

---

# **Introduction au c++ Documentation**

***Version 1.5***

**Bruno Gas**

**mars 09, 2020**



---

## Table des matières:

---

<b>1</b>	<b>Mon premier programme en C++</b>	<b>1</b>
1.1	Explications pas à pas . . . . .	1
1.2	Saisir des données au clavier . . . . .	2
1.3	Compiler le programme . . . . .	3
1.4	Ecrire une fonction . . . . .	3
1.4.1	Déclaration du prototype . . . . .	4
<b>2</b>	<b>Objets en C++</b>	<b>5</b>
2.1	Pointeurs et objets . . . . .	7
2.1.1	Allocation dynamique . . . . .	8
2.2	Passage d'objets en argument de fonction . . . . .	8
2.2.1	Passage d'argument par valeur . . . . .	8
2.2.2	Passage d'argument par adresse . . . . .	9
2.2.3	Passage d'argument par référence . . . . .	10
2.3	Constructeurs et destructeurs . . . . .	11
2.3.1	Constructeur par défaut . . . . .	12
2.3.2	Constructeur spécifique . . . . .	13
2.3.3	Surcharge du constructeur . . . . .	13
2.3.4	Constructeurs et arguments par défaut . . . . .	14
2.3.5	Constructeur par copie . . . . .	16
2.3.6	Constructeur de conversion . . . . .	17
<b>3</b>	<b>Surcharge des fonctions, méthodes et opérateurs</b>	<b>19</b>
3.1	Surcharge de fonctions et principe de la surcharge . . . . .	19
3.2	Surcharge de méthodes . . . . .	20
3.3	Surcharge des opérateurs . . . . .	20
3.3.1	Définition en tant que méthode de la classe . . . . .	21
3.3.2	Surcharge de l'opérateur d'addition . . . . .	23
3.3.3	Surcharge de l'opérateur d'insertion de flux . . . . .	24
3.4	Les tableaux et la surcharge de l'opérateur crochet [ ] . . . . .	25
3.4.1	Exemple : la classe <code>Array</code> . . . . .	26
<b>4</b>	<b>Héritage</b>	<b>31</b>
4.1	Accessibilité des attributs et méthodes . . . . .	32
4.1.1	Accessibilité par les méthodes de la classe descendante . . . . .	32
4.1.2	Accessibilité par les fonctions extérieures . . . . .	32
4.2	Visibilité des méthodes . . . . .	32

4.3	Constructeurs et destructeurs . . . . .	33
4.4	Héritage et accessibilité . . . . .	35
4.5	Conversions . . . . .	36
<b>5</b>	<b>Polymorphisme</b>	<b>39</b>
5.1	Résolution statique de liens (ligature statique) . . . . .	39
5.2	Résolution dynamique de liens (ligature dynamique) . . . . .	40
5.2.1	Le mécanisme . . . . .	41
5.2.2	Règles . . . . .	41
5.2.3	Utilisation avec les tableaux . . . . .	42
5.3	Fonctions virtuelles pures et classes abstraites . . . . .	44
<b>6</b>	<b>Compléments C/C++</b>	<b>47</b>
6.1	Compilation séparée . . . . .	47
6.1.1	Architecture . . . . .	47
6.1.2	Fichiers d'en-tête . . . . .	48
6.1.3	Librairies . . . . .	49
6.1.4	Questions/réponses . . . . .	49
6.2	Compilation conditionnelle . . . . .	50
6.3	Chaînes de caractères . . . . .	51
6.3.1	Gestion dynamique . . . . .	52

# CHAPITRE 1

---

## Mon premier programme en C++

---

Dans ce cours, nous nous considérons sous un système Unix, Linux ou OSX (Mac), nous disposons d'un éditeur de texte et d'une console de commande. Bien entendu, tous ce qui est écrit est valable sous environnement windows et/ou tout environnement de développement, sur tout système d'exploitation.

Considérons le programme ci-dessous :

```
#include <iostream>

int main( void ) {
    std::cout << "Hello world!" << std::endl;
    return 0;
}
```

Sur la console, on demande la compilation du source, puis l'exécution du programme :

```
$ > cc intro.cc -lstdc++
$ > ./a.out
$ Hello world!
$ >
```

## 1.1 Explications pas à pas

```
int main( void ) {...}
```

Il s'agit de la fonction principale. L'exécution du programme commence par l'exécution de cette fonction dont la présence est obligatoire. Toutes les fonctions comportent un encadrement par accolades `{...}` des instructions à exécuter. Le type `int` placé devant le nom de la fonction signifie que la fonction doit retourner une valeur entière à la fin de son exécution. Ce retour est assuré par la dernière ligne :

```
    return 0;
}
```

La première ligne de commande

```
std::cout << "Hello world!" << std::endl;
```

affiche à l'écran la chaîne de caractères *Hello world!*. **cout**, forme condensée de *console out* (sortie console), désigne le flux de sortie standard qui envoie des données à l'écran. La présence du préfixe **std ::** est obligatoire (pour le moment) car *std* spécifie qu'il faut employer l'objet **cout** qui appartient à l'espace de nommage **std** qui contient toutes les fonctions standards.

Les deux chevrons << désignent un opérateur appelé *opérateur d'insertion de flux*. Il indique qu'il faut envoyer la chaîne de texte « *Hello world!* » dans le flux *std :: cout*.

Il est possible d'envoyer consécutivement plusieurs données à l'affichage en utilisant autant de fois qu'il le faut l'opérateur <<. C'est le cas ici avec l'envoi de l'objet **std :: endl** à la suite de la chaîne de caractères. Cet objet désigne un *retour à la ligne* (end of line). La commande suivante ajoute deux retour à la ligne :

```
std::cout << "Hello world!" << std::endl << std::endl;
```

Dans ce mini programme nous avons pu utiliser le flux **cout** car nous avons préalablement inséré en tête de programme le fichier d'en-tête qui contient la déclaration :

```
#include <iostream>
```

Cette commande est une *directive* qui s'adresse au préprocesseur (un programme qui effectue des traitements préliminaires sur le fichier source avant de l'envoyer à la compilation). Elle permet d'insérer dans le fichier source, donc avant sa compilation, le fichier d'en-tête *iostream* qui contient la déclaration d'objets comme **cout**, **endl**, **cin** (que nous verrons plus bas), etc.

Si nous ajoutons à la suite de cette directive la ligne suivante :

```
using namespace std;
```

alors nous n'avons plus besoin d'utiliser le préfixe **std ::**. Nous indiquons en effet que nous travaillons explicitement dans l'espace de nom **std** et qu'il n'est plus nécessaire de le spécifier à chaque emploi d'un objet de cet espace de nom. Le nouveau programme peut alors s'écrire :

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

int main( void ) {
    cout << "Hello world!" << endl;
    return 0;
}
```

## 1.2 Saisir des données au clavier

Soit à récupérer une valeur entière tapée au clavier pour l'afficher ensuite.

On utilise pour cela le flux d'entrée standard et cette fois l'opérateur d'extraction de flux (les deux chevrons dans l'autre sens). Avant son affichage la donnée saisie devra être rangée dans une variable de type **int** que nous appellerons **data**.

Voici le programme :

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

int main( void ) {
    int data;                // déclaration de la variable entière data
    cout << "tapez un nombre entier: "; // affiche un message de sollicitation
    cin >> data;              // attend la donnée du clavier
    cout << "la donnée saisie au clavier est: " << data << endl;    // affiche la
    ↪ donnée saisie

    return 0;
}
```

Au passage, les lignes de commentaires en C++ sont signalées par le symbole `//`. Pour commenter un bloc de plusieurs lignes, on utilisera `/*...*/`.

Notez la simplicité lorsque l'on compare aux instructions équivalentes en langage C : on n'utilise plus les fonctions `printf()` et `scanf()`, plus le formatage des données (`%d`), plus non plus l'opérateur d'adressage (`&`).

## 1.3 Compiler le programme

Dans un environnement intégré, les librairies et chemins d'accès aux fichiers d'en-tête sont déclarés. Il n'y a donc qu'à utiliser le bouton de compilation.

Sous unix/linux on appelle le compilateur (`cc` dans notre exemple). Le nom du programme source doit avoir `.cc` pour extension pour que le compilateur compile en C++. Il faut également spécifier la librairie standard (`stdc++` ici) pour que l'édition de lien puisse se faire à l'issue de la compilation et ainsi qu'un exécutable soit généré.

Par défaut sous unix/linux l'exécutable généré a pour nom `a.out`. Si l'on veut donner un nom particulier au programme, par exemple `hello` :

```
$ > cc intro.cc -o hello -lstdc++
$ > ./hello
$ Hello world!
$ >
```

Le programme de saisie d'une donnée au clavier doit donner quelque chose comme ceci :

```
$ > cc intro.cc -o saisir -lstdc++
$ > ./saisir
$ tapez un nombre entier: 12
$ la donnée saisie au clavier est: 12
$ >
```

## 1.4 Ecrire une fonction

La fonction principale porte le nom réservé `main()`. Comme en langage C, et comme dans tout langage, on ne se contente surtout pas de la fonction `main()`. On écrit des fonctions afin de modulariser les traitements.

L'exemple ci-dessous reprend la saisie d'une donnée tapée au clavier, mais dans une fonction :

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

(suite sur la page suivante)

```
int saisir();

int main( void ) {
    int data;

    data = saisir();

    cout << "la donnée saisie au clavier est: " << data << endl;
}

/*
 * @function saisir()
 * @brief saisie d'une valeur entière
 * @return retourne l'entier saisi au clavier
 */
int saisir() {
    int data;
    cout << "Tapez un nombre entier: ";
    cin >> data;
    return data;
}
```

Observez les lignes de commentaires qui précèdent la fonction. Ces lignes obéissent à un format particulier et utilisent des mots clés appropriés (**@function**, **@brief**, **@return**) pour décrire la fonction. Cela permet tout simplement d'utiliser des logiciels de génération de documentation automatique afin de documenter le code. Ce n'est pas obligatoire mais vivement conseillé au point de devoir devenir un réflexe lorsque l'on écrit du code.

Plusieurs remarques.

### 1.4.1 Déclaration du prototype

La ligne que l'on peut voir en haut du programme, avant la fonction **main()** :

```
int saisir();
```

déclare le prototype de la fonction **saisir()**. Cette déclaration du prototype de la fonction est obligatoire avant d'utiliser la fonction (ici la fonction est utilisée dans le **main()**). C'est précisément la raison d'être des fichiers d'en-tête que l'on déclare en haut des programmes. Ils contiennent des définitions de constantes, de classes, et des déclarations de prototypes.

En revanche, le corps de la fonction peut être défini après son utilisation (ici après la fonction **main()** qui l'utilise), voire dans d'autres fichiers. On parle alors de *compilation séparée*.

Dans le cas présent la fonction ne prend pas d'arguments (parenthèses vides) et retourne un entier



## CHAPITRE 2

---

### Objets en C++

---

Le langage C++ apporte au langage C la programmation orientée objet. La syntaxe du langage C est complètement admise par les compilateurs C++. Cependant, certains outils, ou certaines façons de faire, ne sont plus de mise en C++.

Part exemple, on n'utilise plus les *structures* du langage C mais les *classes* du langage C++.

Dans l'exemple ci-dessous nous définissons une classe *Complex* permettant de représenter les nombres complexes. Comme en C, il n'existe pas de type permettant de représenter les nombres complexes en C++ :

```
#include <iostream>

class Complex {
private:
    double re, im;
};

int main( void ) {
    Complex c;
}
```

Notez que la définition d'une classe s'effectue toujours entre accolades terminées par un point virgule. Ci-dessus nous avons déclaré une classe composée de deux attributs en double précision (*re* et *im*) pour représenter la partie réelle et la partie imaginaire du nombre complexe. Dans la fonction principale nous avons déclaré une *instance* de la classe, ou *objet*, ou encore une variable de type *Complex*.

