



université
de BORDEAUX

COMPTE-RENDU DE TD PROGRAMMATION ORIENTÉE OBJET

NOTIONS DE BASE : CONSTRUCTEURS, DESTRUCTEURS, FONCTIONS MEMBRES ET AMIES

Date : 10 mars 2020

M1 Bioinformatique

Réalisé par : Abou KOUTA et Julie PRATX

Enseignantes : Jenny Benois-Pineau & Marie Beurton-Aimar

Années : 2019/2020

SOMMAIRE

I. Introduction	2
II. Cahier des charges	2
III. Mise en œuvre	4
IV. Tests & Résultats	5
1. Exercice 1	5
2. Exercice 2	6
3. Exercice 3	6
4. Exercice 4	7
5. Exercice 5	8
6. Exercice 6	9
7. Exercice 7	10
8. Exercice 8	11
9. Exercice 9	12
10. Exercice 10	13
11. Exercice 11	14
12. Exercice 12	15
13. Exercice 13	16
V. Conclusion.....	17
VI. Perspectives	17
VII. Références bibliographiques	17

I. Introduction

Le langage C++ a été développé au cours des années 1980 par Bjarne Stroustrup [1]. Ce langage peut être considéré comme une « amélioration » du langage C dont certaines fonctions ont été ajoutées, comme, par exemple :

- ✓ la gestion de l'allocation de la mémoire avec les opérateurs **new** et **delete**,
- ✓ les références **&**,
- ✓ la notion de **classe** et d'**héritage** (constructeurs et destructeurs),
- ✓ la surcharge des opérateurs,
- ...

Le langage C++ utilise la **programmation orienté objet** et permet d'utiliser les **classes**, l'**encapsulation** (public, privé, protégé) et bien d'autres choses.

Dans ce compte-rendu, nous allons traiter les notions (de base) suivantes :

- savoir définir des **constructeurs** et des **destructeurs** dans une classe,
- découvrir le rôle des mot-clés **const** et **static**,
- comprendre le fonctionnement et l'intérêt des fonctions **en ligne**,
- comprendre le concept de **surdéfinition**,
- savoir définir une fonction **amie**.

II. Cahier des charges

Dans l'*exercice 1*, il nous est demandé de créer un main ainsi qu'une classe *Personne* avec trois constructeurs et un destructeur associé. Le premier prendra pour paramètre le nom, le prénom et l'âge (qui seront préalablement déclarés avant : nom et prénom sous forme de tableau de 20 caractères et l'âge par un entier) ; le deuxième n'aura pas de paramètre et le dernier sera un constructeur de copie. Enfin, pour tester la validité de notre classe, on nous propose de créer une fonction d'affichage.

Dans l'*exercice 2*, on devra remplacer le tableau (précédemment créé) auquel on a défini un nombre maximum de caractère en tableau dynamique (dont la taille peut varier). On devra alors modifier les constructeurs pour qu'ils allouent la mémoire nécessaire lors de la création d'un objet mais aussi le destructeur dont le rôle est de libérer cette même mémoire.

Dans l'*exercice 3*, on doit créer une classe *pile_entier* pour gérer une pile d'entiers que l'on devra ensuite stocker dans un tableau dynamique. La classe devra comporter plusieurs fonctions membres dont un constructeur allouant dynamiquement un emplacement de *n* entiers (*pile_entier(int n)*), de même avec un emplacement de 20 entiers (*pile_entier()*), un constructeur de copie créant une copie de la pile d'entier *pile* (*pile_entier (pile_entier &pile const)*), un destructeur (*~pile_entier()*), une fonction qui empile l'entier *p* sur la pile (*void empile(int p)*), une fonction qui fournit la valeur de l'entier en haut de la pile et le supprime de la pile (*int depile()*), une fonction qui renvoie 1 si la pile est pleine et 0 si elle est vide (*int pleine()*) et enfin, une fonction qui renvoie 1 si la pile est vide et 0 si elle est pleine (*int vide()*).

Dans l'exercice 4, il nous est demandé de faire la comparaison entre les deux constantes MAX1 et MAX2 définies de deux façons différentes. Il nous est également demandé d'accéder à l'adresse de ces pointeurs et de commenter ce qui se passe lorsqu'on le fait.

Dans l'exercice 5, on doit écrire une fonction qui a pour rôle d'afficher un entier avec une fonction affiche dont l'un de ces paramètres est une constante. Puis, de commenter ce qui arrive lorsqu'on tente de modifier l'entier à l'intérieur de la fonction. Enfin, il nous est demandé de rajouter dans la classe Personne (de l'exercice 2) une fonction membre d'affichage comme ceci *void affiche () const* et décrire ce qui se passe quand on tente de modifier l'objet courant à l'intérieur de la fonction.

