Survol de C++

(version βeta)

Louise Laforest

1 Introduction

La syntaxe du langage Java a été inspirée de celle du langage C dont C++ est issu. Le texte qui suit suppose une bonne connaissance du langage Java.

2 Éléments de base

- **Identificateurs**
  Le compilateur est sensible à la casse. Les identificateurs doivent être composés d'une lettre (majuscule ou minuscule, non accentuée) suivie d'une suite (pouvant être vide) de chiffres, lettres et du caractère de soulignement.

- **Commentaires**
  Comme en Java soit /* ...*/ ou //.

- **Types de base**
  Il y a quelques différences avec Java. Il y a les types désignés par un seul identificateur : `char`, `bool`, `short`, `int`, `float`, `long`, `double`. Aussi, `short` peut servir de préfixe à `int` mais on peut écrire tout simplement `short` qui désignera la même chose. l'identificateur `long` peut aussi servir de préfixe à `int` et `double` mais `long int` correspond à `long`. Tout type contenant `short` ou `long` peut être précédé de l'identificateur `unsigned`. Les types `char` et `int` peuvent aussi être précédés de `unsigned`. Une remarque s'impose concernant le type `bool`. Une variable de ce type peut prendre comme valeurs possibles `false`, `true`, toute valeur entière, réelle, d'un type énumératif ! La valeur `0` représente la valeur logique `false` alors que toute valeur différente de zéro est interprétée comme `true` ! Ceci peut mener à des bogues difficiles à détecter dans certains cas. En voici un exemple (qui ne compilerait pas en Java).

```
int i = 0;
if ( i = 0 ) {
    cout << "i est nul";
} else {
    cout << "i est non nul";
}
```

  Ce bout de code affichera `i est non nul` ! L'erreur ici est qu'il fallait employer l'opérateur de comparaison `==` au lieu de l'opérateur d'affectation `=`. Ici, l'affectation retourne comme résultat la valeur affectée, soit 0. La valeur 0 étant interprétée comme la valeur logique `false`, la partie `else` est donc exécutée ! Ce bout de code ne compile pas en Java puisque la valeur retournée par l'instruction `i = 0` est de type `int` et non `boolean`.

  L'espace mémoire utilisé par chacun des types dépend du système utilisé.

- **Variables et constantes**
  Elles se déclarent comme en Java. Par exemple `int x; const float PI = 3.1416;`. Voir aussi la section 10.

- **Types énumération**
  Les types énumération existent en Java mais il faut déclarer une classe ou le mot clé `enum` (depuis la version 5.0) pour ce faire. En C++, il s'agit d'énumérer les constantes faisant partie du type. Voici deux exemples.

```
enum Jour {
    DIMANCHE, LUNDI, MARDI, MERCREDI, JEUDI, VENDREDI, SAMEDI
}; // les constantes sont numerotées à partir de 0 par défaut

enum Mois {
    JANVIER = 1, FEVRIER, MARS, AVRIL, MAI, JUIN, JUILLET, AOUT, SEPTEMBRE,
    OCTOBRE, NOVEMBRE, DECEMBRE
}; // les constantes seront numerotées à partir de 1
```

  L'utilisation de types énumération augmente la lisibilité d'un programme.

- **Expressions**
  On utilise les mêmes opérateurs qu'en Java. Une différence existe avec l'opérateur modulo % qui ne peut s'utiliser qu'avec des entiers contrairement à Java (ce qui est un peu étrange puisqu'il s'agit du reste après division entière !)

- **Transtypage**
  On peut utiliser deux formes comme par exemple `(double) i` ou `double (i)` qui sont équivalents.

3 Entrée/sortie simple

La bibliothèque `iostream` contient plusieurs possibilités pour les entrées/sorties. Plusieurs fichiers d'entêtes contiennent des parties de l'interface de cette bibliothèque. On parlera ici des fichiers `<iostream>`, `<iomanip>` et `<fstream>`. Si on veut inclure le premier fichier, on met, en début de fichier la ligne `#include <iostream>` (On peut mettre au lieu cette ligne `#include<iostream.h>` mais la normalisation préconise l'usage de la première.) On peut ajouter après la ligne `using namespace std;` qui permettra d'omettre le préfixe `std::` devant toute utilisation d'un objet de la bibliothèque. Cette dernière inclut notamment l'objet `cout` de la classe `ostream` permettant d'afficher à la console standard ainsi que l'objet `cin` de la classe `istream` qui permet de lire sur la même console.

- **Sortie standard**
  Voici un exemple d'utilisation.

```
cout << "La reponse est " << x << endl;
```

  Chaque élément à afficher est séparé par l'opérateur « (opérateur d'insertion de flux). La constante `endl` vide le tampon de sortie. Pour mettre en forme les sorties, on peut utiliser la bibliothèque `iomanip`. Voici quelques manipulateurs de sortie.

`endl`	Écrit une nouvelle ligne et vide le tampon de sortie
`setw(n)`	Fixe la largeur de champ minimale pour le prochain item
`left`	Justifie à gauche le prochain item lorsque `setw` est utilisé
`setfill(ch)`	Caractère à utiliser pour remplir un champ défini avec `setw`

- **Entrée standard**

  Par défaut, l'entrée se fait au clavier. L'objet `cin` peut être manipulé avec différentes procédures et fonctions. Voici l'exemple d'une fonction qui retourne `true` si le paramètre de sortie contient une valeur entière lue correctement, `false` dans le cas contraire. L'opérateur ≫ se nomme l'opérateur d'extraction de flux.

```
/**
 * Lit un entier.  La lecture se fera jusqu'à ce que l'entrée représente un nombre
 * entier ou fin de flux atteinte.
 * n : paramètre de sortie qui contiendra l'entier lu si pas de fin de flux.
 * Retourne true si une valeur entière valide a été lue, faux sinon.
 */
bool lireInt ( int & n ) {
    const int GROS_NOMBRE = numeric_limits<streamsize>::max();
    cout << "Entrez un nombre entier : " ;
    bool erreur = !(cin >> n); // tentative de lecture d'un nombre entier placé dans n
    bool finFlux = cin.eof();  // fin de flux lorsque fin de fichier ou ^D ou ^Z (Windows)
    // on arrête la lecture lorsque fin de flux ou lecture avec succès
    while ( erreur && !finFlux ) {
            cout << "Saisie incorrecte, recommencez : ";
            cin.clear();
            // ignorer au plus les GROS_NOMBRES prochains caractères jusqu'à <enter>
            cin.ignore( GROS_NOMBRE, '\n' );
            erreur = !(cin >> n); // lire une nouvelle donnée
            finFlux = cin.eof();
    }
    return !(finFlux && erreur); // succès ou échec de lecture
}
```

- **Fichiers**

  Pour ouvrir un fichier en lecture, on déclare un objet de la classe `ifstream` et pour ouvrir un fichier en écriture, on déclare un objet de la classe `ofstream`. Voici un exemple d'ouverture d'un fichier en lecture et d'un autre en écriture.

```
#include <fstream>
#using std
...
ifstream ficEntree ( "donnees.txt", ios :: in );
if ( ! ficEntree ) {
    cerr << "Erreur à l'ouverture du fichier d'entrée.";
    exit ( 1 );
}
ofstream ficSortie ( "resultats.txt", ios :: out );
if ( ! ficSortie ) {
    cerr << "Erreur à l'ouverture du fichier de sortie.";
    exit ( 1 );
}
```

  On manipule ensuite ces deux objets comme on le ferait pour `cin` et `cout`. Par exemple, pour lire l'entier `n` (sans validation comme dans l'exemple plus haut), on aurait `ficEntree >> n;` et pour écrire la chaîne "Le nombre est" suivie de la variable `n`, on ferait `ficSortie << "Le nombre est" << n << endl;`.

