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

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 cœur 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 * *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 }
```

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

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

// Programme qui calcule les images de la
// fonction 4x^3 - 3,5/x + 1
int main()
{
    float nombre;

    cout << "Veuillez rentrer un nombre (x) :" << endl;
    cin >> nombre;

    cout << "4 * x^3 - 3.5 / x + 1 = " << 4 * nombre * nombre * nombre - 3.5 / nombre +
        float(1);

    return 0;
}
```

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	$a >= b$
$a <= b$	a INFÉRIEUR OU ÉGAL À b	$a > b$
$a >= b$	a SUPÉRIEUR OU ÉGAL À b	$a < b$
$a > b$	a STRICTEMENT SUPÉRIEUR À b	$a <= b$
$a == b$	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 :

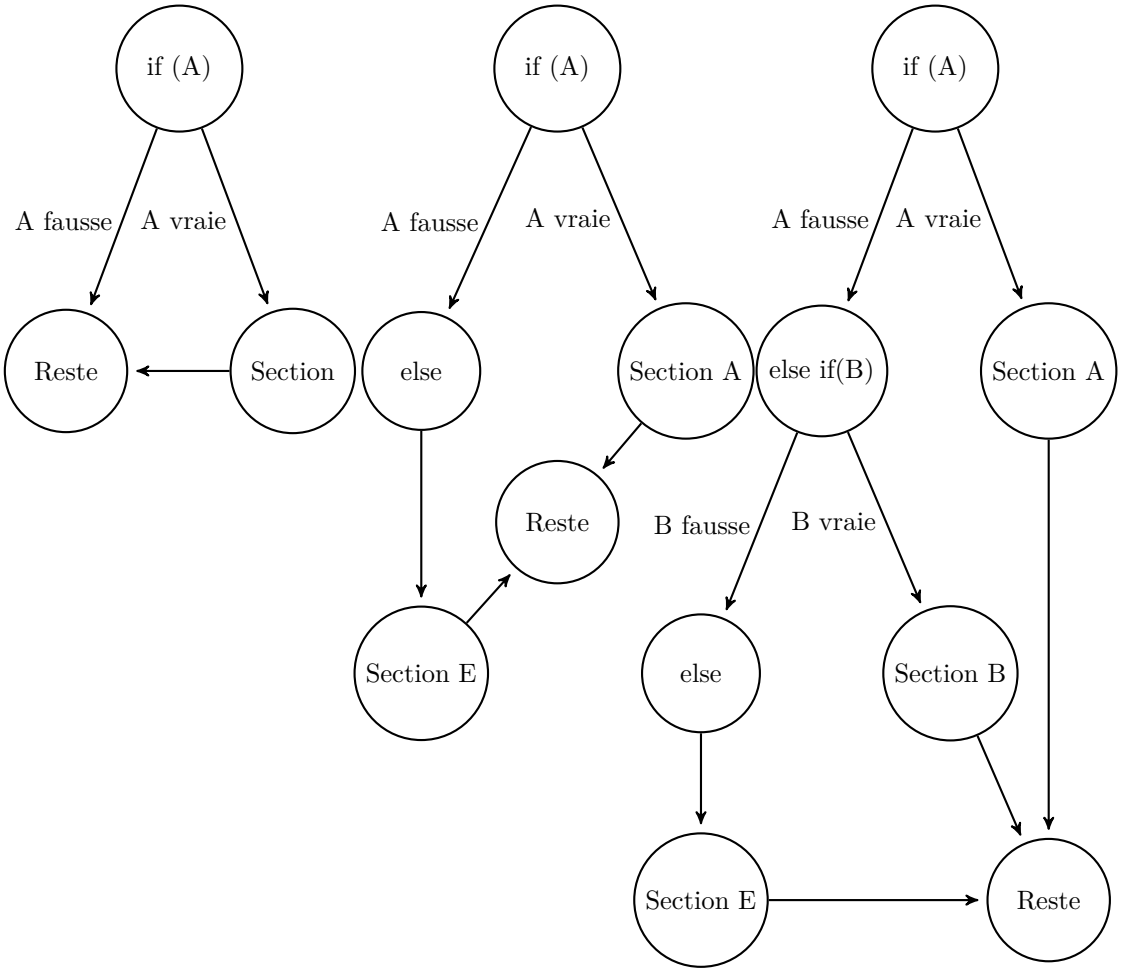

FIGURE 1.1 – Graphe illustrant le fonctionnement des structures conditionnelles

Commentaires sur le graphe :

— Les lettres A et B représentent différentes expression 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 :

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

int main()
{
    int nombre;

    cout << "Veuillez donner un nombre : ";
    cin >> nombre;

    if (nombre % 2 == 0)
    {
        cout << nombre << " est un nombre pair.";
    }

    else
    {
        cout << nombre << " est un nombre impair.";
    }

    return 0;
}
```

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.

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

int main()
{
    float nombre;

    cout << "Veuillez donner un nombre : ";
    cin >> nombre;

    if (nombre < 2)
    {
        cout << "Le double de " << nombre << " est " << 2 * nombre;
    }

    else if (nombre < 8 && nombre * nombre > 12)
    {
        cout << "Le triple de " << nombre << " est " << 3 * nombre;
    }

```

```
    else if (nombre * nombre * nombre <= 2685.619)
    {
        cout << "Le carré de " << nombre << " est " << nombre * nombre;
    }

    else
    {
        cout << "Le cube de " << nombre << " est " << nombre * nombre * nombre;
    }

    return 0;
}
```

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 commançant à 1) :

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

int main()
{
    int i = 1;
    int compteur = 1;

    while (compteur <= 100)
    {
        if (i % 2 == 1)
        {
            cout << i << endl;
            compteur++;
        }

        i++;
    }

    return 0;
```

```
}
```

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

int main()
{
    for (int i = 1; i < 201; i += 2)
    {
        cout << i << endl;
    }

    return 0;
}
```

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

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

int main()
{
    int compteur = 0;
    int entierActuel = 2;
    bool premier;

    while (compteur < 100)
    {
        premier = true;

        for (int i = 2; i < entierActuel; i++)
        {
            if (entierActuel % i == 0)
            {
                premier = false;
            }
        }

        if (premier)
        {
            cout << "Le nombre premier #" << compteur + 1 << " est " << entierActuel <<
                endl;
            compteur++;
        }

        entierActuel++;
    }

    return 0;
}
```

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

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

int main()
{
    float nombre;

    cout << "Veuillez rentrer un nombre : ";
    cin >> nombre;

    float resultat = nombre;
    int compteur = 0;

    while (resultat > -1000 && resultat < 1000)
    {
        if (compteur == 100)
        {
            break;
        }

        cout << resultat << endl;

        resultat *= nombre;

        compteur++;
    }

    return 0;
}
```