Dans l'exercice 6, il nous est demandé de définir un pointeur sur un entier *p (int *p)*, un autre sur un entier constant *q (const int *q)* et un pointeur constant sur un entier *r (int* const r)*. On doit ensuite faire une allocation de mémoire à l'aide de l'opérateur *new*. Puis définir un tableau de dix pointeurs sur des entiers, refaire une allocation de mémoire et une désallocation de mémoire (en vérifiant l'état de la mémoire). Enfin, on doit définir une référence sur un entier *s (const int& t)* et expliquer les résultats entre toutes les affectations.

Dans l'exercice 7, on doit donner la différence entre deux types de « fonction s » et préciser si la copie s'effectue dans les deux cas. Ensuite, on doit changer le type de paramètre de ces deux « fonctions » et dire ce qui se passe. On nous demande enfin si on a le droit de définir des fonctions récursives de cette manière.

Dans l'exercice 8, nous devons créer trois fonctions « somme » dont la première additionnera deux entiers, la deuxième deux réels de type float et enfin la troisième, deux tableaux de dix entiers. Nous allons devoir faire quelques tests dont celui de changer le type des paramètres.

Dans l'exercice 9, on doit écrire une classe Vecteur3D qui contient des données privées, deux fonctions membres ainsi que deux constructeurs. Puis, nous devons modifier le programme pour intégrer les fonctions en ligne.

Dans l'exercice 10, nous devons ajouter au programme de l'exercice précédent des fonctions permettant d'accéder aux coordonnées d'un vecteur, puis des fonctions de les modifier.

Dans l'exercice 11, on nous demande de rajouter à la classe Vecteur3D de l'exercice 10 les fonctions suivantes : pour calculer le produit scalaire de *v* avec l'objet courant (*double produit_scalaire(Vecteur3D v)*), pour calculer le vecteur somme de *v* avec l'objet courant (*Vecteur3D somme(Vecteur3D v)*).

Dans l'exercice 12, nous devons changer une fonction de l'exercice 10 en utilisant le prototype suivant *friend int coincide (Vecteur3D v1, Vecteur3D v2)* et donner la définition du mot-clé *friend*.

Dans l'exercice 13, on nous donne un script des classes Matrice et Vecteur pour réaliser le produit d'une matrice avec un vecteur. On nous pose ensuite des questions auxquelles on doit répondre.

III. Mise en œuvre[2]

La première partie de ce compte-rendu traite des fonctions **constructeur** et **destructeur**. Tout d'abord, intéressons-nous aux **constructeurs**. Rappelons, tout d'abord, que lors de la création d'un objet, l'une des opérations les plus courantes est l'initialisation de ses membres de données. Le programme n'a, toutefois, pas accès à la totalité du programme. En effet, il y a des « parties » de la classe qui sont privées et où seul les constructeurs y ont accès. Par définition, une fonction constructeur est une fonction spéciale appelée automatiquement à chaque création d'objet. Elle a le même nom que la classe de l'objet, ainsi, si la classe de l'objet s'appelle *animal* alors le constructeur s'appelle *animal*. Une fois le constructeur (`nom_de_classe`) défini, une déclaration d'objet lui transmet des paramètres comme ceci : **`nom_de_classe objet (valeur1, valeur2, valeur3)`**

Ensuite, parlons des **destructeurs**. Une fonction destructeur sert à libérer de la mémoire en supprimant l'objet créé par le constructeur. Cette fonction a le même nom que la classe de l'objet mais elle est précédée par le caractère tilde (~) comme ceci : **`~nom_de_classe()`**

La deuxième partie de ce compte-rendu sert à découvrir le rôle des mot-clés **const** et **static**. Commençons par le mot-clé **const**. Pour plus de lisibilité dans le programme, on préférera employer une constante plutôt qu'une valeur numérique (quand cela est nécessaire). Cela permettra de mieux « lire » le programme et surtout de mieux le comprendre. Une constante est, par définition, un nom auquel le programme attribue une valeur numérique (mais aussi une chaîne de caractères) qui ne peut évidemment pas varier pendant son exécution. La constante est définie à l'aide de la directive¹ de préprocesseur **`#define`**. Par exemple, l'instruction suivante affecte à la constante **CLASSE** la valeur intangible de 50 : **`#define CLASSE 50`**

Si on souhaite modifier la valeur numérique de cette constante, il suffit de changer sa valeur avant l'exécution du programme et non pas de parcourir le programme pour modifier toutes les valeurs.

Pour le cas du mot-clé **static**, le but étant de réduire la quantité de mémoire nécessaire en éliminant un certain nombre d'une même information. Grâce à ce mot-clé, une méthode de classe peut être appelée même si le programme n'a encore défini aucun objet du type de la classe.

