À la découverte du C++

PRUVOST Benjamin [1] REPORT Tanguy [2]

1. benjamin.pruvost@etu.univ-littoral.fr
2. Tanguy.Report@etu.univ-littoral.fr

Table des matières

Table des figures

Liste des tableaux

Chapitre 1

Introduction

1.1 L'histoire du C++

FIGURE 1.1 – Bjarne Stroustrup (2013)
()

Le langage *C++* a été conçu au début des années 1980 par Bjarne Stroustrup (figure 1.1), un ingénieur d'origine danoise, alors qu'il travaillait au sein des laboratoires Bell Labs, un centre de recherche réputé pour ses innovations. Stroustrup souhaitait améliorer le langage C en y intégrant des fonctionnalités de programmation orientée objet, qui permettent de structurer le code de manière plus modulaire et intuitive, notamment grâce à des concepts tels que les classes et les objets. L'objectif principal était de combiner la puissance, la flexibilité et l'efficacité du C avec une organisation et une maintenabilité accrues, afin de faciliter l'écriture et la gestion de programmes complexes. Aujourd'hui, le *C++* demeure l'un des langages de programmation les plus populaires et polyvalents, largement utilisé dans le développement de logiciels, de jeux vidéo, d'applications haute performance, ainsi que dans les systèmes embarqués et d'autres domaines technologiques avancés.

1.2 Différences entre C et C++

Le C et le *C++* sont des langages de programmation étroitement liés, mais ils diffèrent par leurs caractéristiques et leurs usages. Le C est un langage procédural qui se concentre sur les fonctions et les structures de données, idéal pour les systèmes bas-niveau comme les systèmes d'exploitation. En revanche, le *C++* étend le C en introduisant la programmation orientée objet, avec des concepts comme les classes, l'héritage et le polymorphisme, ce qui le rend plus adapté à des projets complexes nécessitant une organisation modulaire. De plus, le *C++* offre des fonctionnalités modernes comme les templates, les exceptions et la gestion automatique des ressources, absentes en C. Malgré leurs différences, *le* C++ *reste compatible avec le C*, permettant d'utiliser du code C dans des projets *C++* .

1.3 Installation de l'environnement de développement

Afin d'écrire, compiler et exécuter du code *C++* sur notre machine il est nécessaire d'installer deux outils essentiels : un éditeur de texte et un compilateur. Nous ne verrons pas l'installation de l'éditeur de texte dans ce cours il vous suffit de vous rendre sur le site de votre éditeur de texte de préférence, de télécharger le wizard d'installation correspondant à votre système et de suivre les indications pour pouvoir l'utiliser sur votre machine. Ces éditeurs incluent (mais ne se limitent pas à) :

— Visual Studio Code
— Notepad++
— Sublime Text

Concentrons nous donc sur l'installation du compilateur. Pour les utilisateurs de Mac, MacOs possède déjà un compilateur intégré, nous ne nous soucierons donc que des installations sur Windows et Linux.

1.3.1 Windows

1. Tout d'abord, rendez vous sur le site de MSYS2 et téléchargez l'installateur.
2. Lancez l'installateur et suivez les étapes.
3. Dans l'installateur, choisissez votre dossier d'installation, gardez le en mémoire pour plus tard. Dans la plus part des cas le dossier de base est acceptable.
4. Une fois l'installation terminée assurez vous que le choix *Run MSYS2 now* est coché puis sélectionnez *Finish*. Cela vous ouvrira un terminal MSYS2.
5. Dans ce terminal, installez les outils de compilation de MinGW en lançant la commande suivante :

```
pacman -S --needed base-devel mingw-w64-ucrt-x86_64-toolchain
```

6. Acceptez l'installation des packets en pressant la touche "Entrée".
7. Entrez 'Y' quand le programme vous demande si vous voulez poursuivre l'installation.
8. Ajoutez le chemin du dossier *bin* de votre installation de MinGW au PATH.
   (a) Dans la barre de recherche Windows, cherchez *Modifier les variables d'environnement système*.
   (b) Dans vos variables d'utilisateur, sélectionnez la variable *Path* et cliquez sur *Modifier*.

(c) Sélectionnez *Nouveau* et ajoutez le chemin du dossier bin. Si vous n'avez pas changé l'adresse de celui proposé lors de l'installation le chemin devrait être le suivant : *C :\msys64\ucrt64\bin*.