4 Instructions

On retrouve les mêmes instructions en C++ qu'en Java : `if`, `while`, `do-while`, `for`, `switch`.

5 Sous-programmes

Ce qui s'appelle méthode en Java se nomme procédure (le type de retour est `void`) ou fonction. La syntaxe est la même qu'en Java. Par contre, la section des paramètres peut être beaucoup plus complexe car C++ offre plus que le passage par valeur de Java : le passage par référence. Dans un fichier, on peut regrouper les différentes entêtes de procédures et fonctions en début de fichier. On appelle ces entêtes le ***prototype*** de la procédure ou fonction. Bien que non obligatoire, cette pratique ajoute de la lisibilité au code. Voici un exemple illustrant les prototypes et un passage de paramètre par copie et un autre par référence.

```
void echanger1 ( int a, int b );
void echanger2 ( int & a, int & b );

void echanger1 ( int a, int b ) {
    int temp = a;
    a = b;
    b = temp;
}

void echanger2 ( int & a, int & b ) {
    int temp = a;
    a = b;
    b = temp;
}

int main () {
    int x = 10;
    int y = 5;
    echanger1 ( x, y );
    cout << x << ' ' << y << endl;
    echanger2 ( x, y );
    cout << x << ' ' << y << endl;
    return 0;
}
```

La fonction `main` affichera

```
10 5
5 10
```

car la procédure `echanger1` utilise des copies de `x` et de `y` (placées dans `a` et `b` respectivement) tandis que la procédure `echanger2` utilise une référence vers les paramètres effectifs. Donc, toute modification apportée au paramètre formel `a` dans `echanger2` sera en fait apportée au paramètre effectif `x` (il en va de même pour `b` et `y`).

On peut omettre le nom des paramètres dans un prototype. Ici, on peut avoir ceci

```
void echanger1 ( int , int  );
void echanger2 ( int &, int & );
```

6 Tableaux

6.1 Une dimension

Un tableau est une suite d'éléments du même type comme en Java. Les éléments sont numérotés à partir de 0. Pour déclarer un tableau d'entiers de taille 22 en Java et en C++, on fait,
`int[] toto = new int [22];` en Java,
`int toto[22];` en C++.

En Java, on peut obtenir la taille du tableau `toto` en faisant `toto.length` alors qu'en C++, on doit la calculer avec l'expression `sizeof(toto)/sizeof(int)`. C++ permet d'initialiser un tableau lors de sa déclaration comme par exemple

```
int tab[10] = {23, -1, 34, 500}; // les 6 derniers éléments seront initialisés à 0
int tableau[] = {1, 4, 8}; // le tableau a une longueur de 3
```

Attention ! En C++, aucune vérification d'indice n'est effectuée. Les deux instructions `tab[11] = 18;` et `tab[-1] = 9;` compilent et s'exécutent ! Leur exécution peut produire des résultats catastrophiques car l'indexation (erronée) donnera accès à des octets n'appartenant pas au tableau considéré mais peut appartenir à une autre variable du programme !

6.2 Plusieurs dimensions

Pour déclarer un tableau à deux dimensions de trois lignes et cinq colonnes on fait `int grille[3][5];`. Les lignes et les colonnes sont numérotées à partir de 0. Pour accéder à l'élément de la dernière ligne et de la première colonne, on fait `grille[2][0];`.

6.3 Chaînes de caractères

On peut définir des chaînes de caractères comme en C en utilisant un tableau de `char`. Le caractère de code 0 détermine la fin de la chaîne. Ce caractère est ajouté automatiquement ou non selon le type de déclaration utilisée. Voici des exemples de déclaration de chaînes de caractères.

```
char chaine[5]; // Une chaîne de longueur 5, non initialisée
char message[] = "Bonjour"; // Chaîne de longueur 8 (le caractère 0 est ajouté à la fin)
char s = {'a', 'l', 'l', 'o'}; // Chaîne de longueur 4 sans le caractère 0 à la fin
```

Il vaut mieux ajouter le caractère de code 0 à la fin de tout tableau de `char`.

En C++, il est plus naturel et plus facile d'utiliser la classe **string** qui s'utilise de façon semblable à la classe String de Java. Cependant, contrairement à Java, les objets de la classe string ne sont pas immuables, donc ils peuvent être modifiés. Pour l'utiliser il faut ajouter la directive suivante : `#include <string>`. Voici quelques exemples.

```
string message = "Bonjour"; // utilisation de l'opérateur =
string toto ( "Bonsoir" ); // utilisation d'un constructeur
int longueur = toto.length(); // longueur de toto
int reponse = message.compare ( "allo" ); // retourne 0 si les deux chaînes sont égales
string chaine = toto + message; // concaténation
getline ( cin, toto ); // lecture d'une chaîne
```

La dernière instruction permet de saisir une chaîne de caractères, la fin de la saisie étant déterminée par la fin de la ligne.

7 Structures

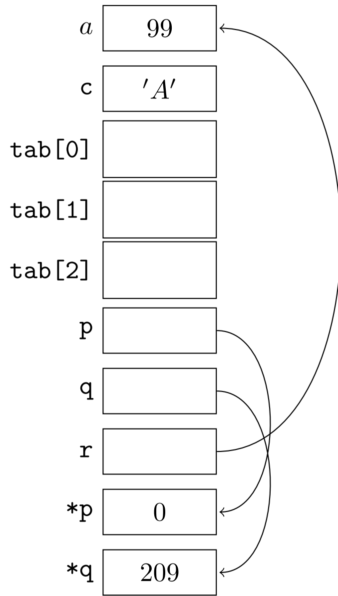
Une structure est un regroupement d'éléments pas nécessairement de même type. C'est ce qu'on nomme parfois, dans d'autres langages des articles (ou enregistrements ou records). C'est la forme primitive en C des classes en C++. Voici un exemple simple.