La troisième partie de ce compte-rendu sert à comprendre à quoi sert et quel est l'intérêt d'utiliser des fonctions **en ligne** lors de la programmation en C++. Pour commencer, on pourrait se poser la question suivante : *A quoi sert une fonction en ligne ?* Tout simplement, à permettre au programme de s'exécuter un peu plus rapidement. En effet, les fonctions, en général, ont un inconvénient majeur, celui d'alourdir les programmes et d'augmenter leur temps d'exécution. Pour bien comprendre comment ça se passe, il faut savoir plusieurs choses. Lorsque nous définissons une fonction, le compilateur en convertit le code en langage machine et ne prend qu'une copie de ses instructions. Il remplace ensuite chaque appel de fonction par des instructions qui ont pour effet d'envoyer les paramètres sur la pile et de brancher le programme sur les instructions de la fonction. Une fois la fonction exécutée, le programme « saute » à l'instruction qui en suit l'appel et reprend son cours normal. Tout ceci ralentit l'exécution du programme. On utilise donc le mot-clé **inline** avant la définition d'une fonction pour que le compilateur remplace dans le code exécutable les appels à cette fonction par les instructions que cette fonction exécute. Nous avons donc un programme plus rapide et dont le programme conserve la clarté qu'offre l'usage des fonctions.

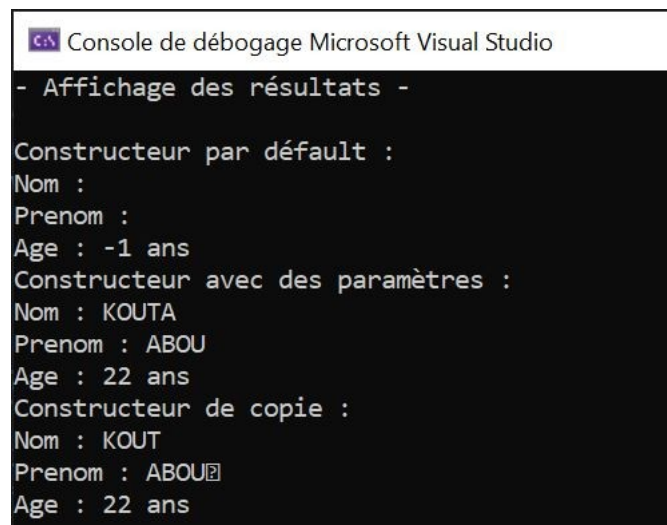
1 Directive : c'est une instruction spéciale à l'intention du préprocesseur du compilateur

La quatrième partie de ce compte-rendu sert à comprendre le concept de **surdéfinition**. En effet, une fonction membre d'une classe est une fonction qui a sa définition ou son prototype dans la définition de classe comme toute autre variable. Il opère sur tout objet de la classe dont il est membre et a accès à tous les membres d'une classe pour cet objet [3]. La notion de surdéfinition est due au fait que l'on va changer le type des paramètres de la fonction sans réécrire cette fonction. Comme par exemple : **void nom_de_fonction (int valeur1, int valeur2)** en **void nom_de_fonction (long valeur1, long valeur2)**

La cinquième et dernière partie de ce compte-rendu va nous servir à définir ce qu'est une fonction **amie**. En règle générale, le programme ne peut accéder aux membres privés d'une classe qu'en passant par les fonctions d'interface de cette classe. La privatisation des membres d'une classe à l'aide du mot-clé *private* permet de limiter les risques d'erreur puisque les fonctions d'interface en contrôlent l'accès. Certaines classes auraient toutefois intérêt à accéder plus librement à des membres privés d'autres classes. Une classe peut être définie comme *friend* d'une autre classe et ainsi accéder directement aux membres privés. Cela accélérerait également le temps d'exécution du programme.

IV. Tests & Résultats

1. Exercice 1 :



```
CA Console de débogage Microsoft Visual Studio
- Affichage des résultats -
Constructeur par défaut :
Nom :
Prenom :
Age : -1 ans
Constructeur avec des paramètres :
Nom : KOUTA
Prenom : ABOU
Age : 22 ans
Constructeur de copie :
Nom : KOUT
Prenom : ABOU
Age : 22 ans
```

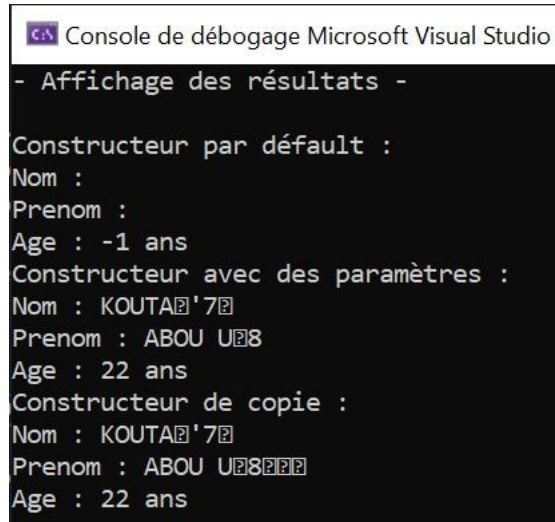
Figure 1: Exercice 1 test du script

Le script étant assez long, il ne sera pas présenté directement ici, mais vous pourrez le trouver dans le zip fourni avec ce document.