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	*int*	42, -6622, 0...
Type de base	*float*	1.22, -42.6...
Type de base	*double*	3.14159, -726.82102...
Type de base	*bool*	*true* ou *false*
enum	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 :

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

int main()
{
    int resultat;

    cout << "Veuillez entrer le résultat du lancer : ";
    cin >> resultat;

    if (resultat == 1)
    {
        cout << "Vous avez fait 1 !";
    }

    else if (resultat == 2)
    {
        cout << "Vous avez fait 2 !";
    }

    else if (resultat == 3)
    {
        cout << "Vous avez fait 3 !";
    }

    else if (resultat == 4)
    {
        cout << "Vous avez fait 4 !";
    }

    else if (resultat == 5)
    {
        cout << "Vous avez fait 5 !";
    }

    else if (resultat == 6)
    {
        cout << "Vous avez fait 6 !";
    }

    else
    {
        cout << "Votre dé est étonnant !";
    }

    return 0;
}
```

```cpp
#include <iostream>
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()
```

```
{
    float vitesse = 6.5;
    float tempsEcoule = 0;
    float distanceParcourue = 0;

    ModeVitesse mode = modeRapide;

    while (tempsEcoule < 100)
    {
        if (distanceParcourue < 100 || tempsEcoule > 85)
        {
            mode = modeRapide;
        }

        else if (tempsEcoule < 60 && distanceParcourue > 100)
        {
            mode = modeLent;
        }

        else
        {
            mode = modeMoyen;
        }

        tempsEcoule += 0.5;

        cout << "Temps écoulé : " << tempsEcoule << " Distance parcourue : ";

        switch (mode)
        {
            case modeLent:
                distanceParcourue += 0.5 * vitesse * 0.5;
                cout << distanceParcourue << " Mode lent" << endl;
                break;

            case modeMoyen:
                distanceParcourue += 0.5 * vitesse * 0.75;
                cout << distanceParcourue << " Mode moyen" << endl;
                break;

            case modeRapide:
                distanceParcourue += 0.5 * vitesse * 0.9;
                cout << distanceParcourue << " Mode rapide" << endl;
                break;

            default:
                break;
        }
    }

    return 0;
}
```

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 :

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

int main()
{
    vector<float> nombres = {0.31, 8.12, -38.1, 642.02};

    for (int i = 0; i < nombres.size(); i++)
    {
        nombres.at(i) = nombres.at(i) * nombres.at(i);
    }

    cout << (nombres.at(0) + nombres.at((int)nombres.size() - 1)) / 2;

    return 0;
}
```

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

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

int main()
{
    vector<int> nombrePremiers;
    int indice = 2;
    int x;

    bool pasPremier;

    while (indice <= 100)
    {
        x = 2;
        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 :

```
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 :

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

int main()
{
    int a = 2;
    int* b = new int;

    *b = 2 * a - 3;

    cout << "Valeur de a : " << a << endl;
    cout << "Adresse de a : " << &a << endl;
    cout << "Valeur de b : " << *b << endl;
    cout << "Adresse de b : " << b;

    return 0;
}
```

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

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

int main()
{
    float nombre = 3.14;
    float* ptrNombre = &nombre;
    *ptrNombre = 3.1415;

    cout << nombre;

    return 0;
}
```

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.

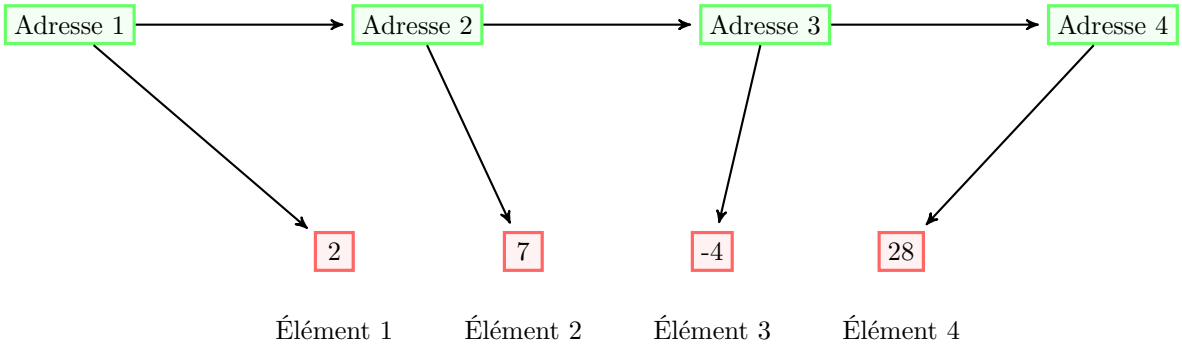

FIGURE 1.2 – Relations entre les éléments d'un tableau et des adresses

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 :

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

int main()
{
    int valeurs[4] = {-71, 42, -13, 56};

    for (int i = 0; i < sizeof(valeurs) / sizeof(int); i++)
    {
        cout << valeurs[i] << endl;
    }

    return 0;
}
```

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

int main()
{
```

```
    int valeurs[4] = {-71, 42, -13, 56};
    int* element = valeurs;

    for (int i = 0; i < sizeof(valeurs) / sizeof(int); i++)
    {
        cout << *element << endl;
        element++;
    }

    return 0;
}
```

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.

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

int main()
{
    float positions[32] = {0.0, 0.36, 0.62, 0.98, 1.73, 3.20, 6.28, 12.03,
                           17.02, 18.38, 19.3, 20.31, 21.38, 23.29, 25.29, 27.82,
                           30.23, 36.92, 42.93, 49.29, 55.92, 62.38, 69.8, 75.42,
                           85.18, 87.52, 89.62, 90.10, 90.15, 90.56, 90.57, 90.62 };

    float* ptr = positions;

    float deltaT = 0.1;
    vector<float> vitesses;

    for (int i = 1; i < 31; i++)
    {
        vitesses.push_back((positions[i + 1] - positions[i - 1]) / (2 * deltaT));
        cout << i << " : " << vitesses.at(i - 1) << endl;
    }

    *ptr = -1000;

    cout << positions[0];

    return 0;
}
```