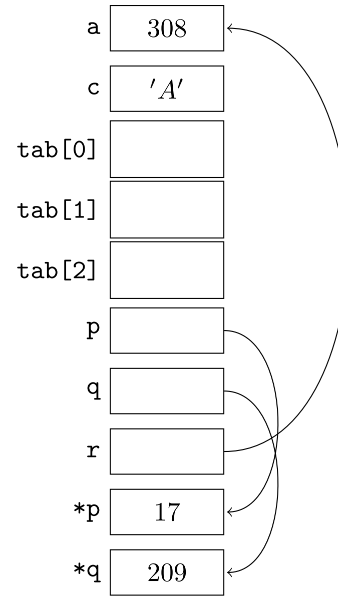
```
struct Date {
    int annee;
    int mois,
    int jour;
};
Date aujourdhui;
aujourdhui.annee = 2007;
```

8 Pointeurs et références

Un pointeur et une référence désignent l'adresse d'un bloc en mémoire statique ou dynamique. Une variable pointeur (ou tout simplement un pointeur) contient l'adresse d'un bloc mémoire. Ces variables permettent, entre autres, de pouvoir réserver de l'espace mémoire lors de l'exécution d'un programme. La forme générale de la déclaration d'une variable pointeur est *type * nom-variable-pointeur*. Voici une série d'instructions en guise d'exemples, suivies de schémas explicatifs.

`int a = 99;`	déclaration de `a` et assignation de son contenu à 99
`char c = 'A';`	déclaration de `c` et assignation de son contenu à 'A' (65)
`int tab[3];`	déclaration du tableau `tab`, de longueur 3
`int * p;`	déclaration de `p` qui contiendra l'adresse d'un entier
`int * q;`	déclaration de `q` qui contiendra l'adresse d'un entier
`p = new int;`	adresse d'un bloc mémoire réservé pour un `int`, donnée à `p`
`q = new int(209);`	adresse d'un bloc mémoire réservé pour un `int`, donnée à `q` et contenu du bloc à 209
`int * r;`	déclaration de `r` qui contiendra l'adresse d'un entier
`r = &a;`	`r` contient l'adresse de la variable `a`
`* p = 17;`	contenu du bloc mémoire pointé par `p` contiendra 17
`* r = a + ( * q );`	contenu du bloc mémoire pointé par `r` contiendra la somme de la valeur de `a` et du contenu du bloc mémoire pointé par `q`

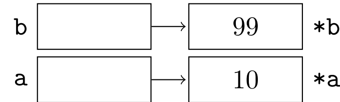




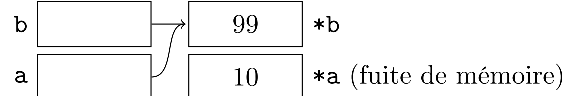
Après avoir exécuté les deux dernières instructions

Il faut bien comprendre ce que signifie l'affectation entre pointeurs. Si on a deux variables `a` et `b` et qu'on fait `a = b;` la variable `a` contiendra une copie du contenu de la variable `b`. Si `a` et `b` sont des variables pointeurs alors `a` contiendra la même adresse que `b`. Voici un exemple.

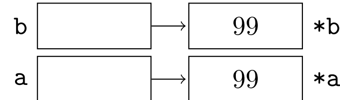
```
int * a = new int ( 10 );
int * b = new int ( 99 );
```



Si on fait `a = b;`



Par contre si on fait `*a = *b;`



Il y a "fuite de mémoire" lorsqu'un bloc mémoire n'est plus accessible par le programme et qu'il n'a pas été libéré pour que le système puisse le réutiliser. En Java, le programmeur n'a pas à s'en préoccuper puisque le système fait intervenir le "ramasse miettes" (garbage collector) au besoin. En C++, il est de la responsabilité du programmeur de récupérer tout espace mémoire qu'il a réservé et dont il ne veut plus. Ceci se fait par l'instruction `delete`. Dans l'exemple ci-dessus, avant de faire `a = b;` il faudrait faire `delete a;`. Pour plus de sûreté, il est recommandé de faire `a = null;` après avoit utilisé `delete` si on n'avait pas donné une nouvelle valeur à `a`, en cas d'utilisation ultérieure.

Attention ! Supposons que l'on ait `int a; int * p = &a;` et que l'on fasse `delete p` ensuite, le compilateur va l'accepter mais le programme peut planter à l'exécution ! Ici, on rend disponible au système la mémoire allouée à la variable `a` qui n'a pas été créée dynamiquement. La variable `a` existe toujours pour le programme mais le système pourra utiliser son espace mémoire pour autre chose.

Une variable peut porter plusieurs noms. Il s'agit d'alias. Si on fait `int a = 27; int & b = a; b = 99; cout << a;`, le programme affichera 99 puisque `a` et `b` désignent la même variable. On dit que `b` est une référence à la variable `a`. **Cette pratique n'est pas recommandée**.

9 Tableaux dynamiques

Dans certains cas, on ne connaît pas la taille du tableau dont nous aurons besoin. On peut réserver l'espace mémoire requis lors de l'exécution du programme. On parle alors de tableau ***dynamique***. Pour ce faire, on déclare un pointeur vers le type des éléments du tableau requis puis on alloue le tableau avec l'opérateur `new`. Voici un exemple de déclaration d'un tableau d'entiers, dynamique de longueur n: `int * ptrTableau = new int [ n ];`. Il faut remarquer l'utilisation d'une variable pour la longueur du tableau. Celle-ci devra contenir une valeur valide lors de l'exécution. La syntaxe de C++ est quelque peu mélangeante lorsqu'il s'agit de pointeurs, de tableaux et de tableaux dynamiques. Voici un bout de code montrant comment on déclare un tableau ***statique*** de `int`, un tableau ***dynamique*** de `int`, un tableau statique de pointeurs vers des `int` et un tableau dynamique de pointeurs vers des `int`, tous de longueur 5, respectivement :

```
int tabStat [ 5 ]; // tableau statique de int
int * tabDyn = new int [ 5 ]; // tableau dynamique de int
int * tabStatDePtr [ 5 ]; // tableau statique de pointeurs
int ** tabDynDePtr = new int * [ 5 ]; // tableau dynamique de pointeurs
```

Supposons que nous voulions placer l'entier 99 dans la première case des tableaux de `int` et un pointeur vers l'entier 99 dans le cas des tableaux de pointeurs vers un `int`. Voici comment faire.

```
tabStat [ 0 ] = 99;
tabDyn [ 0 ] = 99;
tabStatDePtr [ 0 ] = new int ( 99 );
tabDynDePtr [ 0 ] = new int ( 99 );
```

Il ne faut pas oublier de désallouer (libérer) tout tableau dynamique lorsqu'on n'en a plus besoin. Ici, il faudrait faire `delete [] tabDyn;`. Dans le cas du tableau dynamique de pointeurs, il ne faut pas oublier non plus de désallouer toute la mémoire allouée dynamiquement. Pour le tableau `tabDynDePtr`, il faut faire

```
for ( int i; i < 5; ++i ) {
    delete tabDynDePtr [ i ]; // désalloue l'entier tabDynDePtr [ i ];
}
delete [] tabDynDePtr; // désalloue le tableau tabDynDePtr;
```