(d) Sélectionnez *OK* deux fois.

9. Pour vérifier votre installation ouvrez un nouvel invite de commande puis tapez la commande suivante :

```
g++ --version
```

Si vous n'obtenez pas d'erreur alors félicitations ! Vous pouvez désormais compiler du code *C++* !

1.3.2 Linux

Dans un premier temps, mettez à jour votre installateur en exécutant la commande suivante :

```
sudo apt update
```

Ensuite, installez l'outil de compilation *g++* grâce à cette autre commande :

```
sudo apt install g++
```

1.3.3 MacOS

Il est probable que le compilateur *g++* soit déjà installé sur votre système afin d'en être sûr vous pouvez exécuter cette commande et vérifier qu'elle ne vous renvois pas d'erreur :

```
g++ --version
```

Dans le cas contraire vous pouvez l'installer grâce à la commande suivante :

```
xcode-select --install
```

Chapitre 2

Structure d'un programme

Il est important de respecter la structure du code en *C++* afin d'assurer sa lisibilité, sa maintenabilité et son efficacité. Une organisation claire et cohérente du programme facilite son développement et sa compréhension, tant pour le programmeur que pour les membres de l'équipe.

2.1 Les Directives de Préprocesseur

Un programme en *C++* commence généralement par des directives de préprocesseur, qui sont des instructions pré-traitées avant la phase de compilation. Elles permettent d'inclure des bibliothèques et de définir des constantes ou des macros. Ces directives commencent par *le symbole* #.Voici ci-dessous un exemple :

```
//Implementation de la bibliotheque iostream
#include <iostream>
```

2.1.1 Les implémentations de bibliothèques

Les bibliothèques en *C++* jouent un rôle essentiel dans le développement d'applications, car elles offrent des fonctions et des classes pré-définies qui simplifient grandement la gestion des tâches courantes. Une des bibliothèques les plus utilisées est iostream, qui permet la gestion des entrées et sorties, notamment pour la lecture et l'écriture sur la console. L'implémentation de iostream repose sur des flux d'entrée/sortie, tels que std : :cin pour l'entrée et std : :cout pour la sortie mais pas seulement [1].

```
//Implementation de la bibliotheque iostream
#include <iostream>

int main(){
    std::cout << "Hello␣World␣!";
    return 1;
}
```

Ce code permet d'afficher "Hello World" dans la console, ce qui est impossible sans la bibliothèque iostream

1. Documentation de iostream :https://cplusplus.com/reference/iostream/

Il y a bien évidemment plein d'autre bibliothèques utiles, comme par exemple `<cmaths>`,qui fournit un ensemble de fonctions mathématiques standards pour effectuer des calculs numériques.

Par exemple si nous voulons calculer en *C++* $\sqrt{\cos(\frac{1}{3})}$, nous avons maintenant accès aux fonctions `sqrt()` et `cos()` :

```
#include <iostream>
#include <cmath>

int main(){
    std::cout << sqrt(cos(1/3));
    return 0;
}
```

2.1.2 Les Espaces de Noms

Les espaces de noms (namespaces) permettent de regrouper des éléments tels que des fonctions, des classes et des variables pour éviter les conflits de noms. Le namespace standard (std) est couramment utilisé :

```
using namespace std;
```

Cette directive permet d'accéder directement aux fonctions et objets de la bibliothèque standard, comme cout ou cin.

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

int main(){
    cout << "Hello␣World␣!";
    return 1;
}
```

On remarquera maintenant qu'il n'est plus nécessaire d'écrire `std::` devant le cout, qu'il est fastidieux d'écrire quand on appelle ces fonctions un grand nombre de fois.

2.1.3 Les macros

Les macros en *C++* sont définies avec `#define` et permettent de remplacer des parties du code avant la compilation. Elles sont souvent utilisées pour définir des constantes ou des expressions réutilisables.Exemple : `#define PI 3.14159`

2.2 La fonction main()

La fonction main est le point d'entrée de tout programme en *C++* . C'est là que l'exécution commence.Il doit être placer à la fin du code, tout code n'étant pas contenu dedans n'est pas exécuté sauf exception [2]. Elle retourne généralement un entier pour indiquer l'état de fin du programme :

2. Les constructeurs de classes par exemple

```
int main(){
        //Corps du programme
        return 0;
}
```

La valeur de retour 0 signale une exécution réussie, tandis qu'une valeur non nulle indique une erreur.

La fonction main peut également accepter des arguments pour traiter les entrées de la ligne de commande :

