Course: C++ language

Ali ZAINOUL

for AELION
December 6, 2023

Table of contents

1. UTILISER LES OUTILS DE DEVELOPPEMENT ASSOCIES AU LANGAGE C++
   - Historique C++
   - Spécification et corps de main
   - Les différents compilateurs
   - Environnements de développement
   - Affichage de valeurs et de chaînes avec cout
   - Lecture des valeurs avec cin
   - Formatage des sorties avec des manipulateurs de flots
   - Structure d'un Programme C++
   - Création d'un nouveau projet
   - Compilation et Exécution d'un Programme
2. MAITRISER LA SYNTAXE DU LANGAGE C++ (1/4)
   - Syntaxe basique
     - Syntaxe de base
     - Jetons en C++
     - Points-virgules ; en C++
     - Commentaires en C++ - Sur une seule ligne

- Commentaires en C++ - Sur plusieurs lignes
- Identificateurs en C++ - Règles de dénomination
- Mots-clés en C++
- Espaces blancs en C++

- Déclaration et initialisation de variables
- Types de données en C++ - Généralités
- Types de données entiers
- Types de données à virgule flottante
- Calcul arithmétique et affichage des résultats
- Mélange des types entiers et à virgule flottante dans les calculs et affectations
- Utilisation des références pour l'efficacité et des constantes pour la sécurité
- Constantes et littéraux
- Classes de stockage

3 MAITRISER LA SYNTAXE DU LANGAGE C++ (2/4)

- Passage des arguments aux fonctions et retour des valeurs depuis des fonctions
- Passage des arguments : par valeur ou par référence
- Visibilité, durée et valeur initiale des variables temporaires locales et des paramètres
- Portées

4 MAITRISER LA SYNTAXE DU LANGAGE C++ (3/4)
- Opérateurs
- Prise de décision

5 MAITRISER LA SYNTAXE DU LANGAGE C++ (4/4)
- Arrays
- Pointeurs
  - Notion de Pointeurs en C & C++
  - Notion de Smart Pointers en C++
- Déclaration et utilisation de tableaux pointeurs
- Stockage de chaînes dans des tableaux de caractères
  - Strings

Introduction au Langage C++

- Le langage de programmation C++ est une extension du langage de programmation C, développé par **Bjarne Stroustrup** en tant qu'amélioration du C avec des fonctionnalités supplémentaires pour la programmation orientée objet.
- Le C++ hérite de la popularité du C et est largement utilisé dans diverses industries et domaines.

Présentation du Langage C++

- Le C++ est un langage puissant et expressif.
- Il prend en charge les paradigmes de programmation procédurale et orientée objet.
- Offrant des performances élevées et une exécution efficace.
- Compatible avec plusieurs systèmes d'exploitation et plates-formes informatiques.
- **Un programme C++ ou un code source C++ est généralement écrit dans un ou plusieurs fichiers texte avec l'extension ".cpp" pour les fichiers source et ".hpp" pour les fichiers d'en-tête.**

Historique du Langage

- C++ a été développé en tant qu'extension du langage de programmation C, lui-même inventé pour écrire le système d'exploitation **UNIX**.
- C++ hérite de certaines fonctionnalités de son prédécesseur, le **langage C**, qui a été inventé dans les années 1970.
- Le langage C++ a été formalisé par l'**Organisation Internationale de Normalisation** (ISO) en 1998.
- Le **système d'exploitation UNIX a été initialement écrit en C en 1973**, et C++ est devenu largement utilisé dans divers domaines depuis lors.
- C++ est l'un des langages de programmation système les plus largement utilisés et **populaires**.
- De nombreux systèmes d'exploitation Linux et de nombreuses applications industrielles sont écrites en C++.

Utilisations de C++ (Partie 1)

C++ est un langage de programmation **polyvalent** qui est adapté à :

- **Programmation Système :** C++ est couramment utilisé pour des tâches de programmation système telles que le développement de systèmes d'exploitation, de pilotes de périphériques, de systèmes embarqués et de micrologiciels. Ses fonctionnalités de contrôle de bas niveau et de manipulation directe de la mémoire le rendent bien adapté à ces applications.
- **Développement de Jeux :** C++ est largement utilisé dans l'industrie du développement de jeux. Ses performances élevées, sa gestion efficace de la mémoire et son accès aux interfaces matérielles de bas niveau en font un choix optimal.

Utilisations de C++ (Partie 2)

- **Développement d'Applications :** C++ est utilisé pour construire une large gamme d'applications, y compris des applications de bureau, des simulations scientifiques, des systèmes financiers, des logiciels de CAO/FAO, et des applications critiques en termes de performances. Sa capacité à équilibrer performance et productivité permet aux développeurs de créer des applications robustes et efficaces.
- **Calcul Haute Performance (HPC) :** C++ est largement utilisé dans le domaine du calcul haute performance. Il permet aux développeurs de tirer parti du traitement parallèle, d'utiliser des architectures multi-cœurs, et d'optimiser le code pour des tâches intensives en calcul, telles que les simulations, l'analyse de données, et le calcul scientifique.