Sur la **Figure 1**, on a affiché trois résultats pour les trois constructeurs demandés dans l'énoncé de l'exercice. Seul le destructeur ne fonctionne pas.

On remarque que pour le constructeur par défaut, il n'y a pas de donnée. Ceci est normal, car on a initialisé les variables. Pour ce qui est du constructeur avec arguments (Nom, Prénom et Age), tout c'est affiché correctement. Par contre, pour le constructeur de copie, il semblerait qu'il y ait un petit problème.

2. Exercice 2 :



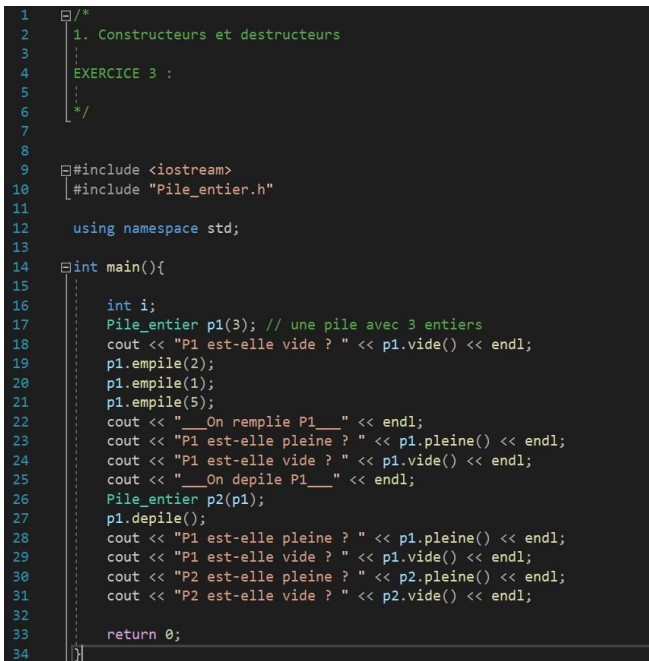
```
Console de débogage Microsoft Visual Studio
- Affichage des résultats -

Constructeur par défaut :
Nom :
Prenom :
Age : -1 ans
Constructeur avec des paramètres :
Nom : KOUTA'7
Prenom : ABOU U8
Age : 22 ans
Constructeur de copie :
Nom : KOUTA'7
Prenom : ABOU U8
Age : 22 ans
```

Figure 2 : Exercice 2 test du script

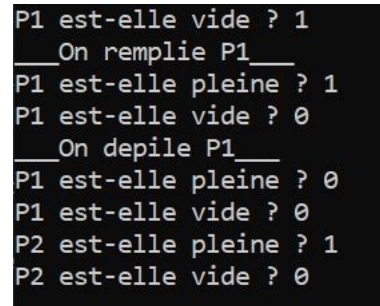
Pour le script, même chose que pour l'exercice 1. Ce qui change par rapport au premier exercice, c'est qu'on voit apparaître des caractères étrangers (**Figure 2**).

3. Exercice 3 :



```
1  /*
2   1. Constructeurs et destructeurs
3
4   EXERCICE 3 :
5
6   */
7
8
9  #include <iostream>
10 #include "Pile_entier.h"
11
12 using namespace std;
13
14 int main(){
15
16     int i;
17     Pile_entier p1(3); // une pile avec 3 entiers
18     cout << "P1 est-elle vide ? " << p1.vide() << endl;
19     p1.empiler(2);
20     p1.empiler(1);
21     p1.empiler(5);
22     cout << "___On remplit P1___" << endl;
23     cout << "P1 est-elle pleine ? " << p1.pleine() << endl;
24     cout << "P1 est-elle vide ? " << p1.vide() << endl;
25     cout << "___On depile P1___" << endl;
26     Pile_entier p2(p1);
27     p1.depiler();
28     cout << "P1 est-elle pleine ? " << p1.pleine() << endl;
29     cout << "P1 est-elle vide ? " << p1.vide() << endl;
30     cout << "P2 est-elle pleine ? " << p2.pleine() << endl;
31     cout << "P2 est-elle vide ? " << p2.vide() << endl;
32
33     return 0;
34 }
```

Figure 3: Exercice 3 main.cpp



```
P1 est-elle vide ? 1
___On remplit P1___
P1 est-elle pleine ? 1
P1 est-elle vide ? 0
___On depile P1___
P1 est-elle pleine ? 0
P1 est-elle vide ? 0
P2 est-elle pleine ? 1
P2 est-elle vide ? 0
```

Figure 4: Exercice 3 test

Sur la **Figure 3**, on peut voir le contenu du main et sur la **Figure 4**, son résultat après compilation et exécution.