```
int main(int argc, char* argv[]) {
        // argc : Nombre d'arguments
        // argv : Tableau contenant les arguments
}
```

Et dont voici un exemple très simple d'utilisation, il faut d'abord taper la ligne de commande ci dessous en remplacer par les noms correspondant à votre fichier et vos arguments :

```
./monProgramme arg1 arg2
```

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

int main(int argc, char* argv[]) {
    //affichage du nombre d'arguments
    cout << "Nombre d'arguments : " << argc << endl;

    // Affichage des arguments
    for (int i = 0; i < argc; ++i) {
        cout << "Argument " << i << " : " << argv[i] << endl;
    }

    return 0;
}
```

2.3 Fonctions et commentaires

Puis finalement pour bien organiser le code, celui-ci va être découper en fonction(détaillées au chapitre ?? p.??)

Mais il faut aussi y insérer des commentaires, notamment pour expliquer les buts des fonctions, mais également pour expliquer des bouts de code qui pourrait être compliqué à comprendre.Cela est nécessaire pour travailler en équipe sur des fichiers de code communs.

```
// Ceci est un commentaire sur une ligne

/*
  Voici un commentaire
  sur plusieurs lignes differentes !
*/
```

Voici finalement à quoi devrait ressembler votre code une fois bien structuré.

```
#include <iostream>
//Ici d'autre inclusion de bibliotheque si necessaire
using namespace std;

void fonction1(){
    //explication fonction1
}

void fonction2(){
    //explication fonction2
}
int main(){
    fonction1();
    fonction2();
    return 0;
}
```

Chapitre 3

Types de données et variables

3.1 Les types primitifs

Les *types de données* constituent une base essentielle pour la programmation, permettant de manipuler différentes formes d'informations. Ces types se divisent principalement en types dit *primitifs*, qui gèrent les données simples, et en types complexes (comme les tableaux, les structures et les pointeurs) que nous verrons plus tard et qui offrent une plus grande flexibilité pour représenter des données structurées. Pour l'instant, concentrons nous sur les quatre types primitifs de base :

- `int`, ou *'integer'* (*entier* en français). Qui permet de stocker des nombres entiers, donc sans virgule.
- `float` ou `double`, un nombre à virgule *flottante*.
- `char`, un caractère unique qui peut être une lettre, un nombre ou un caractère spécial.
- `bool`, soit *boolean* qui ne peut contenir que deux valeurs : *vrai* (*true*) ou faux (*false*).

Ces types sont encodés sur un *nombre de bit* différent, et ne peuvent donc pas contenir des valeurs de même taille, à titre d'information l'encodage ainsi que la portée maximale de chacun de ces types est donné dans le tableau ci-dessous :

Type	Encodage	Portée
`int`	32 bits	$-2\,147\,483\,648$ à $2\,147\,483\,648$
`float`	32 bits	$-3{,}4 \times 10^{38}$ à $3{,}4 \times 10^{38}$
`double`	8 bits	$-1{,}7 \times 10^{300}$ à $1{,}7 \times 10^{300}$
`char`	8 bits	0 à 255
`bool`	1 bit	0 (faux) ou 1 (vrai)

TABLE 3.1 – Encodage et portée des types primitifs

3.2 Les variables

Les variables sont des conteneurs permettant de stocker et manipuler des données en mémoire. Chaque variable est définie par un type, qui détermine la nature des données qu'elle peut contenir, et un nom, utilisé pour y accéder. Les variables peuvent être locales, globales ou statiques. Il est aisé de définir une variable, il suffit simplement de spécifier son *type* puis son *nom* l'un à la suite de l'autre.

```
int age;
float moyenne;
char lettre;
bool majeur;
```

Cette étape s'appelle la *déclaration* de la variable, elle consiste à faire comprendre à notre ordinateur que nous souhaitons réserver une place ne mémoire pour une donnée d'un certain type et nous lui attribuons un nom afin de pouvoir l'utiliser dans les lignes suivantes. Ainsi, après la déclaration nous pouvons *initialiser* notre variable, soit lui donner une valeur, comme suit :

```
age = 18;
moyenne = 12.3;
lettre = 'a';
majeur = true;
```

Notez qu'il est tout à fait possible (et recommandé) de combiner le processus de déclaration et d'initialisation.

```
int age = 18;
float moyenne = 12.3;
char lettre = 'a';
bool majeur = true;
```

L'endroit du code où une variable est définie importe tout autant que son type ou son nom, voire plus, il existe deux cas distincts.