L'accès aux attributs d'une classe suit la même syntaxe qu'en C pour l'accès aux champs d'une structure. A la différence du C cependant, les accès suivants vont être refusés par le compilateur :

```
int main( void ) {
    Complex c;

    cout << "c=(" << c.re << ", " << c.im << ")" << endl;
}
```

En effet, l'affichage requiert l'accès en lecture des attributs de l'objet, or ces attributs sont des attributs *privés* de la classe *Complex*. Ils ne sont donc pas accessibles.

On définit donc des *méthodes* de la classe qui vont permettre de réaliser une *interface*, c'est à dire un moyen d'accéder aux attributs de la classe. Une *méthode* est une fonction membre d'une classe. Ci-dessous on définit deux méthodes permettant de lire l'état des attributs :

```
#include <iostream>

class Complex {
private:
    double re, im;
public:
    double real() { return re; }
    double ima() { return ima; }
};
```

Ces méthodes sont déclarées dans la partie *public* de la classe, on peut donc les utiliser cette fois pour réaliser l'affichage :

```
int main( void ) {
    Complex c;

    cout << "c=(" << c.real() << "," << c.ima() << ")" << endl;
}
```

Lorsqu'une méthode d'une classe ne modifie pas les attributs de la classe, on dit que c'est une méthode *constante* et on l'indique au compilateur en ajoutant le qualifieur *const* comme ci-dessous :

```
#include <iostream>

class Complex {
private:
    double re, im;
public:
    double real() const { return re; }
    double ima() const { return im; }
};
```

L'ajout du qualifieur *const* n'est pas obligatoire. Cette information supplémentaire apportée au compilateur est pourtant très importante et il faudra s'y conformer pour réaliser des programmes plus complexes.

Comment faire à présent pour accéder aux attributs en écriture ?

Réponse : définir de nouvelles méthodes de la classe, qui ne seront pas constantes cette fois, et qui permettront l'accès en écriture aux attributs. En général on appelle *accesseurs* les méthodes qui permettent d'accéder aux attributs d'une classe sans plus de traitements, voire *getters* ou *setters* lorsque l'on précise les accès en lecture et écriture.

Ci-dessous la classe implémente deux *getters* et deux *setters* :

```
#include <iostream>

class Complex {
private:
    double re, im;
public:
    double get_re() const { return re; }
    double get_im() const { return im; }
    void set_re( double r) { re = r; }
    void set_im( double i) const { im = i; }
};
```

Il est maintenant possible de déclarer un objet complexe et de lui affecter ses parties réelles et imaginaires :

```
int main( void ) {
    Complex c;
    c.set_re( 2. );
    c.set_im( 2. );
    cout << "c=(" << c.get_re() << "," << c.get_im() << ")" << endl;
    return 0;
}
```

Le programme affichera :

```
$ > cc objets.cc -lstdc++
$ > ./a.out
$ c=(2,2)
$ >
```

## 2.1 Pointeurs et objets

De même qu'il est possible de manipuler des variables par leur adresse, il est possible de manipuler des objets par leur adresse.

L'exemple ci-dessous déclare une variable pointeur vers un objet *Complex* et utilise ce pointeur pour réaliser le même traitement que ci-dessus :

```
int main( void ) {
    Complex c;           // déclaration d'une variable Complex
    Complex *pc;         // déclaration d'une variable pointeur vers un Complex

    pc = &c;             // affectation au pointeur de l'adresse de la variable c
    pc->set_re( 2. );     // accès aux setters avec la notation pointeur
    pc->set_im( 2. );
    cout << "c=(" << pc->get_re() << "," << pc->get_im() << ")" << endl;
}
```

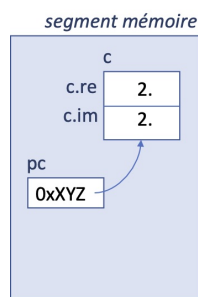


Fig. 1 – Représentation des variables et objet en mémoire au moment de l'affichage.

Lorsque l'on manipule un objet par son adresse, on utilise l'opérateur `->` et non pas le point pour accéder aux attributs ou aux fonctions membres de l'objet.

### 2.1.1 Allocation dynamique

L'allocation dynamique des objets requiert l'emploi des opérateurs *new* et *delete*. On reprend l'exemple ci-dessus en allouant dynamiquement cette fois la variable complexe :

```
int main( void ) {
    Complex *pc;           // déclaration d'une variable pointeur vers un Complex
    pc = new Complex;      // allocation de l'objet

    pc->set_re( 2. );       // accès aux setters avec la notation pointeur
    pc->set_im( 2. );
    cout << "c=(" << pc->get_re() << "," << pc->get_im() << ")" << endl;

    delete pc;             // destruction de l'objet

    return 0;
}
```

La variable *pc* contient l'adresse de l'objet alloué dynamiquement, telle que la retourne l'opérateur *new*. A l'issue du traitement il faut détruire l'objet, tâche réalisée par l'opérateur *delete*.

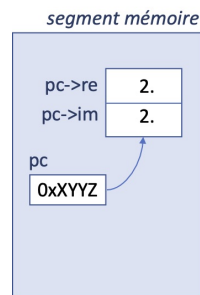


Fig. 2 – Représentation des variables et objet en mémoire au moment de l'affichage.

## 2.2 Passage d'objets en argument de fonction

Soit à afficher un objet de la classe *Complex*. On propose pour cela d'écrire une fonction qui prend en argument un objet et l'affiche. Nous avons trois possibilités pour le passage d'argument : \* le passage par valeur \* le passage par adresse \* le passage par référence

### 2.2.1 Passage d'argument par valeur

Le programme suivant donne un exemple de passage par valeur. A l'appel de la fonction, un objet de classe *Complex*, appelé *c* et local à la fonction, est créé par copie de l'objet *c1* donné en argument par le programme appelant.

La fonction affiche alors les attributs de l'objet local. L'objet local *c* est ensuite détruit avant retour au programme appelant :

```
void afficher( Complex c ) {
    cout << "c=(" << c.get_re() << "," << c.get_im() << ")" << endl;
}

int main() {
```

(suite sur la page suivante)

(suite de la page précédente)

```

Complex c1;

c1.set_re( 2. );
c1.set_im( 3. );
afficher( c1 );

return 0;
}

```

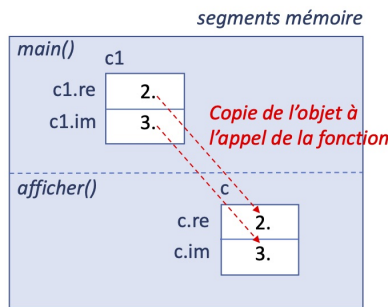


Fig. 3 – Les objets en mémoire au moment de l'exécution de la méthode `afficher()`. La copie locale `c` sera détruite dès le retour de la fonction d'affichage.

Ce mécanisme de passage par valeur coûte cher en temps d'exécution du fait de la création de la copie locale de l'objet. Par ailleurs, comme **la fonction n'accède qu'à une copie de l'objet, elle ne peut pas modifier l'objet d'origine.** S'il fallait par exemple écrire une fonction de saisie des attributs de l'objet, ce ne serait pas possible avec le passage par valeur.

## 2.2.2 Passage d'argument par adresse

Pour modifier l'objet dans la fonction, une méthode possible est d'utiliser le passage des arguments par adresse.

Soit à écrire une fonction de saisie des attributs d'un nombre complexe. Avec un passage par adresse, on écrira :

```

void saisir( Complex *pc ) {
    double r, i;
    cout << "Saisir les parties réelle et imaginaire de l'objet: ";
    cin >> r >> i;

    pc->set_re( r );
    pc->set_im( i );
}

int main() {
    Complex c1;

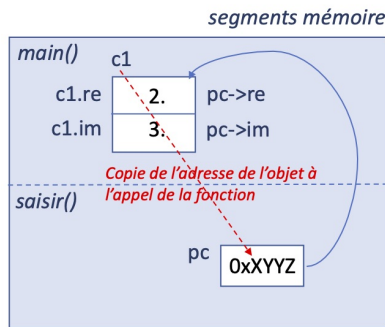
    saisir( &c1 );
    afficher( c1 );

    return 0;
}

```

La fonction `saisir()` sollicite l'utilisateur et affecte les valeurs qu'il tape au clavier aux attributs de l'objet. Cette fonction connaît l'adresse de l'objet (et non pas l'objet lui-même) qui est passée en argument. L'accès aux attributs de l'objet se fait donc par l'opérateur `->`.

Ci-dessous le schéma donne une représentation du segment de mémoire des données du programme lors de l'exécution de la fonction `saisir()` :



Une variable pointeur `pc` apparaît dans l'espace mémoire de la fonction. Cette variable est initialisée avec l'adresse de l'objet `c1` lors de l'appel. La fonction peut donc accéder aux membres de l'objet `c1` déclaré dans le `main()` et non plus aux membres d'un objet local comme lors du passage par valeur.

### 2.2.3 Passage d'argument par référence

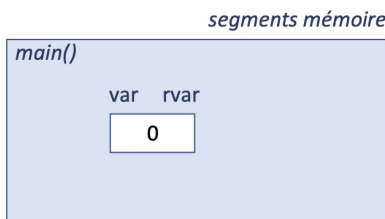
Les références sont une nouvelle méthode de référencement des variables introduites dans le langage C++. Déclarer une référence sur une variable, c'est en quelque sorte donner un nouveau nom à la variable :

```
int main() {
    int var = 0;           // déclare une variable entière
    int &rvar = var;       // déclare une référence sur la variable entière var
}
```

Tout ce que vous ferez sur la référence sera en réalité fait sur la variable d'origine.

Une référence doit obligatoirement se référer à une variable. Ainsi, la déclaration suivante qui ne réfère aucune variable est rejetée par le compilateur :

```
int main() {
    int var = 0;           // déclare une variable entière
    int &rvar;              // pas de variable référencée => erreur de compilation
}
```



C'est le symbole `&` qui indique que `rvar` est une référence.

Les références sont beaucoup utilisées en passages d'arguments des fonctions et méthodes. C'est un moyen commode de réaliser l'équivalent d'un passage par adresse sans utiliser les pointeurs.

Réécrivons la fonction `saisir()` avec un passage par référence de l'objet :

```

void saisir( Complex &rc ) {
    double r, i;
    cout << "Saisir les parties réelle et imaginaire de l'objet: ";
    cin >> r >> i;

    rc.set_re( r ); // on agit ici directement sur l'objet référencé
    rc.set_im( i ); // idem
}

int main() {
    Complex c1;

    saisir( c1 );
    afficher( c1 );

    return 0;
}

```

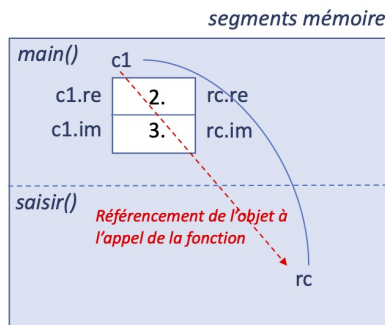
la déclaration `Complex &rc` en argument de la fonction indique au compilateur que `rc` est cette fois une référence vers un objet de type `Complex`.

Notez que la référence ne réfère aucune variable. C'est normal. Le référencement s'effectue lors de l'appel de la fonction. Ainsi, c'est à la ligne :

```
saisir( c1 );
```

du `main()` qu'il est établi que `Complex &rc` sera une référence vers l'objet `c1`.

Voici ce qu'il se passe en mémoire lors de l'exécution de la fonction :



Dans la fonction `saisir()` il est donc établi que `rc` devient un deuxième nom pour l'objet `c1` défini dans le `main()`. Il devient possible d'accéder aux attributs de l'objet d'origine par l'opérateur `.` : `rc.set_re()` et `rc.set_im()`.

