std::vector<std::string> tab;

    sql::Driver *driver {nullptr};

    /* Ouverture d'une connexion MySQL à la database "test" */
    driver = get_driver_instance();
    std::unique_ptr<sql::Connection> uconn {driver->connect(
        "localhost/test", "root", "pwd")};
    std::cout<< "Connecté à la base test\n";

    /* Création et exécution d'une requête SQL à la database "test" */
    std::unique_ptr<sql::Statement> ustmt { uconn->createStatement() };

    std::unique_ptr<sql::ResultSet> ures {ustmt->executeQuery(
        "SELECT nom FROM employe")};
    while (ures->next()) {
        tab.push_back(ures->getString(1));
    }
    //les zones mémoire pointées par uconn, ustmt et ures sont libérées ici implicitement
    // (lecbase() en est propriétaire)
    return tab;
}
```

Ch 8 (hors programme) : Surcharge d'opérateur, foncteurs, fonctions lambda

Plan du chapitre:

1) La surcharge d'opérateur (operator overloading)

- a) Généralités
 - b) Les opérateurs concernés
 - c) L'opérateur << (stream insertion operator)
 - d) Les fonction amies (friend functions)
- ### 2) Les objets-fonctions ou « foncteurs » (functors)

3) Les fonctions lambda (lambda functions)

- a) Généralités
- b) Les fermetures (closures)

1- La surcharge d'opérateur (operator overloading)

a) Généralités

But : permettre d'utiliser les opérateurs habituels (comme +, -, =, etc.) de façon à « nommer » (lors de l'appel) une fonction de façon plus naturelle pour un humain.

Ex: on veut, étant donnée une classe Point, créer une fonction pour additionner deux points et mettre le résultat dans un troisième point.

Sans la surcharge d'opérateur on pourrait écrire une fonction add() :

```
Point add ( const Point& p1, const Point& p2 ){
    Point p3;
    p3.x = p1.x + p2.x; // ici on suppose que x et y sont publics
    p3.y = p1.y + p2.y;
    return p3;
}
// Appel :
Point a, b, c;
c = add ( a, b );
```

Avec la surcharge d'opérateur on peut définir un opérateur + pour additionner deux points :

```

Point operator+ ( const Point& p1, const Point& p2 ){
    // code identique
    ...
}
// Appel :
Point a, b, c;
c = a + b; // le compilateur traduit cette ligne en : c = operator+ ( a, b )

```

L'intérêt se voit au moment de l'appel, qui est beaucoup plus intuitif et naturel pour le lecteur du programme.

L'opérateur + montré ci-dessus a été défini en fonction globale, en dehors de la classe (pour la simplicité de l'explication). Cependant il est plus habituel de définir ces opérateurs en tant que méthode de la classe concernée (ici `Point` par ex.). De ce fait, pour un opérateur binaire (comme +), au lieu d'avoir deux paramètres d'entrée comme ci-dessus (p1 et p2), qui correspondent à l'opérande de gauche et de droite de l'opérateur +, il n'y aura qu'un seul paramètre (=opérande de droite) car l'opérande de gauche correspondra à l'objet courant (*this). Ce qui donnera le code suivant (`operator+()` est maintenant une méthode de `Point`):

```

Point Point::operator+ ( const Point& p2 ) const { // on ne modifie pas l'objet courant
    Point p3;
    p3.x = this->x + p2.x; // x et y peuvent être privés
    p3.y = this->y + p2.y;
    return p3;
}
// Appel (identique) :
Point a, b, c;
c = a + b; // le compilateur traduit maintenant cette ligne en : c = a.operator+ ( b )

```

b) Les opérateurs concernés

Sont concernés tous les opérateurs habituels : arithmétiques: +, -, *, /, etc., de comparaison: ==, <, etc., d'affectation: =, +=, etc., mais aussi des opérateurs comme << (*put*, cf. Ch. 2, §5), >> (*get*, *id.*), () (appel de fonction) ou [] (opérateur d'indilage de tableau).

Quelques opérateurs ne peuvent pas être surchargés : :: (résolution de portée), . (accès à un membre), .* (accès à un membre à travers un pointeur vers le membre), et ?: (opérateur ternaire).

Les opérateurs surchargés peuvent être **binaires**, comme + dans l'exemple d'introduction, ou **unaires**, auquel cas la méthode ne prend pas de paramètre.

Ex : l'opérateur unaire ! (renvoie un booléen : soit vrai soit faux) :

