

FRC - Programmer des robots en C++

Raphaël Pothier

Étudiant en Mathématiques à l'Université de Montréal
Ancien membre et mentor à Stan Robotix 6622

Document présenté par Stan Robotix 6622



Table des matières

Avant-propos	3
1 Les bases de la programmation en C++	7
1.1 Syntaxe, variables et I/O	8
1.1.1 Analyse d'un programme de base	8
1.1.2 Variables	9
1.1.3 Opérations	10
1.1.4 Conversion de type	10
1.1.5 Entrées et sorties	11
1.1.6 Commentaires	12
1.1.7 Exemple récapitulatif de la section 1.1	13
1.1.8 Conseils pour rédiger un programme	13
1.2 Structures conditionnelles et boucles	14
1.2.1 Expressions conditionnelles et opérateurs logiques	14
1.2.2 Structures conditionnelles	14
1.2.3 Boucles	17
1.2.4 Énumérations et structures conditionnelles à états	19
1.2.5 Exemple récapitulatif de la section 1.2	22
1.2.6 Conseils pour rédiger un programme	24
1.3 Vecteurs, tableaux et pointeurs	25
1.3.1 Vecteurs et tableaux	25
1.3.2 Pointeurs	28
1.3.3 Pointeurs et tableaux	29
1.3.4 Exemple récapitulatif de la section 1.3	30
1.3.5 Conseils pour rédiger un programme	30
1.4 Fonctions	32
1.4.1 Types de retour et paramètres	32
1.4.2 Paramètres par défaut et passage de paramètres par référence	35
1.4.3 Exemple récapitulatif de la section 1.4	37
1.4.4 Conseils pour rédiger un programme	38
1.5 Classes	39
1.5.1 Idée de base	39
1.5.2 Attributs, constructeurs et destructeurs	39
1.5.3 Encapsulation et opérateur de résolution de portée	41
1.5.4 Fichier d'en-tête et architecture classique d'un programme en POO	43
1.5.5 Pointeur sur un objet et attribut pointeur	45
1.5.6 Compléments sur le constructeur	47
1.5.7 Héritage	49
1.5.8 Exemple récapitulatif de la section 1.5	52
1.5.9 Conseils pour rédiger un programme	55
1.6 Espaces de noms	56

1.6.1	Idée de base	56
2	Utiliser Git et Github	59
2.1	Git	60
2.1.1	Idée de base	60
2.1.2	Définitions de base	60
2.1.3	Commandes de base	60
2.2	Github	62
2.2.1	Présentation	62
3	Le Command-Based Programming et WPILIB	63
3.1	Architecture	64
3.2	Subsystems	64
3.3	Digression sur les PID	68
3.3.1	Idée de base	68
3.3.2	Présentation détaillée	68
3.4	Commands	68
3.4.1	Idée de base	68
3.4.2	Présentation des méthodes de la classe <i>Command</i>	68
3.4.3	Enchaîner des <i>Commands</i> (nouvelle méthode)	72
3.4.4	Enchaîner des <i>Commands</i> (ancienne méthode)	73
3.5	RobotContainer	75
3.6	Robot	75
4	VS Code	77
4.1	Présentation	78
4.2	Liste de raccourcis indispensables	78
4.3	Utilisation avec WPILIB	78
4.3.1	Création d'un projet	78
4.3.2	Création d'un <i>Subsystem</i> ou d'une <i>Command</i>	79
4.3.3	Compilation et déploiement	79
4.3.4	Gestion des bibliothèques de tiers	79

Avant-propos

ATTENTION :

Ce manuel ne constitue pas un cours complet. Il simplifie ou ignore parfois certaines notions dans le but d'être compréhensible pour tous.

Objectifs du manuel :

1. Permettre aux jeunes d'accéder à une formation sur les bases du C++ et son utilisation dans le cadre de la compétition FIRST.
2. Encourager les STIM.

Ce manuel permet à n'importe quelle personne n'ayant aucune connaissance en programmation de parvenir à concevoir efficacement un programme pour faire fonctionner un robot utilisant un *RoboRIO*. Le langage C++, très utilisé notamment dans les systèmes embarqués et l'industrie du jeu vidéo, est d'abord présenté pour s'assurer que de solides bases soient établies. Vient par la suite le coeur du sujet, à savoir programmer un robot en utilisant le *Command-Based Programming*. On a privilégié cette méthode plutôt qu'une autre pour sa clarté, la possibilité de faire de gros programmes facilement et la possibilité de réutiliser du code très aisément.

Chapitre 1

Les bases de la programmation en C++

Définitions :

- La **console** est l'interface graphique de base dans laquelle sont affichées les différentes informations, s'il y a lieu. C'est également l'endroit où il faut écrire pour communiquer d'éventuelles données.
- Une **sortie** est une information affichée par le programme dans la console.
- Une **entrée** est une information que doit rentrer l'utilisateur dans la console.
- Un **mot-clé** est un mot spécifique directement reconnu par le programme. Les mots-clés sont habituellement affichés en bleu dans un éditeur.

1.1 Syntaxe, variables et I/O

1.1.1 Analyse d'un programme de base

```
1 #include <iostream>
2 using namespace std;
3
4 int main()
5 {
6     cout << "Mon premier programme.";
7
8     return 0;
9 }
```

Essayez d'exécuter le programme ci-dessus. Que remarquez-vous ?
L'information *Mon premier programme.* est une sortie.

Définition : On appelle **bloc** tout contenu qui se trouve entre deux accolades ({ et }).

Exemples :

1. Les lignes 5 à 9 du programme ci-dessus forment un bloc.
2. Considérons le programme suivant :

```
1 int main()
2 {
3     int nbr = 17;
4
5     for(int i = 0; i < 10; i++)
6     {
7         nbr += i * 2;
8     }
9
10    return 0;
11 }
```

Les lignes 2 à 11 forment un bloc. Les lignes 6 à 8 forment un autre bloc.

Remarque : Il est important de noter que des blocs peuvent se trouver à l'intérieur de d'autres blocs.

Définitions :

- La **fonction *main***, ou **fonction principale**, est le bloc qui est appelé par le programme lors de son exécution. Elle doit se trouver dans un fichier avec l'extension *.cpp*.
- L'**en-tête** du programme est la partie composée des lignes commençant par un croisillon (#). On peut également inclure les lignes commençant par le mot-clé *using*.

Exemples :

1. Dans les exemples précédents, la fonction *main* est directement visible. Elle constitue leur premier bloc.
2. Les lignes 1 et 2 du premier programme constituent une en-tête.

Proposition : Toute ligne ne précédant pas un bloc ou ne commençant pas par un croisillon doit se

terminer par un point-virgule (;). Les lignes constituées seulement d'une accolade ne sont pas non plus concernées.

1.1.2 Variables

Définitions :

- Une **variable** est un triplet **type**, **nom** et **valeur**. Il s'agit d'une structure permettant de stocker de l'information et de l'utiliser plus tard dans le programme. La valeur doit être spécifiée avant que la variable soit utilisée.
- La **déclaration** d'une variable est la ligne spécifiant le type et le nom de cette dernière.
- La **définition** d'une variable est la ligne spécifiant la valeur de la variable si elle est déclarée. Dans le cas contraire, cette ligne spécifie également le type et le nom de la variable.
- L'opérateur `=` est appelé opérateur d'**affectation**. Il assigne la valeur à sa droite à la variable qui est à sa gauche.

Exemple : Considérons le programme suivant :

```

1 int main()
2 {
3     int age;
4     float taille = 1.81;
5     double pi = 3.1415926535;
6     bool enTrainDapprendre;
7
8     return 0;
9 }
```

Les variables *age* et *enTrainDapprendre* sont déclarées. Les variables *taille* et *pi* sont définies. Elles ont pour valeur 1.81 et 3.1415926535.

Proposition : Voici une liste non exhaustive des types de variables en C++ :

- **int** (entier) : valeurs entières, y compris négatives.
- **float** (réel) : valeurs décimales, y compris négatives.
- **double** (réel) : valeurs décimales, y compris négatives. La précision est supérieure à *float*.
- **bool** (booléen) : valeurs **true** (**vrai**, ou 1) et **false** (**faux**, ou 0).

Remarques :

- Définir une variable avec le mauvais type conduit à une troncature (perte d'information). Il est primordial de s'assurer que le bon type est utilisé.
- N'utilisez pas d'accent, de cédille dans vos noms de variables, puisque ceux-ci pourraient ne pas être reconnus. De manière générale, éviter d'inclure des caractères spéciaux dans vos noms de variables pour éviter un comportement inattendu. Un nom de variable doit toujours commencer par un caractère qui n'est pas un chiffre.
- Deux variables peuvent avoir le même nom tant qu'elles sont dans des blocs disjoints. Dans le cas contraire, veillez à utiliser des noms différents. Une majuscule peut faire la différence !

1.1.3 Opérations

Le tableau suivant répertorie les différentes opérations majeures :

Expression	Résultat
$a + b$	a PLUS b
$a - b$	a MOINS b
$a * b$	a FOIS b
a / b	a DIVISÉ par b
$a \% b$	RESTE de a DIVISÉ par b

Remarques :

- Assurez-vous que vos variables soient bien définies avant de les utiliser dans des opérations !
- L'opération **modulo** (%) est seulement définie si a et b sont des entiers !
- Attention aux priorités opératoires ! Elles sont les mêmes qu'en mathématiques, donc vous pouvez utiliser au besoin les parenthèses qui ont la même fonction.

Exemple : Considérons les opérations suivantes avec les variables du programme de la page précédente :

```

1 age = 18;
2 int valeur = pi * 2 - 1;
3 float division = age / 12;
4 float age2 = age;
5 float division2 = age2 / 12;
```

La variable *valeur* contient 5. C'est une version tronquée du résultat attendu. Il se passe un phénomène similaire pour *division*. En effet, $age/12$ vaut 1 car *age* est un entier. La variable *age2* contient elle une version avec décimales du résultat.

Proposition : Il est possible de raccourcir certaines opérations précises. Utilisez le tableau suivant pour les connaître :

Version longue	Version courte
$a = a + b$	$a += b$
$a = a - b$	$a -= b$
$a = a * b$	$a *= b$
$a = a / b$	$a /= b$
$a = a \% b$	$a \% = b$
$a += 1$	$a++$
$a -= 1$	$a--$

Remarque : Attention ! Les lignes suivantes ne sont pas équivalentes :

```

1 a *= 2 + 1;
2 a = a * 2 + 1;
```

La ligne 1 renvoie $3a$ alors que la ligne 2 renvoie $2a + 1$.

1.1.4 Conversion de type

Définition : Une **conversion de type** est un changement forcé d'un type à un autre. Lors de cette conversion, une perte d'information peut se produire. Attention, il faut que la valeur de base ait un

sens dans le nouveau type.

Exemple : Considérons les lignes suivantes :

```
1 int age = 18;
2 float division = float(age) / 12;
3 int valeur = int(3 * division) % 2;
```

Ici, *division* stocke bien un nombre décimal (1,5) contrairement à un exemple précédent. Par ailleurs, l'opération modulo à la ligne 3 est bien définie car on s'assure de convertir $3 * \textit{division}$ en entier.

Remarque : Attention ! La conversion d'un nombre réel en nombre entier se fait toujours en arrondissant par défaut (on enlève la partie décimale).

1.1.5 Entrées et sorties

Propositions :

- L'instruction **cout** (*character output*) utilisée avec l'opérateur << permet d'afficher dans la console la valeur à la droite de l'opérateur. La valeur peut être stockée dans une variable mais cela n'est pas nécessaire.
- Il est possible d'afficher plusieurs informations à la suite en rajoutant l'opérateur <<.
- L'instruction **endl** (*end line*) est utilisée pour marquer un retour à la ligne.

Exemple : Le programme suivant permet d'afficher (sortie) des informations dans la console :

```
1 #include <iostream>
2 using namespace std;
3
4 int main()
5 {
6     int age = 18;
7
8     cout << "----" << endl;
9     cout << "L'âge de la personne est " << age << " ans." << endl;
10    cout << "----";
11
12    return 0;
13 }
```

Remarque : Le texte entre guillemets anglais (" ") est un type de valeur que l'on appelle chaîne de caractères. Nous ne verrons pas comment stocker ce type dans une variable.¹

Proposition : L'instruction **cin** (*character input*) utilisée avec l'opérateur >> permet de rentrer une valeur dans la console qui sera assignée à la variable à la droite de l'opérateur. Pour rentrer la valeur, on écrit tous les caractères nécessaires, puis on appuie sur la touche **Entrée** du clavier

Exemple : Le programme suivant permet de préciser le contenu d'une variable (entrée) lors de l'exécution :

```
1 #include <iostream>
2 using namespace std;
3
```

1. Voir au besoin le module *cstring*

```
4 int main()
5 {
6     int age;
7
8     cout << "Quel est votre âge ?" << endl;
9     cin >> age;
10    cout << "Votre âge est : " << age;
11
12    return 0;
13 }
```

Remarques :

- De manière similaire avec l'instruction *cout*, il est possible de rentrer plusieurs valeurs pour plusieurs variables à la suite en rajoutant à chaque fois l'opérateur *>>*.
- Assurez-vous de rentrer une valeur cohérente avec le type pour éviter des comportements imprévus !

1.1.6 Commentaires

Définition : Un **commentaire** est un ensemble d'une ligne ou plus n'ayant aucune influence sur le programme.

Exemples :

1. On utilise deux barres obliques (*//*) pour un commentaire sur une ligne :

```
1 #include <iostream>
2 using namespace std;
3
4 int main()
5 {
6     // Ceci est un commentaire
7     int nombre = 2; // La variable nombre vaut 2.
8     cout << nombre * 4; // On affiche le quadruple de nombre;
9
10    return 0;
11
12 }
```

2. On utilise la structure barre oblique-étoile... étoile-barre oblique (*/*...*/*) pour un commentaire multiligne :

```
1 int main()
2 {
3     int nombre = 1;
4
5     /* Voici un commentaire multiligne.
6     Le programme créer une variable nombre qu'il initialise à1.
7     Il calcule ensuite le résultat de l'opération 4 * nombre - 5 et le stocke
8     dans une variable resultat. */
9
10    int resultat = 4 * nombre - 5;
11    return 0;
12 }
```

1.1.7 Exemple récapitulatif de la section 1.1

```
1 #include <iostream>
2 using namespace std;
3
4 // Programme qui calcule les images de la
5 // fonction  $4x^3 - 3,5/x + 1$ 
6 int main()
7 {
8     float nombre;
9
10    cout << "Veuillez rentrer un nombre (x) :" << endl;
11    cin >> nombre;
12
13    cout << "4 * x * x - 3.5 / x + 1 = " << 4 * nombre * nombre * nombre - 3.5 / nombre +
14        float(1);
15
16    return 0;
17 }
```

1.1.8 Conseils pour rédiger un programme

1. Lister toutes les variables nécessaires et leur type.
2. Déterminer toutes les opérations nécessaires à effectuer.
3. Déclarer toutes les variables au début du programme.
4. S'assurer que toutes les variables sont définies avant d'être utilisées. La valeur peut provenir d'une valeur précise ou d'une entrée.
5. Effectuer les opérations sur les variables.
6. Utiliser une sortie si nécessaire.

Mots-clés de la section : *int*, *float*, *double*, *bool*, *include*, *using*, *main*

1.2 Structures conditionnelles et boucles

1.2.1 Expressions conditionnelles et opérateurs logiques

Définitions :

- Une **expression conditionnelle** est une expression dont les résultats sont booléens. Une telle expression peut être composée de plusieurs expressions de base (voir tableau).
- Un **opérateur logique** est un opérateur agissant sur des valeurs et / ou variables renvoyant comme résultat *true* ou *false*. Lorsqu'un opérateur logique est appliqué à des variables / valeurs, cela donne une expression conditionnelle.

Expression de base	Test	Négation
$a < b$	a STRICTEMENT INFÉRIEUR À b	$a \geq b$
$a \leq b$	a INFÉRIEUR OU ÉGAL À b	$a > b$
$a \geq b$	a SUPÉRIEUR OU ÉGAL À b	$a < b$
$a > b$	a STRICTEMENT SUPÉRIEUR À b	$a \leq b$
$a == b$	a ÉQUIVALENT À b	$a != b$
$a != b$	a DIFFÉRENT DE b	$a == b$

Définitions :

- Un **opérateur booléen** est un opérateur n'agissant que sur des expressions conditionnelles. Il renvoie une valeur booléenne.
- Une **table de vérité** est un tableau donnant les résultats possibles d'un opérateur booléen sur des expressions conditionnelles.

a ET b		
a	b	a && b
false	false	false
false	true	false
true	false	false
true	true	true

a OU b		
a	b	a b
false	false	false
false	true	true
true	false	true
true	true	true

NON a	
a	!a
false	true
true	false

Remarque : Les autres opérations logiques (non et, non ou...) sont constructibles à partir de ces trois opérations de base. Elles ne sont pas traitées car peu utiles dans notre situation.