## 2.3 Constructeurs et destructeurs

Reprenons la classe `Complex` ci-dessous :

```

class Complex {
private:
    double im;
    double re;
public:

```

(suite sur la page suivante)

(suite de la page précédente)

```
double get_re() const { return re; }
double get_im() const { return im; }
};
```

Lorsque l'on déclare une instance de cette classe :

```
int main() {
    Complex c;

    return 0;
}
```

un objet `c` est créé mais non initialisé. Les attributs de `c` ont donc des valeurs indéterminées.

Le constructeur d'une classe a pour tâche d'initialiser les attributs des instances de la classe au moment de leur déclaration. La syntaxe du langage impose que : \* le constructeur est une méthode de la classe \* le constructeur porte obligatoirement le nom de la classe \* le constructeur peut avoir des arguments \* le constructeur ne retourne aucune type, même pas void \* la définition d'un constructeur n'est pas obligatoire lorsque cela n'est pas utile

### 2.3.1 Constructeur par défaut

Le *constructeur par défaut* est un constructeur qui ne prend aucun argument. Il initialise des objets lorsqu'ils sont déclarés comme dans le `main()` vu dessus.

Pour un nombre complexe, l'initialisation par défaut pourrait être simplement d'initialiser à 0 les parties réelle et imaginaire :

```
class Complex {
private:
    double im;
    double re;
public:
    Complex() {
        im = 0;
        re = 0;
        cout << "Complex: Constructeur par défaut" << endl;
    }
};
```

Remarquez que l'affichage n'a aucune utilité ici, il est là pour aider à comprendre les mécanismes en jeu. Ainsi, l'exécution du programme suivant va donner :

```
int main() {
    Complex c;
}
```

```
> $ prog
> Complex: Constructeur par défaut
> $
```

La constructeur est donc bien appelé à la déclaration de l'objet. On peut être certain que les attributs sont initialisés à 0.

Dans la définition précédente de la classe nous n'avions pas défini de constructeur par défaut, et pourtant la déclaration `Complex c;` ne générerait pas d'erreur.



Réponse : lorsque vous ne déclarez pas de constructeur par défaut, le compilateur en définit un à votre place. Le soucis avec cette déclaration implicite est que vous n'êtes jamais assuré que vos attributs seront initialisés aux valeurs que vous souhaitez.

### 2.3.2 Constructeur spécifique

Un constructeur peut avoir des arguments. Par exemple dans le cas de la classe `Complex` il est avantageux de disposer d'un constructeur permettant d'initialiser les attributs à des valeurs spécifiques :

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

class Complex {
private:
    double im;
    double re;
public:
    Complex( double _re, double _im ) {
        re = _re;
        im = _im;
        cout << "Complex: Constructeur spécifique" << endl;
    }
};

int main() {
    Complex c1( 1., 2. );
    cout << "c1 = ( " << c1.get_re() << ", " << c1.get_im() << " )" << endl;
}
```

Exécution :

```
Complex: Constructeur spécifique
c1 = ( 1, 2 )
```

### 2.3.3 Surcharge du constructeur

Initialiser des objets par défaut et initialiser d'autres objets de façon spécifique est possible à condition de doter la classe des deux constructeurs à la fois. On dit que l'on procède à une surcharge du constructeur :

```
class Complex {
private:
    double im; double re;
public:
    Complex() { re = im = 0;} // constructeur par défaut
    Complex( double _re, double _im ) { // surcharge du constructeur
        re = _re; // par un constructeur spécifique
        im = _im;
    }
};

int main() {
    Complex c1; // initialisation par défaut
    Complex c2( 1., 2. ); // initialisation spécifique
}
```

Si on envisage d'initialiser un complexe avec une partie réelle spécifique et une partie imaginaire nulle par défaut, alors on surcharge à nouveau le constructeur en ajoutant un constructeur spécifique pour cela :

```
class Complex {
private:
    double im; double re;
public:
    Complex() { re = im = 0; }           // constructeur par défaut
    Complex( double _re ) {             // 1er constructeur spécifique
        re = _re;
        im = 0;
    }
    Complex( double _re, double _im ) { // 2eme constructeur spécifique
        re = _re;
        im = _im;
    }
};

int main() {
    Complex c1;           // initialisation par défaut
    Complex c2( 1. );     // initialisation spécifique (réel pur)
    Complex c3( 1., 2. ); // initialisation spécifique
}
```

## 2.3.4 Constructeurs et arguments par défaut

Le C++ admet les arguments de fonction par défaut (voir le chapitre *surcharge des fonctions, méthodes et opérateurs*). Ainsi, les deux surcharges précédentes peuvent être évitées en définissant un unique constructeur par défaut qui accepte des arguments initialisés par défaut :

```
class Complex {
private:
    double im; double re;
public:
    Complex( double _re=0, double _im=0 ) { // unique constructeur
        re = _re;
        im = _im;
    }
};

int main() {
    Complex c1;           // initialisation par défaut
    Complex c2( 1. );     // initialisation spécifique (réel pur)
    Complex c3( 1., 2. ); // initialisation spécifique
}
```

Notez bien qu'il y a un ordre d'évaluation des arguments. Lorsqu'un seul argument est spécifié, c'est l'argument le plus à gauche qui prend la valeur, et ainsi de suite :

```
Complex c2( 1. ); // 1. est affecté à l'argument _re
```


### Attention !

Lorsque le constructeur est défini en dehors de la classe, où place-t-on l'initialisation par défaut des arguments ?

Réponse :

```
class Complex {
private:
    double im; double re;
public:
    Complex( double _re=0, double _im=0 ); // valeurs par défaut ici!
};

Complex::Complex( double _re, double _im ) {
    re = _re;
    im = _im;
}
```




Si vous ne placez pas l'initialisation par défaut dans la déclaration de prototype, il n'y a plus de constructeur par défaut mais un seul constructeur spécifique à deux arguments. Lorsqu'au moins un constructeur est défini (ici le constructeur spécifique), le compilateur ne génère pas de constructeur par défaut. D'où :

```
int main() {
    Complex c1;           // erreur: pas de constructeur par défaut
    Complex c2( 1. );     // erreur: pas de constructeur à un seul argument
    Complex c3( 1., 2. ); // correcte: constructeur spécifique
}
```

Etant donné que dans une déclaration de prototype, seule la signature de la fonction/méthode compte pour le compilateur (la signature est la liste des types des arguments d'une fonction et son qualifieur `const` éventuel), l'écriture suivante qui omet le nom des paramètres est encore valide :

```
class Complex {
private:
    double im; double re;
public:
    Complex( double=0, double=0 ); // valeurs par défaut ici!
};

Complex::Complex( double _re, double _im ) {
    re = _re;
    im = _im;
}
```



### Remarque : le pointeur “this”



Vous avez noté le nom donné aux arguments `_re` et `_im` plutôt que `re` et `im` dans :

```
Complex::Complex( double _re, double _im ) {
    re = _re;
    im = _im;
}
```

Cela permet de ne pas confondre les paramètres de la méthode avec les attributs de la classe.

Il est possible d'utiliser les mêmes noms, à condition toutefois de distinguer les attributs des paramètres en utilisant le pointeur `this` :

```
Complex::Complex( double re, double im ) {
    (*this).re = re;
    (*this).im = im;
}
```

Plus de détail sur ce sujet dans le chapitre *surcharge des fonctions, méthodes et opérateurs*.

### 2.3.5 Constructeur par copie

Il arrive fréquemment que l'on initialise une variable à partir d'une autre variable :

```
int main() {
    int i = 0;
    int j = i;
}
```

Rappelons déjà que l'opérateur = dans ce contexte ne désigne pas une *affectation* mais une *initialisation*. La différence est importante car l'initialisation appelle un constructeur, contrairement à l'affectation.

j est donc initialisée par copie de la variable i.

La même chose peut également se produire pour un objet que l'on souhaite déclarer et initialiser par copie d'un autre objet. C'est dans ce cas le *constructeur par copie* qui est appelé.

Le constructeur par copie est un constructeur qui prend obligatoirement un seul argument qui doit être un objet de la même classe.

Dans l'exemple ci-dessous nous déclarons une classe `Complex` dotée d'un constructeur par copie :

```
class Complex {
private:
    double im;
    double re;
public:
    Complex() {
        re = im = 0;
        cout << "Complex: Constructeur par défaut" << endl;
    }
    Complex( const Complex &c ) {
        re = c.re; im = c.im;
        cout << "Complex: Constructeur par copie" << endl;
    }
};
```

et le programme :

```
int main() {

    Complex c1( 1., 2.);
    Complex c2 = c1;

    return 0;
}
```

donnera à l'exécution :

```
Complex: Constructeur par défaut
Complex: Constructeur par copie
```

#### Remarque

Le constructeur de copie déclare comme argument `Complex( const Complex &c )`, donc une référence. On aurait pu imaginer un passage par valeur de l'argument : `Complex( Complex c )`.

C'est impossible :

Le passage par valeur implique la copie locale de l'objet donné en argument, donc un appel au constructeur de copie, constructeur qu'on est précisément en train de définir. On aurait donc un constructeur de copie qui doit s'appeler lui-même pour résoudre la copie, soit une boucle sans fin.

### 2.3.6 Constructeur de conversion

Ce que l'on appelle le *cast* est une *conversion explicite de type*. Convertir un type *double* vers un type *int* requiert de réaliser un *\*cast* car la ligne suivante est refusée par le compilateur :

```
double d = 3.2;
int i = 1;
i = d;           // erreur de compilation (-> requiert un cast)
d = i;           // accepté (c'est une conversion implicite)
```

Convertir un *int* en *double* est accepté : le compilateur réalise une conversion dite *implicite*.

Pour des raisons évidentes (un nombre décimal n'est pas une sorte d'entier), la conversion inverse est interdite par le compilateur.

Le *cast* consiste à forcer la conversion, mais en la faisant proprement, en prenant la partie entière par exemple :

```
double d = 3.2;
int i = 1;
i = int( floor( d ) );
```

La fonction `floor()` prend la partie entière du nombre `d` et le *cast* consiste à employer le type final comme une fonction : `int(...)`.

Nous devons pouvoir faire la même chose avec les objets : convertir un objet en un autre objet. C'est le rôle du *constructeur de conversion*

Le constructeur de conversion prend comme argument un type ou un objet de classe différente. Soit par exemple à convertir un nombre réel en un nombre complexe, supposant donc qu'il sera réel pur, donc à partie imaginaire nulle.

Il faudrait pouvoir écrire :

```
double d = 1.5;
Complex c = Complex( d ); // cast de double -> Complex
```

Il nous faut donc écrire un constructeur qui prend comme argument un *double*. Mais en réalité, ce constructeur nous l'avons déjà défini sans le savoir, c'est le constructeur spécifique à un seul argument *double* vu plus haut, ou encore le constructeur par défaut avec le premier argument *double* affecté.

En résumé, le programme ci-dessus fonctionne sans ajouter aucun code. L'écriture condensée ci-dessous fonctionne également :

```
double d = 1.5;
Complex c = d; // cast de double -> Complex
```

Enfin, au cours d'une affectation, et non plus d'une initialisation, le constructeur de conversion va tout de même être appelé :

```
double d = 1.5;
Complex c;           // déclaration du complexe c

c = Complex( d );    // cast double -> Complex puis affectation dans c
```

La conversion est faite, dans un premier temps par création d'un objet temporaire `Complex` à partir de la donnée double `d`, puis dans un deuxième temps par affectation des valeurs des attributs de l'objet temporaire dans ceux de l'objet destination `c`. L'affichage à l'exécution montre la trace de cette création d'objet temporaire puisque deux appels au constructeur apparaissent :

```
$ > ./a.out  
Complex: Constructeur spécifique  
Complex: Constructeur spécifique
```

---

## Surcharge des fonctions, méthodes et opérateurs

---

La surcharge de fonctions, méthodes et opérateurs permet d'adopter une programmation plus générique. Dans le principe il s'agit de donner le même nom à des fonctions qui réalisent un traitement identique sur des données différentes.

### 3.1 Surcharge de fonctions et principe de la surcharge

*Surcharger* une fonction consiste à écrire deux ou plus fonctions portant le même nom, mais acceptant en argument des types différents.

Exemple : soit la fonction `afficheInt()` permettant d'afficher un entier :