10 Passage de paramètres

Comme mentionné brièvement à la section 5, il y a deux sortes de passage de paramètre en C++, le passage par copie (ou valeur) et le passage par référence (ou adresse). Habituellement, on utilise un passage par valeur lorsqu'on ne désire pas modifier la valeur du paramètre effectif et on utilise un passage par référence lorsqu'on veut modifier le paramètre effectif. À moins d'avis contraire (utilisation de &), un paramètre est passé par copie. Une exception cependant concernant les tableaux, ils sont toujours passés par référence et on n'utilise pas le symbole & dans ce cas. De ce fait, puisque c'est l'adresse du tableau qui est passée, on ne peut déduire la taille d'un tableau passé en paramètre à l'aide de `sizeof` comme décrit à la section 6 car au lieu d'avoir la taille du tableau en nombre d'octets, on aura la taille d'une adresse en nombre d'octets. C'est pourquoi il faut passer aussi en paramètre à toute fonction manipulant un tableau, la longueur de celui-ci. Si l'on veut éviter de modifier par mégarde le contenu d'un tableau dans une fonction, on ajoute le mot réservé `const`. Voici un exemple.

```
int somme ( const int tableau[], int longueur ) {
    int reponse = 0;
    for ( int i = 0; i < longueur; ++i ) {
        reponse += tableau [ i ];
    }
    return reponse;
}
```

Si on met le prototype de la fonction, on peut choisir l'une des deux façons suivantes

```
int somme ( const int tableau[], int longueur );
int somme ( const int[], int );
```

L'entête de la fonction précédente aurait pu s'écrire, sans modifier le code

`int somme ( const int * tableau, int longueur )`.

En ce qui concerne les `struct` (voir la section 7), leur passage en paramètre se fait par copie à moins d'ajouter le symbole & pour signifier un passage par référence. Il est recommandé, surtout pour de grosses structures, de les passer par référence, mais en utilisant le mot réservé `const` dans le cas où on ne veut pas les modifier par accident. Il en est de même avec les objets que nous verrons dans la prochaine section.

En résumé, hormis les types de base (comme `int`, `bool`, etc.), on devrait toujours passer les paramètres par référence. On ajoute le mot réservé `const` dans le cas où le passage serait logiquement par copie.

Le cas des pointeurs en paramètre est plus délicat. Si un pointeur est passé par copie, ceci veut dire qu'on peut le modifier dans la fonction sans que le paramètre effectif correspondant en soit affecté. Par contre, le bloc mémoire dont le pointeur a la référence n'est pas copié. On peut indiquer au compilateur qu'on ne veut pas modifier le bloc mémoire en utilisant encore `const`, au bon endroit. Voici quatre cas possibles.

`int * p;`	`p` et `*p` peuvent être modifiés
`int * const p;`	`p` ne peut être modifié, `*p` peut être modifié
`const int * p;`	`p` peut être modifié, `*p` ne peut être modifié
`const int * const p;`	`p` et `*p` ne peuvent être modifiés

11 Pointeurs à des sous-programmes

Il est possible d'avoir des pointeurs à des sous-programmes. Le nom d'une fonction correspond en fait à l'adresse de la fonction en mémoire tout comme le nom d'un tableau correspond en fait à l'adresse du premier élément du tableau en mémoire. Voyons d'abord comment on peut déclarer de tels pointeurs en regardant l'exemple qui suit.

```
int somme ( int a, int b ) { return a + b; }
int produit ( int a, int b ) { return a * b; }

int ( * p ) ( int, int ); // déclaration de la variable pointeur p vers une fonction
p = &somme; // ou p = somme
cout << "4 + 7 = " << ( *p ) ( 4, 7 ) << endl;
p = &produit; // ou p = produit
cout << "4 * 7 = " << ( *p ) ( 4, 7 ) << endl;
```

Voici un autre exemple d'utilisation de pointeur de fonction comme paramètre de fonction.

```
bool plusPetit ( int a, int b ) { return a < b; }
bool plusGrand ( int a, int b ) { return a > b; }
void interchanger ( int & a, int & b ) { int temp = a; a = b; b = temp; }

void trier ( int tab [], int longueur, bool ( *enOrdre ) ( int, int ) ) {
    // tri-bulle non optimisé
    for ( int i = longueur - 1; i > 0; --i ) {
        for ( int j = 0; j < i; ++j ) {
            if ( !( *enOrdre ) ( tab [ j ], tab [ j + 1 ] ) ) {
                interchanger ( tab [ j ], tab [ j + 1 ] );
            } // if
        } // for j
    } // for i
} // trier
```

Si on a le tableau `int t[10]` et que l'on veuille le trier de façon croissante on ferait l'appel `trier ( t, 10, &plusPetit )` et si on veut plutôt un tri décroissant, on n'a qu'à faire l'appel `trier ( t, 10, &plusGrand )`. L'intérêt ici est qu'on n'a pas à écrire deux procédures de tri mais une seule.

12 Classes

Pour définir une classe, on utilise deux fichiers. L'un possède l'extension `.hpp` dans lequel ont définit l'interface de la classe contenant les attributs et les fonctions membres (méthodes) de celle-ci. Le second possède l'extension `.cpp` et on y retrouve le code des méthodes. La syntaxe de l'interface d'une classe est la suivante

```
class nomDeClasse {
  public :
     déclaration du ou des constructeur(s)
     déclaration du destructeur s'il y a lieu
     prototypes des fonctions membres
  private :
     attributs et fonctions privés
};
```

On met le code des fonctions membres dans le fichier d'extension `.cpp`. On doit ajouter dans le nom de la fonction (constructeur(s) et destructeur aussi) le nom de la classe suivie de l'***opérateur de résolution de portée*** `::`.

On peut mettre une liste d'initialisation de champs (attributs) dans un constructeur. Ces valeurs initiales, données au moyen de cette liste, n'ont pas à être placées sous forme d'affectation dans le corps de la fonction.

Voici un petit exemple complet illustrant ces concepts. Voici le contenu du fichier `Point.hpp` contenant la spécification de la classe `Point`. Les deux premières lignes et la dernière ne font par partie de la déclaration de l'interface de la classe mais sont des directives au précompilateur lui signifiant de ne pas inclure les lignes de codes jusqu'au `#endif` si ces lignes ont déjà été incluses via un autre fichier du projet.

```
#ifndef POINT_HPP
#define POINT_HPP
class Point {
    public :
        Point();                     // constructeur par defaut
        Point(int, int);             // constructeur
```

```
        int coorX() const;            // observateur
        int coorY() const;            // observateur
        double distance() const;      // observateur
        double distance(const Point) const;  // observateur
        void modifierX ( int x );     // modificateur
        void modifierY ( int y );     // modificateur
        //virtual void afficher() const; // virtual : polymorphisme dynamique
        void afficher() const; // sans virtual : polymorphisme statique
    private :
        int x;         // variable d'instance
        int y;         // variable d'instance
};
#endif
```

Le mot réservé `const`, apparaissant dans les prototypes des fonctions `coorX`, `coorY` et `distance` indiquent que celles-ci ne modifieront pas l'objet. Il est recommandé d'utiliser `const` pour tout observateur (*getter*).