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*** apparait. 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 fonction [2] et pour calculer la moyenne d'un tableau sans renvoyer de valeur :

```
#include <iostream>
#include <cmath> // On utilise ce module pour la fonction sinus
using namespace std;

float f(float x)
{
    if (x == 0)
    {
        return 1;
    }

    return sin(x) / x;
}

void moyenne(float* valeurs, float* resultat, int taille)
{
    // Bonne pratique pour éviter d'ajouter des valeurs à une variable potentiellement
    // pas initialisée.
    *resultat = 0;

    for (int i = 0; i < taille; i++)
    {
        *resultat += valeurs[i];
    }

    *resultat /= (float)taille;
}

int main()
```

2. Je parle ici de fonction mathématique

```
{
    float resultat = 0, x = -1;
    float valeurs[4];

    for (int i = 0; i < sizeof(valeurs) / sizeof(float); i++)
    {
        valeurs[i] = f(x);
        x += 0.5;
    }

    moyenne(valeurs, &resultat, 4);

    cout << "La moyenne des valeurs est : " << resultat;

    return 0;
}
```

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...) :

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

int* inverser(int* tableau, int taille)
{
    int valeurTemporaire;

    for (int i = 0; i < taille / 2; i++)
    {
        valeurTemporaire = tableau[i];
        tableau[i] = tableau[taille - i - 1];
        tableau[taille - i - 1] = valeurTemporaire;
    }

    return tableau;
}

int main()
{
    int tableau[4] = {6622, 3990, 296, 2626};
    int* tableauInverse = inverser(tableau, (sizeof(tableau) / sizeof(int)));

    for (int i = 0; i < sizeof(tableau) / sizeof(int); i++)
    {
        cout << tableauInverse[i] << endl;
    }

    return 0;
}
```

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

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

```

```
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.

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

double logTaylor(double x, int iterations = 5)
{
    if (x >= 1 || x <= -1)
    {
        return 0;
    }

    double resultat = 0;

    double xi = x;
    x *= x;
    double x2 = x;

    for (int i = 2; i <= iterations; i += 2)
    {
        resultat -= x / i;
        x *= x2;
    }

    x = xi;

    for (int i = 1; i <= iterations; i += 2)
    {
        resultat += x / i;
        x *= x2;
    }

    return resultat;
}

int main()
{
    cout << "Ordre 5 : " << logTaylor(0.1) << endl << "Ordre 10 : " << logTaylor(0.1, 10);

    return 0;
}
```

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.

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

float polynome(float& x)
{
    x *= x;

    return 3 * x * x * (x - 8) + x + 1;
}

int main()
{
    float x = 3.2;

    cout << "x initial :" << x << endl;
    cout << "P(x)    = " << polynome(x) << endl;
    cout << "x final :" << x;

    return 0;
}
```

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.

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

void logTaylor(double& x, int iterations = 5)
{
    if (x >= 1 || x <= -1)
    {
        x = 0;
    }

    else
    {
        double resultat = 0;

        double xi = x;
        x *= x;
        double x2 = x;

        for (int i = 2; i <= iterations; i += 2)
        {
            resultat -= x / i;
            x *= x2;
        }

        x = xi;

        for (int i = 1; i <= iterations; i += 2)
        {
```

```
            resultat += x / i;
            x *= x2;
        }

        x = resultat;
    }
}

int main()
{
    double x = 0.1;

    cout << "Ordre 5 :";
    logTaylor(x);
    cout << x;

    return 0;
}
```

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 :

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

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


double f(double x)
{
    return sqrt(1 - x * x);
}

vector<double> aire(double a, double b, int points = 10)
{
    vector<double> aires;
    double aire, h;

    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 évènement 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 sections [3] 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 créé 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.

```
    ~Rectangle()
    {
        cout << "Objet détruit !" << endl;
    }
};

int main()
{
    Rectangle rect;

    return 0;
}
```

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 :

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

class Rectangle
{
    float largeur;

public:
    Rectangle()
    {
        cout << "Objet créé !" << endl;
    }

    ~Rectangle()
    {
        cout << "Objet détruit !" << endl;
    }

    float hauteur;
};

int main()
{
    Rectangle rect;
    rect.hauteur = 3.1;
    cout << rect.hauteur << endl;

    return 0;
}
```

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* :

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

class Rectangle
{
    float largeur;
    float hauteur;

public:
    // Il n'est pas nécessaire de remplir le constructeur
    // et le destructeur.
    Rectangle() {  }
    ~Rectangle() {  }

    float getLargeur()
    {
        return largeur;
    }

    float getHauteur()
    {
        return hauteur;
    }

    void setLargeur(float Largeur)
    {
        largeur = Largeur;
    }

    void setHauteur(float Hauteur)
    {
        hauteur = Hauteur;
    }
```

```
};

int main()
{
    Rectangle rect;
    rect.setLargeur(3.1);
    rect.setHauteur(4.5);

    cout << "L'aire du rectangle est : " << rect.getLargeur() * rect.getHauteur();

    return 0;
}
```

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 :

```
class Rectangle
{
    float largeur;
    float hauteur;

public:
    Rectangle();
    ~Rectangle();

    float getLargeur();
    float getHauteur();

    void setLargeur(float Largeur);
    void setHauteur(float Hauteur);
};

Rectangle::Rectangle() { }

Rectangle::~Rectangle() { }

float Rectangle::getLargeur()
{
    return largeur;
}

float Rectangle::getHauteur()
{
    return hauteur;
}

```

```
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 à 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* !).

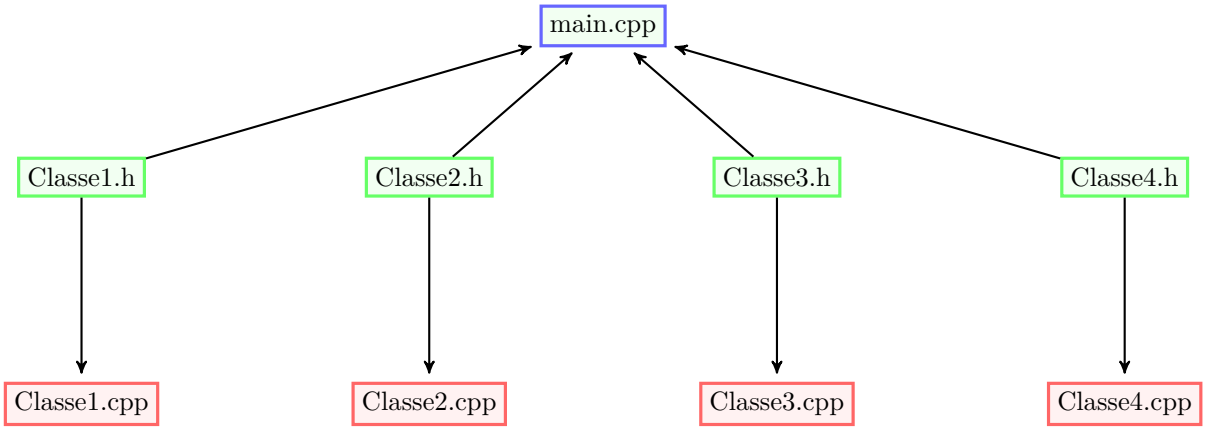

FIGURE 1.3 – Architecture classique en POO

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