```
void afficheInt( int val ) {  
    cout << val << endl;  
}
```

Afficher une valeur de type `double` obligerait à définir la fonction adaptée :

```
void afficheDouble( double val ) {  
    cout << val << endl;  
}
```

Dans un programme principal :

```
int main() {  
    int val = 10;  
    double d = 25.2;  
  
    afficheInt(val);  
    afficheDouble(d);  
}
```

Le principe de la surcharge consiste à donner le même nom aux deux fonctions :

```
void affiche( int val ) {  
    cout << val << endl;  
}  
  
void affiche( double val ) {  
    cout << val << endl;  
}
```

Conduisant à :

```
int main() {  
    int val = 10;  
    double d = 25.2;  
  
    afficher(val);  
    afficher(d);  
}
```

Du point de vue du programmeur, tout se passe comme si nous disposions d'une unique fonction capable d'afficher les entiers autant que les flottants.

C'est le compilateur qui choisit à votre place la bonne fonction à appeler. L'important à présent est de comprendre comment s'en sort le compilateur pour savoir quelle fonction appeler.

La méthode est simple. Au moment de traiter la ligne

```
afficher(val);
```

le compilateur analyse le type de l'argument donné à la fonction, ici `val` est de type `int`. il analyse ensuite les types des arguments des fonctions disponibles et portant le même nom. Dans le cas présent, une seule fonction est définie avec le type `int` en argument. Il construit donc un appel à cette fonction.

Le nom de la fonction et la liste des types de ses arguments et, éventuellement, le qualifieur `const`, forment ce que l'on appelle la *signature* de la fonction.

Si plusieurs fonctions du même nom ont même signature, alors il y a ambiguïté et le compilateur signale une erreur.

### Attention !

Le type retourné par une fonction ne fait pas partie de sa signature. Autrement dit, il n'est pas possible de surcharger deux fonctions qui auraient même nom et mêmes types d'arguments, même si elles retournent un type différent.

## 3.2 Surcharge de méthodes

La surcharge de méthode désigne le même principe que la surcharge de fonction, appliqué aux fonctions membres d'une classe. Le cas déjà présenté de la surcharge des constructeurs d'une classe donne un bon exemple de surcharge de méthode.

## 3.3 Surcharge des opérateurs

Les opérateurs sont les symboles du langage qui permettent de représenter des opérations sur les données et objets. Ils peuvent être unaire, binaire, ternaire.

De même que nous pouvons surcharger des fonctions et méthodes, nous pouvons surcharger des opérateurs.



Pour comprendre le principe de la surcharge des opérateurs, il faut avant tout se représenter ces opérateurs comme des fonctions. Par exemple, soit à opérer une addition entre deux nombre entiers. Deux écritures sont possibles :

```
int main() {
    int a=1, b=2;
    int c;

    c = a + b;           // notation opérateur
    c = operator+(a, b); // notation fonction
}
```

Ces deux lignes sont strictement équivalentes pour le compilateurs et la deuxième (notation fonctionnelle) est parfaitement admise à la compilation.

On comprend ici l'intérêt de la surcharge des opérateurs : nous allons pouvoir les définir et les surcharger pour les classes que nous écrivons.

Soit à définir l'opération d'addition de deux nombres complexes de sorte que l'opération suivante soit admise :

```
int main() {
    Complex c1(1,0), c2(3,2);
    int c3;

    c3 = c1 + c2;
}
```

c1, c2 et c3 étant des objets, nous avons deux façons de voir l'opération :

- 1) l'opérateur d'addition est une définition de l'opérateur d'addition en tant que méthode de la classe Complex :

```
int main() {
    Complex c1(1,0), c2(3,2);
    int c3;

    c3 = c1.operator+(c2); // notation méthode
}
```

- 2) l'opérateur d'addition est une surcharge de l'opérateur d'addition standard :

```
int main() {
    Complex c1(1,0), c2(3,2);
    int c3;

    c3 = operator+(c1, c2); // notation fonction
}
```

La méthode 1) est à utiliser lorsque nous sommes concepteur de la classe Complex et la méthode 2) lorsque nous sommes utilisateur de la classe Complex.

### 3.3.1 Définition en tant que méthode de la classe

En tant que concepteur de la classe Complex, nous définissons donc l'opérateur + ainsi :

```
class Complex {
    double re, im;
private:
```

(suite sur la page suivante)

(suite de la page précédente)

```

Complex operator+( const Complex & );
};
type returned class name Function
Complex Complex::operator+( const Complex &c ) {
    return Complex( re + c.re, im + c.im );
}

```

La méthode `Complex::operator+` construit un nouvel objet `Complex` à partir de la somme des attributs de l'objet courant et de l'objet passé en argument.

Point important, la méthode n'effectue pas un retour par référence de l'objet créé mais un retour par valeur. En effet, cet objet créé dans la méthode a une portée locale à la méthode et n'existe donc plus dès lors que l'on quitte la méthode. Impossible de référencer un objet qui n'existe plus. Le retour par valeur engendre une copie qui pourra être affectée à `c3`.

Comment se passe l'affectation dans `c3` ? Nous n'avons pas surchargé d'opérateur d'affectation. C'est donc le compilateur qui effectue cette opération en copiant tous les attributs de l'objet source dans l'objet destination.

Attention de penser à surcharger l'opérateur d'affectation lorsque des opérations particulières doivent être réalisées qui ne le seraient pas par le compilateur (allocation dynamique de mémoire par exemple).

Afin d'observer l'exécution de ces différentes étapes, définissons volontairement un constructeur par copie et surchargeons l'opérateur d'affectation. Toutes ces méthodes comporteront une affichage permettant de visualiser les opérations :

```

class Complex {
    double re, im;
private:
    Complex( double=0, double=0 );           // constructeur par défaut
    Complex( const Complex &c );             // constructeur par copie
    Complex operator+( Complex & );          // opérateur addition
    Complex & operator=( const Complex & ); // opérateur d'affectation
    double get_re() const { return re; };
    double get_im() const { return im; };
};

Complex::Complex( double r, double i ) {
    cout << "Constructeur par défaut" << endl;
    re = r;
    im = i;
}

Complex::Complex( const Complex &c ) {
    cout << "Constructeur par copie" << endl;
    re = c.re;
    im = c.im;
}

Complex Complex::operator+( const Complex &c ) {
    cout << "Opérateur addition" << endl;
    return Complex( re + c.re, im + c.im );
}

Complex &Complex::operator=( const Complex &c ) {
    if( &c == this )
        return *this;
}

```

(suite sur la page suivante)

(suite de la page précédente)

```

    cout << "Opérateur affectation" << endl;
    re = c.re;
    im = c.im;
    return *this;
}

int main() {

    Complex c1(10, 2), c2(5, 2);
    Complex c3;

    c3 = c1 + c2;

    cout << "c1 + c2 = " << "(" << c3.get_re() << ", " << c3.get_im() << ")" << endl;
    return 0;
}

```

A l'exécution :

```

$ > ./a.out
Constructeur par défaut
Constructeur par défaut
Constructeur par défaut
Opérateur addition
Constructeur par défaut
Opérateur affectation
Destructeur
c1 + c2 = (15, 4)
Destructeur
Destructeur
Destructeur

```



Apparaissent :

- Trois appels au constructeur par défaut consécutifs aux déclarations des trois variables c1, c2 et c3;
- L'appel à l'opérateur d'addition;
- Le constructeur par défaut pour l'objet créé localement par l'opérateur d'addition;
- L'appel à l'opérateur d'affectation;
- Le destructeur de l'objet créé localement par l'opérateur d'addition;
- l'affichage du résultat;
- Trois appels au destructeur pour la destruction des 3 objets c1, c2 et c3.

### 3.3.2 Surcharge de l'opérateur d'addition

Nous nous plaçons en tant qu'utilisateur de la classe `Complex`. Se placer dans cette position signifie en réalité que la classe `Complex` ne dispose pas d'opérateur d'addition et que nous n'avons pas la possibilité de modifier la classe pour en ajouter un.

La surcharge de l'opérateur d'addition dans ce contexte signifie donner la possibilité de réaliser l'opération suivante :

```
c3 = operator+(c1, c2);
```

Soit :

```

Complex operator+( const Complex &c1, const Complex &c2 ) {
    cout << "Surcharge opérateur addition" << endl;

```

(suite sur la page suivante)

(suite de la page précédente)

```

return Complex( c1.get_re() + c2.get_re(), c1.get_im() + c2.get_im() );
}

```

A l'exécution :

```

$ > ./a.out
Constructeur par défaut
Constructeur par défaut
Constructeur par défaut
Surcharge opérateur addition
Constructeur par défaut
Opérateur affectation
Destructeur
c1 + c2 = (15, 4)
Destructeur
Destructeur
Destructeur

```

Notez que nous avons dû faire appel aux méthodes d'accès aux attributs dans l'opérateur (`get_re()` et `get_im()`). Les attribut étant privés et la surcharge externe à la classe, il n'est plus possible d'y accéder sans passer par l'interface de la classe.

### 3.3.3 Surcharge de l'opérateur d'insertion de flux

On souhaite pouvoir afficher un nombre complexe en l'envoyant directement dans le flux `cout`. On est là dans le cas d'une surcharge qu'il faut définir à l'extérieur de la classe `ostream` - dont nous sommes utilisateur de l'instance `cout`, car nous ne pouvons pas la modifier.

L'opération d'insertion :

```

int main() {

    Complex c(10, 2);

    cout << c;

}

```

s'écrit sous sa forme fonctionnelle :

```

int main() {

    Complex c(10, 2);

    operator<<( cout, c );

}

```

On écrit donc la surcharge de l'opérateur d'insertion de la façon suivante :

```

ostream &operator<<( ostream &o, Complex &c ) {
    o << "(" << c.get_re() << ", " << c.get_im() << ") ";
    return o;
}

```

ce qui amène plusieurs commentaires :

- `operator<<` est un opérateur binaire qui prend deux arguments : le flux à gauche et l'objet à insérer à droite.
- `ostream` est la classe de l'instance `cout` que nous utilisons pour afficher des données.

- Pour information, `istream` est la classe de l'instance `cin` que nous utilisons pour saisir des données à partir du clavier.
- Le flux est passé obligatoirement en référence. Un passage par valeur occasionnerait une copie locale du flux.
- La fonction retourne une référence sur le flux. Obligatoire également pour permettre l'envoi d'autres données sur le même flux les unes après les autres. Cela s'illustre bien par l'exemple suivant où l'on ajoute au flux un retour à la ligne :

```
int main() {
    Complex c(10, 2);

    cout << c << endl;
}
```

En écriture fonctionnelle :

```
int main() {
    Complex c(10, 2);

    operator<<( operator<<( cout, c ), endl ;
}
```

où l'on voit que pour insérer `endl`, le premier appel à l'opérateur `operator<<` requiert d'avoir un flux à gauche. Il faut donc bien que le deuxième appel retourne le flux.

- enfin nous devons faire appel à l'interface de la classe `Complex` (les méthodes `get_re()` et `get_im()`) car ses attributs sont privés et que l'on n'y a donc pas accès.

Ce dernier point peut être corrigé en déclarant *amie* la fonction `operator<<()`. Nous pouvons le faire puisque si nous sommes utilisateurs de la classe `ostream`, en revanche nous sommes concepteur de la classe `Complex` et nous pouvons donc la modifier en conséquence :