Voici le contenu du fichier `Point.cpp` correspondant.

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

Point :: Point ( int xx, int yy ) :
    x ( xx ),
    y ( yy ) {}

int Point :: coorX() const {
    return x;
}

int Point :: coorY() const {
    return y;
}

double Point :: distance () const {
    Point p;
    return distance ( p );
}

double Point :: distance ( const Point p ) const {
    return sqrt ( pow ( x - p.x, 2.0 ) + pow ( y - p.y, 2.0 ) );
}

void Point :: modifierX ( int x ) {
    this->x = x;
}

void Point :: modifierY ( int y ) {
    this->y = y;
}
```

```
void Point :: afficher () const {
    cout << "(" << x << ", " << y << ")";
}
```

Cet exemple contient des variantes de code possibles. Prenons le constructeur avec deux paramètres, voici des variantes possibles :

```
Point :: Point ( int xx, int yy ) :
    x ( xx ),
    y ( yy ) {}

Point :: Point ( int xx, int yy ) {
    x = xx;
    y = yy;
}

Point :: Point ( int xx, int yy ) {
    (*this).x = xx;
    (*this).y = yy;
}

Point :: Point ( int xx, int yy ) {
    this->x = xx;
    this->y = yy;
}
```

La première façon utilise une liste d'initialisation des variables d'instance. Les troisième et quatrième variantes utilisent la référence à l'objet courant `this`. Puisqu'il s'agit d'un pointeur, on doit l'utiliser comme tel, soit avec `*`, soit avec `->`.

12.1 Constructeur de copie

Lors de l'appel d'une fonction, une copie des paramètres passés par copie est effectuée. S'il s'agit d'un objet, si le constructeur de copie est absent (ou si on n'a pas surchargé l'opérateur =), le constructeur de copie par défaut sera utilisé. Ceci est aussi vrai si on utilise une affectation comme par exemple `Point p (1, 2); Point p2 = p;`. Ce constructeur fait une ***copie en surface***. Ceci veut dire que toute partie dynamique de l'objet ne sera pas copiée, seul le pointeur sera copié. Pour effectuer une ***copie en profondeur***, on doit fournir le code pour ce faire. On suggère de le faire avec un constructeur de copie. Par exemple, le constructeur de copie pour la classe Point (celui-ci n'est pas nécessaire puisqu'il n'y a pas de partie dynamique) serait le suivant :

```
Point :: Point ( const Point & p ) {
    x = p.x;
    y = p.y;
}
```

Il faut noter que le paramètre du constructeur de copie doit être passé par référence afin d'éviter la récursion infinie. Le fait d'ajouter `const` montre bien que l'objet copié ne sera pas modifié même s'il est passé par référence.

Prenons un autre exemple de classe ayant une partie dynamique comme ce qui suit :

```
class Tableau1D {
    public :
        Tableau1D ( int longueur, int valInit = 0 );
        Tableau1D ( const Tableau1D & ); // constructeur de copie
        ...
    private :
        int lon;
        int * tab;
 };
Tableau1D :: Tableau1D ( int longueur, int valInit ) {
    lon = longueur;
    tab = new int [lon];
    for ( int i = 0; i < lon; ++i ){
        tab[i] = valInit;
    }
}
Tableau1D :: Tableau1D ( const Tableau1D & t ) {
    lon = t.lon;
    tab = new int [lon];
    for ( int i = 0; i < lon; ++i ){
        tab[i] = t.tab[i];
    }
}
```

Si on n'avait pas de constructeur de copie, toute copie d'objet ne copierait que l'attribut `lon` et l'attribut pointeur vers le tableau dynamique. Il n'y aurait pas de nouveau tableau dynamique. L'attribut `tab` de chacun des deux objets pointerait vers le même tableau en mémoire dynamique. Une copie en profondeur doit copier aussi la partie dynamique.

Si on fournit un constructeur de copie parce qu'une classe a des membres dynamiques, il faut aussi fournir la surcharge de l'opérateur d'affectation `=` (voir la section 13 plus loin sur la surcharge des opérateurs).

12.2 Destructeur

Dans une fonction, au moment de sortir de celle-ci, les copies effectuées des paramètres passés par copie sont détruites. S'il s'agit d'objets et s'il n'y a pas de destructeur, le destructeur par défaut sera utilisé. Il s'agira d'une ***destruction en surface***. Donc, si l'objet ne contient pas de partie dynamique, ce sera suffisant. Dans le cas contraire, on devrait fournir le code à exécuter pour effectuer une ***destruction en profondeur*** pour éviter ainsi les fuites de mémoire. Dans l'exemple précédent, il aurait l'allure suivante :

```
Tableau1D :: ~Tableau1D () {
    delete[] tab;
    tab = 0; // plus securitaire
}
```

12.3 Héritage et polymorphisme

On peut dériver une classe d'une autre classe. Reprenons l'exemple de la classe `Point`. Supposons que nous voulons ajouter un attribut supplémentaire, la couleur du point. On crée une nouvelle classe qui utilisera la classe `Point`. Voici cette nouvelle classe.

```
class PointColore : public Point {
    public :
        PointColore();
        PointColore(int x, int y);
        PointColore(int x, int y, int coul);
        int laCouleur() const;
        void modifierCouleur ( int coul );
        void afficher() const; // est virtual si afficher de Point l'est
    private :
        int couleur;
};
```

```
PointColore:: PointColore() {
    couleur = 0;
}
PointColore:: PointColore (int x, int y) : Point ( x, y ) { // appel du constructeur Point
    couleur = 0;
}
PointColore:: PointColore (int x, int y, int coul) : Point ( x, y ) {
    couleur = coul;
}
int PointColore:: laCouleur () const {
    return couleur;
}
void PointColore:: modifierCouleur ( int coul ) {
    couleur = coul;
}
void PointColore:: afficher () const {
    Point::afficher(); // appel de la méthode de la classe supérieure Point
    cout << " couleur " << couleur;
}
```

Voici un exemple d'utilisation.