1.2.2 Structures conditionnelles

Définitions :

- Une **structure conditionnelle** est une portion de code, composée d'un ou plusieurs blocs, qui évalue une ou plusieurs expressions conditionnelles et exécute différentes sections en fonction du résultat.
- Une structure **if** est un bloc évaluant une expression conditionnelle. Si l'expression renvoie la valeur *true*, alors une section supplémentaire de code est exécutée. Sinon, le programme continue son exécution classique.
- Une structure **if...else** est une section composée d'un bloc *if* et d'un bloc *else*, c'est-à-dire un bloc gérant les cas où l'expression conditionnelle est fausse. Une section supplémentaire, mais différente dans les deux cas, est exécutée.

- Une structure **if...else if.....else** avec n blocs **else if** est une section composée d'un bloc **if**, d'un bloc **else** et de n blocs **else if**, c'est-à-dire des blocs évaluant des expressions conditionnelles différentes à chaque fois des précédentes.

Le graphe suivant résume les différentes structures :

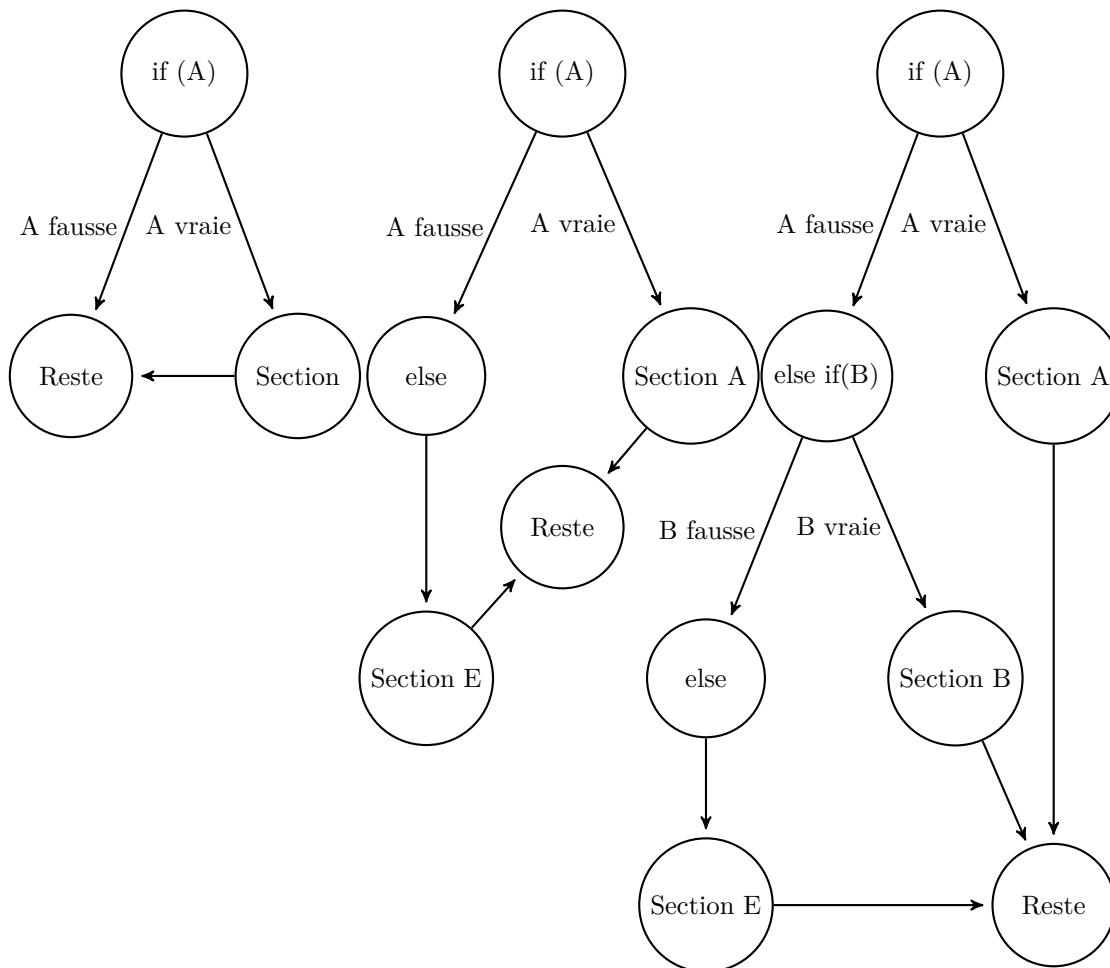


FIGURE 1.1 – Graphe illustrant le fonctionnement des structures conditionnelles

Commentaires sur le graphe :

- Les lettres A et B représentent différentes expressions conditionnelles.
- Les "sections" sont les portions de code exécutées lorsque l'expression évaluée est vraie. La section E est la section exécutée par le bloc **else**.
- Le terme **reste** signifie **reste du programme**, c'est-à-dire toutes les lignes de codes situées après des structures conditionnelles.
- Chaque évaluation d'une expression conditionnelle constitue une action qui prend un certain temps. Il est conseillé de déterminer les cas les plus probables et de disposer judicieusement les blocs pour optimiser le programme.

Exemples :

1. Le programme suivant détermine si le nombre donné par l'utilisateur est pair (divisible par 2) ou non :

```
1 #include <iostream>
2 using namespace std;
3
4 int main()
5 {
6     int nombre;
7
8     cout << "Veuillez donner un nombre : ";
9     cin >> nombre;
10
11     if (nombre % 2 == 0)
12     {
13         cout << nombre << " est un nombre pair.";
14     }
15
16     else
17     {
18         cout << nombre << " est un nombre impair.";
19     }
20
21     return 0;
22 }
```

2. Le programme suivant demande un nombre et renvoie :
 - (a) Son double s'il est inférieur à 2.
 - (b) Son triple s'il est inférieur à 8, si son carré est supérieur à 12 et s'il ne satisfait pas la condition 1.
 - (c) Son carré si son cube est inférieur ou égal à 2685.619 et ne satisfait pas la condition 2.
 - (d) Son cube sinon.

```
1 #include <iostream>
2 using namespace std;
3
4 int main()
5 {
6     float nombre;
7
8     cout << "Veuillez donner un nombre : ";
9     cin >> nombre;
10
11     if (nombre < 2)
12     {
13         cout << "Le double de " << nombre << " est " << 2 * nombre;
14     }
15
16     else if (nombre < 8 && nombre * nombre > 12)
17     {
18         cout << "Le triple de " << nombre << " est " << 3 * nombre;
19     }
20 }
```

```

21     else if (nombre * nombre * nombre <= 2685.619)
22     {
23         cout << "Le carré de " << nombre << " est " << nombre * nombre;
24     }
25
26     else
27     {
28         cout << "Le cube de " << nombre << " est " << nombre * nombre * nombre;
29     }
30
31     return 0;
32 }

```

1.2.3 Boucles

Définitions :

- Une **boucle** est un bloc répétant une série de ligne de code tant qu'une expression conditionnelle est vraie.
- Une boucle **while** continue d'**itérer** (de parcourir la boucle) à "l'inconnu". On ne sait pas exactement combien de fois la boucle sera itérée. La tête de cette boucle est constituée d'une condition à vérifier à chaque passage.
- Une boucle **for** itère de manière bornée. Il est possible de déterminer à l'avance le nombre d'itérations de la boucle. La tête de cette boucle est constituée d'une borne inférieure, d'une condition à vérifier à chaque itération et d'une action à effectuer après chaque passage.
- L'instruction **break** permet de quitter la boucle de la dernière itération.

Remarque : Attention à l'expression conditionnelle ! Prévoyez, lorsque cela est nécessaire, une condition d'arrêt supplémentaire pour éviter d'itérer indéfiniment !

Exemples :

1. Les programmes suivants affichent les 100 premiers nombres impairs de différentes manières (en commençant à 1) :

```

1  #include <iostream>
2  using namespace std;
3
4  int main()
5  {
6      int i = 1;
7      int compteur = 1;
8
9      while (compteur <= 100)
10     {
11         if (i % 2 == 1)
12         {
13             cout << i << endl;
14             compteur++;
15         }
16
17         i++;
18     }
19
20     return 0;

```

```
21 }  
  
1 #include <iostream>  
2 using namespace std;  
3  
4 int main()  
5 {  
6     for (int i = 1; i < 201; i += 2)  
7     {  
8         cout << i << endl;  
9     }  
10  
11     return 0;  
12 }
```

2. Le programme suivant affiche les 100 premiers nombres premiers (nombres seulement divisible par eux-même et 1) :

```
1 #include <iostream>  
2 using namespace std;  
3  
4 int main()  
5 {  
6     int compteur = 0;  
7     entierActuel = 2;  
8     bool premier;  
9  
10    while (compteur < 100)  
11    {  
12        premier = true;  
13  
14        for (int i = 2; i < entierActuel; i++)  
15        {  
16            if (entierActuel % i == 0)  
17            {  
18                premier = false;  
19            }  
20        }  
21  
22        if (premier)  
23        {  
24            cout << "Le nombre premier #" << compteur + 1 << " est " << entierActuel <<  
25                endl;  
26            compteur++;  
27        }  
28        entierActuel++;  
29    }  
30  
31    return 0;  
32 }
```

3. Le programme suivant affiche les puissances d'un nombre tant que le résultat est compris entre -1000 et 1000.

```

1  #include <iostream>
2  using namespace std;
3
4  int main()
5  {
6      float nombre;
7
8      cout << "Veuillez rentrer un nombre : ";
9      cin >> nombre;
10
11     float resultat = nombre;
12     int compteur = 0;
13
14     while (resultat > -1000 && resultat < 1000)
15     {
16         if (compteur == 100)
17         {
18             break;
19         }
20
21         cout << resultat << endl;
22
23         resultat *= nombre;
24
25         compteur++;
26     }
27
28     return 0;
29 }

```

Remarques :

- Le premier programme du premier exemple est nettement moins efficace que le second, puisqu'il effectue 201 itérations de boucle et autant de tests logiques ! L'autre programme lui ne fait que 100 itérations.
- Le programme du dernier exemple s'arrête, peu importe la situation, à 100 itérations. En perdant un peu de performance, on s'assure de quitter la boucle.

1.2.4 Énumérations et structures conditionnelles à états

Définition : Une **énumération** est une manière de créer des **types** de variables à valeurs finies. Plus précisément, on dit qu'elles peuvent créer des variables stockant des **états** spécifiques.

Origine	Type	Valeurs possibles
Type de base	<i>int</i>	42, -6622, 0...
Type de base	<i>float</i>	1.22, -42.6...
Type de base	<i>double</i>	3.14159, -726.82102...
Type de base	<i>bool</i>	<i>true</i> ou <i>false</i>
<i>enum</i>	TypeExemple	Valeur1, Valeur2, ..., ValeurN

Définition : Une structure **switch...case** est un bloc évaluant une expression conditionnelle d'équivalence sur une variable à états ou de type *int*. Elle est composée de plusieurs sections **case** qui gèrent les différents cas possibles.

Exemples :

1. Les programmes suivants affichent un message différent en fonction du résultat d'un dé, mais de manières différentes :

```
1 #include <iostream>
2 using namespace std;
3
4 int main()
5 {
6     int resultat;
7
8     cout << "Veuillez entrer le résultat du lancer : ";
9     cin >> resultat;
10
11     if (resultat == 1)
12     {
13         cout << "Vous avez fait 1 !";
14     }
15
16     else if (resultat == 2)
17     {
18         cout << "Vous avez fait 2 !";
19     }
20
21     else if (resultat == 3)
22     {
23         cout << "Vous avez fait 3 !";
24     }
25
26     else if (resultat == 4)
27     {
28         cout << "Vous avez fait 4 !";
29     }
30
31     else if (resultat == 5)
32     {
33         cout << "Vous avez fait 5 !";
34     }
35
36     else if (resultat == 6)
37     {
38         cout << "Vous avez fait 6 !";
39     }
40
41     else
42     {
43         cout << "Votre dé est étonnant !";
44     }
45
46     return 0;
47 }
```

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

```
3
4  int main()
5  {
6      int resultat;
7
8      cout << "Veuillez rentrer le résultat du lancer : ";
9      cin >> resultat;
10
11     switch(resultat)
12     {
13         case 1:
14             cout << "Vous avez fait 1 !";
15             break;
16
17         case 2:
18             cout << "Vous avez fait 2 !";
19             break;
20
21         case 3:
22             cout << "Vous avez fait 3 !";
23             break;
24
25         case 4:
26             cout << "Vous avez fait 4 !";
27             break;
28
29         case 5:
30             cout << "Vous avez fait 5 !";
31             break;
32
33         case 6:
34             cout << "Vous avez fait 6 !";
35             break;
36
37         default:
38             cout << "Votre dé est étonnant !";
39     }
40
41     return 0;
42 }
```

2. Le programme suivant affiche la vitesse d'un robot après application d'un "mode de vitesse" :

```
1  #include <iostream>
2  using namespace std;
3
4  enum ModeVitesse
5  {
6      modeLent,
7      modeMoyen,
8      modeRapide
9  };
10
11  int main()
12  {
13      float vitesse;
```

```

14     ModeVitesse mode = modeLent;
15
16     cout << "Veuillez rentrer la vitesse du robot : ";
17     cin >> vitesse;
18
19     switch(mode)
20     {
21         case modeLent:
22             vitesse *= 0.5;
23             break;
24
25         case modeMoyen:
26             vitesse *= 0.75;
27             break;
28
29         case modeRapide:
30             vitesse *= 0.9;
31             break;
32
33         default:
34             break;
35     }
36
37     cout << "La nouvelle vitesse est " << vitesse;
38
39     return 0;
40 }

```

Remarques :

- Le premier programme du premier exemple est nettement moins efficace que le second ! En effet, si l'utilisateur rentre 6, le programme va tester 6 conditions avant d'afficher le bon message ! Ce problème est réglé avec l'utilisation de la structure *switch...case* puisque le programme "sait" directement où aller, réduisant donc le temps d'exécution.
- Les instructions *break* sont très importantes ! Si elles ne sont pas présentes, tous les cas en-dessous du bon seront également exécutés !
- Après chaque mot-clé *case*, on place un double point " :".

1.2.5 Exemple récapitulatif de la section 1.2

Le programme suivant mesure la distance parcourue par un robot fictif. Ce robot est soumis à divers changements de vitesse en fonction du temps écoulé et de la distance parcourue. Essayez de retranscrire les expressions conditionnelles sous forme de mots.

```

1  #include <iostream>
2  using namespace std;
3
4  enum ModeVitesse
5  {
6      modeLent,
7      modeMoyen,
8      modeRapide
9  };
10
11 int main()

```

```
12 {
13     float vitesse = 6.5;
14     float tempsEcoule = 0;
15     float distanceParcourue = 0;
16
17     ModeVitesse mode = modeRapide;
18
19     while (tempsEcoule < 100)
20     {
21         if (distanceParcourue < 100 || tempsEcoule > 85)
22         {
23             mode = modeRapide;
24         }
25
26         else if (tempsEcoule < 60 && distanceParcourue > 100)
27         {
28             mode = modeLent;
29         }
30
31         else
32         {
33             mode = modeMoyen;
34         }
35
36         tempsEcoule += 0.5;
37
38         cout << "Temps écoulé : " << tempsEcoule << " Distance parcourue : ";
39
40         switch (mode)
41         {
42             case modeLent:
43                 distanceParcourue += 0.5 * vitesse * 0.5;
44                 cout << distanceParcourue << " Mode lent" << endl;
45                 break;
46
47             case modeMoyen:
48                 distanceParcourue += 0.5 * vitesse * 0.75;
49                 cout << distanceParcourue << " Mode moyen" << endl;
50                 break;
51
52             case modeRapide:
53                 distanceParcourue += 0.5 * vitesse * 0.9;
54                 cout << distanceParcourue << " Mode rapide" << endl;
55                 break;
56
57             default:
58                 break;
59         }
60     }
61
62     return 0;
63 }
```

1.2.6 Conseils pour rédiger un programme

1. Déterminer clairement l'objectif du programme. Au besoin, séparer l'objectif en plusieurs sous-objectifs.
2. Déterminer les opérations répétées sur les variables. Anticipez les boucles infinies !
3. Déterminer les opérations à exécuter seulement dans certains cas. Transcrire ces cas sous forme d'expressions conditionnelles.
4. Ne pas hésiter à afficher le contenu des variables pour contrôler les résultats.

Mots-clés de la section : *if, else, while, for, break, enum, switch, case, default*

1.3 Vecteurs, tableaux et pointeurs

1.3.1 Vecteurs et tableaux

Définition : Un **vecteur** est un type de donnée pouvant stockant un nombre indéterminé de valeurs d'un type précisé. Le type *vector* n'est pas un type de base et nécessite l'inclusion du **module** *vector* (*iostream* est aussi un module). On ne parlera pas de module avant la section 1.5.