```
class Complex {
    friend ostream &operator<<( ostream &o, Complex &c );
    double re, im;
public:
    ...
};

ostream &operator<<( ostream &o, Complex &c ) {
    o << "(" << c.re << ", " << c.im << ")"; // accès direct aux attributs
    return o;
}
```

### 3.4 Les tableaux et la surcharge de l'opérateur crochet []

Tous les opérateurs peuvent être surchargés. Mais il en est un qui présente un intérêt particulier, l'opérateur *crochet*.

Dans l'opération suivante `a=tab[i]`, l'opérateur *crochet* (`operator[]`) signifie : *contenu de l'adresse tab incrémentée i fois*. Ce qui s'écrit donc également en notation pointeur : `a = *(tab+i)`. Rappelons que le nom d'un tableau désigne également son adresse.

Avec cet opérateur défini pour tous les types standards du langage (`int`, `double`, ...) on est très vite limité car la dimension des tableaux n'est pas gérée en natif et que c'est donc toujours à nous de le faire. Cette gestion alourdi le code.

La solution élégante en C++ sera de créer une classe `Array` ayant pour rôle la gestion des tableaux de sorte à ne plus avoir à s'en occuper.

### 3.4.1 Exemple : la classe `Array`

Soit à créer la classe `Array` permettant les opérations suivantes sur des types double :

```
int main() {

    Array t(10);           // déclaration d'un tableau de 10 éléments
    cout << t << endl;     // affichage du tableau

    t[15] = 1;             // affectation d'un élément et affichage
    cout << "t[15]=" << t[15] << endl;
    cout << "dimension du tableau: " << t.get_dim() << endl;

    Array t2(t);           // construction par copie;
    Array t3;
    t3 = t2;               // affectation d'un tableau

    cout << "t3=" << t3 << endl;
}
```

On souhaite obtenir à l'exécution : l'exécution :

```
$ > ./a.out
[0,0,0,0,0,0,0,0,0,0]
t[15]=1
dimension du tableau: 16
t3=[0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1]
```

La dimension du tableau est connue lors de sa création (allocation dynamique), il faut pouvoir affecter et lire des éléments séparément et afficher tout le tableau. Il faut également pouvoir déclarer un tableau à partir d'un autre tableau et réaliser des affectations de tableau.

Voici un exemple de code qui illustre les différents points abordés plus haut :

```
#include <iostream>

using namespace std;

/**
 * @class Array
 * @brief implémente les tableaux de double
 */
class Array{
    friend ostream &operator<<( ostream &, Array &);

    double *tab;    // le tableau alloué dynamiquement
    int dim;        // dimension du tableau
    void raz();     // initialisation du tableau
public:
    Array( int=0 );           // constructeur par défaut
    Array( const Array& );    // constructeur par copie
    ~Array();
    Array &operator=(const Array &);    // opérateur d'affectation
    int get_dim() const { return dim; } // getter dimension
```

(suite sur la page suivante)

(suite de la page précédente)

```

    double &operator[] ( int );
};

/**
 * @function raz()
 * @brief private: remise à 0 des éléments du tableau
 */
void Array::raz() {
    for (int i=0; i<dim; tab[i++]=0 );
}

/**
 * @function Array( int d )
 * @brief Constructeur par défaut
 * @param d dimension du tableau
 */
Array::Array( int d ) {
    if( ( dim = d )==0 )
        tab = NULL;
    else{
        tab = new double [dim];
        raz();
    }
}

/**
 * @function Array( const Array &t )
 * @brief Constructeur par copie
 * @param a tableau à copier
 */
Array::Array( const Array &a ) {
    if( ( dim = a.dim )==0 )
        tab = NULL;
    else {
        tab = new double [dim];
        for (int i=0; i<dim; i++)
            tab[i] = a.tab[i];
    }
}

/**
 * @function ~Array()
 * @brief destructeur
 */
Array::~~Array() {
    if( dim != 0 )
        delete [] tab;
}

/**
 * @function Array &Array( const Array &t )
 * @brief Constructeur par défaut
 * @param a tableau à copier
 */
Array &Array::operator=( const Array &a ) {
    // pas d'auto-affectation:
    if( &a == this )

```

(suite sur la page suivante)

(suite de la page précédente)

```

        return *this;

        // si il existe déjà un tableau non vide -> delete:
        if( dim > 0 )
            delete [] tab;

        // allocation + copie:
        if( ( dim = a.dim )==0 )
            tab = NULL;
        else {
            tab = new double [dim];
            for (int i=0; i<dim; i++)
                tab[i] = a.tab[i];
        }

        return *this;
    }

    /**
     * @function double &Array::operator=( int i )
     * @brief Opérateur d'accès
     * @param i indice
     */
    double &Array::operator[]( int i ) {
        if ( i<0 )
            i = 0;

        // indice en dehors du tableau -> on agrandi le tableau:
        if ( i>=dim ) {
            double *buff = new double[i+1];
            for( int j=0; j<dim; j++ )
                buff[j] = tab[j];
            for( int j=dim; j<=i; j++ )
                buff[j] = 0;
            delete [] tab;
            tab = buff;
            dim = i+1;
        }

        return tab[i];
    }

    /**
     * @function ostream &operator<<( ostream &o, Array &a)
     * @brief Affichage sur cout du tableau entier
     * @param ostream &o flux
     * @param Array &a tableau à afficher
     */
    ostream &operator<<( ostream &o, Array &a ) {
        o << '[';
        for( int i=0; i<a.dim-1; i++ )
            o << a.tab[i] << ',';
        if( a.dim > 0 )
            o << a.tab[a.dim-1];
        o << ']';

        return o;
    }

```

(suite sur la page suivante)



(suite de la page précédente)

```
}
```



# CHAPITRE 4

## Héritage

L'héritage en POO désigne le mécanisme qui permet de créer une classe à partir d'une autre étant entendu que la classe *dérivée* peut satisfaire la relation *est une sorte de...* relativement à la classe *parente*.

Si par exemple, la classe `Article` permet de représenter tous les articles en vente dans un magasin, la classe `Boisson` peut hériter de la classe `Article` dans la mesure où l'on peut écrire qu'une boisson est une sorte d'article :

```
class Article {
    double prix;
public:
    Article( double p=0 ) : prix(p) {} Constructeur
    double quelprix() const { return prix; }
};

class Boisson : Article {
    double volume;
public:
    Boisson( double v=0 ) : volume(v) {}
    double quelvolume() { return volume; }
};

int main() {
    Article a;
    Boisson b;

    cout << "prix = " << a.quelprix() << endl;
    cout << "volume = " << b.quelvolume() << endl;
}
```

L'écriture

```
class Boisson : Article {...
```

indique que la classe `Boisson` hérite de la classe `Article`. Dans les faits, l'objet `b` déclaré ci-dessus comporte les attributs de sa classe (`volume`) ET les attributs de la classe mère (`prix`).

En revanche, l'accès à ces attributs ne va pas de soi et nécessite des explications.

## 4.1 Accessibilité des attributs et méthodes

Pour comprendre les lois d'accessibilité aux attributs, il faut bien séparer les deux contextes suivants :

- l'accessibilité des attributs d'une classe parente par les méthodes de la classes enfant ;
- l'accessibilité des attributs d'une classe parente par une fonction extérieure à la classe.

### 4.1.1 Accessibilité par les méthodes de la classe descendante

Ecrivons une méthode de la classe `Boisson` permettant d'afficher son prix :

```
class Boisson : Article {
    double volume;
public:
    Boisson( double v=0) : volume(v) {}
    double quelvolume() { return volume; }
    void afficher_prix() { cout << "prix de la boisson = " << prix << endl; }
};
```

Cette méthode **génèrera une erreur de compilation** car les attributs privés, dont `prix` de la classe `Article`, ne sont pas accessibles aux méthodes de la classe fille `Boisson`.

En revanche, les attributs et méthodes `public` d'une classe parente son accessibles aux méthodes de la classe enfant. A condition d'utiliser l'interface de la classe mère, on peut donc afficher le prix :

```
void afficher_prix() { cout << "prix de la boisson = " << quelprix() << endl; }
```

### 4.1.2 Accessibilité par les fonctions extérieures

Les méthodes déclarées `public` sont-elles accessibles aux fonctions extérieures à la classe fille ?

Réponse : **non**.

La commande suivante qui appelle la méthode `Article::quelprix()` n'est pas acceptée par le compilateur :

```
int main() {
    Boisson b;

    cout << "prix = " << b.quelprix() << endl; // err. de compilation
}
```

La règle générale est donc :

- que les attributs et méthodes `private` de la classe mère deviennent innaccessibles pour la classe fille ;
- que les attributs et méthodes `public` de la classe mère deviennent `private` pour la classe fille : ses méthodes peuvent encore y accéder, mais plus les fonctions externes à la classe.

Nous verrons plus bas que cette règle de l'héritage peut être assouplie.

## 4.2 Visibilité des méthodes

Un moyen de rendre accessible les méthodes d'une classe parente à l'extérieur de la classe consiste à redéfinir une interface pour la classe fille.

Dans l'exemple suivant, on réécrit la méthode `quelprix()` dans la classe fille pour la rendre accessible :

```
class Boisson : Article {
    double volume;
public:
    Boisson( double v=0 ) : volume(v) {}
    double quelvolume() { return volume; }
    double quelprix() { return Article::quelprix(); }
};

int main() {
    Boisson b;

    cout << "prix = " << b.quelprix() << endl; // plus d'erreur
}
```

Notez que dans la méthode `Boisson::quelprix()` il est obligatoire d'utiliser l'opérateur de résolution de portée `Article::` pour accéder à la méthode `Article::quelprix()`. Ne pas le faire conduirait à un appel récursif de la méthode `Boisson::quelprix()` par elle-même.

## 4.3 Constructeurs et destructeurs

Avec le constructeur

```
Boisson::Boisson( double v=0 ) : volume(v) {}
```

et la déclaration

```
int main() {
    Boisson b(10);
}
```

l'attribut `Boisson::volume` est initialisé, mais pas l'attribut `Article::prix` car le constructeur de la classe mère n'est pas appelé.

Cet appel doit se faire explicitement en respectant la syntaxe suivante :

```
Boisson::Boisson( double v=0 ) : Article, volume(v) {}
```

C'est le constructeur de `Boisson` qui appelle le constructeur de `Article`. Dans le cas présent c'est le constructeur par défaut de `Article` qui est appelé. Mais si l'on souhaite initialiser le prix à partir d'une valeur donnée en argument, on procède en ajoutant un argument au constructeur de `Boisson` :

```
Boisson::Boisson( double p=0, double v=0 ) : Article(p), volume(v) {}
```

Le programme suivant permet alors d'initialiser correctement l'objet `b` :


```
int main() {
    Boisson b( 5, 10 );
    cvout << "prix de la boisson: " << b.quelprix() << endl;
}
```

Avec ce schéma tous les constructeurs des classes descendantes sont appelés, celui de la classe de base s'exécutant d'abord, puis celui de la classe fille immédiate, et ainsi de suite.

C'est le schéma exactement inverse qui se produit dans la chaîne d'appel des destructeurs. Le destructeur de la classe fille est d'abord appelé, puis celui de la classe mère, et ainsi de suite jusqu'au destructeur de la classe de base.

Lorsqu'il s'agit d'objets statiques, c'est le compilateur qui s'occupe de détruire les objets, donc d'appeler les destructeurs.

Insérons l'affichage d'un message dans chacun des destructeurs ci-dessous :

```
class Article {
    double prix;
public:
    Article( double p=0 ) : prix(p) {}
    ~Article() { cout << "~Article()" << endl; }
};

class Boisson : Article {
    double volume;
public:
    Boisson( double p=0, double v=0 ) : Article(p), volume(v) {}
    ~Boisson() { cout << "~Boisson()" << endl; }
};
```

Le programme suivant qui déclare deux objets, respectivement de la classe Article et de la classe Boisson :

```
int main() {

    Article a( 10 );
    Boisson b( 5, 10 );
}
```

Va générer l'affichage suivant à l'exécution :