```
int main () {
    Point p;             // p <- (0,0)
    Point p2 ( 19, 45 ); // p <- (19,45)
    cout << p.coorX() << endl;
    cout << p.distance ( p2 ) << endl; // distance entre p et p2
    Point p3 = p2; // copie en surface (ok ici)
    p3.afficher();
    cout << endl;

    PointColore pc1 ( 1, 2, 3 );
    Point p4 = pc1;  // un PointColore est un Point
    // pc1 = p4; // non valide // un Point n'est pas nécessairement un PointColore
    p4.afficher(); // Quelle fonction afficher sera utilisée ?
    cout << endl;

    // Illustration du polymorphisme

    Point * ptrPoint = new Point ( 5, 29 ); // un Point est un Point
```

```
    Point * ptrPoint2 = new PointColore ( 2, 4, 63 ); // un PointColore est un Point
    PointColore * ptrPointCol = new PointColore ( 1, 2, 3 ); // un PointColore est un PointColore
    // PointColore * ptrPointCol2 = new Point ( 1, 2, 3 ); // non valide
    // un Point n'est pas nécessairement un PointColore
    ptrPoint -> afficher(); // Quelle fonction afficher sera utilisée ?
    cout << endl;
    ptrPoint2 -> afficher(); // Quelle fonction afficher sera utilisée ?
    cout << endl;
    ptrPointCol -> afficher(); // Quelle fonction afficher sera utilisée ?
    cout << endl;
}
```

L'exécution de la fonction `main` affiche ceci :

```
0
48.8467
(19, 45)
(1, 2)
(5, 29)
(2, 4) couleur 63
(1, 2) couleur 3
```

Ici, les variables `p`, `p2`, `p3` et `p4` sont des variables de type `Point` et `pc1` est une variable de type `PointColore`. Lorsque la fonction `afficher()` est invoquée avec l'une ou l'autre de ces variables, on utilise le type de la variable pour déterminer quelle fonction sera utilisée. Les appels `p->afficher()`, `p2->afficher()`, `p3->afficher()` et `p4->afficher()` font référence à la même fonction, soit celle déclarée dans la classe `Point` même si, dans le cas de la variable `p4` on a fait l'affectation `Point p4 = pc1` où `pc1` est une variable de type `PointColore`. L'appel `pc1->afficher()` fait référence à la fonction de la classe `PointColore` puisque la variable `pc1` est de type `PointColore`.

Par contre, le cas des pointeurs vers un objet de la classe `Point` est différent. On peut considérer que l'espace mémoire vers lequel pointe le pointeur peut être de type `Point` ou de type `PointColore` selon le constructeur invoqué à l'exécution. Pour que le type dynamique (celui utilisé lors du `new`, donc à l'exécution) soit choisi pour faire exécuter la fonction qui lui est associée, on doit faire précéder l'entête de la fonction par le mot `virtual`. Comme ici, si on met le mot `virtual` devant la fonction `afficher` de `Point`, toutes les classes dérivées qui masqueront (redéfiniront) la fonction `afficher` seront elles aussi considérées comme `virtual` (même si le mot `virtual` n'apparaît pas) et ce sera le type dynamique qui sera utilisé. Si la fonction de `Point` n'est pas `virtual`, ce sera le type statique (le type de la variable à la déclaration) qui sera utilisé. Lorsqu'on crée une nouvelle classe qui hérite d'une classe qui contient des fonctions virtuelles, on recommande fortement d'utiliser le mot `virtual` devant toute fonction virtuelle redéfinie de la classe parent même si c'est facultatif. À ce moment-là, il n'est pas nécessaire d'examiner l'arborescence de la hiérarchie des classes pour déterminer si une fonction est virtuelle ou non.

12.4 Classes abstraites

Une classe abstraite permet de définir des attributs et des méthodes de base. On ne peut créer d'objet d'une classe abstraite. On peut, par contre, dériver des classes, abstraites ou concrètes, d'une classe abstraite. Une classe concrète dérivant d'une classe abstraite devra définir les fonctions virtuelles, dites pures, de la classe de laquelle elle dérive. Voici des classes permettant d'illustrer ces concepts et leur implantation en C++.

```
class FigureGeometrique {
    public :
        virtual int laCouleur() const;
        virtual bool estVide () const;
```

```
        virtual void changerCouleur ( int c );
        virtual void changerContenu ( bool r ); //true = plein
    private :
        int couleur;
        bool estPleine;
};
int FigureGeometrique :: laCouleur() const {
    return couleur;
}
bool FigureGeometrique :: estVide () const {
    return !estPleine;
}
void FigureGeometrique :: changerCouleur ( int c ) {
    couleur = c;
}
void FigureGeometrique :: changerContenu ( bool r ) {
    estPleine = r;
}
```

```
class Figure2D : public FigureGeometrique {
    public :
        virtual Point position() const;
        virtual void changerPosition ( Point p );
        virtual double surface () const = 0; // fonction virtuelle pure
        virtual void afficher() const = 0;   // fonction virtuelle pure
    private :
        Point pos;
};

Point Figure2D :: position() const {
    return pos;
}
void Figure2D :: changerPosition ( Point p ) {
    pos = p;
}
```

```
class Figure3D : public FigureGeometrique {
    public :
        virtual Point3D position() const;
        virtual void changerPosition ( Point3D p );
        virtual double surface () const = 0; // fonction virtuelle pure
        virtual double volume () const = 0;  // fonction virtuelle pure
        virtual void afficher() const = 0;   // fonction virtuelle pure
    private :
        Point3D pos;
};

Point3D Figure3D :: position() const {
    return pos;
}
void Figure3D :: changerPosition ( Point3D p ) {
```

```
    pos = p;
}
```

```
class Cercle : public Figure2D {
    public :
        Cercle ( Point centre, double r );
        virtual double surface () const;
        virtual void afficher() const;
    private :
        double rayon;
};

#define PI 3.14159265358979323846264338332
Cercle :: Cercle ( Point centre, double rayon ) {
    this->rayon = rayon;
    changerPosition ( centre );
    changerCouleur ( 0 );
    changerContenu ( false );
}
double Cercle :: surface () const {
    return rayon * rayon * PI;
}
void Cercle :: afficher () const {
    cout << "Position du centre : ";
    position().afficher();
    cout << ", rayon : " << rayon;

}
```

Ici, les classes `FigureGeometrique` et `Cercle` sont des classes concrètes tandis que les classes `Figure2D` et `Figure3D` sont abstraites. Une classe est abstraite si au moins une de ses fonctions est virtuelle pure comme par exemple la fonction surface dans la classe `Figure2D`. Pour rendre une fonction virtuelle pure, on ajoute `= 0` à la suite de son prototype dans l'interface de la classe. Une classe qui dérive d'une classe abstraite sera elle-même abstraite si elle n'implémente pas toutes les fonctions virtuelles pures de la classe de laquelle elle dérive.

12.5 Héritage multiple

On peut créer des classe qui héritent de plusieurs classes. On parle alors d'héritage multiple. Par exemple, supposons que nous ayions déclaré une classe `Employe` puis une classe `Etudiant`. Un auxiliaire d'enseignement est un étudiant qui est aussi un employé. On pourrait donc définir une classe `AuxiliaireEnseignement` qui hériterait des deux classes `Employe` et `Etudiant`. En supposant que ces deux classes soient déclarées, on définirait la classe `AuxiliaireEnseignement` de cette façon :