```

bool Point::operator! () const { // pas de paramètre
    return this->x == 0 && this->y == 0;
}
// Appel :
Point p;
if ( !p ) { // le compilateur traduit cette ligne en : if ( p.operator!() )
    std::cout << "Le point est (0,0)\n";
}

```

Cas particulier : opérateur = (assignment operator). Il fait partie des méthodes spéciales d'une classe (avec le constructeur par défaut, constructeur de copie, constructeur par déplacement, destructeur,...) : pour chacune de ces méthodes une version par défaut est générée automatiquement par le compilateur si le programmeur ne l'a pas définie explicitement. En clair, si vous ne l'écrivez pas vous même, le compilateur le crée en arrière-plan. Cet opérateur « = » par défaut (implicite) affecte membre à membre les attributs de l'objet courant (à gauche de =) par les attributs de l'objet paramètre (à droite de =). De ce fait, même si on n'écrit pas d'opérateur « = » explicite, le code suivant fonctionne :

```
Point p1{12,5}, p2;  
p2 = p1; // le compilateur traduit : p2.operator=(p1)  
p2.affiche(); // x=12,y=5
```

On ne doit écrire explicitement l'opérateur « = » que si il doit effectuer des opérations non triviales, par exemple acquérir ou libérer une ressource mémoire. Sa « forme canonique » doit :

- prendre en paramètre une référence constante,
- retourner la référence de l'objet courant,
- ne rien faire si l'appel implique l'affectation d'un objet par lui-même, ex. : `a = a;`

Ex. sur une classe T :

```
T& operator= ( const T& other ) {  
    if ( &other == this ) return *this; // traitement du cas a = a;  
    // faire le nécessaire pour copier other dans l'objet courant  
    ...  
    return *this;  
}
```

c) Opérateur << (stream insertion operator)

Jusqu'à présent, pour afficher un objet, nous avons utilisé une méthode « banale », comme `affiche()` pour la classe `Point`, dans laquelle on fait l'affichage de `x` et `y` (des entiers) sur la sortie standard.

Pour obtenir une fonction d'affichage d'utilisation plus intuitive, on peut surcharger l'opérateur << pour une classe.

Quand on écrit « `std::cout << obj` », où `obj` est un objet, le compilateur regarde s'il existe (dans la classe `std::ostream` puis en global) une surcharge de l'opérateur << pour le type de `obj`. Si ce n'est pas le cas (il n'y a de surcharges prédéfinies que pour les types de base `int`, `char`, `double`, etc. et les chaînes de caractères), il faut donc la définir pour notre classe.

Cependant l'opérateur << ne peut pas être une méthode de la classe `Point` (dans notre exemple). En effet, c'est un opérateur binaire et l'opérande de gauche n'est pas de la classe `Point` : dans l'instruction `std::cout << obj`, convertie par le compilateur en `std::cout.operator<<(obj)`, l'objet sur lequel est appelé la fonction (`std::cout`) est de la classe `std::ostream`.

Une première idée serait donc de modifier la classe `std::ostream` en surchargeant l'opérateur << pour qu'il accepte un paramètre de type `Point` (dans notre exemple). Or, il est assez complexe de modifier (ou dériver) une classe standard et ce n'est pas obligatoirement une bonne idée.

En fait la solution est d'implémenter l'opérateur << en tant que fonction globale (c'est-à-dire non-membre d'une classe). Sa « forme canonique » doit :

- prendre en premier paramètre : une référence sur un objet la classe `std::ostream`,

- prendre en second paramètre : une référence constante sur l'objet à afficher,
- retourner la référence de l'objet de la classe `std::ostream` passé en paramètre.

Ex. sur la classe `Point` :

```
std::ostream& operator<< ( std::ostream& sortie, const Point& obj ){
    sortie << "x=" << obj.getX() << ",y=" << obj.getY();
    return sortie;
}
// Appel :
Point p;
std::cout << p; // le compilateur traduit cette ligne en : operator<<(std::cout,p)
```

L'intérêt de faire retourner un l'objet de classe `ostream` par la fonction est que ce retour permet de chainer plusieurs fois l'opérateur `<<`. Ex. :