```
// rectangle.h

class Rectangle
{
    float largeur;
    float hauteur;

public:
    Rectangle();
    ~Rectangle();

    float getLargeur();
    float getHauteur();

    void setLargeur(float Largeur);
    void setHauteur(float Hauteur);
};
```

```
// rectangle.cpp

#include "rectangle.h"

Rectangle::Rectangle() { }

Rectangle::~Rectangle() { }

float Rectangle::getLargeur()
{
    return largeur;
}

float Rectangle::getHauteur()
{
    return hauteur;
}

void Rectangle::setLargeur(float Largeur)
{
    largeur = Largeur;
}

void Rectangle::setHauteur(float Hauteur)
{
    hauteur = Hauteur;
}
```

```
// main.cpp

#include "rectangle.h"

#include <iostream>
using namespace std;

```

```
int main()
{
    Rectangle rect;
    rect.setLargeur(3.1);
    rect.setHauteur(4.5);

    cout << "L'aire du rectangle est : " << rect.getLargeur() * rect.getHauteur();

    return 0;
}
```

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 classe 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* :

```
// main.cpp

#include "rectangle.h"

#include <iostream>
using namespace std;

int main()
{
    Rectangle* rect = new Rectangle;
    rect->setLargeur(3.1);
    rect->setHauteur(4.5);

    cout << "L'aire du rectangle est : " << rect->getLargeur() * rect->getHauteur();

    return 0;
}
```

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 :

```
// rectangle.h

class Rectangle
{
    float* largeur;
    float* hauteur;

public:
    Rectangle();
    ~Rectangle();

    float getLargeur();
    float getHauteur();

    void setLargeur(float Largeur);
    void setHauteur(float Hauteur);
};
```

```
// rectangle.cpp

#include "rectangle.h"

Rectangle::Rectangle()
{
    largeur = new float;
    hauteur = new float;
}

Rectangle::~Rectangle() { }

float Rectangle::getLargeur()
{
    return *largeur;
}

float Rectangle::getHauteur()
{
    return *hauteur;
}

void Rectangle::setLargeur(float Largeur)
{
    *largeur = Largeur;
}

void Rectangle::setHauteur(float Hauteur)
{
    *hauteur = Hauteur;
}
```

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* :

```
// cercle.h

class Cercle
{
    float rayon;

public:
    Cercle();
    Cercle(float Rayon);
    ~Cercle();

    float getRayon();

    void setRayon(float Rayon);
};
```

```
// cercle.cpp

#include "cercle.h"

Cercle::Cercle() { }

Cercle::Cercle(float Rayon)
{
    rayon = Rayon;
}

Cercle::~Cercle() { }

float Cercle::getRayon()
{
    return rayon;
}

void Cercle::setRayon(float Rayon)
{
    rayon = Rayon;
}
```

```
// main.cpp

#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()
```

```
{
    return *largeur;
}

float Rectangle::getHauteur()
{
    return *hauteur;
}

void Rectangle::setLargeur(float Largeur)
{
    *largeur = Largeur;
}

void Rectangle::setHauteur(float Hauteur)
{
    *hauteur = Hauteur;
}
```

```
// main.cpp

#include "rectangle.h"

#include <iostream>
using namespace std;

int main()
{
    Rectangle* rect = new Rectangle(3, 4);
    // Ici on appelle carrément le constructeur pour qu'il retourne
    // un objet de type Rectangle avec les informations spécifiées

    cout << "Aire du rectangle : " << rect->getLargeur() * rect->getHauteur();

    return 0;
}
```

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 :

```
Rectangle rect(2, 3);
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 :

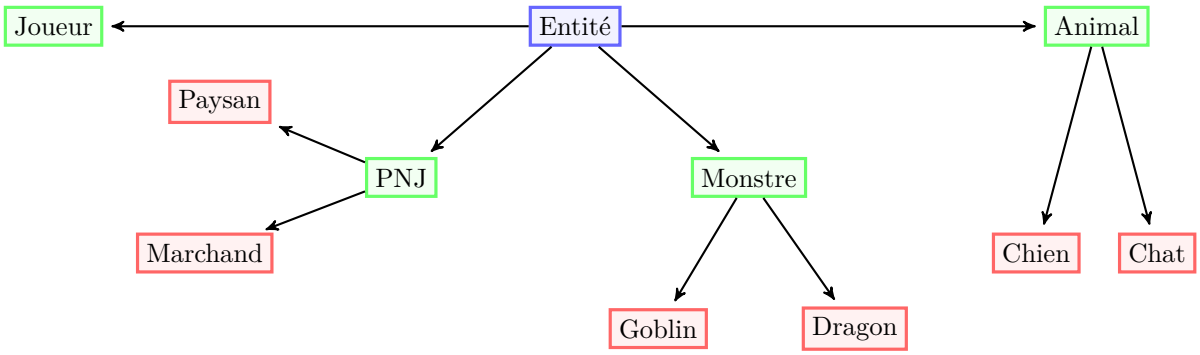

FIGURE 1.4 – Schéma de dépendance des classes

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 :

```
class B : public A
{
    // Reste de la classe
};
```