```
class AuxiliaireEnseignement : public Employe, public Etudiant { ...
```

L'héritage multiple est beaucoup plus complexe que l'héritage simple. Certains auteurs vont jusqu'à dire que celui-ci ne devrait jamais être utilisé. Dans l'exemple ci-dessus on pourrait avoir plutôt une classe interne à la classe `AuxiliaireEnseignement` que l'on pourrait nommer `PartieEmploye` qui hériterait simplement de la classe `Employe` :

```
class AuxiliaireEnseignement : public Etudiant {
    ...
```

```
        private :
            class PartieEmploye;
            PartieEmploye * ptrEmploye;
};
class AuxiliaireEnseignement :: PartieEmploye : public Employe { ...
```

Ceci permet donc d'éviter l'héritage multiple.

12.6 Fonctions et classes amies

Une fonction amie (`friend`) d'une classe se définit hors de la portée de cette classe même si celle-ci se déclare dans la définition de la classe. Cette fonction amie n'est pas un membre de la classe. Elle a tout de même accès aux membres privés de la classe en question. On peut aussi déclarer une classe entière `friend` d'une autre classe. Bien que pratique dans certains cas, on ne devrait pas abuser des fonctions et classes amies. Nous verrons, dans la section suivante, l'utilité de fonctions amies dans le cas de la surcharge de certains opérateurs.

13 Surcharge d'opérateurs

Presque tous les opérateurs peuvent être surchargés. Ceux ne le pouvant pas sont : `.`, `.*`, `::`, ?:, et `sizeof`. Le nombre d'opérandes des opérateurs surchargés doit rester le même. Voici un exemple de classe qui définit le type `Fraction` et les différentes opérations de base que l'on peut effectuer sur celles-ci.

```
class Fraction {
    friend ostream & operator << ( ostream &, const Fraction & ); // pour l'affichage
    public :
        Fraction ( int num = 0, int denom = 1 ); // Constructeur
        int num () const; // retourne le numérateur;
        int denom () const; // retourne le dénominateur;
        Fraction operator - () const; // négation d'une fraction
        Fraction operator - ( const Fraction & ) const; // soustraction de deux fractions
        Fraction operator + ( const Fraction & ) const; // addition de deux fractions
        Fraction operator * ( const Fraction & ) const; // multiplication de deux fractions
        Fraction operator / ( const Fraction & ) const; // division de deux fractions
        const Fraction & operator = ( const Fraction & ); // const pour empêcher (a=b)=c
        Fraction inverse () const;
    private :
        int numerateur;
        int denominateur;
};
```

```
ostream & operator << ( ostream & sortie, const Fraction & f ) {
    sortie << f.numerateur << "/" << f.denominateur;
    return sortie;
}
Fraction :: Fraction ( int num, int denom ) {
    numerateur = num;
    denominateur = denom;
}
int Fraction :: num () const { // retourne le numérateur;
    return numerateur;
```

```
}
int Fraction :: denom () const { // retourne le dénominateur;
    return denominateur;
}
Fraction Fraction :: operator - () const {
    return Fraction ( -numerateur, denominateur );
}
Fraction Fraction :: operator + ( const Fraction & f ) const {
    return Fraction ( numerateur * f.denominateur +
                      f.numerateur * denominateur, denominateur * f.denominateur );
}
Fraction Fraction :: operator - ( const Fraction & f ) const {
    return (*this) + (-f);
}
Fraction Fraction :: operator * ( const Fraction & f ) const {
    return Fraction ( numerateur * f.numerateur, denominateur * f.denominateur );
}
Fraction Fraction :: operator / ( const Fraction & f ) const {
    return (*this) * f.inverse();
}
const Fraction & Fraction :: operator = ( const Fraction & f ) {
    if ( &f != this ) { // ne rien faire si a=a
        numerateur = f.numerateur;
        denominateur = f.denominateur;
    }
    return *this;
}
Fraction Fraction :: inverse () const {
    return Fraction ( denominateur, numerateur );
}
```

Toutes les fonctions membres d'une classe ont comme premier opérande un objet de cette classe. Il n'y a pas de restrictions quant au type des autres opérandes. L'opérateur d'insertion de flux `<<` s'utilise, de façon naturelle, avec un opérande gauche de la classe `ostream`. De ce fait, on ne pourra surcharger cet opérateur en tant que fonction membre de la classe. On doit donc en faire une fonction hors classe qui devra utiliser les méthodes d'accès de la classe. Voici cette version :

```
ostream & operator << ( ostream & sortie, const Fraction & f ) {
        sortie << f.num() << "/" << f.denom();
    return sortie;
}
```

Cette façon de faire, bien que correcte, n'est pas très performante. C'est pourquoi il est préconisé d'en faire une fonction amie qui pourra alors accéder à tout membre de la classe dans passer par les méthodes d'accès comme il est fait dans l'exemple précédent.

Il faut mentionner les notions de ***valeur gauche*** (lvalue) et ***valeur droite*** (rvalue) pour bien comprendre. Lorsqu'on fait une affectation, l'expression écrite à gauche du symbole = est considérée comme la destination (contenant) ou valeur gauche qui recevra le résultat de l'évaluation de l'expression à droite du symbole = qui elle est considérée comme la source ou valeur droite. Le nom d'une variable est considéré comme une valeur gauche. Les constantes sont considérées comme des valeurs droites. Il faut noter qu'une valeur gauche peut être utilisée comme valeur droite mais non l'inverse. Si `a` et `b` désignent deux variables entières on peut avoir `a = b`, `b = 8` mais on ne peut écrire `8 = b` car `8` n'est pas une valeur gauche.

Le paragraphe précédent permet de comprendre l'effet de l'utilisation du symbole `const` dans le type de retour de certaines fonctions comme la surcharge de l'opérateur `=` dans la classe `Fraction`. Ceci indique que le résultat retourné par l'opérateur `=` est une constante, donc une valeur droite qui ne peut donc être utilisée pour recueillir une nouvelle valeur. C'est pourquoi l'expression `( a = b ) = c` ne compilera pas puisque l'expression `a = b` retourne une constante.

Une autre remarque importante concerne le symbole `&`. Son absence fera en sorte que la valeur de l'expression retournée par la fonction sera copiée par le constructeur de copie, et c'est cette copie qui sera retournée. Dans le cas de l'opérateur `=` de la classe `Fraction`, la valeur retournée est l'objet qui invoque la fonction et il n'est pas une variable locale. Il est donc inutile d'en faire une copie avant de le retourner. Dans le cas ou le résultat retourné est une expression ou une variable locale, le résultat retourné ne doit pas l'être par référence car tout espace mémoire alloué localement lors de l'exécution d'une fonction n'est plus accessible après son exécution et que, de ce fait, si l'on retourne la référence à une variable créée localement, il y aura problème lors de l'exécution du programme qui invoquera cette fonction.

Voici l'exemple de la section 12.1, augmenté.