Local Une variable déclarée *localement* est déclarée dans une fonction ou tout autre bloc de code inclus dans un autre (une fonction par exemple), ainsi, cette variable ne sera accessible que dans le bloc où elle à été déclarée et ses enfants.

Exemple Variable déclarée localement

```
#include<iostream>

void ma_fonction(){
    int a = 1;
}

int main(){

    int a = 2;
    std::cout << a;     // Affichera 2.

    return 1;
}
```

()

Global Une variable déclarée *globalement* est déclarée directement dans le programme, en dehors de tout bloc de code. Ainsi, elle est accessible partout dans le code.

Exemple Variable déclarée globalement

```
#include<iostream>

int a = 1;

void ma_fonction(){
    std::cout << a; // Je connais 'a'
}

void mon_autre_fonction(){
    std:: cout << a; // Moi aussi !
}

int main(){

    std::cout << a; // Et moi aussi !

    return 1;
}
```

3.3 Préfixes particuliers

Les préfixes comme *long*, *short*, *signed*, et *unsigned* permettent de modifier les types de données de base pour ajuster leur taille ou leur plage de valeurs. Par exemple, short et long réduisent ou augmentent respectivement la taille des entiers (int), tandis que unsigned supprime les valeurs négatives pour doubler la plage des nombres positifs. Le mot-clé *const* est utilisé pour déclarer des variables dont la valeur est immuable, garantissant qu'elle ne pourra pas être modifiée après initialisation. Ces préfixes offrent une flexibilité accrue dans la gestion de la mémoire et l'adaptation aux besoins spécifiques du programme.

3.4 Opérateurs simples

Les opérateurs arithmétiques en *C++* incluent des symboles comme '+', '-', '*', '/' et '%', utilisés pour effectuer des opérations mathématiques de base. L'opérateur '+' sert à l'addition, '-' à la soustraction, '*' à la multiplication, '/' à la division et '%' à obtenir le reste d'une division entière. *C++* propose aussi des opérateurs d'assignation comme '=', qui permet d'affecter une valeur à une variable, et des opérateurs d'incrémentation ('++') et de décrémentation ('–') qui augmentent ou diminuent la valeur d'une variable de 1. Ces opérateurs sont essentiels pour les calculs et les manipulations de données dans les programmes.

Exemple Addition, incrémentation, multiplication

```
#include<iostream>

int a = 1;
int b = 2;

int main(){
```

```
    int c = a + b;       // c = 3

    b = b - a;           // b = 1

    a++;                 // a = 2

    c = a*2;             // c = 4


    return 1;
}
```

3.5 Conversion entre types

La conversion entre types permet de changer la représentation d'une variable d'un type à un autre. Par exemple, si vous voulez convertir un `char` en `int`, vous pouvez simplement faire une conversion explicite comme `(int)(variable)` où variable est un caractère. Cette conversion transforme le caractère en son code ASCII correspondant, comme `'A'` qui devient `65`. À l'inverse, pour convertir un `int` en `char`, on utilise `(char)(variable)` où variable est un entier. Cela prend l'entier et le convertit en caractère correspondant à son code ASCII. Il faut cependant être vigilant avec ces conversions, car si la valeur est hors des limites du type cible, cela peut entraîner des résultats inattendus ou une perte de données.

Chapitre 4

Instructions de contrôle

Les instructions de contrôle sont fondamentales en programmation. Elles permettent de diriger le flux d'exécution d'un programme en fonction de conditions ou de boucles. En *C++* , ces instructions incluent les comparateurs, les structures conditionnelles, les boucles, ainsi que certaines commandes spéciales.

4.1 Les expressions booléennes et les comparateurs

Les expressions booléennes évaluent une condition pour produire un résultat soit `true`, soit `false`. Ces expressions sont essentielles pour déterminer quel chemin emprunter dans un programme. Les comparateurs les plus couramment utilisés sont :

- `==` : Vérifie si deux valeurs sont égales.
- `!=` : Vérifie si deux valeurs sont différentes.
- `<` : Vérifie si une valeur est strictement inférieure à une autre
- `>` : Vérifie si une valeur est strictement supérieure à une autre.
- `<=` : Vérifie si une valeur est inférieure ou égale à une autre
- `>=` : Vérifie si une valeur est supérieure ou égale à une autre.