```
$ > ./a.out
~Boisson()
~Article()
~Article()
```

Le dernier objet déclaré est d'abord détruit (b), le destructeur de Boisson est donc exécuté en premier, puis celui de la classe parente Article. A ce niveau l'objet b est entièrement détruit. Vient ensuite la destruction de l'objet a (dernière ligne), donc appel au destructeur ~Article().

Même exemple ci-dessous mais en procédant à une allocation dynamique des objets. Pour les détruire il faut cette fois appeler explicitement l'opérateur delete :

```
int main() {

    Article *pa;
    Boisson *pb;

    pa = new Article( 10 );
    pb = new Boisson( 5, 10 );

    delete pa;
    delete pb;
}
```

L'exécution produit exactement le même affichage. Les objets sont intégralement détruits :

```
$ > ./a.out
~Boisson()
```

(suite sur la page suivante)

(suite de la page précédente)

```
~Article()
~Article()
```

Ceci signifie que dans une hiérarchie de classes, l'appel au destructeur d'une classe fille appelle automatiquement le destructeur de la classe mère, et ainsi jusqu'à la classe de base.

## 4.4 Héritage et accessibilité

Nous avons vu précédemment que tout attribut ou toute méthode déclaré en `private` dans une classe parente devient inaccessible dans la classe dérivée. Que déclarés en `public` ils deviennent `private` dans la classe dérivée.

Cette règle découle du mode de dérivation qui est par défaut le mode `private`. Pour le faire apparaître explicitement il aurait fallu écrire :

```
class Boisson : private Article {...
```

Ce mode de dérivation peut être changé par un mode de dérivation moins restrictif. Il en existe au total trois :

- le mode de dérivation `private` : le mode par défaut utilisé jusqu'à présent. Ce qui prévaut dans cette règle c'est que les utilisateurs de la classe dérivée n'ont pas à savoir qu'elle dérive d'une autre classe. Cela relève uniquement de la construction du code par le concepteur de la classe ;
- le mode de dérivation `protected` : l'implémentation de la classe de base reste accessible aux concepteurs des classes dérivées, mais pas aux utilisateurs des classes dérivées ;
- le mode de dérivation `public` : L'interface de la classe de base reste accessible aux concepteurs des classes dérivées, ainsi qu'aux utilisateurs des classes dérivées.

Ces trois règles se caractérisent par les trois schémas d'accessibilité suivants :

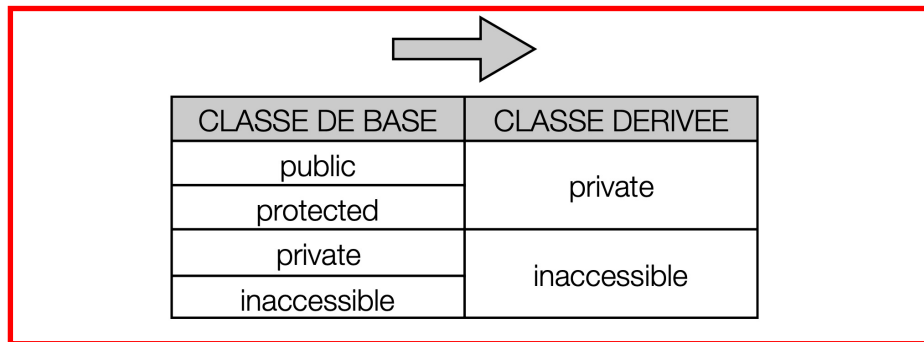


Fig. 1 – Le schéma de dérivation `private`.

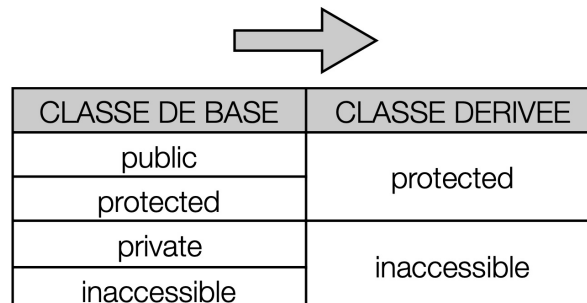
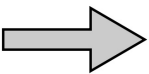


Fig. 2 – Le schéma de dérivation `protected`.



CLASSE DE BASE	CLASSE DERIVEE
public	public
protected	protected
private	inaccessible
inaccessible	

Fig. 3 – Le schéma de dérivation public.

## 4.5 Conversions

Les conversions de type sont courantes en programmation. En C++, deux conversions de type existent :

- les conversions implicites : elles sont réalisées par le compilateur ;
- les conversions explicites : elles sont imposées par le programmeur (*cast*).

Ainsi, la conversion suivante est implicite car réalisée par le compilateur :

```
int main() {
    int i = 10;
    double d;

    d = i;          // conversion implicite int -> double

    return 0;
}
```

En revanche la conversion inverse doit être imposée explicitement car non admise par le compilateur :

```
int main() {
    int i;
    double d = 5.5;

    i = int( d ); // conversion explicite (cast) double -> int

    return 0;
}
```

Le résultat de cette conversion reste incertain. Pour être correcte il faut écrire le code qui permet de convertir correctement un double en un entier, en décidant par exemple d'extraire la partie entière :

```
int main() {
    int i;
    double d = 5.5;

    i = int( floor( d ) ); // conversion explicite (cast) double -> int

    return 0;
}
```



Le même principe de conversion existe pour les objets définis en C++, **mais uniquement dans le cadre de la dérivation “public”**. Le principe est que la conversion d’un objet d’une classe fille vers un objet d’une classe mère est implicite tandis que la conversion contraire doit être explicitement définie via un constructeur de conversion.

Exemple :

```
class Boisson : public Article {
    ...
}

int main() {

    Article a;
    Boisson b(5, 10);

    a = b;          // conversion implicite Boisson -> Article
    cout << "prix de a = " << a.quelprix() << endl;

    return 0;
}
```

Les attributs de la partie `Article` de l’objet `b` sont copiés dans l’objet `a`. Si l’inverse n’est pas possible (`b = a`), c’est parce que le compilateur ne serait pas quoi copier dans la partie `Boisson`.

Notez que l’affectation est générée par le compilateur car nous n’avons pas défini d’opérateur d’affectation.

Dans le cas des dérivations `private` et `protected` la conversion implicite n’est plus possible. Il faut définir un constructeur de conversion :



```
// obligatoire pour déclarer l'existence de la classe Boisson utilisée dans Article:
class Boisson;

class Article {
    double prix;
public:
    Article( double p=0 ) : prix(p) {}

    // constructeur de conversion Boisson -> Article:
    Article( const Boisson &);
    double quelprix() const { return prix; }
};

class Boisson : private Article {
    double volume;
public:
    Boisson( double p=0, double v=0 ) : Article(p), volume(v) {}

    double quelvolume() { return volume; }
    double quelprix() const { return Article::quelprix(); }
};

// constructeur défini en dehors de la classe:
Article::Article( const Boisson &b) {
    prix = b.quelprix();
}

int main() {
    Article a;
    Boisson b (5, 10);
}
```

(suite sur la page suivante)

(suite de la page précédente)

```
a = Article( b );           // conversion explicite obligatoire
cout << "prix de a = " << a.quelprix() << endl;

return 0;
}
```

Le constructeur de conversion est défini en dehors de la classe et obligatoirement après la définition de la classe **Boisson**. Ceci pour pouvoir faire référence aux méthodes de la classe **Boisson**.

### 5.1 Résolution statique de liens (ligature statique)

Avec le polymorphisme, on se donne la possibilité d'écrire du code dont le comportement change en fonction du type des données auxquelles il est appliqué.

Reprenons le jeu de classes définies au chapitre *Héritage* auquel on ajoute la classe fille `Biscuit` :

```
class Article {
    double prix;
public:
    Article( double p=0 ) : prix(p) {}
    void printype() const { cout << "Ceci est un article " << endl; }
};

class Boisson : public Article {
    double volume;
public:
    Boisson( double p=0, double v=0 ) : Article(p), volume(v) {}
    void printype() const { cout << "Ceci est une boisson " << endl; }
};

class Biscuit : public Article {
    double poids;
public:
    Biscuit( double p=0, double pds=0 ) : Article(p), poids(pds) {}
    void printype() const { cout << "Ceci est un biscuit " << endl; }
};
```

Le programme suivant permet d'afficher le type des articles sans soucis :

```
int main() {

    Boisson boisson;
```

(suite sur la page suivante)

(suite de la page précédente)

```

    boisson.printtype();

    Biscuit biscuit;
    biscuit.printtype();

    return 0;
}

```

```

$ > ./a.out
Ceci est une boisson
Ceci est un biscuit

```

Ecrivons maintenant une fonction générique qui permette d’afficher le type d’un article quelque soit l’objet passé en argument. Pour ce faire il convient de déclarer la classe de base en argument de la fonction dans le but d’obtenir une conversion implicite des objets qui seront passés ensuite en argument de la fonction. Rappelons que la conversion implicite n’est possible que dans le cas d’une dérivation public, sauf à définir un constructeur de conversion :

```

void affichetype( const Article &article) {
    article.printtype();
}

int main() {

    Boisson boisson;
    Biscuit biscuit;

    affichetype( boisson );
    affichetype( biscuit );

    return 0;
}

```

A l’exécution :

```

$ > ./a.out
Ceci est un article
Ceci est un article

```

Nous voyons donc dans cet exemple que c’est la méthode de la classe parente `Article::printtype()` qui est appelée, quelque soit le type de l’objet réel donné en argument.

Nous avons affaire ici à une *résolution statique des liens* : le choix de la méthode à appeler est opéré par le compilateur au moment de la compilation du programme. Le compilateur voit que le type de l’argument de la fonction est `Article`, il appelle donc la méthode `printtype()` de `Article`.

## 5.2 Résolution dynamique de liens (ligature dynamique)

Pour générer un code qui puisse prendre en compte le type réel des objets, c’est à dire le type connu au moment de l’exécution du programme, on met en oeuvre le mécanisme de *résolution dynamique des liens*.

Cette opération s’effectue très simplement à l’aide d’un mot clé nouveau, le mot clé du langage `virtual` que l’on doit placer en tête du prototype des méthodes concernées. Ici nous souhaitons que la bonne méthode `printtype()` des objets soit appelée. On les déclare donc *méthodes virtuelles* dans les classes :

```

class Article {
    double prix;
public:
    Article( double p=0 ) : prix(p) {}
    virtual void printype() const { cout << "Ceci est un article " << endl; }
};

class Boisson : public Article {
    double volume;
public:
    Boisson( double p=0, double v=0 ) : Article(p), volume(v) {}
    virtual void printype() const { cout << "Ceci est une boisson " << endl; }
};

class Biscuit : public Article {
    double poids;
public:
    Biscuit( double p=0, double pds=0 ) : Article(p), poids(pds) {}
    virtual void printype() const { cout << "Ceci est un biscuit " << endl; }
};

```

Le même programme donne à l'exécution :

A l'exécution :