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 :

```
// entite.h

class Entite
{
    float x;
    float y;

public:
    Entite();
    ~Entite();

    float getX();
    float getY();

    void setX(float X);
    void setY(float Y);
};
```

```
// entite.cpp

#include "entite.h"

Entite::Entite() { }

Entite::~Entite() { }

float Entite::getX()
```

```
{
    return x;
}

float Entite::getY()
{
    return y;
}

void Entite::setX(float X)
{
    x = X;
}

void Entite::setY(float Y)
{
    y = Y;
}
```

```
// joueur.h

#include "entite.h"

class Joueur : public Entite
{
    float pointDeVie;

public:
    Joueur();
    ~Joueur();

    float getPointDeVie();

    void setPointDeVie(float PointDeVie);
};
```

```
// joueur.cpp

#include "joueur.h"

Joueur::Joueur() { }

Joueur::~Joueur() { }

float Joueur::getPointDeVie()
{
    return pointDeVie;
}

void Joueur::setPointDeVie(float PointDeVie)
{
    pointDeVie = PointDeVie;
}
```

```cpp
// main.cpp

#include "joueur.h"

int main()
{
    Joueur joueur;

    // Il est possible d'accéder aux méthodes de la classe Entite car
    // Joueur hérite de Entite
    joueur.setX(0);
    joueur.setY(0);
    joueur.setPointDeVie(30);

    return 0;
}
```

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 créé 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.

```cpp
// composant.h

enum EtatComposant
{
    Fonctionnel,
    Endommage
};

class Composant
{
    EtatComposant etat;
    bool actif;
    int dureeVie;

    const int dureeVieMaxComposant = 500;

public:
    Composant();
    ~Composant();

    EtatComposant getEtat();

    bool getActif();

    int getDureeVie();

    void activer();
    void desactiver();
};
```

```cpp
// composant.cpp

#include "composant.h"

Composant::Composant()
{
    etat = EtatComposant::Fonctionnel;
    actif = false;
    dureeVie = 0;
}

Composant::~Composant() { }

EtatComposant Composant::getEtat()
{
    return etat;
}

bool Composant::getActif()
{
    return actif;
}

int Composant::getDureeVie()
{
    return dureeVie;
}

void Composant::activer()
{
    if (etat == EtatComposant::Fonctionnel)
    {
        if (!actif)
        {
            actif = true;
        }

        dureeVie++;
    }

    if (dureeVie == dureeVieMaxComposant)
    {
        actif = false;
        etat = EtatComposant::Endommage;
    }
}

void Composant::desactiver()
{
    actif = false;
}
```

```cpp
// moteur.h

#include "composant.h"
```

```

class Moteur : public Composant
{
    float voltage;
    int port;

public:
    Moteur();
    Moteur(int Port);
    ~Moteur();

    float getVoltage();

    void setVoltage(float Voltage);
};
```

```
// moteur.cpp

#include "moteur.h"

Moteur::Moteur()
{
    voltage = 0;
    port = 0;
}

Moteur::Moteur(int Port)
{
    voltage = 0;
    port = Port;
}

Moteur::~Moteur() { }

float Moteur::getVoltage()
{
    return voltage;
}

void Moteur::setVoltage(float Voltage)
{
    voltage = Voltage;

    if (voltage == 0 || getEtat() == EtatComposant::Endommage)
    {
        desactiver();
        voltage = 0;
    }

    else
    {
        activer();
    }
}
```

```cpp
// main.cpp

#include "moteur.h"

#include <iostream>
using namespace std;

int main()
{
    Moteur* moteur = new Moteur(1);

    moteur->setVoltage(1);

    while(moteur->getActif())
    {
        moteur->setVoltage(1);
        cout << moteur->getDureeVie() << endl;
    }

    return 0;
}
```

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 "boite". Pour atteindre un élément, il suffit alors d'inclure le module et de préciser le nom de cette "boite".

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 :

```
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 varibale *pi* :

```
// alice.h

namespace Alice
{
    float pi = 3.14;
}
```

```
// bob.h

namespace Bob
{
    float pi = 3.142;
}
```

```
// main.cpp

#include "alice.h"
#include "bob.h"

#include <iostream>
// std est un espace de noms ! Voici ce que cela donne sans
// la ligne habituelle :