Exemple Comparaison simple

```
int x = 10, y = 20;
//ex1 sera egale a true car la comparaison est vraie
bool ex1 = x < y ;
//ex2 sera egale a false car la comparaison est fausse
bool ex2 = x == y;
```

Ces comparateurs peuvent également être combinés avec des opérateurs logiques tels que `&&` (et logique), `||` (ou logique), et `!` (non logique)

Exemple Opérateurs logiques

```
int x = 10, y = 20;
bool ex3 = x < y && x == y ;
/*ex3 sera egale a false avec le et logique
il faut que les deux comparaisons soient vraient*/
bool ex4 = x < y && x == y ;
/*ex4 sera egale a true car le ou logique
```

4.2 Les structures conditionnelles

Les structures conditionnelles permettent de prendre des décisions en fonction des résultats d'une ou plusieurs expressions booléennes.

— `if/else` : La structure la plus basique et flexible pour contrôler le flux d'exécution.
Exemple If/else simple

```
int age = 18;
if (age >= 18) {
    cout << "Vous␣etes␣majeur" << endl;
} else {
    cout << "Vous␣etes␣mineur" << endl;
}
```

— `else if` : Permet de vérifier plusieurs conditions dans une structure imbriquée.
Exemple Else if

```
int note = 85;
if (note >= 90) {
    cout << "Excellent␣!" << endl;
} else if (note >= 75) {
    cout << "Tres␣bien␣!" << endl;
} else {
    cout << "Besoin␣d'amelioration." << endl;
```

— `switch` : Utile pour comparer une seule expression à plusieurs valeurs possibles
Exemple Switch

```
char grade = 'B';
switch (grade) {
    case 'A':
        cout << "Excellent" << endl;
        break;
    case 'B':
        cout << "Tres␣bien" << endl;
        break;
    case 'C':
        cout << "Bien" << endl;
        break;
    default:
        cout << "Non␣valide" << endl;
}
```

4.3 Les boucles

Les boucles permettent de répéter un bloc de code plusieurs fois. Elles sont très utiles pour traiter des tableaux, des collections, ou pour des calculs répétitifs.

— Boucle `for` : Idéale pour un nombre défini d'itérations.
Exemple For

```
int compteur = 0;
for (int i = 0; i < 100; i++) {
    compteur += 1;
}
```

Pour bien comprendre l'incrémentation à chaque tour de i ,voici la représentation de la variable compteur en fonction de i :

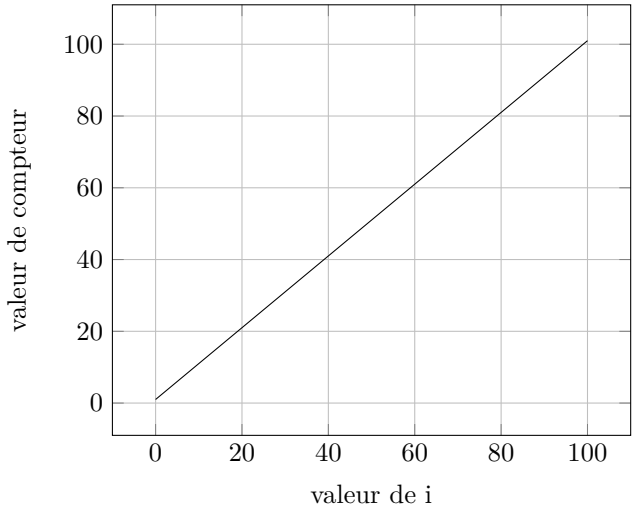

FIGURE 4.1 – Courbe de i en fonction de compteur

— Boucle `while` : Idéale pour un nombre défini d'itérations.

Exemple While

```
int compteur = 5;
while (compteur > 0) {
    cout << "Compteur : " << compteur << endl;
    compteur--;
}
```

— Boucle `do while` : Idéale pour un nombre défini d'itérations.