4. Exercice 4 :

```
1  /*
2  2. Mot-clés const et static
3  ...
4  EXERCICE 4 : Quelles différences il y a t-il entre les deux constantes MAX1 et MAX2 définies de la façon suivante ?
5  */
6
7  #include <iostream>
8
9  using namespace std;
10
11 #define MAX1 100
12 static const int MAX2 = 100;
13
14 int main(){
15
16     int* p1 = &MAX1;
17     int* p2 = &MAX2;
18
19 }
20
```

Figure 5: Exercice 4 script

Sur la **Figure 5**, `#define MAX1 100` définit la constante. Une fois la constante définie, le programme peut utiliser sa valeur en donnant seulement son nom, mais on ne peut pas accéder à sa valeur. Aucun espace mémoire n'est créé pour cette variable. Tandis que `static const int MAX2 = 100;` est une variable dont la valeur est à 100 (pareil que MAX1) dont on a alloué une place dans la mémoire et qui peut être appelée par le programme à tout moment.

E0150	l'expression doit être une lvalue ou un désignateur de fonction	main	main.cpp	16
E0144	impossible d'utiliser une valeur de type "const int *" pour initialiser une entité de type "int *"	main	main.cpp	17

Figure 6: Erreurs de compilation du script de l'exercice 4

Lorsqu'on compile le programme présenté en **Figure 6**, on obtient deux erreurs (présentées en Figure*).

La première concerne la ligne 16 du programme et stipule : *l'expression doit être une lvalue ou un désignateur de fonction*, on peut expliquer cela par on ne peut pas attribuer une valeur à gauche du signe '='. On ne peut donc pas créer de pointeurs sur une variable définie avec `#define`.

Rappelons qu'un pointeur est une variable dont la particularité est de contenir une adresse mémoire de type donné. C'est en général une adresse virtuelle attribuée par le processus en cours. Elle permet de 'pointer' sur un contenu pour pouvoir le lire ou le modifier.

La deuxième concerne la ligne 17 du programme et stipule : *impossible d'utiliser une valeur de type « const int » pour initialiser une entité de type « int * »*, on peut expliquer cela car on ne peut pas créer un pointeur (p2) contenant l'adresse de MAX2.

5. Exercice 5 :

```
1  /*
2  2. Mot-clés const et static
3  ...
4  EXERCICE 5 :
5  - Ecrire une fonction affichant un entier : void affiche(const int& n). Que signifie le const
6  dans le prototype de la fonction affiche ? Que se passe t-il s'il ont tente de modifier l'entier
7  n à l'intérieur de la fonction affiche ?
8  ...
9  */
10
11 #include <iostream>
12 #include "main.h"
13
14 using namespace std;
15
16 int main()
17 {
18     affiche(4);
19 }
20
21 void affiche(const int& n) {
22     // int n = 4;
23     cout << n << endl;
24 }
25
26 }
```

Figure 7: Exercice 5 script

Que signifie le const dans le prototype de la fonction affiche ?

Cela signifie que cette une valeur constante et qu'elle ne peut pas être modifiée dans la fonction. (voir **Figure 7**)

Que se passe t-il s'il ont tente de tente de modifier l'entier n à l'intérieur de la fonction affiche ?

On a une erreur qui nous dit « redéfinition du paramètre formel 'n' » ce qui est normal.

C2082 redéfinition du paramètre formel 'n' main main.cpp 24

Figure 8: Erreur sur le script

```
void Personne::affiche() const {
    this->age = 0;
    cout << "Nom : " << nom << endl;
    cout << "Prenom : " << prenom << endl;
    cout << "Age : " << age << " ans" << endl;
}
```

Figure 9: Ajout d'un objet courant

E0137 l'expression doit être une valeur modifiable

Figure 10: Erreur dû à l'ajout d'un objet courant

Lorsque l'on essaie de modifier l'objet courant à l'intérieur de la fonction membre affiche, on se retrouve avec cette erreur (voir **Figure 10**).

6. Exercice 6 :

```
1  /*
2  2. Mot-clés const et static
3  ...
4  EXERCICE 6 :
5  ...
6  */
7
8  #include <iostream>
9
10 using namespace std;
11
12 int main(){
13
14     int s = 1;
15     const int t = 1;
16
17     int *p = &s;
18     const int *q = &s;
19     int* const r = &s;
20
21     int *p = &t;
22     const int *q = &t;
23     int* const r = &t;
24
25     p = new int;
26     q = new int;
27     r = new int;
28
29     int** tableau = new int* [10];
30     for (int i = 0; i < 10; i++) { tableau[i] = new int; }
31     cout << tableau << endl;
32     for (int i = 0; i < 10; i++) { delete tableau[i]; }
33     cout << tableau << endl;
34
35     return 0;
36 }
```

Figure 11: Exercice 6 script

abc	E0144	impossible d'utiliser une valeur de type "const int *" pour initialiser une entité de type "int *"
abc	E0144	impossible d'utiliser une valeur de type "const int *" pour initialiser une entité de type "int *const"
abc	E0137	l'expression doit être une valeur modifiable

Figure 12: Erreurs de l'exercice 6

Le script de l'exercice 6 est en **Figure 11**. Les erreurs rencontrées à la compilation sont présentées sur la **Figure 12**.