int main()
{
    float pi = 3.1415;
```

```
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  }
```

Mots-clés de la section : ***namespace*, *using***

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.

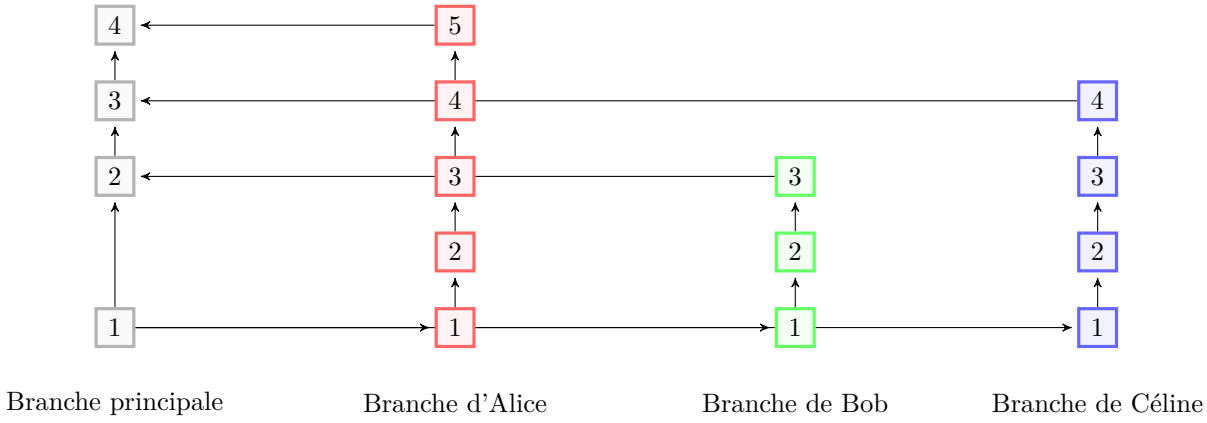

FIGURE 2.1 – Schéma de branches

À chaque modification conséquente, chaque membre créé 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.

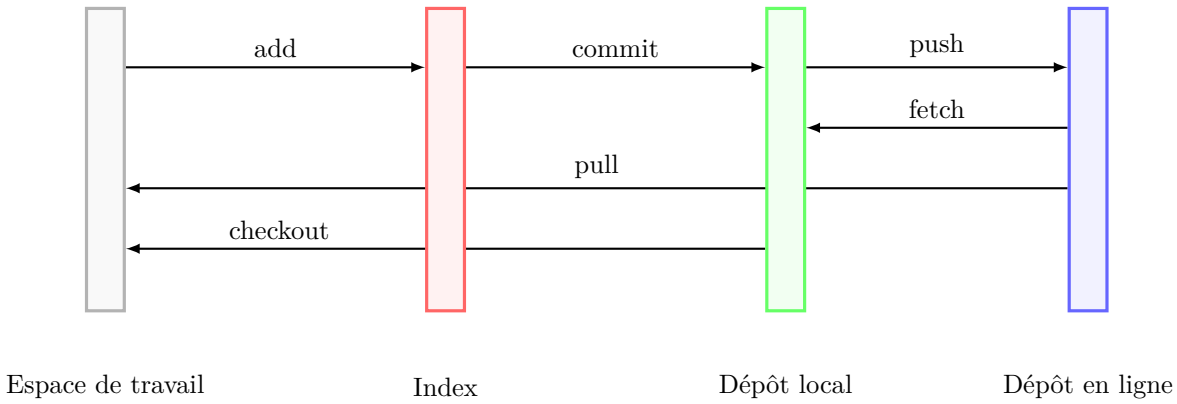

FIGURE 2.2 – Résumé des commandes Git de base

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 :

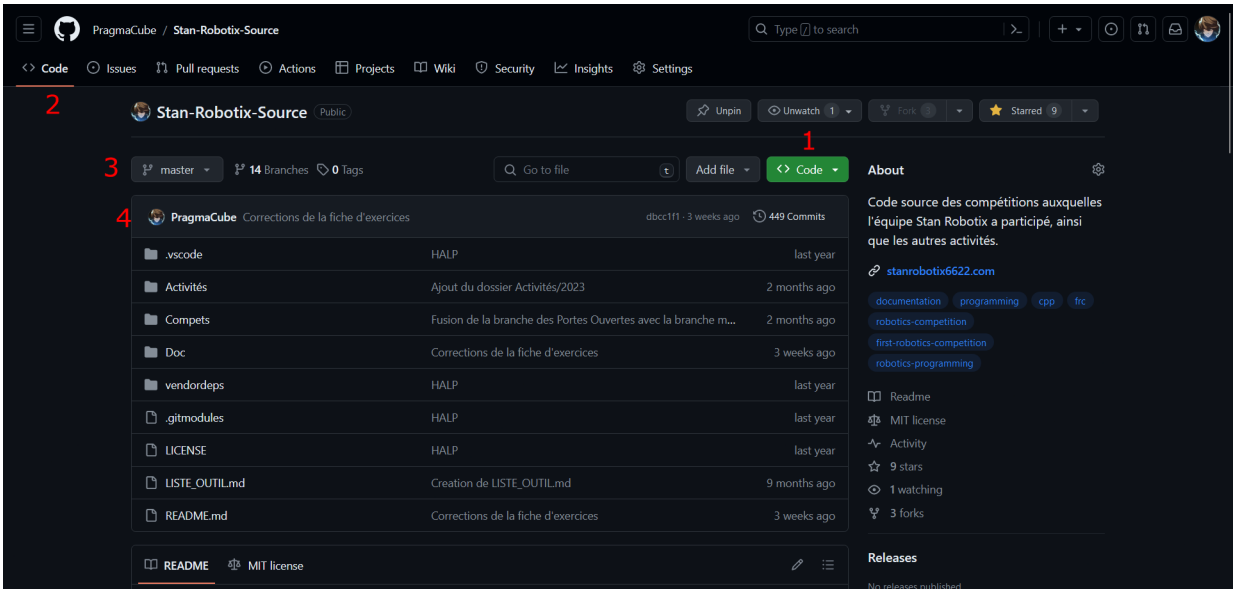

FIGURE 2.3 – Interface d'un dépôt sur Github

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 et 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 indépendemment 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

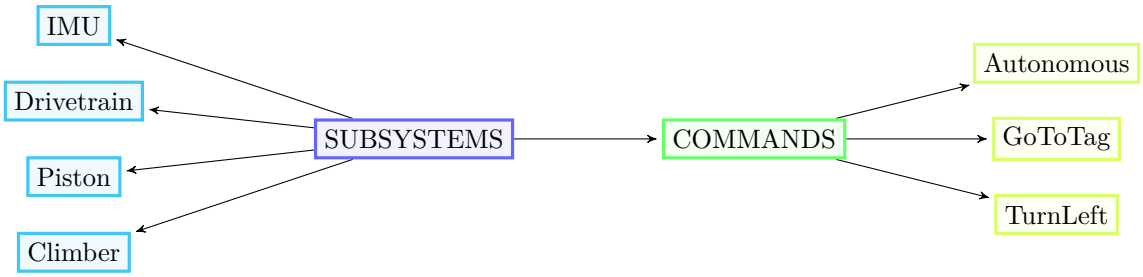

FIGURE 3.1 – Schéma du *Command-Based Programming*

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)é

```
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
```

```
#pragma once

namespace DriveTrainConstants
{
    constexpr int kMotorL1Id = 2;
    constexpr int kMotorL2Id = 4;
    constexpr int kMotorR1Id = 3;
    constexpr int kMotorR2Id = 1;
}
```

```
// SubDriveTrain.h

// Copyright (c) FIRST and other WPILib contributors.
// Open Source Software; you can modify and/or share it under the terms of
// the WPILib BSD license file in the root directory of this project.

#pragma once

#include <frc2/command/SubsystemBase.h>
#include <frc/drive/MecanumDrive.h>
#include <ctre/phoenix/motorcontrol/can/WPI_VictorSPX.h>
#include <ctre/phoenix/motorcontrol/can/WPI_TalonSRX.h>

#include "Constants.h"

class SubDriveTrain : public frc2::SubsystemBase
{
public:
    SubDriveTrain();

    void Periodic() override;

    void mecanumDrive(float iX, float iY, float iZ, frc::Rotation2d iRotation2d);

private:
    ctre::phoenix::motorcontrol::can::WPI_VictorSPX* mMotorL1{};
    ctre::phoenix::motorcontrol::can::WPI_VictorSPX* mMotorL2{};
    ctre::phoenix::motorcontrol::can::WPI_VictorSPX* mMotorR1{};
    ctre::phoenix::motorcontrol::can::WPI_VictorSPX* mMotorR2{};

    frc::MecanumDrive* mDrive;
};
```

```
// SubDriveTrain.cpp

// Copyright (c) FIRST and other WPILib contributors.
// Open Source Software; you can modify and/or share it under the terms of
// the WPILib BSD license file in the root directory of this project.