Exemple Do/While

```
int compteur = 1;
do {
    cout << "Valeur : " << compteur << endl;
    compteur++;
} while (compteur <= 5);

for (int i = 0; i < 10; i++) {
    if (i == 5) break; // Quitte la boucle si i vaut 5
    if (i % 2 == 0) continue; // Ignore les nombres pairs
    cout << "i = " << i << endl;
}
```

Chapitre 5

Fonctions

Jusque ici nous avons employé le terme de *fonction* sans réellement l'expliquer. Une fonction est un bloc de code réutilisable qui exécute une tâche spécifique. Elle est définie par un nom, un type de retour (indiquant ce qu'elle renvoie), et éventuellement des paramètres pour transmettre des données. Les fonctions permettent d'organiser et de structurer le code en le divisant en morceaux plus simples et modulaires, facilitant la compréhension, la maintenance et la réutilisation dans un programme.

5.1 Déclaration

La déclaration d'une fonction est très similaire à celle d'une variables à quelques points près : premièrement, il est nécessaire d'ajouter une paire de parenthèses après le nom de cette dernière afin de faire comprendre à notre programme que nous souhaitons déclarer une fonction.

```
// La variable 'a'
int a;

// La fonction 'a'
int a();
```

Tout comme les variables il y a une différence entre la *déclaration* et l'*initialisation*, pour initialiser notre fonction il suffit d'ajouter une paire de crochets qui vont encadrer le *corps* de notre fonction. Enfin, on ajoute le mot clé `return` qui devra renvoyer une valeur [1] ou une variable du type définit avant le nom de la fonction.

```

    // Corps de la fonction

    return 1;
}
```

Mais comment utilise-ton nos fonctions ? Il suffit simplement de l'*appeler* grâce au nom qu'on lui a donné et d'ajouter une paire de parenthèse à la suite. Alors le corps de la fonction sera exécuté directement.

1. Dans le cas d'une déclaration avec `void` la fonction ne renvois rien, donc un `return` n'est pas nécessaire.

```
void bonjour(){
    std::cout << "Bonjour␣!";
}

int main(){

    bonjour();

    return 1;
}
```

5.2 Paramètres

Nous savons désormais comment initialiser des fonctions simples, mais une fonction sans argument n'est pas très utiles, on appelle ces arguments *paramètres*, ce sont eux qui donnent toute leur utilité aux fonctions. Afin de déclarer un paramètre pour notre fonction il suffit simplement de donner son type puis le nom que l'on voudra utiliser pour le référencer dans le corps.

```
int tres_utile(int mon_nombre){

    return mon_nombre;
}

int b = tres_utile(1);
```

Pour déclarer plusieurs paramètres il suffit de les séparer d'une virgule.

```
int calcul(int a, int b){
    return a+b;
}
```

Notons qu'il est tout à fait possible de donner une valeur de *base* aux arguments dans le cas où ceux-ci ne sont pas donné lors de l'appel de la fonction.

```
float calcul(float a = 1.0, float b = 1.0){
    return a*b;
}
```

5.3 Notion de "surcharge"

Contrairement aux variables, il est possible de déclarer plusieurs fonctions possédant le même nom tant que le type ou les paramètres sont différents. Cela peut s'avérer très utile car le programme saura automatiquement laquelle utiliser en fonction des paramètres donnés.

```
int main(){

    // Retournera 3
    calcul(1, 2);
```

```
    // Retournera 9
    calcul(1.5, 6.0f);

    return 1;
}
```

Chapitre 6

Pointeurs

Une des notions les plus importantes de la programmation en *C++* est très probablement celle de *pointeurs*.

6.1 Qu'est ce qu'un pointeur

Un pointeur est une variable qui contient l'*adresse mémoire* d'une autre variable. Au lieu de stocker directement une valeur, il pointe vers l'emplacement où cette valeur est située en mémoire. Les pointeurs sont utilisés pour manipuler directement la mémoire, partager des données entre différentes parties du programme, ou gérer des structures complexes comme les tableaux et les objets dynamiques.

6.1.1 Fonctionnement de la mémoire

Pour comprendre comment fonctionne un pointeur en *C++* il est premièrement nécessaire de comprendre comment fonctionne la mémoire lorsque l'on exécute un programme. Intéressons nous donc au programme suivant.

```
#include<iostream>

void change_valeur(bool a){
    a = true;
}

int main(){

    bool a = false;

    change_valeur(a);

    std::cout << a;

    return 1;
}
```

Maintenant essayons de prévoir comment le programme va se dérouler

— La valeur de 'a' est initialisée à `false`
— On passe 'a' dans la fonction `change_valeur()`
— La valeur de 'a' passe à `true`
— On retrouve `true` en sortie du programme.