Sur la **Figure 11**, pour les lignes 17, 18 et 19, cela fonctionne. Mais pas pour une association avec une constante, à savoir, les lignes 21, 22 et 23.

Pour expliquer ce phénomène, prenons comme exemples :

- `const Fred* p` signifie « p pointe vers un Fred qui est constant » : l'objet Fred ne peut pas être modifié par p.
- `Fred* const p` signifie « p est un pointeur constant vers un Fred » : l'objet Fred peut être modifié par p mais ne peut pas changer le pointeur lui-même.
- `const Fred* const p` signifie « p est un pointeur constant vers un Fred qui est constant » : on ne peut pas changer le pointeur p et on ne peut pas changer l'objet Fred par p.

Les lignes 29 à 33 représente le tableau de dix pointeurs sur des entiers.

7. Exercice 7 :

```
10 using namespace std;
11
12 #define copie1(source,dest) source = dest;
13
14 inline void copie2(int source, int dest) {
15     source = dest;
16 }
17
18
19
20 /* ... */
21
22 int main(){
23     double a = 1.01;
24     double b = 5.04;
25
26     int c = 1;
27     int d = 56;
28
29     cout << "Copie 1" << endl;
30     copie1(a, b);
31     copie1(c, d);
32
33     cout << a << endl;
34     cout << b << endl;
35     cout << c << endl;
36     cout << d << endl;
37
38     cout << "\nCopie 2" << endl;
39     copie2(a, b);
40     copie2(c, d);
41
42     cout << a << endl;
43     cout << b << endl;
44     cout << c << endl;
45     cout << d << endl;
46
47     return 0;
48 }
```

Figure 13: Exercice 7 script

Console de débogage Microsoft Visual Studio

```
Copie 1
5.04
5.04
56
56

Copie 2
5.04
5.04
56
56
```

Figure 14: Exercice 7 tests

La différence des deux fonctions (voir **Figure 13**) est dans les mots-clés qui les précèdent. Une est définie avec le mot-clé *define* et l'autre avec le mot-clé *inline* (ces deux termes ont déjà été définis dans la partie III. Mise en œuvre à la page 4).

Lorsque l'on appelle les deux fonctions, elles fonctionnent (voir **Figure 14**) que ce soit avec des entiers de type *int* ou de type *double*.

On peut tout à fait définir une fonction récursive de cette façon, mais le compilateur l'implémentera normalement donc sans sa propriété *inline*. Du fait que les fonctions *inline* sont insérées telles quelles aux endroits où elles sont appelées, il est nécessaire qu'elles soient complètement définies avant leur appel. Cela signifie que, contrairement aux fonctions classiques, il n'est pas possible de se contenter de les déplacer pour les appeler mais de fournir leur définition dans un fichier séparé.

8. Exercice 8 :

```
6   int somme(int a, int b);
7   float somme(float a, float b);
8   int somme(int* tab_A, int* tab_B);
9   int somme(int a, int b, int c);
10  int somme(float a, int b);
```

Figure 15: Exercice 8 : Prototypes des fonctions

Que se passe-t-il lorsqu'un appel est fait avec comme arguments deux short ?

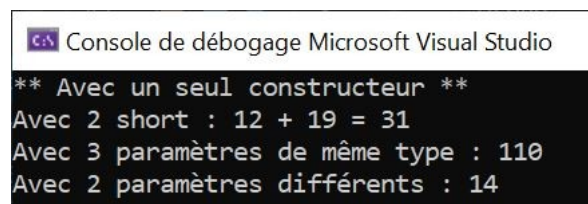
Le constructeur choisi par le programme est le 1^{er}, celui qui prend pour paramètre deux int (voir **Figure 15**).

Est-il possible de créer une fonction somme prenant 3 paramètres de même type ?

Oui, c'est possible et ça fonctionne sans retourner d'erreur (voir Figure).

Est-il possible de définir une fonction somme ayant deux paramètres de types différents (par exemple un int et un float) ?

Oui, c'est possible et ça fonctionne (voir **Figure 16**). Cependant, si l'addition se fait avec une fonction `int somme()`, les décimales du `float` seront tronquées. C'est le compilateur qui décide quelle fonction il appelle et ceci en fonction du nombre et du type de paramètre.



```
Console de débogage Microsoft Visual Studio
** Avec un seul constructeur **
Avec 2 short : 12 + 19 = 31
Avec 3 paramètres de même type : 110
Avec 2 paramètres différents : 14
```