```

$ > ./a.out
Ceci est une boisson
Ceci est un biscuit

```

### 5.2.1 Le mécanisme

Comment le compilateur peut-il savoir à l'avance quel sera le type des objets envoyés à la fonction ?

En définissant dans la classe - dont une méthode au moins est déclarée `virtual` - une table supplémentaire appelée `vtable` qui contient l'adresse de toutes des mêmes méthodes définies dans les classes descendantes.

L'objet `boisson` comporte deux parties : la partie `Article` et la partie `Boisson`. La partie `Article` incorpore la `vtable` qui sauvegarde les adresses des méthodes virtuelles. Ces méthodes restent appelables, y compris lorsque seule la première partie de l'objet est transmise en argument d'une fonction.

Ce mécanisme mis en place est intégralement géré par le compilateur et reste invisible au programmeur et à l'utilisateur des classes. Il n'y a donc pas lieu de s'en préoccuper.

### 5.2.2 Règles

Des règles sont à respecter en revanche lorsque l'on déclare une ou plusieurs méthodes virtuelles :

- La ligature dynamique n'a de sens que dans un contexte d'héritage ;
- Lorsqu'une méthode d'une classe est déclarée virtuelle, toutes les méthodes surchargées dans les classes descendantes sont également virtuelles. Il n'est d'ailleurs pas obligatoire de les déclarer `virtual`, mais c'est fortement conseillé pour la lisibilité du code ;
- Lorsqu'une méthode d'une classe est déclarée virtuelle, le destructeur doit l'être également. Dans le cas contraire, il y aurait risque de ne détruire qu'une partie de l'objet ;
- Les attributs et les constructeurs d'une classe ne peuvent pas être déclarés `virtual` ;
- Les méthodes virtuelles sont également gérées lorsque l'on référence des objets par leur adresse. Par exemple, la version de l'exemple ci-dessus avec passage par adresse fonctionne :

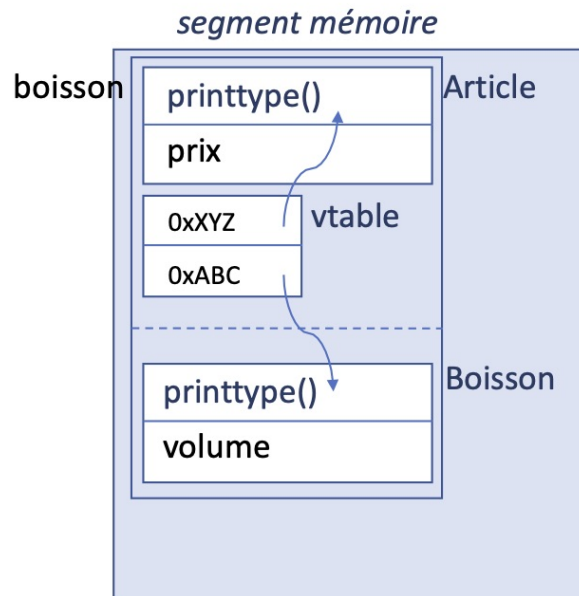


Fig. 1 – Représentation en mémoire de l'objet boisson.

```

void affichetype( Article *article) {
    article->printtype();
}

int main() {

    Boisson boisson;
    Biscuit biscuit;

    affichetype( &boisson );
    affichetype( &biscuit );

    return 0;
}

```

### 5.2.3 Utilisation avec les tableaux

La ligature dynamique prend tout son sens lorsque l'on désire réaliser des collections d'objets de classes différentes, la seule contrainte étant qu'ils dérivent toujours d'une même classe parente au sein de la hiérarchie des classes.

La première et plus simple collection est le *tableau* qui permet l'accès directe aux données via un indice (Mais un autre exemple serait de considérer par exemple des *listes*).

```

class Article {
    double prix;
public:
    Article( double p=0 ) : prix(p) {}
    double getPrix() const { return prix; }
    virtual void printtype() const { cout << "Ceci est un article " << endl; }
    virtual ~Article() {};
};

```

(suite sur la page suivante)

(suite de la page précédente)

```

class Boisson : public Article {
    double volume;
public:
    Boisson( double p=0, double v=0 ) : Article(p), volume(v) {}
    virtual void printtype() const { cout << "Ceci est une boisson (volume: " <<
    ↪ volume << "L, prix: " << getPrix() << "€)" << endl; }
};

class Biscuit : public Article {
    double poids;
public:
    Biscuit( double p=0, double pds=0 ) : Article(p), poids(pds) {}
    virtual void printtype() const { cout << "Ceci est un biscuit (poids: " << poids <
    ↪ "gr, prix: " << getPrix() << "€)" << endl; }
};

```

Notez que l'on a ajouté une méthode `getPrix()` dans l'interface de la classe de base pour pouvoir afficher le prix à partir des objets enfants. On a également affiné l'affichage en précisant le prix, le volume et le poids des articles.

Ci-dessous on construit donc un tableau permettant de stocker des objets dérivés de la classe `Article`. A titre d'exemple, deux objets sont déclarés en statique, puis deux autres en allocation dynamique. Les adresses de ces objets sont rangées dans le tableau. On peut à présent effectuer des traitements sur tous les éléments du tableau (ici leur affichage), et ce en tenant compte de leur classe réelle :

```

int main() {

    // déclaration du tableau des objets:
    Article* tab[100];

    // déclaration de deux objets en statique:
    Boisson boisson(2, 1);
    Biscuit biscuit(50, 2);

    // stockage dans le tableau:
    tab[0] = &boisson;
    tab[1] = &biscuit;

    // déclaration de deux autres objets en dynamique:
    Boisson *boisson2 = new Boisson(5, 5);
    Biscuit *biscuit2 = new Biscuit(100, 5);

    // stockage dans le tableau:
    tab[2] = boisson2;
    tab[3] = biscuit2;

    // affichage des données:
    for (int i=0; i<4; i++ )
        tab[i]->printtype();

    // destruction des objets dynamiques:
    delete tab[2];
    delete tab[3];

    return 0;
}

```

A l'exécution, on obtient nos informations sur tous les objets :

```
$ > ./a.out
Ceci est une boisson (volume: 1L, prix: 2€)
Ceci est un biscuit (poids: 2gr, prix: 50€)
Ceci est une boisson (volume: 5L, prix: 5€)
Ceci est un biscuit (poids: 5gr, prix: 100€)
```

### 5.3 Fonctions virtuelles pures et classes abstraites

Dans l'exemple présenté au paragraphe précédent, nous pouvons constater que la méthode `printype()` définie dans la classe de base n'est jamais utilisée.

Dès lors, il est inutile de la définir et nous pouvons la remplacer par quelque chose comme :

```
class Article {
    double prix;
public:
    Article( double p=0 ) : prix(p) {}
    double getPrix() const { return prix; }
    virtual void printype() const {} // méthode vide car jamais utilisée
    virtual ~Article() {};
};
```

Il existe une autre façon de faire qui consiste à la déclarer comme étant une méthode virtuelle *pure*. La syntaxe est la suivante :

```
class Article {
    double prix;
public:
    Article( double p=0 ) : prix(p) {}
    double getPrix() const { return prix; }
    virtual void printype() const = 0; // méthode virtuelle pure
    virtual ~Article() {};
};
```

Mais cela a une conséquence qui doit être sérieusement examinée :

Toute classe comportant au moins une méthode virtuelle pure est une *classe abstraite*. Or une *classe abstraite* est une classe à partir de laquelle on ne peut pas instancier d'objets.

Par exemple la ligne suivante sera refusée par le compilateur :

```
int main() {
...
    Article a; // error: variable type 'Article' is an abstract class
...
}
```

Donc la question qu'il faut se poser avant de déclarer une méthode virtuelle pure c'est de savoir s'il est intéressant d'avoir une classe de base *abstraite*. Si la réponse à la question :

```
au vu de ma définition de classe, très générale, cela a-t-il en sens de déclarer des
→ instances de cette classe ?
```

est oui, alors la réponse est oui, on peut déclarer dans cette classe une méthode virtuelle pure et faire ainsi de la classe une *classe abstraite*.



Dans notre exemple, nous ne déclarerons jamais d'objet `Article` car ce type est trop général. Lorsqu'on parle d'article, on envisage un objet concrêt, c'est à dire un biscuit ou une boisson.

Définir une classe abstraite revient à imposer aux concepteurs des classes dérivées une interface obligatoire. En effet, toutes les méthodes qui sont déclarées *virtuelles pures* dans la classe abstraite doivent être obligatoirement implémentées dans les classes descendantes sous peine d'erreur à la compilation.



### 6.1 Compilation séparée

Lorsqu'un programme commence à devenir important, on a intérêt à l'organiser en modules distincts que l'on va devoir compiler séparément. Chaque module est écrit dans un fichier qui lui est propre. La compilation du fichier génère alors un fichier *objet* qui sous linux porte le nom du fichier avec l'extension `.o` (`.obj` avec d'autres compilateurs). Ce fichier contient le code binaire du code source, mais il n'est pas exécutable pour autant car il manque les codes binaires des fonctions et classes définies dans les autres fichiers.

C'est l'éditeur de lien qui permet de générer un exécutable à partir des fichiers objets générés.

Soit par exemple une application composée d'un fichier source `prog.cc` contenant la définition de la fonction `main()`, et d'un deuxième fichier source `piledechar.cc`.

Les commandes de compilation seront les suivantes :

```
$ > cc -c prog.cc
$ > cc -c piledechar.cc
```

A ce stade les deux fichiers objets `prog.o` et `piledechar.o` sont créés.

On peut procéder à l'édition de lien :

```
$ > cc prog.o piledechar.o -lstdc++ -o prog
```

Cette fois un fichier exécutable `prog` est créé que l'on peut exécuter :

```
$ > ./prog
```

#### 6.1.1 Architecture

On décompose l'application en modules, mais comment procéder ? quels modules identifier ?

La programmation objet apporte une solution naturelle à cette question : une classe et son implémentation sont définies dans un unique fichier source dédié.

Par exemple un programme doit utiliser des objets de type `PileDeChar` sera défini sur deux fichiers à compiler séparément : un fichier source qui contient la fonction principale et un fichier source qui contient la définition de la classe.

Pour autant, cette organisation ne suffit pas. Le programme contenant la fonction `main()` et déclarant donc des objets de type `PileDeChar` ne peut pas être compilé correctement si la classe et son interface ne lui sont pas connues. C'est le rôle des fichier d'en-tête.

### 6.1.2 Fichiers d'en-tête

Les fichiers d'en-tête sont des fichiers c++ comportant des définitions de classe, des définitions de constantes, des déclarations de fonctions, etc., susceptibles d'être utilisées dans votre programme. La directive au pré-compilateur `#include` permet d'inclure ces fichiers :

```
#include <iostream.h>

int main() {
    ...
}
```

Il faut comprendre que le pré-compilateur insère le fichier `iostream.h` avant l'appel au compilateur, et que donc ce dernier analysera votre programme source auquel aura été ajouté préalablement le fichier d'en-tête, à la position du code où se situe la directive `include`.

Attention, une autre notation est possible qui utilise les guillemets : `#include "iostream.h"`. La différence est que dans ce cas le pré-compilateur cherche le fichier dans votre répertoire courant, sans utiliser le chemin de recherche défini lors de l'installation du compilateur et qui définit l'emplacement des libraries.

En résumé, pour l'inclusion de vos fichiers personnels, utilisez `#include "monfichier.h"` sans quoi le pré-compilateur ne les trouvera pas. A l'inverse, pour les librairies standards, utilisez `#include <iostream.h>` sans quoi le pré-compilateur ne les trouvera pas non plus.

Lorsque l'on définit une classe et son interface, on écrit un fichier d'en-tête, par exemple `piledechar.h`, qui doit être inclu dans tous les programmes qui utilisent la classe `PileDeChar` :

*piledechar.h*

```
class PileDeChar {
private:
    char *mPile;      // tableau contenant les caractères
    int mMax;         // dimension du tableau
    int mSommet;      // niveau de la pile
public:
    PileDeChar( int d=10 );
};
```

On ne développe pas les méthodes dans la classe mais dans le fichier source destiné à cela et compilé séparément : `piledechar.cc` :

*piledechar.cc*

```
#include "piledechar.h"

PileDeChar::PileDeChar(int d) {
    mMax = d;
```

(suite sur la page suivante)

(suite de la page précédente)

```

    mSommet = 0;
    mPile = new char [d];
}

```

Notez l'inclusion de `piledechar.h` en tête du fichier, indispensable au compilateur. De la même façon, le fichier `prog.cc` qui utilise la classe `PileDeChar` doit l'inclure :

*prog.cc*

```

#include "piledechar.h"

int main() {
    PileDeChar p(100);

    return 0;
}

```

Après compilation on obtient donc deux fichiers objets. L'un qui contient le code binaire de l'application (`prog.o`), l'autre le code binaire de la classe (`piledechar.o`).

Lorsque le code de la classe est au point, il n'a plus vocation à changer et il n'est donc plus utile de le compiler à chaque écriture d'un programme applicatif. Compilation et édition de lien peuvent alors se faire en une seule étape :