#include "subsystems/SubDriveTrain.h"

SubDriveTrain::SubDriveTrain()
{
    mMotorL1 = new
```

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

    mDrive = new frc::MecanumDrive{*mMotorL1, *mMotorL2, *mMotorR1, *mMotorR2};

    mMotorL1->SetInverted(true);
    mMotorL2->SetInverted(true);
}

// This method will be called once per scheduler run
void SubDriveTrain::Periodic() {}

void SubDriveTrain::mecanumDrive(const float iX, const float iY, const float iZ, const
    frc::Rotation2d iRotation2d)
{
    mDrive->DriveCartesian(iY, -iX, -iZ, -iRotation2d);
}
```

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

```
// SubIMU.h

// Copyright (c) FIRST and other WPILib contributors.
// Open Source Software; you can modify and/or share it under the terms of
// the WPILib BSD license file in the root directory of this project.

#pragma once

#include <frc2/command/SubsystemBase.h>

#include <ctre/Phoenix.h>
#include <ctre/phoenix6/Pigeon2.hpp>
#include <units/acceleration.h>

class SubIMU : public frc2::SubsystemBase
{
public:
    SubIMU();

    void Periodic() override;

    units::standard_gravity_t getAccelX();
    units::standard_gravity_t getAccelY();
    units::standard_gravity_t getAccel();

```

```
    frc::Rotation2d getRotation2d();

    void ResetAngle();

private:
    ctre::phoenix6::hardware::Pigeon2* mIMU{0};
};
```

```
// SubIMU.cpp

// Copyright (c) FIRST and other WPILib contributors.
// Open Source Software; you can modify and/or share it under the terms of
// the WPILib BSD license file in the root directory of this project.

#include "subsystems/SubIMU.h"
#include <cmath>

SubIMU::SubIMU()
{
    mIMU = new ctre::phoenix6::hardware::Pigeon2{0};
}

// This method will be called once per scheduler run
void SubIMU::Periodic() {}

void SubIMU::ResetAngle()
{
    mIMU->Reset();
}

units::standard_gravity_t SubIMU::getAccelX()
{
    return mIMU->GetAccelerationX().GetValue();
}

units::standard_gravity_t SubIMU::getAccelY()
{
    return mIMU->GetAccelerationY().GetValue();
}

units::standard_gravity_t SubIMU::getAccel()
{
    return units::standard_gravity_t(std::sqrt(GetAccelerationX().GetValueAsDouble() *
        GetAccelerationX().GetValueAsDouble() + GetAccelerationY().GetValueAsDouble() *
        GetAccelerationY().GetValueAsDouble()));
}

frc::Rotation2d SubIMU::getRotation2d()
{
    return mIMU->GetRotation2d();
}
```

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, mais 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* existent, 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...).

— ***AddRequirements(Subsystem* 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. Il faut **impérativement** utiliser cette fonction pour éviter des conflits entre commandes utilisant le même *Subsystem*.

Exemples :

1. On souhaite créer une *Command* permettant faire tourner le robot de 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.

```cpp
// TurnRight.h

#pragma once

#include <frc2/command/Command.h>
#include <frc2/command/CommandHelper.h>

#include "subsystems/SubDriveTrain.h"
#include "subsystems/SubIMU.h"

#include "Constants.h"

class TurnRight
    : public frc2::CommandHelper<frc2::Command, TurnRight> {
public:
    TurnRight(SubDriveTrain* iDriveTrain, SubIMU* iIMU);

    void Initialize() override;

    void Execute() override;

    void End(bool interrupted) override;

    bool IsFinished() override;

private:
    SubDriveTrain* mDriveTrain;
    SubIMU* mIMU;

    double mStartingAngle;
};
```

```cpp
// TurnRight.cpp

#include "commands/TurnRight.h"

TurnRight::TurnRight(SubDriveTrain *iDriveTrain, SubIMU *iIMU)
{
    mDriveTrain = iDriveTrain;
    mIMU = iIMU;
    AddRequirements(mDriveTrain);
```

```
    AddRequirements(mIMU);
}

void TurnRight::Initialize()
{
    mStartingAngle = mIMU->getAngleYaw();
}

void TurnRight::Execute()
{
    mDriveTrain->mecanumDrive(0, 0, 0.8, mIMU->getRotation2d());
}

void TurnRight::End(bool interrupted)
{

}

bool TurnRight::IsFinished()
{
    return (mIMU->getAngleYaw() - mStartingAngle) > 72;
}
```

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 controleurs *CANSparkMax*.
   — Le *Subsystem* en question possède des méthodes *setTargetPosition()*, *getTargetPosition()* et *moveToPosition()* pour contrôler la position du pivot. Des méthodes *getEncoderPosition-MotorL()* et *getEncoderPositionMotorR()* sont là pour déterminer la position actuelle.
   — Un *PID* lié aux contrôleurs *CANSparkMax* est utilisé pour atteindre la position de manière optimale.

```
// Constants.h

#pragma once

namespace ElevatorConstants {

    constexpr int kMotorId1 = 1;
    constexpr int kMotorId2 = 3;

    constexpr double kElevatorLimitUp = 82 - 83;
    constexpr double kElevatorLimitMiddle = 55 - 83;
    constexpr double kElevatorLimitDown = 0 - 83 ;

    constexpr double kP = 0.00009999999873689376,
                kI = 19.999999974752427e-7,
                kD = 0,
                kIz = 0,
                kFF = 0.000155999994603917,
                kMaxOutput = 1,
                kMinOutput = -1;
}
```

```h
// ElevatorUp.h

// Copyright (c) FIRST and other WPILib contributors.
// Open Source Software; you can modify and/or share it under the terms of
// the WPILib BSD license file in the root directory of this project.

#pragma once

#include <frc2/command/Command.h>
#include <frc2/command/CommandHelper.h>

#include "subsystems/SubElevator.h"

class ElevatorUp
    : public frc2::CommandHelper<frc2::Command, ElevatorUp> {
public:
    ElevatorUp(SubElevator* iElevator);

    void Initialize() override;

    void Execute() override;

    void End(bool interrupted) override;

    bool IsFinished() override;

private:
  SubElevator* mElevator;
};
```

```cpp
// ElevatorUp.cpp

// Copyright (c) FIRST and other WPILib contributors.
// Open Source Software; you can modify and/or share it under the terms of
// the WPILib BSD license file in the root directory of this project.

#include <cmath>

#include "commands/ElevatorUp.h"

ElevatorUp::ElevatorUp(SubElevator* iElevator) {

  mElevator = iElevator;
  AddRequirements(mElevator);
}

// Called when the command is initially scheduled.
void ElevatorUp::Initialize()
{
  mElevator->setTargetPosition(ElevatorConstants::kElevatorLimitUp);
}

// Called repeatedly when this Command is scheduled to run
void ElevatorUp::Execute()
{
```

```
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() -
          mElevator->getTargetPosition()) < 0.05;
38  }
```

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.

```
#include <frc2/command/Commands.h>

#include "commands/Forward.h"
#include "commands/ElevatorDown.h"
#include "commands/PickBall.h"
...
.
.
.
...
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 évènement 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* :

```
// Path.h

// Copyright (c) FIRST and other WPILib contributors.
// Open Source Software; you can modify and/or share it under the terms of
// the WPILib BSD license file in the root directory of this project.