Utilisations de C++ (Partie 3)

- **Graphiques et Multimédia :** C++ est un choix populaire pour la programmation graphique, la vision par ordinateur, le traitement d'images et les applications multimédias. Des bibliothèques telles qu'OpenGL et DirectX offrent des liaisons en C++, permettant aux développeurs de créer des applications graphiques attrayantes et interactives.
- **Réseaux et Communications :** C++ est utilisé dans le développement de protocoles réseau, de programmation réseau et de systèmes de communication. Sa capacité à gérer efficacement la programmation de sockets de bas niveau, à implémenter des bibliothèques réseau et à construire des applications serveur le rend adapté aux tâches liées au réseau.

Utilisations de C++ (Partie 4)

- **Développement de Bibliothèques :** C++ est souvent utilisé pour créer des bibliothèques et des cadres pouvant être utilisés par d'autres développeurs dans leurs projets. De nombreuses bibliothèques et API largement utilisées, telles que Boost, Qt et la Standard Template Library (STL), sont écrites en C++.

Ce ne sont que quelques exemples parmi les nombreuses applications où C++ excelle en raison de ses performances, de sa flexibilité et de sa capacité à gérer des opérations de bas niveau. Sa large utilisation dans différentes industries et domaines témoigne de sa polyvalence en tant que langage de programmation.

Les différents compilateurs en C++

- Les compilateurs C++ jouent un rôle crucial dans le processus de développement.
- Les compilateurs traduisent le code source C++ en langage machine exécutable.
- Plusieurs compilateurs C++ sont largement utilisés dans l'industrie du développement logiciel.

Les différents compilateurs en C++ (suite)

- **GCC (GNU Compiler Collection):** Un compilateur open source populaire, disponible sur de nombreuses plateformes. Très utilisé sur Linux et Unix.
- **Clang:** Un autre compilateur open source, réputé pour sa rapidité et sa conformité aux normes. Souvent utilisé sur macOS et disponible sur d'autres plateformes.
- **Microsoft Visual C++:** Le compilateur fourni avec l'environnement de développement intégré (IDE) Visual Studio de Microsoft, principalement utilisé sur Windows.
- **Intel C++ Compiler:** Conçu pour les processeurs Intel, il offre des optimisations spécifiques à l'architecture. Disponible sur différentes platesformes, y compris Linux et Windows.

Configuration de l'environnement

Pour programmer en C++, vous avez besoin de ce qui suit :

- Soit :
  - (1) : **un éditeur de texte**
    - **Éditeur de texte :** Atom (consommant beaucoup de mémoire), Sublime Text, Visual Studio, gEdit, VIM, Notepad++.
  - (2) : un **compilateur C++** en utilisant le terminal en ligne de commande.
- Soit : un **Environnement de Développement Intégré** (IDE).

Configuration de l'environnement - Linux/Unix

- **Compilateur C++ :** Systèmes basés sur Linux/Unix (**recommandé**).
  - Ouvrez une fenêtre Shell/ Terminal et mettez à jour le système avec :

```
sudo apt update
```

  - Installez les outils nécessaires, y compris **GCC** :

```
sudo apt install build-essential
sudo apt-get install manpages-dev
```

  - Vérifiez votre version de GCC avec :

```
gcc --version
```

Configuration de l'environnement - MacOS

- **Compilateur C++ :** Systèmes basés sur MacOS.

```
# Open a Terminal window
# Set the URL for the Homebrew installation script
# Homebrew (package manager for macOS)
SCRIPTURL="https://raw.githubusercontent.com/Homebrew/install/
    HEAD/install.sh"

# Run the installation script using bash
/bin/bash -c "$(curl␣-fsSL␣$SCRIPTURL)"

# Install the GCC compiler using Homebrew
brew install gcc

# Display the installed GCC version
gcc --version
```

Configuration de l'environnement - Windows

- **Compilateur C++ :** Systèmes basés sur Windows.
  - Suivez les étapes sur ce site web :
    Installer GCC en utilisant la ligne de commande sur Windows

Configuration de l'environnement - IDE

Si vous avez choisi d'utiliser la deuxième option (c'est-à-dire installer un Environnement de Développement Intégré), vous pouvez installer l'un des suivants :