```
std::cout << p << std::endl;
// le compilateur traduit cette ligne en : operator<<(std::cout,p).operator<<(std::endl);
```

operator<< ami de la classe Point
qui retourne std::cout
operator<< de la classe
std::ostream

Le fait que notre opérateur `<<` soit une fonction globale, non-membre de la classe `Point`, entraine deux inconvénients :

- il n'est pas déclaré dans le bloc de déclaration de la classe `Point`,
- il n'a accès aux attributs privés (ici `x` et `y`) qu'au travers d'accesseurs.

Pour éviter ces inconvénients, cet opérateur est souvent implémenté en tant que fonction amie (*friend function*), cf. § ci-dessous.

d) Les fonctions amies (*friend functions*)

Le mot-clé « **friend** » devant une déclaration de fonction dans un bloc de déclaration de classe autorise à la fonction l'accès aux membres privés (`private`) et `protected` de cette classe.



Une fonction amie ne devient pas pour autant une fonction membre de la classe, elle reste non-membre mais a un statut particulier par rapport à cette classe.

Ex. avec l'opérateur `<<` sur la classe `Point` :

```
class Point{
    public :
        // Constructeurs et autres méthodes
        ...
        // Fonction amie (la définition pourrait aussi être ici, inline)
        friend std::ostream& operator<<( std::ostream&, const Point& );
    private :
        int x,y;
};
```



```
// Définition hors du bloc de déclaration de Point,
// mais ne surtout pas mettre le préfixe Point:: devant operator<< (pas une fonction membre)
std::ostream& operator<< ( std::ostream& sortie, const Point& obj ){
    sortie << "x=" << obj.x << ",y=" << obj.y; // Accès direct à x et y
    return sortie;
}
```

2- Les objets-fonctions ou « foncteurs » (*functors*)

Définition : un foncteur (*functor*), ou objet-fonction (*function object*), est un objet qui se comporte comme une fonction. Pour créer un foncteur, il suffit de créer une classe qui **surcharge l'opérateur ()**. Un foncteur est appelé en utilisant la syntaxe traditionnelle d'une fonction, c'est-à-dire suivi par des parenthèses qui encadrent les éventuels paramètres.

Ex. :

```
class Add_x { // classe foncteur
public :
    // Constructeur à un argument
    Add_x(int x) : x(x) {}
    // Surcharge de l'opérateur ()
    int operator()(int y) const { // "y" est le paramètre de la fonction
        return x + y;
    }
private:
    int x;
};

// Maintenant vous pouvez l'utiliser comme ceci:
Add_x add42{42}; // création d'une instance de la classe foncteur avec l'argument 42 (ira dans x)
int i = add42(8); // appel de cette instance comme si c'était une fonction, avec le paramètre 8
std::cout << i; // affiche 50
```

Intérêt d'un foncteur :

- permet de passer une « fonction » en paramètre d'une autre fonction : c'est un mécanisme analogue à celui de pointeur de fonction, mais différent car c'est un objet qu'on passe. Ce type de paramètre est souvent appelé un « *callback* ».
- permet de garder un « état » entre chaque appel du foncteur, sous forme d'attribut de l'objet. Ici, par exemple l'attribut x vaut 42 à la création du foncteur, mais il pourrait être modifié et cette modification serait conservée à l'appel suivant.

Les foncteurs sont très utilisés par les algorithmes de la STL, cf. Ch. 6, §2.c.3. Beaucoup de ces algorithmes acceptent des foncteurs en paramètre. La librairie fournit un grand nombre de **foncteurs prédéfinis**, qui sont en fait des *templates* de classes : `std::plus`, `std::minus`, `std::less`, `std::greater`, `std::equal_to`, etc.

Vous pouvez aussi définir vos propres foncteurs à passer en paramètre aux algorithmes standard. Ex. une fonction de comparaison d'entiers fournie en paramètre à l'algorithme `std::sort` :