```
$ > cc prog.c piledechar.o -lstdc++ -o prog
```

### 6.1.3 Bibliothèques

Lorsqu'un ensemble de classes permettant d'implémenter une fonctionnalité est au point, le concepteur peut réaliser une bibliothèque à partir de l'ensemble des fichiers objets générés. L'utilisateur de la classe n'aura alors plus qu'à insérer la bibliothèque dans sa ligne de commande, exactement comme on le fait déjà avec la bibliothèque standard `stdc++`. La commande qui permet de réaliser une bibliothèque (archive) sous unix/linux est `ar`.

Ci-dessous nous créons une archive qui contient le binaire de la classe `PileDeChar`. l'option `cr` permet de créer l'archive si elle n'existe pas déjà (`c`) et d'insérer le code (`r`) présent dans `piledechar.o` :

```
$ > ar cr libpiledechar.a piledechar.o
```

Notez que la bibliothèque générée est *statique* par défaut (le code est inséré dans l'exécutable au moment de l'édition de lien plutôt qu'au moment de l'exécution). Le nom donné à l'archive reprend celui du fichier (`piledechar`) précédé des trois lettres `lib` (obligatoire) et prend pour extension `a` qui signifie qu'il s'agit d'une archive statique.

A présent notre programme peut être compilé comme suit :

```
$ > cc prog.cc -lpiledechar -lstdc++ -L./
```

La partie `lib` n'est pas mentionnée. l'option `-L` demande au compilateur de chercher les bibliothèques dans le répertoire courant.

Pour que tout cela fonctionne correctement, le concepteur de la classe devra mettre à disposition des utilisateurs le fichier à inclure `piledechar.h`.

### 6.1.4 Questions/réponses

Pourquoi ne peut-on inclure directement `piledechar.cc` dans `prog.cc` ?

Parce que cela reviendrait à renoncer à la compilation partagée. En effet le programme suivant se suffirait à lui-même pour générer un exécutable :

```
#include "piledechar.cc"

int main() {
    PileDeChar p(100);

    return 0;
}
```

Et si vous continuez à faire de la compilation séparée malgré cela, alors vous incluez deux fois le code binaire de votre classe, une fois dans le programme objet `prog.o`, une deuxième fois dans le programme objet `piledechar.o`. L'éditeur de lien renverra une erreur.

## 6.2 Compilation conditionnelle

Il peut arriver que des inclusions multiples de fichiers d'en-tête `.h` surviennent. Un moyen commode d'éviter cela est d'ajouter quelques lignes dans les fichiers d'en-tête permettant de réaliser des inclusions conditionnelles de la forme *inclure le fichier XXX.h s'il n'a pas déjà été inclu*.

Quand une inclusion multiple peut elle survenir ?

Votre programme utilise la classe `PileDeChar` et inclue donc le fichier `piledechar.h`. Mais il requiert également des éléments définis dans un autre fichier d'en-tête, mettons `xxx.h`.

Il se trouve que `piledechar.h` utilise également des éléments définis dans `xxx.h`. Il inclue donc également ce fichier, mais vous n'êtes pas sensé le savoir.

En résumé, les inclusions suivantes dans votre programme :

```
#include "piledechar.h"
#include "xxx.h"

int main() {
    ...
}
```

engendrent une double inclusion de `xxx.h` puisque `xxx.h` est déjà inclu dans `piledechar.h`.

La solution consiste à encadrer systématiquement votre code dans tous vos fichier d'en-tête par :

*piledechar.h*

```
#ifndef PILEDECHAR_H
#define PILEDECHAR_H

    ... // votre code
#endif
```

Votre code sera inclu seulement si la constante `PILEDECHAR_H` n'est pas déjà définie. Si une inclusion de votre fichier a déjà eu lieu, alors la constante `PILEDECHAR_H` est déjà définie et votre code n'est pas inclu.

Le nom de la constante est arbitraire. On choisit de donner un nom lié au fichier (ici son nom en majuscules) afin d'éviter toute collision avec d'autres constantes qui porteraient le même nom. Enfin c'est une convention très largement adoptée que d'utiliser les majuscules pour définir les constantes.

## 6.3 Chaînes de caractères

On hérite en C++ de ce qui n'existait déjà pas en C, à savoir l'absence d'un type spécifique pour représenter les chaînes de caractères. A défaut on utilise les tableaux de caractères. Certaines conventions existent qui doivent être comprises et respectées.

Les constantes chaînes de caractères sont toujours représentées entre guillemets. A ne pas confondre avec les caractères qui sont représentés entre cotes :

```
int main() {

    char a = 'a';           // variable de type char (caractère) initialisée avec
    ↪ le caractère 'a'
    char str[10] = "bonjour"; // tableau de caractères initialisé avec la constante
    ↪ chaîne de caractères "bonjour"
    return 0;
}
```

Dans `char a = 'a'` c'est en fait le code ASCII du caractère 'a' qui est affecté à la variable. Ca peut se voir facilement avec l'affichage suivant en convertissant le caractère en entier avant de l'envoyer sur le flux :

```
#include <iostream>
int main() {

    char a = 'a';
    str::cout << "le code ascii du caractère " << a << " est: " << int(a) <<
    ↪ str::endl;
    return 0;
}
```

A l'exécution :

```
$ > le code ascii du caractère a est: 97
```

`str` est un tableau de caractères pouvant contenir 10 caractères. La chaîne "bonjour" en contient 7 qui sont donc positionnés dans les 7 premières cases du tableau `str`.

Par convention les chaînes de caractère se terminent obligatoirement par le caractère spécial `'\0'` qu'on appelle le *caractère nul*. C'est ce qui permet au flux `cout` dans l'exemple ci-dessous de n'afficher que les 7 premières cases du tableau et non les 10 : le flux arrête l'affichage lorsqu'il rencontre le caractère nul. Mais cette convention est utilisée par toutes les fonctions de traitement des chaînes de caractères. Son non respect conduit à des erreurs d'exécution.

```
#include <iostream>
int main() {

    char str[10] = "bonjour";
    str::cout << "str = " << str << str::endl;
    return 0;
}
```

Comme pour tout tableau on accède à un élément du tableau, donc à un caractère, en utilisant l'opérateur `[]` :

```
...
cout << "le 3ème caractère de la chaîne est " << str[2] << endl;
...
```

Si l'on ne souhaite pas calculer la longueur de la chaîne à la main, la déclaration suivante laisse le compilateur faire ce travail :

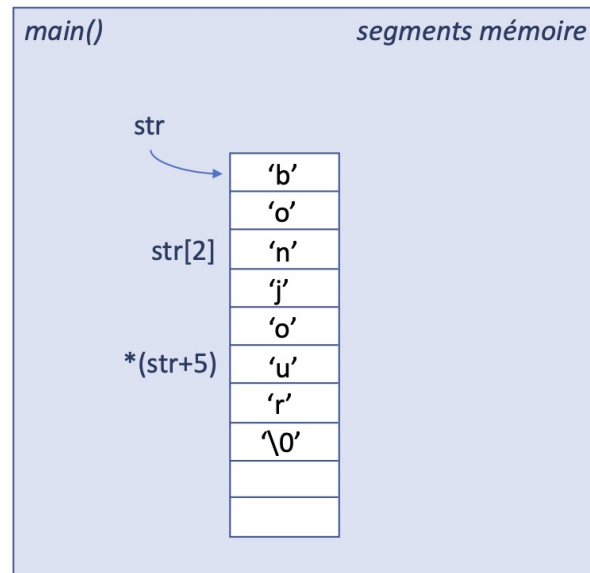


Fig. 1 – Représentation en mémoire du tableau de caractères contenant une chaîne.

```
#include <iostream>
int main() {

    char str[] = "ceci est une chapine de caractères";    // taille du tableau
    // définie par le compilateur
    str::cout << "str = " << str << str::endl;
    return 0;
}
```

### 6.3.1 Gestion dynamique

Représenter des chaînes dans des tableaux statiques comme ci-dessus (la taille du tableau est fixée à la compilation) atteint vite ses limites. Pour représenter des chaînes de longueur variable on en vient donc à utiliser des tableaux dynamiques.

#### Rappels

L'adresse d'un tableau est par convention l'adresse de la première case mémoire du tableau.

Puisque `tab[0]` désigne la première case du tableau, alors `&tab[0]` désigne l'adresse du tableau. Cette adresse est obtenue plus simplement en mentionnant uniquement le nom du tableau. On aura donc toujours `tab` identique à `&tab[0]`.

#### Allocation dynamique

Pour allouer un tableau devant contenir une chaîne d'au maximum 100 caractères :

```
#include <iostream>
int main() {
```

(suite sur la page suivante)



(suite de la page précédente)

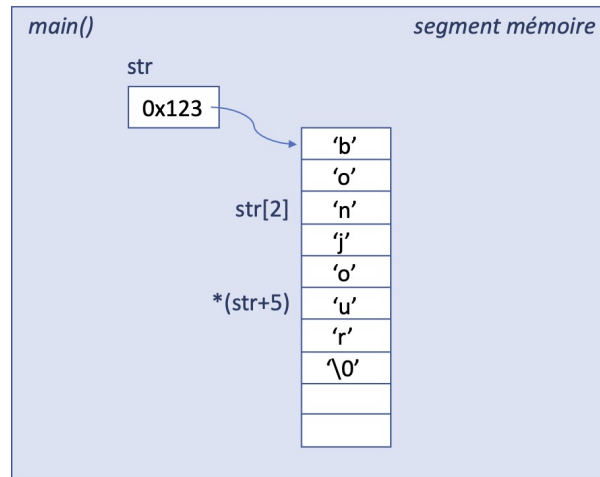
```

char *str = new char [100];

return 0;
}

```

La figure ci-dessous illustre la différence avec le cas statique : `str` devient une variable pointeur sur des caractères. Elle contient l'adresse du tableau alloué dynamiquement.



A ceci près que la chaîne est allouée, mais elle est vide en réalité. Comment affecter la constante chaîne de caractères "bonjour" au tableau que nous venons d'allouer ?

Le type chaîne n'existant pas, une affectation directe comme `str = "ceci est une chaîne de caractères"` n'est pas possible.

La solution est en réalité de copier la chaîne, caractères après caractères, dans le tableau...

Heureusement, la librairie standard du langage C dispose de fonctions pour traiter les chaînes de caractères. En incluant l'en-tête `string.h` nous pouvons faire appel à ces fonctions :

```

#include <iostream>
#include <string.h>

int main() {

    char *str = new char [100];
    strcpy( str, "bonjour" ); // copie la chaîne "bonjour" dans le tableau str
    cout << "str = " << str << endl;

    return 0;
}

```

A l'affichage :

```
$ > str = bonjour
```

Notez bien que la chaîne affichée se limite à celle copiée et non au tableau entier de 100 caractères. Les fonctions de traitement de chaînes gèrent donc correctement le caractère nul de fin de chaîne sans qu'on ait à le faire.

### Vers un type `string`

En C++ il devient possible d'envisager la conception d'une classe dont le rôle serait de représenter les chaînes de caractère et d'implémenter tous les traitements afférant aux chaînes. Il s'agit d'un cas d'école étudié en détail au chapitre qui traite des *objets C++*. Tous les éléments ci-dessus vous sont donnés car ils doivent être connus pour écrire une telle classe.

Mais la librairie standard propose également cette classe, appelée `string`, que l'on utilisera avantageusement. Exemple :

```
#include <iostream>
#include <string>

using namespace std;

int main() {

    string str = "bonjour";           // déclaration et initialisation
    cout << "str = " << str << endl;

    string bye = "aurevoir";

    string hello = str + ' ' + bye;   // concaténation
    cout << "hello = " << hello << endl;

    return 0;
}
```

A l'exécution :

```
bonjour
bonjour aurevoir
$ >
```