Figure 16: Exercice 8 : Résultats

9. Exercice 9 :

```
1  /*
2  4. Fonctions membres et surdéfinition
3
4  EXERCICE 9 : Ecrivez une classe Vector3D comprenant :
5  - trois données membre de type double x, y et z (privées)
6  - deux fonctions membres d'affichage :
7      - void affiche() affichant le vecteur
8      - void affiche(const char* string) affichant le string avant l'affichage du vecteur
9  - deux constructeurs :
10     - l'un sans argument, initialisant chaque composante à 0
11     - l'autre, avec trois arguments correspondant aux coordonnées du vecteur.
12  Modifiez ensuite le code pour que les constructeurs soit des fonction en ligne (inline).
13  */
14
15  #include <iostream>
16  #include "Vecteur3D.h"
17
18  using namespace std;
19
20  int main(){
21
22
23      cout << "- Affichage des résultats - " << endl;
24      cout << "Affiche V1 :" << endl;
25      Vecteur3D V1;
26      V1.affiche();
27      Vecteur3D V2(23, 76, 11);
28      V2.affiche("Affiche moi V2 :");
29
30      return 0;
31  }
```

Figure 17: Exercice 9 script

```
Console de débogage Microsoft Visual Studio
- Affichage des résultats -
Affiche V1 :
x : 0
y : 0
z : 0
Affiche moi V2 :
x : 23
y : 76
z : 11
```

Figure 18: Exercice 9 résultats

Pour que le code puisse être compilé une fois que les constructeurs sont devenus des fonctions en ligne, il faut déplacer le corps des constructeurs dans le fichier main.cpp en dessous de la fonction `int main()`.

10. Exercice 10 :

```
1  /*
2  4. Fonctions membres et surdéfinition
3
4  EXERCICE 10 : Rajouter les fonctions suivantes à la classe Vecteur3D :
5  - Des fonctions permettant d'accéder au coordonnées d'un vecteur :
6    - int abscisse() pour x
7    - int ordonnée() pour y
8    - int cote() pour z
9  - Des fonctions permettant de modifier les coordonnées d'un vecteur :
10   - void fixer_abscisse(int nouvelle_abscisse) pour x
11   - void fixer_ordonnée(int nouvelle_ordonnée) pour z
12   - void fixer_cote(int nouvelle_cote) pour z
13   - bool coincide(Vecteur3D v) qui renvoie true si v et l'objet courant
14   ont les mêmes coordonnées, sinon false.
15  */
16
17 #include <iostream>
18 #include "Vecteur3D.h"
19
20 using namespace std;
21
22 int main(){
23
24     Vecteur3D v1;
25     Vecteur3D v2(4.2, 5, 6.1);
26     v1.affiche();
27     v2.affiche("test");
28     v2.fixer_abscisse(5);
29     v2.fixer_ordonnee(7);
30     v2.fixer_cote(1);
31     cout << "Abscisse v2 : " << v2.abscisse() << endl;
32     cout << "Ordonnee v2 : " << v2.ordonnee() << endl;
33     cout << "Cote v2 : " << v2.cote() << endl;
34     Vecteur3D v3(5, 7, 1);
35     cout << "Coincide v1 et v2 : " << v1.coincide(v2) << endl;
36     cout << "Coincide v2 et v3 : " << v2.coincide(v3) << endl;
37 }
38
39
```

Figure 19: Exercice 10 script

```
Console de débogage Microsoft Visual Studio
x = 0 y = 0 z = 0
String = test
x = 4.2 y = 5 z = 6.1
Abscisse v2 : 5
Ordonnee v2 : 7
Cote v2 : 1
Coincide v1 et v2 : 0
Coincide v2 et v3 : 1
```

Figure 20: Exercice 10 test

11. Exercice 11 :

```
1  /*
2  4. Fonctions membres et surdéfinition
3  ...
4  EXERCICE 11 :
5  ...
6  */
7
8  #include <iostream>
9  #include "Vecteur3D.h"
10
11  using namespace std;
12
13
14  int main(){
15
16      Vecteur3D v1;
17      Vecteur3D v2(4.2, 5, 6.1);
18      v1.affiche();
19      v2.affiche("test");
20      v2.fixer_abcisse(5);
21      v2.fixer_ordonnee(7);
22      v2.fixer_cote(1);
23      cout << "Abscisse v2 : " << v2.abcisse() << endl;
24      cout << "Ordonnee v2 : " << v2.ordonnee() << endl;
25      cout << "Cote v2 : " << v2.cote() << endl;
26      Vecteur3D v3(5, 7, 1);
27      cout << "Coincide v1 et v2 : " << v1.coincide(v2) << endl;
28      cout << "Coincide v2 et v3 : " << v2.coincide(v3) << endl;
29
30      cout << "Produit scalaire v2 et v3 : " << v2.produit_scalaire(v3) << endl;
31      Vecteur3D vaffiche = v2.somme(v3);
32      vaffiche.affiche();
33  }
34
```

Figure 21: Exercice 11 script

```
Console de débogage Microsoft Visual Studio
x = 0 y = 0 z = 0
String = test
x = 4.2 y = 5 z = 6.1
Abscisse v2 : 5
Ordonnee v2 : 7
Cote v2 : 1
Coincide v1 et v2 : 0
Coincide v2 et v3 : 1
Produit scalaire v2 et v3 : 75
x = 10 y = 14 z = 2
```