```
class IntComp
{
public:
    // Pas de constructeur : le constructeur par défaut est implicite
    // (pas d'attribut, donc pas d'argument à prévoir)
    bool operator()(int a, int b){
        return a < b;
    }
};

int main()
{
    std::vector<int> tab { 4, 3, 1, 2 };
    std::sort(tab.begin(), tab.end(), IntComp{}); // IntComp{} crée un foncteur anonyme
    // On pourrait aussi passer par un foncteur « nommé » :
    IntComp fcomp; // Instanciation d'un objet (=foncteur) de classe IntComp
    std::sort(tab.begin(), tab.end(), fcomp);
    return 0;
}
```

3- Les fonctions *lambda* (*lambda functions*)

a) Généralités

Définition : une fonction *lambda* (*lambda function*), ou expression *lambda* (*lambda expression*), est une fonction anonyme créée directement « *in line* » dans le code. En fait une expression *lambda* est un raccourci pour créer un foncteur (sans avoir besoin de définir une classe, de redéfinir l'opérateur `()` ni d'instancier un objet de cette classe). Les fonctions *lambda* n'existent que depuis la **norme C++11**.

La fonction *lambda* est bien adaptée pour une fonction très courte qui n'est appelée qu'une seule fois. C'est en particulier utile pour passer une fonction simple (« *callback* ») en paramètre d'un algorithme : le paramètre passé contient directement le corps de la fonction.

Syntaxe générale (parties obligatoires en rouge, parties optionnelles en noir italique) :

[captures] (paramètres) -> type_retour { corps }

Ex: une expression *lambda* pour l'opération « plus petit que *x* » (*x* étant une variable capturée dans la portée où se trouve l'expression *lambda*) :

Ici pas de type de retour: il est automatiquement inféré (booléen)

```
[ &x ] ( int a ) { return a < x; }
```

Capture (cf. §b)

Type et nom param

Corps de la fonction

Dans le cas le plus simple, il n'y a pas de variables capturées ni de paramètres (rare). Ex :

```
[ ] { std::cout << "Je suis une fonction lambda" << std::endl; }
```

Avantages des fonctions *lambda* :

- code très léger et efficace,
- permet la capture de variables, c'est-à-dire la création d'une « fermeture » (cf. §b ci-dessous).

Inconvénient :

- la syntaxe, inhabituelle, surtout pour la partie « capture », rend le code obscur pour les développeurs qui utilisent peu les fonctions *lambdas*.

Ex. d'utilisation : en reprenant l'exemple présenté ci-dessus pour les foncteurs (fonction de comparaison d'entiers fournie en paramètre à l'algorithme `std::sort`) :

```
int main()
{
    std::vector<int> tab { 4, 3, 1, 2 };

    std::sort( tab.begin(), tab.end(), [](int a, int b){return a < b;} );

    return 0;
}
```

On peut constater en comparant avec le code du § précédent, que celui-ci est beaucoup plus léger.

b) Les fermetures (*closures*)

Définition : une fermeture (*closure*) est une fonction qui « embarque » son environnement au moment de sa création (c'est-à-dire qu'elle « capture » les variables externes présentes dans la portée englobante dans laquelle elle est définie). Au moment de son exécution, la fermeture a accès (en lecture et/ou écriture) à ces variables capturées, même si celles-ci n'existent plus dans son environnement d'exécution.

Certains langages utilisent fréquemment les fermetures, comme JavaScript dans lequel une fermeture est créée en définissant une fonction dans le corps d'une autre fonction. En C++ le concept de fermeture n'existe que depuis le C++11 grâce aux fonctions *lambda* qui ont mis en place la possibilité de capture de variables par une fonction.

La liste de capture

Dans la syntaxe d'une expression *lambda*, la partie entre `[]` est la **liste de capture**. Par défaut, les variables de la portée englobante ne sont pas accessibles dans la fonction *lambda*. La liste de capture définit ce qui, à l'extérieur de la *lambda*, doit être accessible à l'intérieur du corps de la fonction et comment (par copie ou par référence). Elle peut être :

1. une valeur (par copie) : `[x]`
2. une référence : `[&x]`
3. toute variable de la portée englobante, par référence : `[&]`
4. comme en 3, mais par valeur (par copie) : `[=]`

On peut mixer ces différentes possibilités dans une liste séparée par des virgules : `[x, &y]`.

Remarque 1 : une expression *lambda* elle-même n'est pas un foncteur ou une fermeture, c'est une expression qui en crée une. On peut penser à une *lambda* comme à une classe, et la fermeture comme à l'objet réellement instancié.

Remarque 2 : une expression *lambda* dont la liste de capture est vide (cas assez fréquent) ne génère pas de fermeture, mais un simple objet-fonction.

Remarque 3 : on peut stocker une expression *lambda* dans une variable. Il est plus simple dans ce cas de déclarer cette variable de type « auto ».

Remarque 4 : l'expression *lambda* étant en arrière-plan convertie en classe foncteur par le compilateur, chaque variable capturée correspond à un attribut de cette classe. Les variables capturées (si elles sont accessibles en écriture) permettent donc de garder un « état » entre chaque appel de la fonction *lambda*, comme indiqué au §2 pour les foncteurs.

Ex. de listes de capture :