#pragma once

#include <frc2/command/CommandHelper.h>
#include <frc2/command/SequentialCommandGroup.h>

#include "commands/Forward.h"
#include "commands/TurnLeft.h"

class Path : public frc2::CommandHelper<frc2::SequentialCommandGroup, Path>
{
public:
    Path(SubDriveTrain* iDriveTrain, SubIMU* iIMU);
};
```

```
// Path.cpp

// Copyright (c) FIRST and other WPILib contributors.
// Open Source Software; you can modify and/or share it under the terms of
// the WPILib BSD license file in the root directory of this project.

```

```
#include "commands/Path.h"

Path::Path(SubDriveTrain* iDriveTrain, SubIMU* iIMU)
{
    AddCommands(Forward(iDriveTrain, 1), TurnLeft(iDriveTrain, iIMU),
        Forward(iDriveTrain, 4));
}
```

2. *Commands* en parallèle :

```
// Preparation.h

// Copyright (c) FIRST and other WPILib contributors.
// Open Source Software; you can modify and/or share it under the terms of
// the WPILib BSD license file in the root directory of this project.

#pragma once

#include <frc2/command/CommandHelper.h>
#include <frc2/command/ParallelCommandGroup.h>

#include "commands/Forward.h"
#include "commands/ElevatorDown.h"
#include "commands/PickBall.h"

class Preparation : public frc2::CommandHelper<frc2::ParallelCommandGroup, Preparation>
{
public:
    Preparation(SubDriveTrain* iDriveTrain, SubElevator* iElevator, SubPicker* iPicker);
};
```

```
// Preparation.cpp

// Copyright (c) FIRST and other WPILib contributors.
// Open Source Software; you can modify and/or share it under the terms of
// the WPILib BSD license file in the root directory of this project.

#include "commands/Preparation.h"

Preparation::Preparation(SubDriveTrain* iDriveTrain, SubElevator* iElevator, SubPicker*
    iPicker)
{
    AddCommands(Forward(iDriveTrain, 2), ElevatorDown(iElevator), PickBall(iPicker));
}
```

Remarque : Pour utiliser des *CommandGroups* avec des évènements 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 évènements d'une manette ou d'un *Joystick*. Les méthodes importantes à retenir sont :

— ***ConfigureBindings***() : Méthode permettant de lier différents boutons à des *Commands*.

— ***GetAutonomousCommand***() : Méthode renvoyant la *CommandPtr* de la période autonome. Il faut s'assurer que la bonne *CommandPtr* est renvoyée.

3.5.1 Fonctions lambdas

La notion suivante n'a pas été présentée avant car son utilité se limite, dans notre contexte, aux deux sujets qui suivent (les contrôleurs et la DefaultCommand d'un Subsystem).

Définitions :

— Un **évènement** est une condition dont la valeur est testée constamment par le programme de manière **asynchrone** (en parallèle).

— Une **fonction lambda-évènement** [1] est une fonction sans nom spécifié, dont l'utilité est restreinte à donner la valeur de vérité de l'évènement. Cette fonction n'a pas de paramètre.

— Un objet ***Trigger*** (**déclencheur**) est un objet liant la fonction lamda-évènement à la *Command* à exécuter.

Exemple :

On souhaite lier la *Command ElevatorUp* à l'évènement "le bouton 4 du *Joystick* est pressé". Dans la méthode *ConfigureBindings*() on ajoute la ligne suivante :

```
1 frc2::Trigger([this] {
2    return mJoystick.GetRawButtonPressed(4);
3  }).OnTrue(ElevatorUp(&mElevator).ToPtr());
```

Le mot-clé ***this***, renvoyant un pointeur vers l'objet courant (ici l'objet de type *RobotContainer*), présent dans les crochets sert à indiquer que la fonction lambda-évènement a besoin d'avoir accès aux membres (ici *mJoystick*) de l'objet courant. Notez que puisque la fonction n'a pas de paramètre, il n'y a pas de parenthèses.

Définitions :

— Un objet ***RunCommand*** est un objet renvoyant une *Command* sans condition d'arrêt. Dans la plupart des cas, il ne s'agit que d'une instruction à exécuter (ex : arrêter un moteur).

— La ***Default Command*** **(commande par défaut)** d'un *Subsystem* est une méthode exécutant en tout temps une *RunCommand* lorsque celui-ci n'est pas utilisé.

— Une **fonction lambda-commande** [2] est une fonction sans nom spécifié, dont l'utilité est restreinte à regrouper la ou les instructions d'un objet *RunCommand*.

Exemple :

On souhaite pouvoir bouger un robot en mode *MecanumDrive* tant que celui-ci n'exécute pas de *Command* qui le fait bouger. Le robot possède un *IMU* de type *Pigeon2*. Dans le constructeur de *RobotContainer*, on ajoute la ligne suivante :

```
1 mDrivetrain->SetDefaultCommand(frc2::RunCommand(
2     [this]
3     {
4       mDrivetrain->mecanumDrive(joystick.GetX(), joystick.GetY(), joystick.GetZ(),
          mIMU->getRotation2d());
```

1. Terme non officiel. On parle de fonction lambda à valeurs de retour booléennes, donc modélisant une condition à vérifier.
2. Terme non officiel. On parle de fonction lambda sans valeur de retour, donc modélisant une suite d'instructions à exécuter périodiquement.

```
5       },
6       {mDrivetrain}));
```

L'objet *mDrivetrain* présent dans les accolades sert à préciser qu'il s'agit d'un *Subsystem* utilisé par la *Command* (comme on ferait avec la méthode *AddRequirement*()).

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 ***TeleopInit***() (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 importants 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 occurences 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 selectionner 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 déploitment

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