Sauf que non ! La valeur de 'a' est restée la même et cela grâce/à cause de la façon dont la mémoire traite les relations entre les variables et les paramètres de fonctions. En effet, lorsque l'on utilise une variable comme paramètre, le programme n'utilise pas le même emplacement dans la mémoire pour ces deux valeurs, en somme il crée une *copie* de notre variable et c'est cette copie que l'on modifie dans notre fonction. Une fois la fonction exécutée, la copie est détruite et notre variable de base se trouve inchangée ! La question reste de savoir comment remédier à cela.

6.1.2 Syntaxe

Voyons comment les pointeurs peuvent nous aider à régler ce problème. Premièrement comment créer un pointeur ? Un pointeur est une variable comme une autre, la seule différence étant qu'elle content l'adresse d'une autre variable au lieu d'une valeur comme `1`, `true` ou `'a'` il faut donc que l'on récupère l'adresse de notre variable, pour se faire on utilise le caractère `&`. Ensuite, pour initialiser notre pointeur, nous devons préciser vers quelle type de valeur celui-ci va *pointer* puis ajouter un caractère `*` à la suite du type pour préciser qu'il s'agit bien d'un pointeur.

```
#include<iostream>

int main(){

    bool a = false;

    bool* p = &a;

    return 1;
}
```

Nom	Adresse	Valeur
a	0x0001	false
p	0x0002	0x0001
*p	0x0001	false

Table 6.1 – Différence entre pointeur et variable

Nous avons donc le pointeur `p` qui pointe vers l'adresse mémoire de `a`. Voyons comment mettre cela à profit pour régler notre problème.

6.2 Utilisation

Il est nécessaire d'apporter des modifications à notre fonction `change_valeur()`, car nous voulons maintenant accueillir en paramètre non pas 'a' mais *le pointeur vers 'a'*.

```
void change_valeur(bool* p){
```

Comme on peut le voir ça n'est pas différent des autres paramètres, on ajoute simplement une `*` pour signifier que l'on attend un pointeur. Maintenant, nous devons changer la valeur de

'a', sauf que notre pointeur 'p' ne contient pas la valeur mais l'adresse de 'a', il nous faut donc accéder à cette dernière grâce au caractère *.

```
#include<iostream>

void change_valeur(bool* p){
    *p = true;
}

int main(){

    bool a = false;

    bool* p = &a;

    change_valeur(p);

    std::cout << a;

    return 1;
}
```

Maintenant, c'est la valeur de 'a' qui sera changé. Il est aussi intéressant de noter qu'il est possible de faire des *pointeurs de pointeurs* en utilisant deux ** et non pas une, ce pointeur va donc pointer sur l'adresse du pointeur pointant sur notre variable (on s'y perd !).

Chapitre 7

Structures

Les structures en *C++* permettent de regrouper plusieurs variables, potentiellement de types différents, au sein d'un même type de données. Elles sont particulièrement utiles pour organiser des informations complexes tout en conservant une logique claire et lisible.

7.1 Définition d'une structure

Une structure est définie à l'aide du mot-clé `struct`, suivi de son nom et de son contenu, entouré d'accolades `{}`. Les membres d'une structure peuvent être des variables ou même d'autres structures.

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

// Definition d'une structure
struct Person {
    string name;  // Nom
    int age;      // Age
    float height; // Taille en metres
};
```

Ici, la structure `Person` regroupe trois attributs : un `string` pour le nom, un `int` pour l'âge, et un `float` pour la taille. Ces variables sont appelées membres de la structure.

7.2 Utilisation d'une structure

Une fois définie, une structure peut être utilisée pour créer des variables (instances). On peut accéder ou modifier les membres d'une instance à l'aide de l'opérateur `.`.

```
int main() {
    // Creation d'une instance de Person
    Person p1 = {"Alice", 25, 1.68};

    // Affichage des valeurs des membres
    cout << "Nom : " << p1.name << endl;
```

```
    cout << "Age␣:␣" << p1.age << endl;
    cout << "Taille␣:␣" << p1.height << "␣m" << endl;

    // Modification d'un membre
    p1.age += 1;
    cout << "Nouvel␣age␣:␣" << p1.age << endl;

    return 0;
}
```

7.3 Avantages des Structures

— **Organisation des données** :Les structures regroupent les données logiquement liées.
— **Lisibilité** : Elles permettent de donner un sens clair aux regroupements de données.
— **Réutilisation** : Une structure définie peut être utilisée plusieurs fois dans le programme.