Proposition : Le tableau suivant répertorie 3 opérations importantes sur les vecteurs :

Fonction	Code	Type de valeur renvoyé
Ajouter un élément elem	vecteur.push_back(elem)	Rien
Accéder à l'élément i	vecteur.at(i)	Type du vecteur
Obtenir la taille du vecteur	vecteur.size()	size_t

Exemples :

1. Le programme suivant modifie chaque nombre d'un vecteur en son carré. Il affiche ensuite la moyenne du premier et du dernier élément :

```

1 #include <iostream>
2 #include <vector>
3 using namespace std;
4
5 int main()
6 {
7     vector<float> nombres = {0.31, 8.12, -38.1, 642.02};
8
9     for (int i = 0; i < nombres.size(); i++)
10     {
11         nombres.at(i) = nombres.at(i) * nombres.at(i);
12     }
13
14     cout << (nombres.at(0) + nombres.at((int)nombres.size() - 1)) / 2;
15
16     return 0;
17 }
```

2. Le programme suivant stocke tous les nombres premiers entre 1 et 100 dans un vecteur :

```

1 #include <iostream>
2 #include <vector>
3 using namespace std;
4
5 int main()
6 {
7     vector<int> nombrePremiers;
8     int indice = 2;
9     int x;
10
11     bool pasPremier;
12
13     while (indice <= 100)
14     {
15         x = 2;
16         pasPremier = false;
```

```

17
18     while (x < indice)
19     {
20         if (indice % x == 0)
21         {
22             pasPremier = true;
23             break;
24         }
25
26         x++;
27     }
28
29     if (!pasPremier)
30     {
31         nombrePremiers.push_back(indice);
32     }
33
34     indice++;
35 }
36
37 return 0;
38 }

```

Remarques :

- Le premier élément d'un vecteur (et d'un tableau) est à la position 0! Pas en position 1!
- Attention! `size_t` n'est pas un type classique! Il faut le convertir en `int` pour l'utiliser dans des opérations!

Définition : Un **tableau** est un type de donnée pouvant stocker un nombre prédéterminé de valeurs d'un type précisé. On peut voir les tableaux comme une extension des types classiques.

Fonction	Vecteur	Tableau
Ajouter un élément elem	vecteur.push_back(elem)	Impossible (taille fixe)
Accéder à l'élément i	vecteur.at(i)	tableau[i]
Obtenir la taille de la structure	vecteur.size()	sizeof(tableau) / sizeof(type)

Exemples : Les exemples suivants sont une adaptation des exemples de la page précédente mais en utilisant des tableaux :

1.

```

1  #include <iostream>
2  using namespace std;
3
4  int main()
5  {
6      float nombres[4] = {0.31, 8.12, -38.1, 642.02};
7
8      for (int i = 0; i < sizeof(nombres) / sizeof(float); i++)
9      {
10         nombres[i] = nombres[i] * nombres[i];
11     }
12
13     cout << (nombres[0] + nombres[int(sizeof(nombres) / sizeof(float)) - 1] ) / 2;
14

```

```

15     return 0;
16 }

```

2.

```

1  #include <iostream>
2  using namespace std;
3
4  int main()
5  {
6      // On ne peut pas prédire à l'avance le nombre de nombres premiers entre 2 et
7      // 100, mais on peut fixer la limite à 49 (les nombres pairs ne sont pas
8      // premiers).
9      int nombrePremiers[49];
10     int indice = 2;
11     int x;
12     int position = 0;
13
14     bool pasPremier;
15
16     while (indice <= 100)
17     {
18         x = 2;
19         pasPremier = false;
20
21         while (x < indice)
22         {
23             if (indice % x == 0)
24             {
25                 pasPremier = true;
26                 break;
27             }
28
29             x++;
30         }
31
32         if (!pasPremier)
33         {
34             nombrePremiers[position] = indice;
35             position++;
36         }
37
38         indice++;
39     }
40
41     return 0;
42 }

```

Remarques :

- Comme pour les vecteurs, il est possible de faire des tableaux de n'importe quel type, dont des tableaux d'énumérations, de vecteurs ou même de d'autres tableaux !
- `sizeof(...)` renvoie une taille en octet. C'est pour cela que l'on divise par `sizeof(type)`.

1.3.2 Pointeurs

Définition : Un **pointeur** est une variable ne stockant pas directement une valeur mais plutôt l'adresse mémoire où celle-ci est située. Un pointeur nécessite d'être **initialisé**, c'est-à-dire de lui allouer de la mémoire pour stocker quelque chose.

Proposition : Pour créer un pointeur, on doit écrire la ligne suivante :

```
1 type* pointeur = new type;
```

Le tableau suivant donne les opérations équivalentes entre variables classiques et pointeurs :

Action	Variable	Pointeur
Accéder à la valeur	variable	*pointeur
Accéder à l'adresse	&variable	pointeur

Exemples :

1. Le programme suivant affiche la valeur et l'adresse mémoire d'une variable classique puis d'un pointeur :

```
1 #include <iostream>
2 using namespace std;
3
4 int main()
5 {
6     int a = 2;
7     int* b = new int;
8
9     *b = 2 * a - 3;
10
11     cout << "Valeur de a : " << a << endl;
12     cout << "Adresse de a : " << &a << endl;
13     cout << "Valeur de b : " << *b << endl;
14     cout << "Adresse de b : " << b;
15
16     return 0;
17 }
```

2. Le programme suivant modifie la valeur d'une variable *nombre* sans y accéder directement :

```
1 #include <iostream>
2 using namespace std;
3
4 int main()
5 {
6     float nombre = 3.14;
7     float* ptrNombre = &nombre;
8     *ptrNombre = 3.1415;
9
10     cout << nombre;
11
12     return 0;
13 }
```

Remarque : Comme montré dans le deuxième exemple, il est possible d'initialiser un pointeur à partir d'une adresse déjà existante.

1.3.3 Pointeurs et tableaux

Avez-vous essayé d'afficher un tableau et pas un élément du tableau ? Cette sous-section va détailler ce qu'il se passe.

Proposition : Lorsqu'un tableau de taille n est créé, n emplacements mémoire sont alloués pour stocker les valeurs. Ces emplacements se trouvent les uns à la suite des autres.

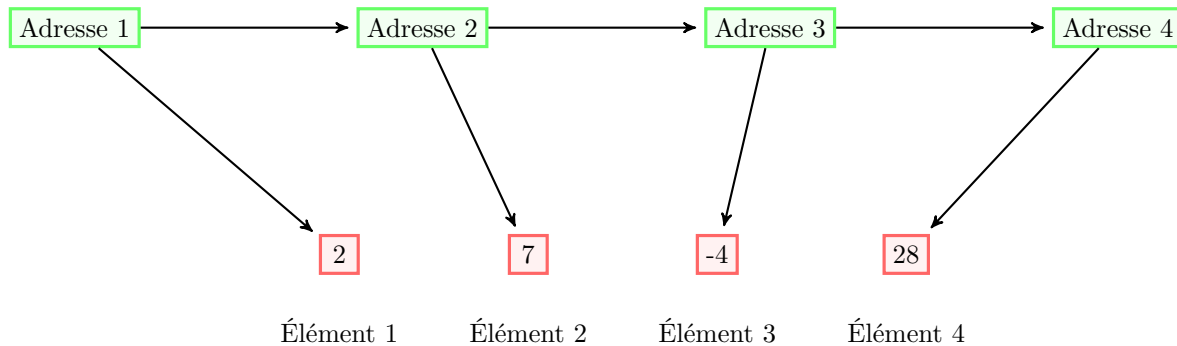


FIGURE 1.2 – Relations entre les éléments d'un tableau et des adresses

Remarque : L'adresse du premier élément d'un tableau est la valeur affichée lorsque l'on essaie d'afficher un tableau.

Exemple : Les programmes suivants affichent les éléments d'un tableau de deux manières différentes :

```

1 #include <iostream>
2 using namespace std;
3
4 int main()
5 {
6     int valeurs[4] = {-71, 42, -13, 56};
7
8     for (int i = 0; i < sizeof(valeurs) / sizeof(int); i++)
9     {
10         cout << valeurs[i] << endl;
11     }
12
13     return 0;
14 }
```

```

1 #include <iostream>
2 using namespace std;
3
4 int main()
5 {
```

```

6     int valeurs[4] = {-71, 42, -13, 56};
7     int* element = valeurs;
8
9     for (int i = 0; i < sizeof(valeurs) / sizeof(int); i++)
10    {
11        cout << *element << endl;
12        element++;
13    }
14
15    return 0;
16 }

```

Remarque : Il est possible d'incrémenter un pointeur. L'adresse stockée change pour la suivante.

1.3.4 Exemple récapitulatif de la section 1.3

Le programme suivant approxime la vitesse d'un objet à différents instants. Les valeurs sont stockées dans un vecteur. Par ailleurs, la première valeur du tableau est modifiée sans y accéder directement.

```

1  #include <iostream>
2  #include <vector>
3  using namespace std;
4
5  int main()
6  {
7      float positions[32] = {0.0, 0.36, 0.62, 0.98, 1.73, 3.20, 6.28, 12.03,
8                           17.02, 18.38, 19.3, 20.31, 21.38, 23.29, 25.29, 27.82,
9                           30.23, 36.92, 42.93, 49.29, 55.92, 62.38, 69.8, 75.42,
10                          85.18, 87.52, 89.62, 90.10, 90.15, 90.56, 90.57, 90.62 };
11
12      float* ptr = positions;
13
14      float deltaT = 0.1;
15      vector<float> vitesses;
16
17      for (int i = 1; i < 31; i++)
18      {
19          vitesses.push_back((positions[i + 1] - positions[i - 1]) / (2 * deltaT));
20          cout << i << " : " << vitesses.at(i - 1) << endl;
21      }
22
23      *ptr = -1000;
24
25      cout << positions[0];
26
27      return 0;
28 }

```

1.3.5 Conseils pour rédiger un programme

1. Utiliser des vecteurs lorsqu'il est difficile de prédire la quantité de données à stocker. Utiliser un tableau si cette quantité est connue.
2. Ne pas abuser des pointeurs. Les utiliser si cela est nécessaire.

3. Bien comprendre que si l'on veut traiter des données (calculer une moyenne...), une boucle *for* est fortement utile pour parcourir la structure.

Mots-clés de la section : *vector*, *new*, *sizeof*

1.4 Fonctions

1.4.1 Types de retour et paramètres

Définitions :

- Une **fonction** est un bloc composé d'une série d'instructions pouvant être exécutées à n'importe quel endroit. Elle peut renvoyer, ou pas, une valeur.
- Le **type de retour** d'une fonction est le type de valeur que peut renvoyer la fonction. Si la fonction ne renvoie rien, son type est le type **void**. La fonction arrête d'être exécutée (peu importe s'il reste des choses à faire) dès qu'une ligne avec le mot-clé **return** apparaît. C'est ce mot-clé qui sert à renvoyer une valeur. Une fonction **void** n'a pas besoin de mot-clé **return**.
- Un **paramètre** est une variable précisée dans l'en-tête de la fonction. Les paramètres servent à transmettre des valeurs à la fonction pour performer des actions spécifiques.
- Un **appel** de fonction est la ligne qui permet d'exécuter la fonction. On appelle une fonction en précisant son nom et en mettant des paramètres dans des parenthèses.

Proposition : Habituellement, la définition de la fonction se place avant la fonction *main*. Elle peut se placer après si l'on prend soin de déclarer la fonction dans le *main*.

Exemples :

1. Le programme suivant utilise des fonctions pour calculer les images d'une fonction² et pour calculer la moyenne d'un tableau sans renvoyer de valeur :

```

1  #include <iostream>
2  #include <cmath> // On utilise ce module pour la fonction sinus
3  using namespace std;
4
5  float f(float x)
6  {
7      if (x == 0)
8      {
9          return 1;
10     }
11
12     return sin(x) / x;
13 }
14
15 void moyenne(float* valeurs, float* resultat, int taille)
16 {
17     // Bonne pratique pour éviter d'ajouter des valeurs à une variable potentiellement
18     // pas initialisée.
19     *resultat = 0;
20
21     for (int i = 0; i < taille; i++)
22     {
23         *resultat += valeurs[i];
24     }
25
26     *resultat /= (float)taille;
27 }
28
29 int main()

```

2. Je parle ici de fonction mathématique


```

30 {
31     float resultat = 0, x = -1;
32     float valeurs[4];
33
34     for (int i = 0; i < sizeof(valeurs) / sizeof(float); i++)
35     {
36         valeurs[i] = f(x);
37         x += 0.5;
38     }
39
40     moyenne(valeurs, &resultat, 4);
41
42     cout << "La moyenne des valeurs est : " << resultat;
43
44     return 0;
45 }

```

2. Le programme suivant renvoie un tableau en ayant inversé l'ordre des éléments (le premier devient le dernier, le deuxième devient l'avant-dernier...) :

```

1  #include <iostream>
2  using namespace std;
3
4  int* inverser(int* tableau, int taille)
5  {
6      int valeurTemporaire;
7
8      for (int i = 0; i < taille / 2; i++)
9      {
10         valeurTemporaire = tableau[i];
11         tableau[i] = tableau[taille - i - 1];
12         tableau[taille - i - 1] = valeurTemporaire;
13     }
14
15     return tableau;
16 }
17
18 int main()
19 {
20     int tableau[4] = {6622, 3990, 296, 2626};
21     int* tableauInverse = inverser(tableau, (sizeof(tableau) / sizeof(int)));
22
23     for (int i = 0; i < sizeof(tableau) / sizeof(int); i++)
24     {
25         cout << tableauInverse[i] << endl;
26     }
27
28     return 0;
29 }

```

3. Le programme suivant génère tous les nombres premiers strictement compris entre deux bornes :

```

1  #include <iostream>
2  #include <vector>
3  using namespace std;
4

```

```

5  vector<int> nombresPremiers(int borneInferieure, int borneSuperieure)
6  {
7      int entierActuel = borneInferieure;
8      bool premier;
9      vector<int> listeNombres;
10
11     for (int j = borneInferieure + 1; j < borneSuperieure; j++)
12     {
13         premier = true;
14
15         for (int i = 2; i < entierActuel; i++)
16         {
17             if (entierActuel % i == 0)
18             {
19                 premier = false;
20             }
21         }
22
23         if (premier && entierActuel > borneInferieure)
24         {
25             listeNombres.push_back(entierActuel);
26         }
27
28         entierActuel++;
29     }
30
31     return listeNombres;
32 }
33
34 int main()
35 {
36     vector<int> liste = nombresPremiers(18, 104);
37
38     for (int i = 0; i < liste.size(); i++)
39     {
40         cout << liste.at(i) << endl;
41     }
42
43     return 0;
44 }

```

Remarques :

- Pour passer un tableau en paramètre, il faut passer un pointeur (donc l'adresse du premier élément). Il est aussi impossible de retourner directement un tableau. Il faut là aussi passer par un pointeur.
- Il ne faut pas utiliser l'opérateur (oui c'en est un!) *sizeof* pour obtenir la taille d'un tableau passé en paramètre. La taille d'un pointeur est en général le double de la taille du type, ce qui n'est pas forcément vrai pour un tableau ! C'est pour cela que l'on passe une taille en paramètre.
- Dans l'exemple 2, on aurait pu ne rien envoyer et seulement appeler la fonction. Les variables `tableau` et `tableauInverse` contiennent les mêmes valeurs.
- Deux fonctions ne peuvent pas avoir le même nom, sauf dans certains cas. Ces cas ne seront pas traités car il est généralement mauvais d'avoir deux fonctions avec le même nom.

1.4.2 Paramètres par défaut et passage de paramètres par référence

Définition : Un **paramètre par défaut** est un paramètre qui n'a pas besoin d'être précisé dans l'appel de la fonction. Une valeur par défaut lui est alors attribué. Un tel paramètre doit être placé en dernier.

Exemple : Le programme suivant calcule des approximations des images de la fonction $\log(1+x)$ en base e (attention, cela n'est valable que dans $] -1; 1[$). Par défaut, la fonction renvoie une approximation d'ordre 5, mais cet ordre peut être modifié au besoin.

```

1  #include <iostream>
2  using namespace std;
3
4  double logTaylor(double x, int iterations = 5)
5  {
6      if (x >= 1 || x <= -1)
7      {
8          return 0;
9      }
10
11     double resultat = 0;
12
13     double xi = x;
14     x *= x;
15     double x2 = x;
16
17     for (int i = 2; i <= iterations; i += 2)
18     {
19         resultat -= x / i;
20         x *= x2;
21     }
22
23     x = xi;
24
25     for (int i = 1; i <= iterations; i += 2)
26     {
27         resultat += x / i;
28         x *= x2;
29     }
30
31     return resultat;
32 }
33
34 int main()
35 {
36     cout << "Ordre 5 : " << logTaylor(0.1) << endl << "Ordre 10 : " << logTaylor(0.1, 10);
37
38     return 0;
39 }

```

Définition : On dit que l'on passe un paramètre par **référence** lorsque l'on adjoint (dans la déclaration et / ou définition de la fonction) juste après le type le caractère "&" (esperluette). On peut le voir comme une sorte de mélange entre le passage par valeur et le passage par pointeur : la fonction traite le paramètre comme une variable classique, mais les changements de celle-ci se propagent en dehors de la fonction.