```
int a {42}; // Variable locale "a"

auto f0 = [](){ return a*2; }; // Erreur: 'a' n'est pas accessible
auto f1 = [a]() { return a*2; }; // "a" est capturée par valeur, "a" ne peut pas être modifiée
auto f2 = [&a]() { return a++; }; // "a" est capturée par référence, la variable "a" sera modifiée
// Rq: le programmeur doit s'assurer que "a" n'est pas détruite avant l'appel de f2()

int b {f2()}; // Appel de la fonction lambda.
std::cout << "a:" << a << ",b:" << b << "\n"; // Affiche a:43,b:42
```

Ex. : Une expression *lambda* est passée en paramètre (type `std::function<...>`) à une fonction « normale » `ftest()` et une variable entière locale du `main()` est capturée par référence et modifiée (incrémentée).

```
#include <iostream>
#include <functional>

void ftest( std::function<int()> lbda ){ // La fonction ftest a une
// expression lambda en paramètre
    std::cout << "a:" << lbda() << std::endl; // Appel de la fonction lambda
// a est incrémenté de 1, pourtant a est inconnu dans ftest
}

int main(){
    int a {42}; // Variable locale au main()
    auto lambda = [&a]() { return ++a; }; // Stockage de l'expression lambda
// dans une variable

    ftest(lambda); // a:43
    ftest(lambda); // a:44
    return 0;
}
```

Ex. d'utilisation réelle : l'algorithme `remove_if` de la STL (cf. Ch.6 §2.c.3) peut prendre en paramètre un pointeur de fonction, un foncteur ou une expression *lambda*. Cet objet-fonction n'accepte qu'un paramètre (le résultat du déréférencement de chaque itérateur dans la plage `[begin(), end())`) et doit retourner un booléen (*true* indique qu'il faut enlever la valeur). Ici le seuil en-dessous duquel la valeur est enlevée d'un vecteur est une variable, qu'on ne peut pas passer en argument à cet objet-fonction (puisqu'il n'accepte qu'un paramètre), donc il faut la capturer.

Une autre expression *lambda*, mais sans capture, est aussi utilisée dans le code ci-dessous, en paramètre de l'algorithme `for_each` de la STL.

```
#include <vector>
#include <iostream>
#include <algorithm>

int main()
{
    std::vector<int> tab {1, 2, 3, 4, 5, 6, 7};
    int seuil;
    std::cout << "seuil en dessous duquel les valeurs sont effacées : ";
    std::cin >> seuil;
    tab.erase (
        std::remove_if(tab.begin(), tab.end(), [seuil](int n){return n<seuil;}),
        tab.end()
    );

    std::cout << "tab: ";
    std::for_each(tab.begin(), tab.end(), [](int i){ std::cout << i << ' '; });
    std::cout << '\n';

    return 0;
}
```

Si le seuil est fixé à 5, le programme va afficher :

tab: 5 6 7

Références

Les livres de référence

Meyers, Scott 2015 : *Effective Modern C++*, O'Reilly, Inc.

Stroustrup, Bjarne 2013 : *The C++ Programming Language fourth ed.*, Addison Wesley, Inc.

Stroustrup, Bjarne 2014 : *A tour of C++*, Addison Wesley, Inc.

Van Weert, Peter, Gregoire, Marc 2016 : *C++ Standard Library Quick Reference*, Apress Inc.

Les sites de référence

<https://en.cppreference.com>. Traduction française (partiellement automatique) <https://fr.cppreference.com> mais utiliser de préférence la version anglaise qui est plus à jour.

<http://www.cplusplus.com/reference>. Uniquement la librairie standard.

<http://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines>. Guide de programmation évolutif par Bjarne Stroustrup et Herb Sutter.