```
class Tableau1D {
    public :
        Tableau1D ( int longueur, int valInit = 0 );
        Tableau1D ( const Tableau1D & ); // constructeur de copie
        ~Tableau1D ();
        const Tableau1D & operator = ( const Tableau1D & );
        int & operator[] ( int );
        const int & operator[] ( int ) const;  // pour les Tableau1D const
     private :
        int lon;
        int * tab;
};
```

```
#include <cassert>
...
const Tableau1D & Tableau1D :: operator = ( const Tableau1D & t ) {
    if ( & t != this ) { // pour éviter l'auto-affectation
        // vérifier la longueur de la source t et celle de la destination (*this)
        if ( lon != t.lon ) {
            delete[] tab; // récupérer l'espace utilisé par la destination this
            lon = t.lon;  // nouvelle longueur de la destination this
            tab = new int [ lon ];
        }
        // destination et source ont la même longueur
        for ( int i = 0; i < lon; ++i ) {
            tab[i] = t.tab[i];
        }
    }
    return *this;
}
int & Tableau1D :: operator[] ( int i ) {
    assert ( 0 <= i && i < lon );
    return tab[i];
}
const int & Tableau1D :: operator[] ( int i ) const {
    assert ( 0 <= i && i < lon );
    return tab[i];
```

```
}
```

Ici, en ce qui concerne l'opérateur `[]`, il est obligatoire de transmettre le résultat par référence en utilisant le symbole `&` pour que le résultat de l'opérateur `[]` soit une valeur gauche (pouvant donc recevoir un résultat).

14 Les fonctions et classes génériques

Comme en Java, il est possible de créer des classes et des fonctions génériques appelées "template" en C++. On appelle aussi les classes ou fonctions génériques classes ou fonctions ***modèles***. Ces modèles font intervenir un ou plusieurs paramètres formels génériques. On utilise les mots clés ***class*** ou ***typename*** pour déclarer un type générique.

14.1 Fonctions génériques

Certains traitements demeurent identiques peu importe le type des données auquel ils s'adressent. On peut surcharger la dite fonction pour chacun des types voulus (voir plusieurs exemples dans la classe Math de Java). Par exemple, faire l'affichage des éléments d'un tableau consiste en une boucle qui affiche chacun des éléments. Voici comment en faire une fonction générique en Java et en C++.

```
public static <T> void afficher ( T[] t ) {
    for ( int i = 0; i < t.length; ++i ) {
        System.out.print ( t[i] + " " );
    }
    System.out.println();
}

template < class T >
void afficher ( const T * tab, const int longueur ) {
    for ( int i = 0; i < longueur; ++i ) {
        cout << tab [ i ] << " ";
    }
    cout << endl;
}
```

Ici, le paramètre formel `T` désigne le type générique. Voici un exemple d'utilisation de la fonction C++.

```
int t1[] = {1, 2, 3, 4, 5, 6, 7, 8, 9};
afficher ( t1, 9 );
string t2[] = { "a", "b", "c"};
afficher ( t2, 3 );
```

Le type `T` est remplacé par le bon type lors de la compilation.

La version en Java ne permet que des tableaux d'objets. Les tableaux de types primitifs ne peuvent être affichés par la méthode. Par contre, les tableaux de types primitifs peuvent l'être avec la version C++.

14.2 Classes génériques

Reprenons l'exemple de la section 12.1 qui définit des tableaux à une dimension de `int`. On ne peut que manipuler des tableaux de `int`. En rendant la classe générique, on pourra déclarer des `Tableau1D` contenant n'importe quel type d'éléments.

```
#include <cassert>
template<typename T>
class Tableau1D {
    public :
        Tableau1D ( int longueur = 0, T valInit = 0 );
        Tableau1D ( const Tableau1D<T> & ); // constructeur de copie
        ~Tableau1D ();
        int longueur () const;
        const Tableau1D<T> & operator = ( const Tableau1D<T> & );
        T & operator[] ( int );
        const T & operator[] ( int ) const;  // pour les Tableau1D const
    private :
        int lon;
        T * tab;
};

template<typename T>
Tableau1D<T> :: Tableau1D ( int longueur = 0, T valInit = 0 ) {
    lon = longueur;
    tab = new T [lon];
    for ( int i = 0; i < lon; ++i ){
        tab[i] = valInit;
    }
}

template<typename T>
Tableau1D<T> :: Tableau1D ( const Tableau1D<T> & t ) {
    lon = t.lon;
    tab = new T [lon];
    for ( int i = 0; i < lon; ++i ){
        tab[i] = t.tab[i];
    }
}

template<typename T>
Tableau1D<T> :: ~Tableau1D () {
    delete[] tab;
    tab = 0; // plus securitaire
}

template<typename T>
int Tableau1D<T> :: longueur () const {
return lon;
}

template<typename T>
const Tableau1D<T> & Tableau1D<T> :: operator = ( const Tableau1D<T> & t ) {
    if ( & t != this ) { // pour éviter l'auto-affectation
        // vérifier la longueur de la source t et celle de la destination (*this)
        if ( lon != t.lon ) {
            delete[] tab; // récupérer l'espace utilisé par la destination this
```

```
            lon = t.lon;  // nouvelle longueur de la destination this
            tab = new T [ lon ];
        }
        // destination et source ont la même longueur
        for ( int i = 0; i < lon; ++i ) {
            tab[i] = t.tab[i];
        }
    }
    return *this;
}

template<typename T>
T & Tableau1D<T> :: operator[] ( int i ) {
    assert ( 0 <= i && i < lon );
    return tab[i];
}

template<typename T>
const T & Tableau1D<T> :: operator[] ( int i ) const {
    assert ( 0 <= i && i < lon );
    return tab[i];
}
```

Il faut remarquer le préfixe de chaque corps de fonction : `template<typename T>`. Aussi, le nom de la classe est `Tableau1D<T>`. Voici un exemple d'utilisation.

```
template < class T >
void afficher ( const Tableau1D<T> tab ) {
    for ( int i = 0; i < tab.longueur(); ++i ) {
        cout << tab [ i ] << " ";
    }
    cout << endl;
}

...
Tableau1D<string> tab( 10, "");
for ( int i = 0; i < tab.longueur(); ++i ) {
    string n = "abc";
    tab[i] = n + ((char) (i + '0'));
}
afficher ( tab );
cout << endl;

Tableau1D<int> tab2 ( 20 );
afficher ( tab2 );
```

On remarquera ici l'utilisation d'une fonction générique pour l'affichage de n'importe quel tableau `Tableau1D<T>`.

Références

- Deitel et Deitel, *C++, Comment programmer - Édition révisée*, 3ème édition, Les éditions Raynald Goulet, 2003.
- Scott Meyers, *Effective C++, Third Edition - 55 Specific Ways to Improve Your Programs and Designs*, Addison-Wesley, 2005.
- Cay Horstmann, Timothy Budd, *Big C++*, John Wiley & Sons, 2005.
- Claude Delannoy, *Apprendre le C++*, Best of Eyroles, 2008.
- Philippe Gabrini, *Structures de données avancées avec la STL*, Loze-Dion, 2005.
- www.fredosaurus.com/notes-cpp/io/omanipulators.html