Exemples :

1. Le programme suivant permet d'évaluer efficacement (en seulement 7 opérations) le polynôme $P(x) = 3x^6 - 8x^4 + x^2 + 1$. La valeur de x est modifiée.

```

1  #include <iostream>
2  using namespace std;
3
4  float polynome(float& x)
5  {
6      x *= x;
7
8      return 3 * x * x * x * (x - 8) + x + 1;
9  }
10
11 int main()
12 {
13     float x = 3.2;
14
15     cout << "x initial :" << x << endl;
16     cout << "P(x)    = " << polynome(x) << endl;
17     cout << "x final  :" << x;
18
19     return 0;
20 }
```

2. Ce programme se base sur celui de la page précédente. Il permet de calculer l'approximation de l'image de $\log(1 + x)$ sans rien retourner.

```

1  #include <iostream>
2  using namespace std;
3
4  void logTaylor(double& x, int iterations = 5)
5  {
6      if (x >= 1 || x <= -1)
7      {
8          x = 0;
9      }
10
11     else
12     {
13         double resultat = 0;
14
15         double xi = x;
16         x *= x;
17         double x2 = x;
18
19         for (int i = 2; i <= iterations; i += 2)
20         {
21             resultat -= x / i;
22             x *= x2;
23         }
24
25         x = xi;
26
27         for (int i = 1; i <= iterations; i += 2)
28         {
```

```

29         resultat += x / i;
30         x *= x2;
31     }
32
33     x = resultat;
34 }
35 }
36
37 int main()
38 {
39     double x = 0.1;
40
41     cout << "Ordre 5 :";
42     logTaylor(x);
43     cout << x;
44
45     return 0;
46 }

```

Remarque : Il est utile d'utiliser le passage par référence lorsque l'on souhaite la modification effectuée dans la fonction. Cela peut permettre de gagner du temps d'exécution.

1.4.3 Exemple récapitulatif de la section 1.4

Le programme qui suit génère 47 approximations (avec chaque fois un nombre de points différents) de l'aire en dessous de la fonction f (il s'agit d'un demi-cercle de rayon 1). La méthode utilisée est appelée *Simpson composée*. Notez bien qu'il est possible d'appeler une fonction dans une autre fonction. L'algorithme suivant vous permettra de reproduire la fonction pour une approximation à n points :

```

aire ← f(a) + f(b)
h ←  $\frac{b-a}{2n}$ 
Pour i allant de 1 à n - 1 :
    aire ← aire + f(a + 2hi)
Pour i allant de 0 à n - 1 :
    aire ← aire + f(a + h(2i + 1))
aire ←  $\frac{\text{aire}}{3}$ 

```

```

1  #include <iostream>
2  #include <vector>
3  #include <cmath>
4  using namespace std;
5
6
7  double f(double x)
8  {
9      return sqrt(1 - x * x);
10 }
11
12 vector<double> aire(double a, double b, int points = 10)
13 {
14     vector<double> aires;
15     double aire, h;
16
17     for (int j = 3; j <= points; j++)

```

```
18     {
19         aire = f(a) + f(b);
20         h = (b - a) / (j * 2);
21
22         for (int i = 0; i < j; i++)
23         {
24             aire += 4 * f(a + h * (2 * i + 1));
25         }
26
27         for (int i = 1; i < j; i++)
28         {
29             aire += 2 * f(a + h * 2 * i);
30         }
31
32         aire *= h / 3;
33
34         aires.push_back(aire);
35     }
36
37     return aires;
38 }
39
40 int main()
41 {
42     vector<double> aires = aire(-1, 1, 50);
43
44     for (int i = 0; i < aires.size(); i += 10)
45     {
46         cout << "Approximation avec " << i + 3 << " points : " << aires.at(i) << endl;
47     }
48
49     return 0;
50 }
```

1.4.4 Conseils pour rédiger un programme

1. Faites très attention au type de retour d'une fonction ! Assurez-vous de ne jamais initialiser une variable en appelant une fonction de type *void*.
2. Faites la liste de tous les paramètres dont vous avez besoin pour réaliser la fonction.
3. Assurez-vous de définir la fonction au bon endroit ! Évitez de le faire dans la fonction *main*.

Mots-clés de la section : *void*, *return*

1.5 Classes

1.5.1 Idée de base

L'idée derrière la notion de **classe** est de pouvoir rassembler plusieurs variables et fonctions (qui agissent sur ces variables) en une seule structure. De la sorte, on peut créer et manipuler beaucoup plus de variables à la fois, sans devoir les gérer une à une.

Un bon exemple serait de considérer un jeu vidéo. Supposons qu'un ennemi ait plusieurs dizaines de statistiques et informations (points de vie, résistances, armes, objets...) et que le joueur puisse se battre contre 30 ennemis en même temps. Dans ce cas, le programme devra gérer plusieurs centaines de variables uniquement pour cet événement spécifique ! Il serait fastidieux de considérer chaque variable indépendamment des autres et de s'assurer en même temps qu'elles soient reliées au bon ennemi. Par contre, en créant une structure de donnée spéciale qui regroupe toutes les variables nécessaires, on peut baisser la quantité de variables à traiter en même temps à "seulement" 30.

1.5.2 Attributs, constructeurs et destructeurs

Définitions :

- Une **classe** est une structure composée d'un bloc (qui doit se terminer par un point-virgule!) contenant des variables, nommées **attributs**, et des fonctions pouvant agir sur ces variables, nommées **méthodes**. Finalement, les attributs et méthodes sont appelés **composants** de la classe.
- Les composants d'une classe sont séparés en deux sections³ d'**accessibilité**. Les composants en **public** sont directement accessibles alors que les composants en **private** nécessitent une méthode en **public** pour y accéder. Pour spécifier l'accessibilité des composants, il faut indiquer au dessus de ceux-ci le mot-clé nécessaire, puis des deux points (:).
- On **instancie** un objet lorsque l'on crée une variable du type de la classe. Il faut procéder d'une manière similaire aux variables classiques : d'abord le nom de la classe, puis le nom de l'objet. La classe devient alors un type de variable à part entière.
- Le **constructeur** est la méthode qui est appelée immédiatement après la création de l'objet. Elle a le même nom que la classe. Le **destructeur** est la méthode qui est appelée immédiatement avant la destruction de l'objet (souvent à la fin du programme). Elle a le même nom que la classe mais elle est précédée d'un tilde (~). Ces méthodes doivent se placer en **public**.

Exemple : Le programme suivant contient une classe contenant deux nombres flottants et les méthodes de base :

```

1 #include <iostream>
2 using namespace std;
3
4 class Rectangle
5 {
6     float largeur;
7     float hauteur;
8
9 public:
10     Rectangle()
11     {
12         cout << "Objet créé !" << endl;
13     }
14

```

3. En réalité il y en a trois, mais seulement deux nous intéressent ici.

```
15     ~Rectangle()
16     {
17         cout << "Objet détruit !" << endl;
18     }
19 };
20
21 int main()
22 {
23     Rectangle rect;
24
25     return 0;
26 }
```

Remarques :

- Par défaut, les composants d'une classe sont en *private*. Il n'est donc pas nécessaire d'indiquer ce mot-clé à moins de placer les composants après ceux en *public*.
- Deux classes ne peuvent pas avoir le même nom.

Proposition : Il est possible d'accéder aux éléments en *public* d'un objet en plaçant un point entre le nom de l'objet et le composant auquel on souhaite accéder.

Exemple : Le programme suivant contient une classe contenant deux nombres flottants et les méthodes de base. Les variables ne sont pas dans la même section d'accessibilité :

```
1 #include <iostream>
2 using namespace std;
3
4 class Rectangle
5 {
6     float largeur;
7
8 public:
9     Rectangle()
10    {
11        cout << "Objet créé !" << endl;
12    }
13
14    ~Rectangle()
15    {
16        cout << "Objet détruit !" << endl;
17    }
18
19    float hauteur;
20 };
21
22 int main()
23 {
24     Rectangle rect;
25     rect.hauteur = 3.1;
26     cout << rect.hauteur << endl;
27
28     return 0;
29 }
```

On obtient la sortie suivante :

Objet créé!

3.1

Objet détruit!

Remarques :

- Essayer d'accéder à l'attribut *largeur* de la classe *Rectangle* n'est pas possible puisqu'il est en *public*!
- Si vous essayez d'accéder à une méthode en *public*, n'oubliez pas les parenthèses!

1.5.3 Encapsulation et opérateur de résolution de portée

Définitions :

- L'**encapsulation** est le procédé par lequel on met tous les attributs d'une classe en *private* et toutes les méthodes d'une classe en *public*.
- Pour manipuler les attributs d'une classe sur laquelle on pratique l'encapsulation, il est nécessaire d'utiliser des méthodes pour le faire. On appelle **getter** une méthode générique qui permet d'obtenir la valeur d'un attribut spécifique. On appelle **setter** une méthode générique qui permet de modifier la valeur d'un attribut spécifique.

Exemple : On va pratiquer l'encapsulation sur la classe *Rectangle* :

```
1  #include <iostream>
2  using namespace std;
3
4  class Rectangle
5  {
6      float largeur;
7      float hauteur;
8
9  public:
10     // Il n'est pas nécessaire de remplir le constructeur
11     // et le destructeur.
12     Rectangle() { }
13     ~Rectangle() { }
14
15     float getLargeur()
16     {
17         return largeur;
18     }
19
20     float getHauteur()
21     {
22         return hauteur;
23     }
24
25     void setLargeur(float Largeur)
26     {
27         largeur = Largeur;
28     }
29
30     void setHauteur(float Hauteur)
31     {
32         hauteur = Hauteur;
33     }
```

```

34 };
35
36 int main()
37 {
38     Rectangle rect;
39     rect.setLargeur(3.1);
40     rect.setHauteur(4.5);
41
42     cout << "L'aire du rectangle est : " << rect.getLargeur() * rect.getHauteur();
43
44     return 0;
45 }

```

Remarques :

- L'encapsulation est un procédé utilisé presque toujours en POO (Programmation Orientée Objet). Dans la suite de ce manuel, on l'utilisera tout le temps.
- En utilisant l'encapsulation, il n'est pas forcément nécessaire de remplir le constructeur et le destructeur. On verra plus tard qu'il est possible de passer des paramètres dans ces méthodes.

Définition : Il est souvent pratique de séparer la déclaration des méthodes de leur définition, et ce pour plus de clarté et pour augmenter la lisibilité du code. L'opérateur **résolution de portée** (::) permet d'accéder aux éléments d'une classe (et non pas d'un objet !) à l'extérieur de celle-ci.

Exemple : On va séparer la définition des méthodes de la classe *Rectangle* de leur déclaration :

```

1  class Rectangle
2  {
3      float largeur;
4      float hauteur;
5
6  public:
7      Rectangle();
8      ~Rectangle();
9
10     float getLargeur();
11     float getHauteur();
12
13     void setLargeur(float Largeur);
14     void setHauteur(float Hauteur);
15 };
16
17 Rectangle::Rectangle() { }
18
19 Rectangle::~~Rectangle() { }
20
21 float Rectangle::getLargeur()
22 {
23     return largeur;
24 }
25
26 float Rectangle::getHauteur()
27 {
28     return hauteur;
29 }
30

```

```

31 void Rectangle::setLargeur(float Largeur)
32 {
33     largeur = Largeur;
34 }
35
36 void Rectangle::setHauteur(float Hauteur)
37 {
38     hauteur = Hauteur;
39 }

```

Remarque : Il faut s'assurer de bien indiquer le bon nom de classe et le bon nom de méthode pour éviter les erreurs ! Le nom des paramètres doivent aussi être le même.

1.5.4 Fichier d'en-tête et architecture classique d'un programme en POO

Définitions :

- Un fichier d'**en-tête** est un fichier pour la déclaration de fonctions, de classes et éventuellement de variables constantes. Un tel fichier a comme extension *.h*.
- Un **module** est un fichier d'en-tête comportant lui-même plusieurs inclusions et permettant de relier plusieurs classes entre elles. Un module sert à permettre un groupe d'actions spécifiques (fonctions mathématiques, types de stockage supplémentaires, temps, gestion du son...).

Propositions :

- L'architecture classique d'un programme en C++ consiste à séparer les classes du fichier principal. Chaque classe est scindée en fichier *.h/.cpp* et l'on ajoute un fichier *main.cpp* contenant la fonction *main*.
- Pour inclure un fichier *.h* dans un fichier *.cpp*, on utilise la ligne :

```

1 #include "nom du fichier.h"

```

- Pour inclure dans le fichier *main.cpp* une classe déclarée dans un fichier d'en-tête, il faut inclure le fichier avec l'extension *.h* (et non pas le *.cpp*!).

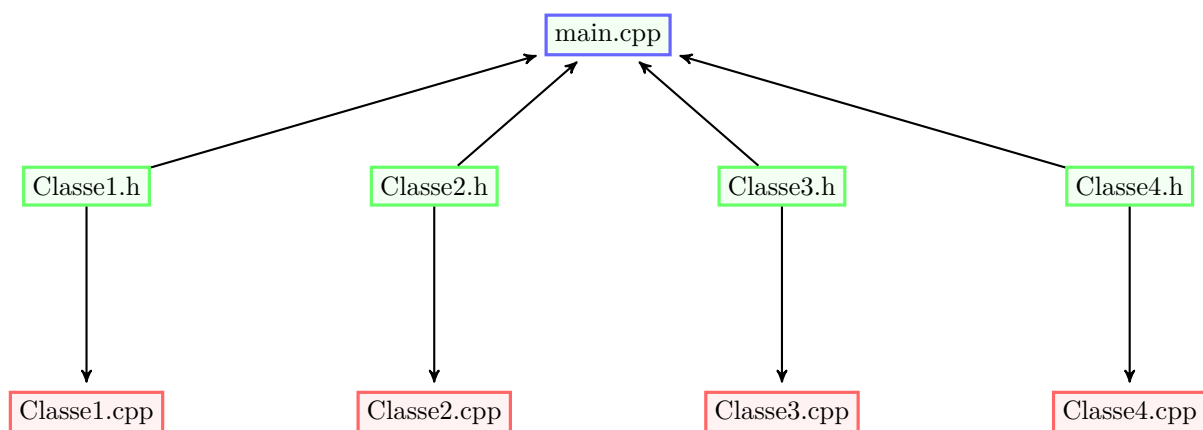


FIGURE 1.3 – Architecture classique en POO

Exemple : On va adopter l'architecture classique sur le programme de la classe *Rectangle* :

```
1 // rectangle.h
2
3 class Rectangle
4 {
5     float largeur;
6     float hauteur;
7
8 public:
9     Rectangle();
10    ~Rectangle();
11
12    float getLargeur();
13    float getHauteur();
14
15    void setLargeur(float Largeur);
16    void setHauteur(float Hauteur);
17 };
```

```
1 // rectangle.cpp
2
3 #include "rectangle.h"
4
5 Rectangle::Rectangle() { }
6
7 Rectangle::~~Rectangle() { }
8
9 float Rectangle::getLargeur()
10 {
11     return largeur;
12 }
13
14 float Rectangle::getHauteur()
15 {
16     return hauteur;
17 }
18
19 void Rectangle::setLargeur(float Largeur)
20 {
21     largeur = Largeur;
22 }
23
24 void Rectangle::setHauteur(float Hauteur)
25 {
26     hauteur = Hauteur;
27 }
```

```
1 // main.cpp
2
3 #include "rectangle.h"
4
5 #include <iostream>
6 using namespace std;
7
```

```

8  int main()
9  {
10     Rectangle rect;
11     rect.setLargeur(3.1);
12     rect.setHauteur(4.5);
13
14     cout << "L'aire du rectangle est : " << rect.getLargeur() * rect.getHauteur();
15
16     return 0;
17 }

```

Remarques :

1. Les lignes du type `#include <iostream>` représentent également des inclusions de fichiers d'en-tête, mais de la bibliothèque standard. Il faut utiliser les chevrons (`<>`) pour ce genre de fichier et les guillemets anglais pour les fichiers créés par vous même.
2. Si vous incluez un fichier d'en-tête qui lui-même en inclus d'autres, les inclusions seront automatiquement transmises dans le fichier de travail.
3. Il est primordial de n'avoir qu'une seule définition de la fonction *main* dans l'ensemble de votre projet. De multiples définitions pourraient créer des erreurs.
4. Ce genre d'architecture est omniprésente en programmation, y compris en utilisant l'API de *FIRST*. On utilisera cette architecture dans la suite manuel.

1.5.5 Pointeur sur un objet et attribut pointeur

Proposition : Il est possible de déclarer un pointeur sur un objet de manière similaire aux autres variables. L'accès aux composants de la classes ne se fait plus via un point mais via une flèche (`->`).

Exemple : On réutilise les fichiers *rectangle.h* et *rectangle.cpp* de l'exemple précédent. La variable *rect* devient un pointeur sur un objet de type *Rectangle* :

```

1  // main.cpp
2
3  #include "rectangle.h"
4
5  #include <iostream>
6  using namespace std;
7
8  int main()
9  {
10     Rectangle* rect = new Rectangle;
11     rect->setLargeur(3.1);
12     rect->setHauteur(4.5);
13
14     cout << "L'aire du rectangle est : " << rect->getLargeur() * rect->getHauteur();
15
16     return 0;
17 }

```

Proposition : Il est possible de faire en sorte que les attributs d'une classe soient des pointeurs. Pour cela, il faut procéder de manière habituelle, en s'assurant que les pointeurs soient bien initialisés avant d'être utilisés.