Figure 22: Exercice 11 test

12. Exercice 12 :

```
1  /*
2   5. Fonctions amies
3   :
4   EXERCICE 12 :
5   */
6
7  #include <iostream>
8  #include "Vecteur3D.h"
9
10 using namespace std;
11
12 int main(){
13
14
15     Vecteur3D v1;
16     Vecteur3D v2(4.2, 5, 6.1);
17     v1.affiche();
18     v2.affiche("test");
19     v2.fixer_abscisse(5);
20     v2.fixer_ordonnee(7);
21     v2.fixer_cote(1);
22     cout << "Abscisse v2 : " << v2.abscisse() << endl;
23     cout << "Ordonnee v2 : " << v2.ordonnee() << endl;
24     cout << "Cote v2 : " << v2.cote() << endl;
25     Vecteur3D v3(5, 7, 1);
26     cout << "Coincide v1 et v2 : " << coincide(v1, v2) << endl;
27     cout << "Coincide v2 et v3 : " << coincide(v2, v3) << endl;
28
29     cout << "Produit scalaire v2 et v3 : " << v2.produit_scalaire(v3) << endl;
30     Vecteur3D vaffiche = v2.somme(v3);
31     vaffiche.affiche();
32
33
34
35
36 }
37
```

Figure 23: Exercice 12 script

```
CA Console de débogage Microsoft Visual Studio
x = 0 y = 0 z = 0
String = test
x = 4.2 y = 5 z = 6.1
Abscisse v2 : 5
Ordonnee v2 : 7
Cote v2 : 1
Coincide v1 et v2 : 0
Coincide v2 et v3 : 1
Produit scalaire v2 et v3 : 75
x = 10 y = 14 z = 2
```

Figure 24: Exercice 12 test

Une fonction amie d'une classe peut accéder à tous ses éléments, même les éléments privés. L'amitié doit être utilisée avec parcimonie en C++, uniquement lorsque cela est nécessaire.

13. Exercice 13 :

```
1  /*
2  5. Fonctions amies
3
4  EXERCICE 13 :
5
6  */
7
8  #include <iostream>
9  #include "VecteurMatrice.h"
10
11  using namespace std;
12
13
14  int main(){
15
16      double t[3];
17      double s[3][3];
18      Matrice m1(s);
19      Vecteur v1(t);
20      Vecteur v2 = m1.produit(v1);
21
22      return 0;
23
24
25
26  }
27
```

Figure 25: Exercice 13 script

Il est d'usage d'implanter les constructeurs dans le .cpp et non dans la classe. Ici, ce n'est pas le cas car il y a un *define* : la portée de ce dernier se limite au fichier. C'est pourquoi on préfère implémenter les constructeurs dans la classe afin d'avoir ce *define* plutôt que de créer deux fichiers .cpp qui contiendraient chacun un *define*.

Si on essaie d'implémenter ces constructeurs à l'extérieur de la classe, cela ne fonctionnerait pas à cause de la portée du *define* qu'ils utilisent.

On peut déclarer la fonction *Vecteur produit(Matrice mat, Vecteur vect)* dans le même fichier que les classes. Pour accéder aux données membres des objets, il suffit de rajouter des accesseurs **getMat()** et **getVect()** qui vont renvoyer ces données privées dans les classes Vecteur et Matrice. C'est le principe du masquage de l'encapsulation. On garantit l'intégrité des données.

Pour accéder aux données membres de l'objet *vect*, il faut utiliser les accesseurs. On crée l'accesseur dans la classe *Vecteur* : *Vecteur getVect() return vect* ; On peut ensuite l'utiliser dans la fonction produite avec *vect.getVe*.

V. **Conclusion**

En réalisant ce TD, nous avons appris à nous servir un peu plus du langage C++. En effet, ces quelques notions de base sont très importantes et elles sont à maîtriser. C'est en faisant beaucoup d'exercices que l'on va arriver à progresser. Ce qui peut nous freiner et nous décourager (un peu), ce sont les erreurs dû au langage et elles sont nombreuses. On ne les comprend pas forcément au premier coup d'œil ce qui rend la tâche difficile. De plus, jongler entre plusieurs langages n'est pas forcément évident. Il est facile de se mélanger les pincesaux de temps à autres. Python, Java et maintenant C++ !

Nous sommes encore des débutants, mais il faut bien commencer quelque part.

VI. **Perspectives**

Ce TD n'est pas une fin en soi car le chemin est encore long. Il faut encore travailler pour que ça soit plus facile pour nous. Donc la seule chose qui nous reste à faire c'est programmer, programmer et encore programmer. Y a que comme ça que ça va marcher. Ce n'est qu'en programmant que l'on devient programmeur !

VII. **Références bibliographiques**

- [1] C++, Wikipédia, (2020).
- [2] K. Jamsa, Toute la puissance de C++ en quarante leçons, 1999.
- [3] Developpement-informatique.com, Fonctions membres en C++ | Développement Informatique, Dev. Inform. - Plateforme Pédagogique En Ligne, (n.d.).