Exemple : On va modifier la classe *Rectangle* de la manière suivante :

```
1 // rectangle.h
2
3 class Rectangle
4 {
5     float* largeur;
6     float* hauteur;
7
8 public:
9     Rectangle();
10    ~Rectangle();
11
12    float getLargeur();
13    float getHauteur();
14
15    void setLargeur(float Largeur);
16    void setHauteur(float Hauteur);
17 };
```

```
1 // rectangle.cpp
2
3 #include "rectangle.h"
4
5 Rectangle::Rectangle()
6 {
7     largeur = new float;
8     hauteur = new float;
9 }
10
11 Rectangle::~Rectangle() { }
12
13 float Rectangle::getLargeur()
14 {
15     return *largeur;
16 }
17
18 float Rectangle::getHauteur()
19 {
20     return *hauteur;
21 }
22
23 void Rectangle::setLargeur(float Largeur)
24 {
25     *largeur = Largeur;
26 }
27
28 void Rectangle::setHauteur(float Hauteur)
29 {
30     *hauteur = Hauteur;
31 }
```

Ici, on initialise les attributs dans le constructeur. C'est généralement le meilleur endroit pour le faire.

1.5.6 Compléments sur le constructeur

Définition : Le constructeur par **défaut** est le constructeur rencontré jusqu'à présent. Il ne possède aucun paramètre.

Proposition : Il est possible d'ajouter des paramètres à un constructeur et d'appeler ce constructeur lors de la création d'un objet. Ce constructeur sera différent du constructeur par défaut.

Exemples :

1. Considérons la classe *Cercle* suivante. On va ajouter un constructeur supplémentaire pour initialiser directement l'attribut *rayon* :

```
1 // cercle.h
2
3 class Cercle
4 {
5     float rayon;
6
7 public:
8     Cercle();
9     Cercle(float Rayon);
10    ~Cercle()
11
12    float getRayon();
13
14    void setRayon(float Rayon);
15 };
```

```
1 // cercle.cpp
2
3 #include "cercle.h"
4
5 Cercle::Cercle() { }
6
7 Cercle::Cercle(float Rayon)
8 {
9     rayon = Rayon;
10 }
11
12 Cercle::~Cercle() { }
13
14 float Cercle::getRayon()
15 {
16     return rayon;
17 }
18
19 void Cercle::setRayon(float Rayon)
20 {
21     rayon = Rayon;
22 }
```

```
1 // main.cpp
2
3 #include "cercle.h"
```

```
4
5 int main()
6 {
7     Cercle cercle1;
8     cercle1.setRayon(2);
9
10    Cercle cercle2(2); // On évite d'appeler la méthode setRayon
11
12    return 0;
13 }
```

2. Revenons à la classe *Rectangle*. On va ajouter un constructeur à deux paramètres et mélanger les concepts :

```
1 // rectangle.h
2
3 class Rectangle
4 {
5     float* largeur;
6     float* hauteur;
7
8 public:
9     Rectangle();
10    Rectangle(float Largeur, float Hauteur);
11    ~Rectangle();
12
13    float getLargeur();
14    float getHauteur();
15
16    void setLargeur(float Largeur);
17    void setHauteur(float Hauteur);
18 };
```

```
1 // rectangle.cpp
2
3 #include "rectangle.h"
4
5 Rectangle::Rectangle()
6 {
7     largeur = new float;
8     hauteur = new float;
9 }
10
11 Rectangle::Rectangle(float Largeur, float Hauteur)
12 {
13     largeur = new float;
14     hauteur = new float;
15
16     *largeur = Largeur;
17     *hauteur = Hauteur;
18 }
19
20 Rectangle::~Rectangle() { }
21
22 float Rectangle::getLargeur()
```

```

23 {
24     return *largeur;
25 }
26
27 float Rectangle::getHauteur()
28 {
29     return *hauteur;
30 }
31
32 void Rectangle::setLargeur(float Largeur)
33 {
34     *largeur = Largeur;
35 }
36
37 void Rectangle::setHauteur(float Hauteur)
38 {
39     *hauteur = Hauteur;
40 }

```

```

1 // main.cpp
2
3 #include "rectangle.h"
4
5 #include <iostream>
6 using namespace std;
7
8 int main()
9 {
10     Rectangle* rect = new Rectangle(3, 4);
11     // Ici on appelle carrément le constructeur pour qu'il retourne
12     // un objet de type Rectangle avec les informations spécifiées
13
14     cout << "Aire du rectangle : " << rect->getLargeur() * rect->getHauteur();
15
16     return 0;
17 }

```

Remarque : Dans l'appel d'un constructeur avec paramètres, il est possible de remplacer les parenthèses par des accolades (`{}`). Les deux lignes suivantes sont donc équivalentes :

```

1 Rectangle rect(2, 3);
2 Rectangle rect{2, 3};

```

1.5.7 Héritage

Idée de base : L'idée derrière l'héritage s'explique plus facilement à l'aide d'un exemple. Considérons la situation suivante :

Une personne souhaite réaliser un *RPG* avec plusieurs sortes de créatures vivantes (humains, animaux, monstres...) qui ont plusieurs sortes de fonctions (protagoniste, *PNJ*, ennemi...). Une manière d'éviter de créer une classe distincte pour chaque créature avec une base de fonctionnalités communes (se déplacer, gérer les collisions...) est de créer une classe **mère** englobant tous les types de créatures : créer une classe pour les **entités**. Les classes **filles** (protagoniste, *PNJ*, monstres...) héritent de cette classe et de tous ses attributs et méthodes. On pourrait donc avoir quelque chose de la forme :

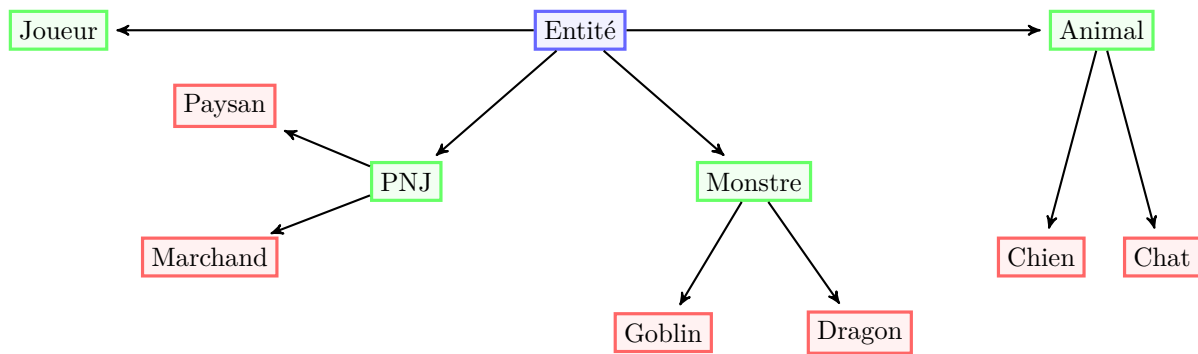


FIGURE 1.4 – Schéma de dépendance des classes

Proposition : Pour indiquer qu’une classe B hérite d’une classe A , il faut modifier la première ligne de la déclaration de B comme suit :

```

1 class B : public A
2 {
3     // Reste de la classe
4 };

```

Exemple : Construisons une classe *Joueur* qui encapsule un attribut *pointDeVie* et qui hérite d’une classe *Entite* (pour entité) qui encapsule les attributs x et y pour la position :

```

1 // entite.h
2
3 class Entite
4 {
5     float x;
6     float y;
7
8 public:
9     Entite();
10    ~Entite();
11
12    float getX();
13    float getY();
14
15    void setX(float X);
16    void setY(float Y);
17 };

```

```

1 // entite.cpp
2
3 #include "entite.h"
4
5 Entite::Entite() { }
6
7 Entite::~Entite() { }
8
9 float Entite::getX()

```

```
10 {
11     return x;
12 }
13
14 float Entite::getY()
15 {
16     return y;
17 }
18
19 void Entite::setX(float X)
20 {
21     x = X;
22 }
23
24 void Entite::setY(float Y)
25 {
26     y = Y;
27 }

```

```
1 // joueur.h
2
3 #include "entite.h"
4
5 class Joueur : public Entite
6 {
7     float pointDeVie;
8
9 public:
10     Joueur();
11     ~Joueur();
12
13     float getPointDeVie();
14
15     void setPointDeVie(float PointDeVie);
16 };

```

```
1 // joueur.cpp
2
3 Joueur::Joueur() { }
4
5 Joueur::~Joueur() { }
6
7 float Joueur::getPointDeVie()
8 {
9     return pointDeVie;
10 }
11
12 void Joueur::setPointDeVie(float PointDeVie)
13 {
14     pointDeVie = PointDeVie;
15 }

```

```
1 // main.cpp

```

```
2
3 #include "joueur.h"
4
5 int main()
6 {
7     Joueur joueur;
8
9     // Il est possible d'accéder aux méthodes de la classe Entite car
10    // Joueur hérite de Entite
11    joueur.setX(0);
12    joueur.setY(0);
13    joueur.setPointDeVie(30);
14
15    return 0;
16 }
```

Remarque : Une classe peut hériter de plusieurs classes en même temps. Pour ce faire, il suffit de rajouter une virgule et de procéder comme avant (*public C*).

1.5.8 Exemple récapitulatif de la section 1.5

Voici un programme qui crée un objet de type *Moteur*, type qui hérite de la classe *Composant*. Le programme fait fonctionner le moteur à pleine puissance tant que sa durée de vie ne dépasse pas 500 (valeur arbitraire). La durée de vie n'est pas modifiable via un *setter* puisque cela ne serait pas réaliste.

```
1 // composant.h
2
3 enum EtatComposant
4 {
5     Fonctionnel,
6     Endommage
7 };
8
9 class Composant
10 {
11     EtatComposant etat;
12     bool actif;
13     int dureeVie;
14
15     const int dureeVieMaxComposant = 500;
16
17 public:
18     Composant();
19     ~Composant();
20
21     EtatComposant getEtat();
22
23     bool getActif();
24
25     int getDureeVie();
26
27     void activer();
28     void desactiver();
29 };
```

```
1 // composant.cpp
2
3 #include "composant.h"
4
5 Composant::Composant()
6 {
7     etat = EtatComposant::Fonctionnel;
8     actif = false;
9     dureeVie = 0;
10 }
11
12 Composant::~Composant() { }
13
14 EtatComposant Composant::getEtat()
15 {
16     return etat;
17 }
18
19 bool Composant::getActif()
20 {
21     return actif;
22 }
23
24 int Composant::getDureeVie()
25 {
26     return dureeVie;
27 }
28
29 void Composant::activer()
30 {
31     if (etat == EtatComposant::Fonctionnel)
32     {
33         if (!actif)
34         {
35             actif = true;
36         }
37
38         dureeVie++;
39     }
40
41     if (dureeVie == dureeVieMaxComposant)
42     {
43         actif = false;
44         etat = EtatComposant::Endommage;
45     }
46 }
47
48 void Composant::desactiver()
49 {
50     actif = false;
51 }

```

```
1 // moteur.h
2
3 #include "composant.h"

```

```
4
5 class Moteur : public Composant
6 {
7     float voltage;
8     int port;
9
10 public:
11     Moteur();
12     Moteur(int Port);
13     ~Moteur();
14
15     float getVoltage();
16
17     void setVoltage(float Voltage);
18 };
```

```
1 // moteur.cpp
2
3 #include "moteur.h"
4
5 Moteur::Moteur()
6 {
7     voltage = 0;
8     port = 0;
9 }
10
11 Moteur::Moteur(int Port)
12 {
13     voltage = 0;
14     port = Port;
15 }
16
17 Moteur::~Moteur() { }
18
19 float Moteur::getVoltage()
20 {
21     return voltage;
22 }
23
24 void Moteur::setVoltage(float Voltage)
25 {
26     voltage = Voltage;
27
28     if (voltage == 0 || getEtat() == EtatComposant::Endommage)
29     {
30         desactiver();
31         voltage = 0;
32     }
33
34     else
35     {
36         activer();
37     }
38 }
```

```
1 // main.cpp
2
3 #include "moteur.h"
4
5 #include <iostream>
6 using namespace std;
7
8 int main()
9 {
10     Moteur* moteur = new Moteur(1);
11
12     moteur->setVoltage(1);
13
14     while(moteur->getActif())
15     {
16         moteur->setVoltage(1);
17         cout << moteur->getDureeVie() << endl;
18     }
19
20     return 0;
21 }
```

1.5.9 Conseils pour rédiger un programme

1. Faire la liste des attributs nécessaires à la classe. S'assurer, si nécessaire, que l'encapsulation est respectée. Également, déterminer les inclusions nécessaires.
2. S'assurer que les attributs pointeurs sont bien initialisés avant d'être utilisés (c'est une erreur fréquente!).
3. Faire un schéma de la classe. Pas besoin de faire très compliqué!
4. En cas de questionnements sur les méthodes, pointeurs, structures conditionnelles, etc..., revenir aux sections précédentes et relire les conseils.

Mots-clés de la section : *class*, *public*, *private*

1.6 Espaces de noms

1.6.1 Idée de base

Imaginons que vous souhaitiez utiliser en même temps deux modules qui comportent deux définitions de variables, fonctions ou classes avec le même nom. Inclure les deux modules en même temps risque de causer une erreur, car deux éléments distincts ont le même nom. Pour résoudre ce problème, on peut cloisonner chaque élément de chaque module dans une "boîte". Pour atteindre un élément, il suffit alors d'inclure le module et de préciser le nom de cette "boîte".

Définition : Un **espace de noms** est une manière de regrouper différents éléments (variables, fonctions et classes) autour d'une même "bannière". Deux variables, fonctions ou classes peuvent alors avoir le même nom si elles sont placées dans des espaces de noms différents.

Propositions :

- Pour atteindre un élément dans un espace de nom, il faut préciser le nom de cet espace, puis utiliser l'opérateur résolution de portée et enfin préciser le nom de l'élément.
- Il est possible d'utiliser normalement les éléments d'un espace de noms en utilisant la ligne suivante :

```
1 using namespace nomDeLespace;
```

Cette méthode est généralement moins bonne que la première puisqu'elle supprime l'utilité de l'espace de noms.

Exemple : Voici un programme qui utilise de multiples définitions la variable π :

```
1 // alice.h
2
3 namespace Alice
4 {
5     float pi = 3.14;
6 }
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100
101
102
103
104
105
106
107
108
109
110
111
112
113
114
115
116
117
118
119
120
121
122
123
124
125
126
127
128
129
130
131
132
133
134
135
136
137
138
139
140
141
142
143
144
145
146
147
148
149
150
151
152
153
154
155
156
157
158
159
160
161
162
163
164
165
166
167
168
169
170
171
172
173
174
175
176
177
178
179
180
181
182
183
184
185
186
187
188
189
190
191
192
193
194
195
196
197
198
199
200
201
202
203
204
205
206
207
208
209
210
211
212
213
214
215
216
217
218
219
220
221
222
223
224
225
226
227
228
229
230
231
232
233
234
235
236
237
238
239
240
241
242
243
244
245
246
247
248
249
250
251
252
253
254
255
256
257
258
259
260
261
262
263
264
265
266
267
268
269
270
271
272
273
274
275
276
277
278
279
280
281
282
283
284
285
286
287
288
289
290
291
292
293
294
295
296
297
298
299
300
301
302
303
304
305
306
307
308
309
310
311
312
313
314
315
316
317
318
319
320
321
322
323
324
325
326
327
328
329
330
331
332
333
334
335
336
337
338
339
340
341
342
343
344
345
346
347
348
349
350
351
352
353
354
355
356
357
358
359
360
361
362
363
364
365
366
367
368
369
370
371
372
373
374
375
376
377
378
379
380
381
382
383
384
385
386
387
388
389
390
391
392
393
394
395
396
397
398
399
400
401
402
403
404
405
406
407
408
409
410
411
412
413
414
415
416
417
418
419
420
421
422
423
424
425
426
427
428
429
430
431
432
433
434
435
436
437
438
439
440
441
442
443
444
445
446
447
448
449
450
451
452
453
454
455
456
457
458
459
460
461
462
463
464
465
466
467
468
469
470
471
472
473
474
475
476
477
478
479
480
481
482
483
484
485
486
487
488
489
490
491
492
493
494
495
496
497
498
499
500
501
502
503
504
505
506
507
508
509
510
511
512
513
514
515
516
517
518
519
520
521
522
523
524
525
526
527
528
529
530
531
532
533
534
535
536
537
538
539
540
541
542
543
544
545
546
547
548
549
550
551
552
553
554
555
556
557
558
559
560
561
562
563
564
565
566
567
568
569
570
571
572
573
574
575
576
577
578
579
580
581
582
583
584
585
586
587
588
589
590
591
592
593
594
595
596
597
598
599
600
601
602
603
604
605
606
607
608
609
610
611
612
613
614
615
616
617
618
619
620
621
622
623
624
625
626
627
628
629
630
631
632
633
634
635
636
637
638
639
640
641
642
643
644
645
646
647
648
649
650
651
652
653
654
655
656
657
658
659
660
661
662
663
664
665
666
667
668
669
670
671
672
673
674
675
676
677
678
679
680
681
682
683
684
685
686
687
688
689
690
691
692
693
694
695
696
697
698
699
700
701
702
703
704
705
706
707
708
709
710
711
712
713
714
715
716
717
718
719
720
721
722
723
724
725
726
727
728
729
730
731
732
733
734
735
736
737
738
739
740
741
742
743
744
745
746
747
748
749
750
751
752
753
754
755
756
757
758
759
760
761
762
763
764
765
766
767
768
769
770
771
772
773
774
775
776
777
778
779
780
781
782
783
784
785
786
787
788
789
790
791
792
793
794
795
796
797
798
799
800
801
802
803
804
805
806
807
808
809
810
811
812
813
814
815
816
817
818
819
820
821
822
823
824
825
826
827
828
829
830
831
832
833
834
835
836
837
838
839
840
841
842
843
844
845
846
847
848
849
850
851
852
853
854
855
856
857
858
859
860
861
862
863
864
865
866
867
868
869
870
871
872
873
874
875
876
877
878
879
880
881
882
883
884
885
886
887
888
889
890
891
892
893
894
895
896
897
898
899
900
901
902
903
904
905
906
907
908
909
910
911
912
913
914
915
916
917
918
919
920
921
922
923
924
925
926
927
928
929
930
931
932
933
934
935
936
937
938
939
940
941
942
943
944
945
946
947
948
949
950
951
952
953
954
955
956
957
958
959
960
961
962
963
964
965
966
967
968
969
970
971
972
973
974
975
976
977
978
979
980
981
982
983
984
985
986
987
988
989
990
991
992
993
994
995
996
997
998
999
1000
1001
1002
1003
1004
1005
1006
1007
1008
1009
1010
1011
1012
1013
1014
1015
1016
1017
1018
1019
1020
1021
1022
1023
1024
1025
1026
1027
1028
1029
1030
1031
1032
1033
1034
1035
1036
1037
1038
1039
1040
1041
1042
1043
1044
1045
1046
1047
1048
1049
1050
1051
1052
1053
1054
1055
1056
1057
1058
1059
1060
1061
1062
1063
1064
1065
1066
1067
1068
1069
1070
1071
1072
1073
1074
1075
1076
1077
1078
1079
1080
1081
1082
1083
1084
1085
1086
1087
1088
1089
1090
1091
1092
1093
1094
1095
1096
1097
1098
1099
1100
1101
1102
1103
1104
1105
1106
1107
1108
1109
1110
1111
1112
1113
1114
1115
1116
1117
1118
1119
1120
1121
1122
1123
1124
1125
1126
1127
1128
1129
1130
1131
1132
1133
1134
1135
1136
1137
1138
1139
1140
1141
1142
1143
1144
1145
1146
1147
1148
1149
1150
1151
1152
1153
1154
1155
1156
1157
1158
1159
1160
1161
1162
1163
1164
1165
1166
1167
1168
1169
1170
1171
1172
1173
1174
1175
1176
1177
1178
1179
1180
1181
1182
1183
1184
1185
1186
1187
1188
1189
1190
1191
1192
1193
1194
1195
1196
1197
1198
1199
1200
1201
1202
1203
1204
1205
1206
1207
1208
1209
1210
1211
1212
1213
1214
1215
1216
1217
1218
1219
1220
1221
1222
1223
1224
1225
1226
1227
1228
1229
1230
1231
1232
1233
1234
1235
1236
1237
1238
1239
1240
1241
1242
1243
1244
1245
1246
1247
1248
1249
1250
1251
1252
1253
1254
1255
1256
1257
1258
1259
1260
1261
1262
1263
1264
1265
1266
1267
1268
1269
1270
1271
1272
1273
1274
1275
1276
1277
1278
1279
1280
1281
1282
1283
1284
1285
1286
1287
1288
1289
1290
1291
1292
1293
1294
1295
1296
1297
1298
1299
1300
1301
1302
1303
1304
1305
1306
1307
1308
1309
1310
1311
1312
1313
1314
1315
1316
1317
1318
1319
1320
1321
1322
1323
1324
1325
1326
1327
1328
1329
1330
1331
1332
1333
1334
1335
1336
1337
1338
1339
1340
1341
1342
1343
1344
1345
1346
1347
1348
1349
1350
1351
1352
1353
1354
1355
1356
1357
1358
1359
1360
1361
1362
1363
1364
1365
1366
1367
1368
1369
1370
1371
1372
1373
1374
1375
1376
1377
1378
1379
1380
1381
1382
1383
1384
1385
1386
1387
1388
1389
1390
1391
1392
1393
1394
1395
1396
1397
1398
1399
1400
1401
1402
1403
1404
1405
1406
1407
1408
1409
1410
1411
1412
1413
1414
1415
1416
1417
1418
1419
1420
1421
1422
1423
1424
1425
1426
1427
1428
1429
1430
1431
1432
1433
1434
1435
1436
1437
1438
1439
1440
1441
1442
1443
1444
1445
1446
1447
1448
1449
1450
1451
1452
1453
1454
1455
1456
1457
1458
1459
1460
1461
1462
1463
1464
1465
1466
1467
1468
1469
1470
1471
1472
1473
1474
1475
1476
1477
1478
1479
1480
1481
1482
1483
1484
1485
1486
1487
1488
1489
1490
1491
1492
1493
1494
1495
1496
1497
1498
1499
1500
1501
1502
1503
1504
1505
1506
1507
1508
1509
1510
1511
1512
1513
1514
1515
1516
1517
1518
1519
1520
1521
1522
1523
1524
1525
1526
1527
1528
1529
1530
1531
1532
1533
1534
1535
1536
1537
1538
1539
1540
1541
1542
1543
1544
1545
1546
1547
1548
1549
1550
1551
1552
1553
1554
1555
1556
1557
1558
1559
1560
1561
1562
1563
1564
1565
1566
1567
1568
1569
1570
1571
1572
1573
1574
1575
1576
1577
1578
1579
1580
1581
1582
1583
1584
1585
1586
1587
1588
1589
1590
1591
1592
1593
1594
1595
1596
1597
1598
1599
1600
1601
1602
1603
1604
1605
1606
1607
1608
1609
1610
1611
1612
1613
1614
1615
1616
1617
1618
1619
1620
1621
1622
1623
1624
1625
1626
1627
1628
1629
1630
1631
1632
1633
1634
1635
1636
1637
1638
1639
1640
1641
1642
1643
1644
1645
1646
1647
1648
1649
1650
1651
1652
1653
1654
1655
1656
1657
1658
1659
1660
1661
1662
1663
1664
1665
1666
1667
1668
1669
1670
1671
1672
1673
1674
1675
1676
1677
1678
1679
1680
1681
1682
1683
1684
1685
1686
1687
1688
1689
1690
1691
1692
1693
1694
1695
1696
1697
1698
1699
1700
1701
1702
1703
1704
1705
1706
1707
1708
1709
1710
1711
1712
1713
1714
1715
1716
1717
1718
1719
1720
1721
1722
1723
1724
1725
1726
1727
1728
1729
1730
1731
1732
1733
1734
1735
1736
1737
1738
1739
1740
1741
1742
1743
1744
1745
1746
1747
1748
1749
1750
1751
1752
1753
1754
1755
1756
1757
1758
1759
1760
1761
1762
1763
1764
1765
1766
1767
1768
1769
1770
1771
1772
1773
1774
1775
1776
1777
1778
1779
1780
1781
1782
1783
1784
1785
1786
1787
1788
1789
1790
1791
1792
1793
1794
1795
1796
1797
1798
1799
1800
1801
1802
1803
1804
1805
1806
1807
1808
1809
1810
1811
1812
1813
1814
1815
1816
1817
1818
1819
1820
1821
1822
1823
1824
1825
1826
1827
1828
1829
1830
1831
1832
1833
1834
1835
1836
1837
1838
1839
1840
1841
1842
1843
1844
1845
1846
1847
1848
1849
1850
1851
1852
1853
1854
1855
1856
1857
1858
1859
1860
1861
1862
1863
1864
1865
1866
1867
1868
1869
1870
1871
1872
1873
1874
1875
1876
1877
1878
1879
1880
1881
1882
1883
1884
1885
1886
1887
1888
1889
1890
1891
1892
1893
1894
1895
1896
1897
1898
1899
1900
1901
1902
1903
1904
1905
1906
1907
1908
1909
1910
1911
1912
1913
1914
1915
1916
1917
1918
1919
1920
1921
1922
1923
1924
1925
1926
1927
1928
1929
1930
1931
1932
1933
1934
1935
1936
1937
1938
1939
1940
1941
1942
1943
1944
1945
1946
1947
1948
1949
1950
1951
1952
1953
1954
1955
1956
1957
1958
1959
1960
1961
1962
1963
1964
1965
1966
1967
1968
1969
1970
1971
1972
1973
1974
1975
1976
1977
1978
1979
1980
1981
1982
1983
1984
1985
1986
1987
1988
1989
1990
1991
1992
1993
1994
1995
1996
1997
1998
1999
2000
2001
2002
2003
2004
2005
2006
2007
2008
2009
2010
2011
2012
2013
2014
2015
2016
2017
2018
2019
2020
2021
2022
2023
2024
2025
2026
2027
2028
2029
2030
2031
2032
2033
2034
2035
2036
2037
2038
2039
2040
2041
2042
2043
2044
2045
2046
2047
2048
2049
2050
2051
2052
2053
2054
2055
2056
2057
2058
2059
2060
2061
2062
2063
2064
2065
2066
2067
2068
2069
2070
2071
2072
2073
2074
2075
2076
2077
2078
2079
2080
2081
2082
2083
2084
2085
2086
2087
2088
2089
2090
2091
2092
2093
2094
2095
2096
2097
2098
2099
2100
2101
2102
2103
2104
2105
2106
2107
2108
2109
2110
2111
2112
2113
2114
2115
2116
2117
2118
2119
2120
2121
2122
2123
2124
2125
2126
2127
2128
2129
2130
2131
2132
2133
2134
2135
2136
2137
2138
2139
2140
2141
2142
2143
2144
2145
2146
2147
2148
2149
2150
2151
2152
2153
2154
2155
2156
2157
2158
2159
2160
2161
2162
2163
2164
2165
2166
2167
2168
2169
2170
2171
2172
2173
2174
2175
2176
2177
2178
2179
2180
2181
2182
2183
2184
2185
2186
2187
2188
2189
2190
2191
2192
2193
2194
2195
2196
2197
2198
2199
2200
2201
2202
2203
2204
2205
2206
2207
2208
2209
2210
2211
2212
2213
2214
2215
2216
2217
2218
2219
2220
2221
2222
2223
2224
2225
2226
2227
2228
2229
2230
2231
2232
2233
2234
2235
2236
2237
2238
2239
2240
2241
2242
2243
2244
2245
2246
2247
2248
2249
2250
2251
2252
2253
2254
2255
2256
2257
2258
2259
2260
2261
2262
2263
2264
2265
2266
2267
2268
2269
2270
2271
2272
2273
2274
2275
2276
2277
2278
2279
2280
2281
2282
2283
2284
2285
2286
2287
2288
2289
2290
2291
2292
2293
2294
2295
2296
2297
2298
2299
2300
2301
2302
2303
2304
2305
2306
2307
2308
2309
2310
2311
2312
2313
2314
2315
2316
2317
2318
2319
2320
2321
2322
2323
2324
2325
2326
2327
2328
2329
2330
2331
2332
2333
2334
2335
2336
2337
2338
2339
2340
2341
2342
2343
2344
2345
2346
2347
2348
2349
2350
2351
2352
2353
2354
2355
2356
2357
2358
2359
2360
2361
2362
2363
2364
2365
2366
2367
2368
2369
2370
2371
2372
2373
2374
2375
2376
2377
2378
2379
2380
2381
2382
2383
2384
2385
2386
2387
2388
2389
2390
2391
2392
2393
2394
2395
2396
2397
2398
2399
2400
2401
2402
2403
2404
2405
2406
2407
2408
2409
2410
2411
2412
2413
2414
2415
2416
2417
2418
2419
2420
2421
2422
2423
2424
2425
2426
2427
2428
2429
2430
2431
2432
2433
2434
2435
2436
2437
2438
2439
2440
2441
2442
2443
2444
2445
2446
2447
2448
2449
2450
2451
2452
2453
2454
2455
2456
2457
2458
2459
2460
2461
2462
2463
2464
2465
2466
2467
2468
2469
2470
2471
2472
2473
2474
2475
2476
2477
2478
2479
2480
2481
2482
2483
2484
2485
2486
2487
2488
2489
2490
2491
2492
2493
2494
2495
2496
2497
2498
2499
2500
2501
2502
2503
2504
2505
2506
2507
2508
2509
2510
2511
2512
2513
2514
2515
2516
2517
2518
2519
2520
2521
2522
2523
2524
2525
2526
2527
2528
2529
2530
2531
2532
2533
2534
2535
2536
2537
2
```



```
13     std::cout << pi << std::endl;
14     std::cout << Alice::pi << std::endl;
15     std::cout << Bob::pi << std::endl;
16
17     return 0;
18 }
```

Chapitre 2

Utiliser Git et Github

2.1 Git

2.1.1 Idée de base

Avez-vous déjà essayé de faire un projet C++ en groupe ? Si oui, vous avez probablement remarqué (ou pas, tant mieux pour vous !) que mettre en commun le code peut difficilement se faire. S'envoyer des programmes n'est pas vraiment la meilleure des manières de procéder et retrouver les anciennes versions n'est pas forcément chose aisée. C'est là qu'entre en jeu Git : un formidable outil de versionnement de code.

Imaginons la situation suivante :

Alice, Bob et Céline veulent travailler sur un projet en même temps. Ils se sont répartis le travail au préalable. Chacun travaille sur sa partie, sa **branche**, effectue ses propres modifications et tests, puis met en commun son travail sur une branche principale.

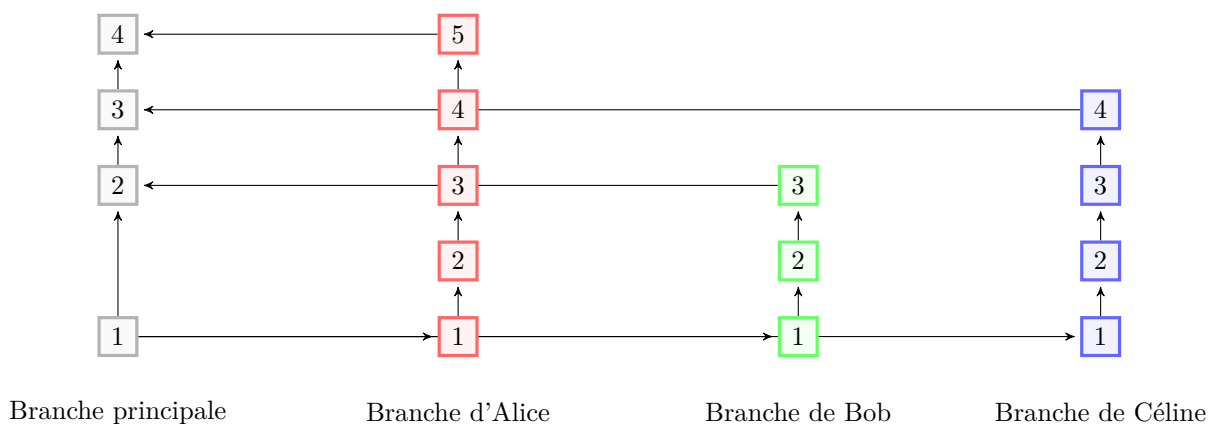


FIGURE 2.1 – Schéma de branches

À chaque modification conséquente, chaque membre crée un **commit**, un **paquet** qui est mis sur la branche de travail. De cette manière, le travail de chacun est mis à jour et versionné (on a accès à toutes les versions du code).

2.1.2 Définitions de base

- L'**espace de travail** constitue l'ensemble des fichiers de votre projet sur votre ordinateur. Il dépend de la branche et de la version.
- L'**index** contient l'ensemble des fichiers ayant été ajoutés pour mettre à jour le dépôt local.
- Le **dépôt local** constitue une version locale de votre dépôt en ligne, contenant toutes les branches et version du code. Il est possible de mettre à jour plusieurs fichiers avec un *commit* sans avoir accès à Internet, puis de téléverser les modifications en ligne avec un *push* dès le retour de la connexion.
- Le **dépôt en ligne** constitue une version en ligne (souvent disponible au public) du dépôt.

2.1.3 Commandes de base

- **git clone** : Clone le dépôt en ligne sur un ordinateur.
- **git status** : Avoir des informations sur la branche actuelle, savoir si des fichiers ne sont pas versionnés / traqués.

- ***git add "nomDuFichier.extension"*** : Ajoute le fichier *nomDuFichier.extension* (l'extension est importante!) à l'index.
- ***git add *.extension*** : Ajoute tous les fichiers présents dans le répertoire avec comme extension *extension* à l'index.
- ***git add **** : Ajoute tous les fichiers présents dans le répertoire à l'index.
- ***git commit -m "Message"*** : Met à jour le dépôt local à partir de l'index.
- ***git rm "nomDuFichier"*** : Supprime le fichier *nomDuFichier* de l'index.
- ***git push*** : Met à jour le dépôt en ligne à partir du dépôt local.
- ***git fetch*** : Met à jour le dépôt local à partir du dépôt en ligne.
- ***git pull*** : Met à jour l'espace de travail à partir du dépôt en ligne.
- ***git checkout "nomDeLaBranche"*** : Bascule sur la branche *nomDeLaBranche* et met à jour l'espace de travail.

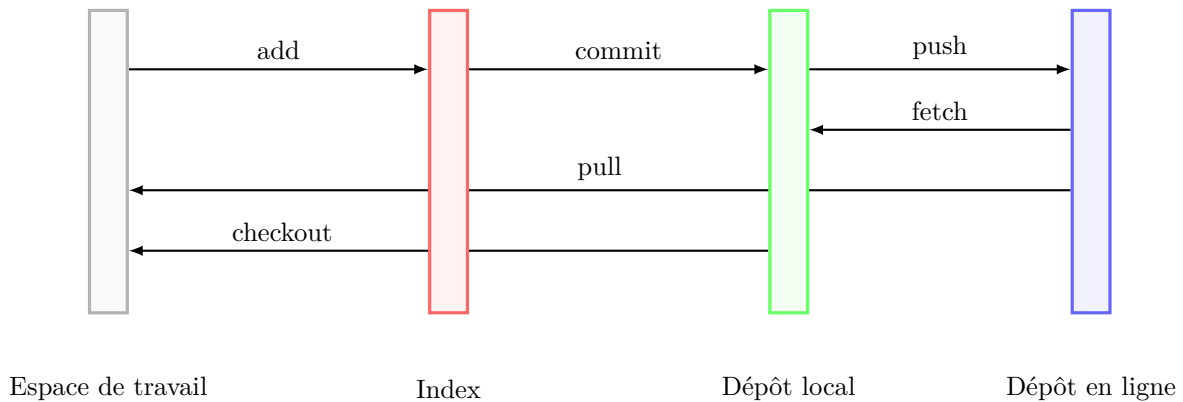


FIGURE 2.2 – Résumé des commandes Git de base

2.2 Github

2.2.1 Présentation

Github est un site qui permet d'héberger des dépôts *Git* en ligne. Pour profiter au maximum de ses fonctionnalités, il est recommandé de se créer un compte *Github*. L'image suivante permet de suivre les fonctionnalités principales de la page d'un dépôt :

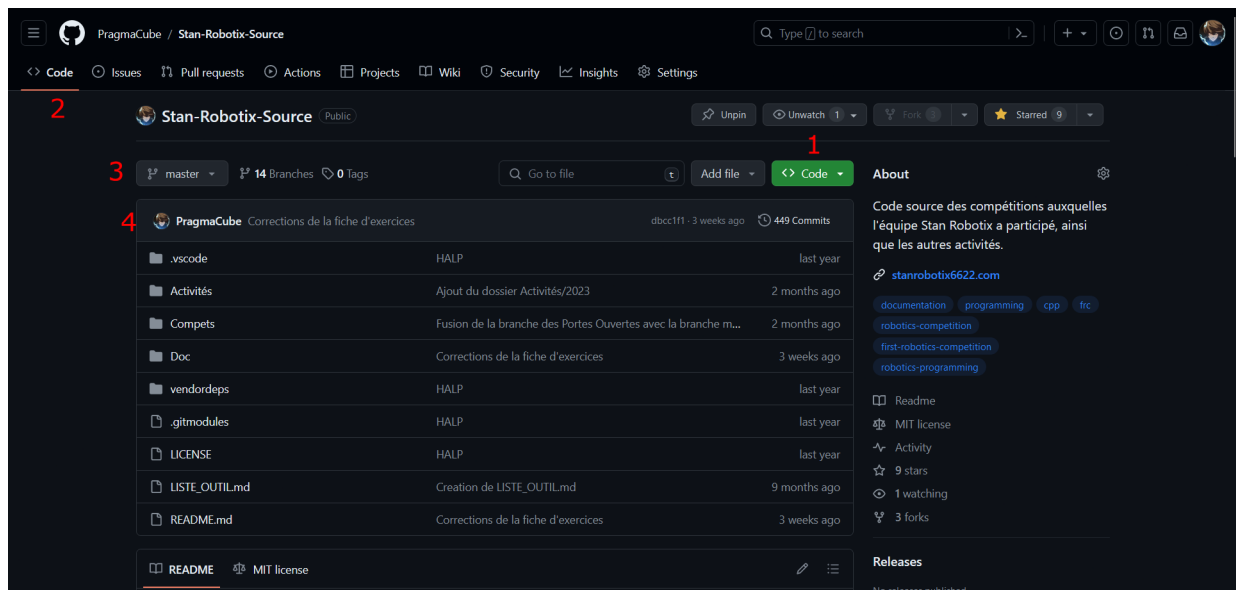


FIGURE 2.3 – Interface d'un dépôt sur Github

1. Menu *Code* : menu permettant d'accéder aux différentes méthodes pour cloner le dépôt.
2. Barre d'onglets : l'onglet *Code* permet d'afficher les différents répertoires et fichiers du dépôt. L'onglet *Issues* permet de créer et suivre les différents problèmes repérés dans le code pour effectuer des modifications. L'onglet *Pull requests* permet de créer et suivre les demandes de fusion de branches. L'utilité des autres onglets ne sera pas traitée dans ce manuel.
3. Menu des branches : permet de basculer entre les différentes branches et ainsi de mettre à jour les répertoires et fichiers affichés.
4. Dernier *commit* : permet d'accéder au dernier *commit* ayant eu lieu sur la branche.

Il est recommandé au lecteur d'explorer par lui-même les autres fonctionnalités de Github.

Remarque : Un dépôt sur Github fonctionne indépendamment du dépôt local. Il est parfaitement possible de modifier du code et de préparer des *commits* sans pour autant avoir accès à une connexion Internet. Il faut par contre s'assurer d'exécuter la commande *push* le plus rapidement possible pour éviter de potentielles désynchronisations.

Chapitre 3

Le Command-Based Programming et WPILIB

Remarque : Il est recommandé d'avoir la documentation de WPILIB à portée de main avant de commencer à programmer quoi que ce soit.

3.1 Architecture

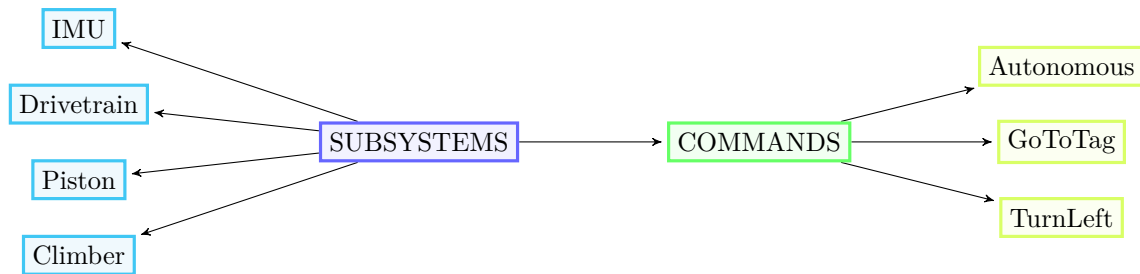


FIGURE 3.1 – Schéma du *Command-Based Programming*

L'idée du **Command-Based Programming** est de regrouper chaque mécanisme d'un robot (base pilotable, piston, capteur...) dans une classe générique permettant de modifier son état. Chaque classe de ce type est appelée un **Subsystem**. Les *Subsystems* sont ensuite utilisés dans des classes appelées **Commands** qui servent à mettre ensemble un certain nombre d'actions devant être exécutées dans un ordre précis (avancer pendant deux secondes, tourner à droite, activer le piston...). Les *Commands* sont appelées lorsqu'un bouton du *joystick* est pressé.

Il est important de noter que le code est contenu dans une boucle plus globale qui exécute chaque itération toutes les 20 ms. Si vous souhaitez placer une boucle de votre propre chef, faites très attention ! Il est également important de s'assurer que chaque *Subsystem* ne soit présent en un seul exemplaire pour ne pas avoir de conflits ou de problèmes de synchronisation. L'utilisation de pointeurs est une solution.

3.2 Subsystems

Un *Subsystem* est une classe qui encapsule un ou plusieurs objet(s) permettant de contrôler un ou plusieurs composant(s) électrique(s) ou mécanique(s) sur le robot. On souhaite que cette classe soit aussi simple que possible, mais qu'elle dispose de toutes les méthodes nécessaires au bon contrôle du composant.

Exemples :

1. On souhaite créer un *Subsystem* contrôlant les 4 moteurs de la base pilotable d'un robot. On souhaite avoir les spécificités suivantes :
 - Contrôler les moteurs avec des contrôleur *VictorSPX*. La classe *VictorSPX* n'étant pas présente dans la bibliothèque de base, il faut l'ajouter. Consulter l'onglet *Third Party Libraries* sur la documentation au besoin.
 - Utiliser des roues *Mecanum*.
 - Utiliser le mode *Field-Oriented* (le robot avance dans la direction du *joystick* peut importe la direction vers laquelle pointe le robot)

```

1 // Constants.h
2
3 // Copyright (c) FIRST and other WPILib contributors.
4 // Open Source Software; you can modify and/or share it under the terms of
5 // the WPILib BSD license file in the root directory of this project.
6
  
```



```

7  #pragma once
8
9  namespace DriveTrainConstants
10 {
11     constexpr int kMotorL1Id = 2;
12     constexpr int kMotorL2Id = 4;
13     constexpr int kMotorR1Id = 3;
14     constexpr int kMotorR2Id = 1;
15 }

```

```

1  // SubDriveTrain.h
2
3  // Copyright (c) FIRST and other WPILib contributors.
4  // Open Source Software; you can modify and/or share it under the terms of
5  // the WPILib BSD license file in the root directory of this project.
6
7  #pragma once
8
9  #include <frc2/command/SubsystemBase.h>
10 #include <frc/drive/MecanumDrive.h>
11 #include <ctre/phenix/motorcontrol/can/WPI_VictorSPX.h>
12 #include <ctre/phenix/motorcontrol/can/WPI_TalonSRX.h>
13
14 #include "Constants.h"
15
16 class SubDriveTrain : public frc2::SubsystemBase
17 {
18 public:
19     SubDriveTrain();
20
21     void Periodic() override;
22
23     void mecanumDrive(float iX, float iY, float iZ, frc::Rotation2d iRotation2d);
24
25 private:
26     ctre::phenix::motorcontrol::can::WPI_VictorSPX* mMotorL1{};
27     ctre::phenix::motorcontrol::can::WPI_VictorSPX* mMotorL2{};
28     ctre::phenix::motorcontrol::can::WPI_VictorSPX* mMotorR1{};
29     ctre::phenix::motorcontrol::can::WPI_VictorSPX* mMotorR2{};
30
31     frc::MecanumDrive* mDrive;
32 };

```

```

1  // SubDriveTrain.cpp
2
3  // Copyright (c) FIRST and other WPILib contributors.
4  // Open Source Software; you can modify and/or share it under the terms of
5  // the WPILib BSD license file in the root directory of this project.
6
7  #include "subsystems/SubDriveTrain.h"
8
9  SubDriveTrain::SubDriveTrain()
10 {
11     mMotorL1 = new

```

```

    ctre::phoenix::motorcontrol::can::WPI_VictorSPX{DriveTrainConstants::kMotorL1Id};
12  mMotorL2 = new
    ctre::phoenix::motorcontrol::can::WPI_TalonSRX{DriveTrainConstants::kMotorL2Id};
13  mMotorR1 = new
    ctre::phoenix::motorcontrol::can::WPI_VictorSPX{DriveTrainConstants::kMotorR1Id};
14  mMotorR2 = new
    ctre::phoenix::motorcontrol::can::WPI_VictorSPX{DriveTrainConstants::kMotorR2Id};

15
16  mDrive = new frc::MecanumDrive{*mMotorL1, *mMotorL2, *mMotorR1, *mMotorR2};
17
18  mMotorL1->SetInverted(true);
19  mMotorL2->SetInverted(true);
20 }
21
22 // This method will be called once per scheduler run
23 void SubDriveTrain::Periodic() {}
24
25 void SubDriveTrain::mecanumDrive(const float iX, const float iY, const float iZ, const
    frc::Rotation2d iRotation2d)
26 {
27     mDrive->DriveCartesian(iY, -iX, -iZ, -iRotation2d);
28 }

```

2. On souhaite créer un *Subsystem* récupérant les données d'un *IMU* (centrale inertielle). On souhaite avoir les spécificités suivantes :

- Utiliser l'*IMU Pigeon2*. La classe *Pigeon2* n'est pas non plus incluse dans la bibliothèque de base.
- Obtenir un angle de rotation (*Rotation2D*) dans un plan parallèle au sol
- Obtenir la norme (valeur) du vecteur accélération

```

1  // SubIMU.h
2
3  // Copyright (c) FIRST and other WPILib contributors.
4  // Open Source Software; you can modify and/or share it under the terms of
5  // the WPILib BSD license file in the root directory of this project.
6
7  #pragma once
8
9  #include <frc2/command/SubsystemBase.h>
10
11 #include <ctre/Phoenix.h>
12 #include <ctre/phoenix6/Pigeon2.hpp>
13 #include <units/acceleration.h>
14
15 class SubIMU : public frc2::SubsystemBase
16 {
17 public:
18     SubIMU();
19
20     void Periodic() override;
21
22     units::standard_gravity_t getAccelX();
23     units::standard_gravity_t getAccelY();
24     units::standard_gravity_t getAccel();
25 }

```

```

26     frc::Rotation2d getRotation2d();
27
28     void ResetAngle();
29
30 private:
31     ctre::phoenix6::hardware::Pigeon2* mIMU{0};
32 };

```

```

1  // SubIMU.cpp
2
3  // Copyright (c) FIRST and other WPILib contributors.
4  // Open Source Software; you can modify and/or share it under the terms of
5  // the WPILib BSD license file in the root directory of this project.
6
7  #include "subsystems/SubIMU.h"
8  #include <cmath>
9
10 SubIMU::SubIMU()
11 {
12     mIMU = new ctre::phoenix6::hardware::Pigeon2{0};
13 }
14
15 // This method will be called once per scheduler run
16 void SubIMU::Periodic() {}
17
18 void SubIMU::ResetAngle()
19 {
20     mIMU->Reset();
21 }
22
23 units::standard_gravity_t SubIMU::getAccelX()
24 {
25     return mIMU->GetAccelerationX().GetValue();
26 }
27
28 units::standard_gravity_t SubIMU::getAccelY()
29 {
30     return mIMU->GetAccelerationY().GetValue();
31 }
32
33 units::standard_gravity_t SubIMU::getAccel()
34 {
35     return units::standard_gravity_t(std::sqrt(GetAccelerationX().GetValueAsDouble() *
36         GetAccelerationX().GetValueAsDouble() + GetAccelerationY().GetValueAsDouble() *
37         GetAccelerationY().GetValueAsDouble()));
38 }
39
40 frc::Rotation2d SubIMU::getRotation2d()
41 {
42     return mIMU->GetRotation2d();
43 }

```

3.3 Digression sur les PID

3.3.1 Idée de base

Comment atteindre une valeur cible à partir d'une mesure le plus efficacement possible ?

Cette question soulève un enjeu important : celui de l'optimisation du déroulement de certaines actions.

Imaginons la situation suivante :

On souhaite qu'un moteur actionnant fasse pivoter une barre métallique à un endroit précis, et ce le plus rapidement et correctement possible. On pourrait se dire naïvement qu'une simple correction à l'aide d'une structure conditionnelle et une boucle peut faire l'affaire, peut comment contrôler efficacement le dépassement, les oscillations, etc... ? En utilisant un **PID** !

3.3.2 Présentation détaillée

Un *PID* (pour **Proportionnel**, **Intégral** et **Dérivé**) est un régulateur permettant de donner une réponse adéquate et optimale face à une **erreur** (différence entre la **mesure** et la **cible**). Il faut fournir une mesure et une manière de modifier l'état du système (généralement un moteur). Trois coefficients entrent en jeu pour ajuster le *PID* :

- **Coefficient proportionnel** : Sert à moduler la réponse proportionnellement à l'erreur. Diminue la durée d'action mais augmente le dépassement.
- **Coefficient intégral** : Sert à moduler la réponse face à l'accumulation des erreurs au cours du temps. Diminue la durée d'action mais augmente le dépassement.
- **Coefficient dérivé** : Sert à moduler les oscillations autour de la cible. Diminue le dépassement.

Le réglage d'un *PID* doit faire partie intégrante des tests pour éviter des comportements imprévus. Plusieurs méthodes pour le réglage d'un *PID* existe, notamment celle de Ziegler–Nichols qui est plutôt simple à mettre en œuvre. Le réglage empirique (en faisant immédiatement les tests pour mesurer l'intérêt du changement) peut aussi être utilisé.

3.4 Commands

3.4.1 Idée de base

Une *Command* est une classe encapsulant un ou plusieurs *Subsystem(s)* pour performer un certain nombre d'actions sur ceux-ci. Le but est de créer une commande faisant automatiquement des actions récurrentes lors d'un match pour alléger la charge du pilote.

3.4.2 Présentation des méthodes de la classe *Command*

- ***Initialize()*** : méthode appelée au début de la commande. Il est conseillé de l'utiliser pour initialiser des valeurs.
- ***Execute()*** : méthode appelée à chaque itération dans la boucle (toutes les 20 ms). C'est la méthode qui doit contenir le coeur de la commande (par exemple actionner un moteur).
- ***IsFinished()*** : méthode appelée immédiatement après la méthode ***Execute()*** qui doit contenir la ou les condition(s) d'arrêt de la commande. Elle doit renvoyer *true* lorsque la commande doit s'arrêter.

- ***End(bool interrupted)*** : méthode appelée à la fin de l'exécution de la commande. Si la commande ne se termine pas correctement, le paramètre *interrupted* est mis à *true* et certaines actions peuvent être engagées dans ce cas pour sécuriser un système (arrêter un compresseur d'air, immobiliser un moteur...).
- ***AddRequirementsSubsystem* subsystem*** : spécifie un *Subsystem* utilisé dans une commande. Si plusieurs *Subsystem* sont nécessaires, il faut appeler plusieurs fois cette méthode. L'appel se fait généralement dans le constructeur de la classe.

Exemples :

1. On souhaite créer une *Command* permettant faire tourner le robot de 90 degrés dans le sens horaire (tourner à droite). Le robot possède les spécificités suivantes :
 - Base pilotable avec roues *Mecanum* avec un *Subsystem* comme dans l'exemple de la section 3.2.
 - *IMU Pigeon2* avec un *Subsystem* comme dans l'exemple de la section 3.2 mais avec une méthode supplémentaire, *GetAngleYaw()* qui permet de déterminer l'angle de rotation du robot selon un plan horizontal dans le sens horaire.

```

1 // TurnRight.h
2
3 #pragma once
4
5 #include <frc2/command/Command.h>
6 #include <frc2/command/CommandHelper.h>
7
8 #include "subsystems/SubDriveTrain.h"
9 #include "subsystems/SubIMU.h"
10
11 #include "Constants.h"
12
13 class TurnRight
14 : public frc2::CommandHelper<frc2::Command, TurnRight> {
15 public:
16     TurnRight(SubDriveTrain* iDriveTrain, SubIMU* iIMU);
17
18     void Initialize() override;
19
20     void Execute() override;
21
22     void End(bool interrupted) override;
23
24     bool IsFinished() override;
25
26 private:
27     SubDriveTrain* mDriveTrain;
28     SubIMU* mIMU;
29
30     double mStartingAngle;
31 };

```

```

1 // TurnRight.cpp
2
3 #include "commands/TurnRight.h"
4
5 TurnRight::TurnRight(SubDriveTrain *iDriveTrain, SubIMU *iIMU)

```

```

6  {
7      mDriveTrain = iDriveTrain;
8      mIMU = iIMU;
9      AddRequirements(mDriveTrain);
10     AddRequirements(mIMU);
11 }
12
13 void TurnRight::Initialize()
14 {
15     mStartingAngle = mIMU->getAngleYaw();
16 }
17
18 void TurnRight::Execute()
19 {
20     mDriveTrain->mecanumDrive(0, 0, 0.8, mIMU->getRotation2d());
21 }
22
23 void TurnRight::End(bool interrupted)
24 {
25 }
26 }
27
28 bool TurnRight::IsFinished()
29 {
30     return (mIMU->getAngleYaw() - mStartingAngle) > 72;
31 }

```

2. On souhaite créer une *Command* permettant de faire monter un ascenseur sur lequel est placé un mécanisme important du robot. Le robot possède les spécificités suivantes :
 - 2 moteurs *NEO Brushless* pour l'ascenseur avec un *Subsystem* contrôlant 2 contrôleurs *CANSparkMax*.
 - Le *Subsystem* en question possède des méthodes *setTargetPosition()*, *getTargetPosition()* et *moveToPosition()* pour contrôler la position du pivot. Des méthodes *getEncoderPositionMotorL()* et *getEncoderPositionMotorR()* sont là pour déterminer la position actuelle.
 - Un *PID* lié aux contrôleurs *CANSparkMax* est utilisé pour atteindre la position de manière optimale.
-

```

1  // Constants.h
2
3  #pragma once
4
5  namespace ElevatorConstants {
6
7      constexpr int kMotorId1 = 1;
8      constexpr int kMotorId2 = 3;
9
10     constexpr double kElevatorLimitUp = 82 - 83;
11     constexpr double kElevatorLimitMiddle = 55 - 83;
12     constexpr double kElevatorLimitDown = 0 - 83 ;
13
14     constexpr double kP = 0.00009999999873689376,
15                     kI = 19.999999974752427e-7,
16                     kD = 0,
17                     kIz = 0,
18                     kFF = 0.000155999994603917,

```

```

19         kMaxOutput = 1,
20         kMinOutput = -1;
21     }

```

```

1 // ElevatorUp.h
2
3 // Copyright (c) FIRST and other WPILib contributors.
4 // Open Source Software; you can modify and/or share it under the terms of
5 // the WPILib BSD license file in the root directory of this project.
6
7 #pragma once
8
9 #include <frc2/command/Command.h>
10 #include <frc2/command/CommandHelper.h>
11
12 #include "subsystems/SubElevator.h"
13
14 class ElevatorUp
15 : public frc2::CommandHelper<frc2::Command, ElevatorUp> {
16 public:
17     ElevatorUp(SubElevator* iElevator);
18
19     void Initialize() override;
20
21     void Execute() override;
22
23     void End(bool interrupted) override;
24
25     bool IsFinished() override;
26
27 private:
28     SubElevator* mElevator;
29 };

```

```

1 // ElevatorUp.cpp
2
3 // Copyright (c) FIRST and other WPILib contributors.
4 // Open Source Software; you can modify and/or share it under the terms of
5 // the WPILib BSD license file in the root directory of this project.
6
7 #include <cmath>
8
9 #include "commands/ElevatorUp.h"
10
11 ElevatorUp::ElevatorUp(SubElevator* iElevator) {
12
13     mElevator = iElevator;
14     AddRequirements(mElevator);
15 }
16
17 // Called when the command is initially scheduled.
18 void ElevatorUp::Initialize()
19 {
20     mElevator->setTargetPosition(ElevatorConstants::kElevatorLimitUp);

```

```

21 }
22
23 // Called repeatedly when this Command is scheduled to run
24 void ElevatorUp::Execute()
25 {
26     mElevator->moveToPosition();
27 }
28
29 // Called once the command ends or is interrupted.
30 void ElevatorUp::End(bool interrupted)
31 {
32
33 }
34
35 // Returns true when the command should end.
36 bool ElevatorUp::IsFinished() {
37     return std::abs(mElevator->getEncoderPositionMotor1() -
38                     mElevator->getTargetPosition()) < 0.05;
39 }

```

3.4.3 Enchaîner des *Commands* (nouvelle méthode)

Il est possible d'exécuter plusieurs *Commands* à la suite de diverses façons :

- De manière séquentielle : chaque *Command* s'exécute l'une à la suite des autres, en attendant d'abord que la précédente soit terminée.
- De manière parallèle : chaque *Command* s'exécute en même temps. On peut soit attendre qu'elles soient toutes terminées ou qu'une seule termine pour mettre fin à l'exécution.
- De manière répétée : la commande (ou séquence de commande) peut s'exécuter tant qu'elle n'est pas interrompue.

Exemples :

1. On souhaite faire avancer le robot pendant 1 seconde, puis le faire tourner à gauche et enfin le faire avancer pendant 4 secondes. Le robot possède les spécificités suivantes :
 - Une *Command Forward* qui permet de faire avancer le robot pendant un certain temps.
 - Une *Command TurnLeft* qui permet de faire tourner le robot de 90 degrés dans le sens anti-horaire.

```

1 #include <frc2/command/Commands.h>
2
3 #include "commands/Forward.h"
4 #include "commands/TurnLeft.h"
5 ...
6 .
7 .
8 .
9 ...
10 frc2::CommandPtr commandSequence = frc2::cmd::Sequence(Forward(&mDriveTrain,
    1).ToPtr(), TurnLeft(&mDriveTrain, &mIMU).ToPtr(), Forward(&mDriveTrain,
    4).ToPtr());

```

2. On souhaite faire avancer le robot pendant 2 secondes, descendre un ascenseur en bas et activer un ramasseur de balles, et ce de manière simultanée . Le robot possède les spécificités suivantes :
 - Une *Command Forward* qui permet de faire avancer le robot pendant un certain temps.

- Une *Command ElevatorDown* qui permet de descendre l'ascenseur à sa position la plus basse.
- Une *Command PickBall* qui permet de ramasser une balle au sol.

```

1 #include <frc2/command/Commands.h>
2
3 #include "commands/Forward.h"
4 #include "commands/ElevatorDown.h"
5 #include "commands/PickBall.h"
6 ...
7 .
8 .
9 .
10 ...
11 frc2::CommandPtr parallelCommand = frc2::cmd::Parallel(Forward(&mDriveTrain,
    2).ToPtr(), ElevatorDown(&mElevator).ToPtr(), PickBall(&mPicker).ToPtr());

```

Remarque : La méthode *ToPtr()* sert à convertir une *Command* en *CommandPtr*, un format utilisé pour la *Command* de période autonome et pour lier une *Command* à un événement d'un *Joystick* (par exemple un bouton pressé).

3.4.4 Enchaîner des *Commands* (ancienne méthode)

La méthode présentée précédemment pour enchaîner des *Commands* est un raccourci qui n'a pas toujours existé. Dans cette section, on va adapter le code des exemples précédents pour constater les différences. Pour ce faire, on utilise à chaque fois un *CommandGroup* qui permet d'exécuter plusieurs *Commands*.

Exemples :

1. Séquence de *Commands* :

```

1 // Path.h
2
3 // Copyright (c) FIRST and other WPILib contributors.
4 // Open Source Software; you can modify and/or share it under the terms of
5 // the WPILib BSD license file in the root directory of this project.
6
7 #pragma once
8
9 #include <frc2/command/CommandHelper.h>
10 #include <frc2/command/SequentialCommandGroup.h>
11
12 #include "commands/Forward.h"
13 #include "commands/TurnLeft.h"
14
15 class Path : public frc2::CommandHelper<frc2::SequentialCommandGroup, Path>
16 {
17 public:
18     Path(SubDriveTrain* iDriveTrain, SubIMU* iIMU);
19 };

```

```

1 // Path.cpp
2
3 // Copyright (c) FIRST and other WPILib contributors.

```

```

4 // Open Source Software; you can modify and/or share it under the terms of
5 // the WPILib BSD license file in the root directory of this project.
6
7 #include "commands/Path.h"
8
9 Path::Path(SubDriveTrain* iDriveTrain, SubIMU* iIMU)
10 {
11     AddCommands(Forward(iDriveTrain, 1), TurnLeft(iDriveTrain, iIMU),
12                 Forward(iDriveTrain, 4));
13 }

```

2. *Commands* en parallèle :

```

1 // Preparation.h
2
3 // Copyright (c) FIRST and other WPILib contributors.
4 // Open Source Software; you can modify and/or share it under the terms of
5 // the WPILib BSD license file in the root directory of this project.
6
7 #pragma once
8
9 #include <frc2/command/CommandHelper.h>
10 #include <frc2/command/ParallelCommandGroup.h>
11
12 #include "commands/Forward.h"
13 #include "commands/ElevatorDown.h"
14 #include "commands/PickBall.h"
15
16 class Preparation : public frc2::CommandHelper<frc2::ParallelCommandGroup, Preparation>
17 {
18 public:
19     Preparation(SubDriveTrain* iDriveTrain, SubElevator* iElevator, SubPicker* iPicker);
20 };

```

```

1 // Preparation.cpp
2
3 // Copyright (c) FIRST and other WPILib contributors.
4 // Open Source Software; you can modify and/or share it under the terms of
5 // the WPILib BSD license file in the root directory of this project.
6
7 #include "commands/Preparation.h"
8
9 Preparation::Preparation(SubDriveTrain* iDriveTrain, SubElevator* iElevator, SubPicker*
10 iPicker)
11 {
12     AddCommands(Forward(iDriveTrain, 2), ElevatorDown(iElevator), PickBall(iPicker));
13 }

```

Remarque : Pour utiliser des *CommandGroups* avec des événements ou pour la période autonome, il est nécessaire d'appeler la méthode *ToPtr()* qui est aussi disponible.

3.5 RobotContainer

La classe *RobotContainer* est la classe qui permet de lier les *Subsystems* aux *Commands* et les *Commands* aux évènement d'une manette ou d'un *Joystick*. Les méthodes importantes à retenir sont :

- ***ConfigureBindings()*** : Méthode permettant de lier différents boutons à des *Commands*. La ligne type est de la forme :

```

1 frc2::Trigger([this] {
2     return mJoystick.GetRawButtonPressed(JoystickBindingsConstants::kElevatorUp);
3 }).OnTrue(ElevatorUp(&mElevator).ToPtr());

```

- ***Init()*** : Méthode exécutée une seule fois au début du programme. Elle sert notamment à réinitialiser les systèmes devant l'être.
- ***GetAutonomousCommand()*** : Méthode renvoyant la *CommandPtr*. Il faut s'assurer que la bonne *CommandPtr* est renvoyée.

Remarque : L'expression dans les parenthèses du constructeur de *Trigger* est une fonction lambda. Ce concept n'a pas été présenté puisqu'il rend le code moins clair. Il est par contre tout à fait possible de remplacer cette expression par une fonction booléenne faisant la même chose (en enlevant les parenthèses car la fonction ne doit pas avoir d'arguments).

3.6 Robot

La classe *Robot* est la classe que vous ne devriez **jamais** toucher, à moins de faire des tests très rapides, pour la simple raison que vous n'avez pas besoin de le faire (la classe *RobotContainer* servant déjà à ça). Cette classe possède des méthodes qui sont appelées en fonction des différentes phases d'un match. Les plus importantes sont :

- ***AutonomousInit()*** et ***AutonomousPeriodic()*** (appelée toutes les 20 ms) pour la période autonome durant 15 s.
- ***TeleopInit()*** et ***TeleopPeriodic()*** (appelée toutes les 20 ms) pour la période téléopérée.

Chapitre 4

VS Code

4.1 Présentation

VS Code est un *IDE*, c'est-à-dire un environnement de développement dans lequel il est facile de programmer. Il est doté d'un formidable outil appelé ***IntelliSense*** permettant de gagner en productivité. Il permet l'autocomplétion (complète le nom des variables, méthodes... seulement à partir de quelques caractères) et l'accès plus facile aux caractéristiques des variables et méthodes (types, paramètres...).

4.2 Liste de raccourcis indispensables

Voici une liste non exhaustive des raccourcis important à connaître pour utiliser *VS Code* au maximum :

- CTRL + SPACE : Permet d'activer le menu d'autocomplétion.
- CTRL + S : Permet de sauvegarder le fichier actuellement ouvert.
- CTRL + C : Permet de copier des caractères sélectionnés.
- CTRL + X : Permet de couper des caractères sélectionnés.
- CTRL + V : Permet de coller des caractères précédemment copiés ou coupés.
- CTRL + Z : Permet de revenir en arrière d'une action.
- CTRL + Y : Permet de revenir en avant d'une action.
- CTRL + F : Permet de chercher une chaîne de caractères précise dans le fichier actuellement ouvert.
- CTRL + H : Permet d'ouvrir un menu remplaçant automatiquement les occurrences d'une chaîne de caractères par une autre.
- CTRL + W : Permet de fermer le fichier actuellement ouvert.
- CTRL + TAB : Permet de basculer au dernier fichier consulté actuellement ouvert. Maintenir la touche CTRL permet de naviguer entre tous les fichiers ouverts en appuyant successivement sur TAB.
- SHIFT + Flèches : Permet de sélectionner un caractère ou une ligne. Maintenir la touche SHIFT permet d'en sélectionner plusieurs en appuyant successivement sur les flèches. On peut appuyer sur CTRL pour sélectionner un mot au lieu d'un seul caractère.

4.3 Utilisation avec WPILIB

La version de *VS Code* utilisée pour programmer un robot est une version modifiée (attention à bien ouvrir la bonne!). Quelques fonctionnalités supplémentaires ont été rajoutées. Pour accéder à la plupart d'entre elles, il suffit d'appuyer sur le bouton en haut à droite avec un "W". Cela va ouvrir un menu déroulant à plusieurs options.

4.3.1 Création d'un projet

1. Appuyer sur le bouton "W"
2. Sélectionner *Create a new project*
3. Sélectionner le type de projet : *Template* pour choisir le modèle de base (choisir l'option *Command-Robot* pour le *Command-Based Programming*) ou *Example* pour utiliser des programmes préfait et comprendre leur fonctionnement.
4. Indiquer le nom du projet.
5. Indiquer le répertoire de base (le placer dans un répertoire lié à Git pour versionner le code).
6. Indiquer le numéro de l'équipe.

4.3.2 Création d'un *Subsystem* ou d'une *Command*

1. Faire CLIC DROIT sur le répertoire (menu à gauche) *subsystems* (respectivement *commands*)
2. Appuyer sur *Create a new class/command*
3. Sélectionner *Subsystem* (respectivement *Command*)
4. Indiquer le nom du *Subsystem* (respectivement de la *Command*)

Pour compiler le projet, il suffit d'appuyer sur le bouton "W" ou le bouton "... " et d'appuyer sur "Build Robot Code".

4.3.3 Compilation et déploiement

Pour déployer le projet sur le RoboRIO, il suffit d'appuyer sur le bouton "W" ou le bouton "... " et d'appuyer sur "Deploy Robot Code". Le raccourci SHIFT + F5 fonctionne aussi.

4.3.4 Gestion des bibliothèques de tiers

- Appuyer sur le bouton "W"
- Sélectionner *Manage Vendor Libraries*
- Sélectionner l'option désirée