- **Visual Studio Code** (**recommandé**)
- Code::Blocks
- CodeLite
- NetBeans

Affichage de valeurs et de chaînes avec `cout`

Utilisez l'objet `cout` pour afficher des valeurs et des chaînes dans la console.

```cpp
#include <iostream>

int main() {
    int age = 35;
    std::string name = "Akuma";
    std::cout << "Hello, my name is " << name << " and I have " <<
        age << " YO." << "\n";
    // REMARK: we call the "::" operator of resolution;
       furthermore, string and cout are classes of the std (
       namespace) library.
    return 0;
}
```

Affichage de valeurs et de chaînes avec `cout`

Utilisez l'objet `cout` pour afficher des valeurs et des chaînes dans la console.

```cpp
#include <iostream>
#include <iomanip>
int main() {
    double price = 49.99;
    // Affichage avec un format monétaire
    std::cout << "The price is: $" << std::fixed << std::
        setprecision(2) << price << std::endl;
    // Affichage d'une chaîne formatée
    std::cout << "Reduction of 20%: $" << (price * 0.20) << std::
        endl;

    return 0;
}
```

Lecture des valeurs avec `cin`

Utilisez l'objet `cin` pour lire les valeurs depuis la console.

```cpp
#include <iostream>

int main() {
    int age;
    std::string name;
    std::cout << "Enter your name : ";
    std::cin >> name;
    std::cout << "Enter your age : ";
    std::cin >> age;
    std::cout << "Hello my name is: " << name << " and I have " <<
        age << " YO." << std::endl;

    return 0;
}
```

Lecture des valeurs avec `cin`

Utilisez l'objet `cin` pour lire les valeurs depuis la console.

```cpp
#include <iostream>

int main() {
    int number;
    // Lecture d'un entier depuis la console
    std::cout << "Enter an integer : ";
    std::cin >> number;
    std::cout << "The square of  " << number << " is : " << (
        number * number) << std::endl;

    return 0;
}
```

Formatage des sorties avec des manipulateurs de flots

Utilisez des manipulateurs de flots pour formater la sortie avec `cout`.

```cpp
#include <iostream>
#include <iomanip>

int main() {
    double rate = 0.08;
    // Affichage en pourcentage avec deux décimales
    std::cout << "Tax␣rate␣:␣" << std::fixed << std::setprecision
        (2) << std::setw(6) << std::right << rate * 100 << "%" <<
        std::endl;
    return 0;
}
```

Formatage des sorties avec des manipulateurs de flots

Utilisez des manipulateurs de flots pour formater la sortie avec `cout`.

```cpp
#include <iostream>
#include <iomanip>

int main() {
    double CUSTOM_PI = 3.14159265359;
    // Affichage de pi avec deux décimales
    std::cout << "The value of PI est : " << std::fixed << std::
        setprecision(2) << CUSTOM_PI << std::endl;
    // Affichage en notation scientifique
    std::cout << "In Scientific notation : " << std::scientific <<
         CUSTOM_PI << std::endl;
    return 0;
}
```

Structure d'un Programme C++ (Partie 1)

Un programme C++ se compose de divers composants, notamment :

- **Directives du Préprocesseur :** Les commandes du préprocesseur, commençant par un symbole dièse (#), sont utilisées pour inclure des fichiers d'en-tête, définir des constantes et effectuer des substitutions de macros avant que le code ne soit compilé.
- **Fonctions :** Les fonctions en C++ sont des blocs de code qui effectuent des tâches spécifiques. Elles peuvent être définies et appelées dans le programme pour obtenir un code modulaire et réutilisable.
- **Variables :** C++ permet la déclaration et l'utilisation de variables pour stocker et manipuler des données. Les variables doivent être déclarées avant leur utilisation et peuvent avoir différents types tels que des entiers, des nombres à virgule flottante, des caractères ou des types définis par l'utilisateur.

Structure d'un Programme C++ (Partie 2)

Un programme C++ comprend également les composants suivants :

- **Instructions :** Les programmes C++ se composent d'une série d'instructions qui définissent les actions à exécuter. Les instructions peuvent inclure des affectations, des conditionnelles, des boucles, des appels de fonctions, et plus encore.
- **Expressions :** Les expressions sont des combinaisons de variables, de constantes et d'opérateurs qui produisent une valeur lorsqu'elles sont évaluées. Elles peuvent être utilisées pour des calculs, des comparaisons et d'autres opérations.
- **Commentaires :** Les commentaires en C++ sont utilisés pour documenter et expliquer le code. Ils ne sont pas exécutés et sont ignorés par le compilateur.

Création d'un nouveau projet

- Pour créer un programme C++, ouvrez un éditeur de texte ou votre IDE préféré, créez un nouveau fichier et nommez-le "**helloCWorld.cpp**". Copiez et collez le code ci-dessous :

```cpp
// FILE NAME: helloCWorld.cpp
// This is a comment
/*
 This is a comment
 in many
 lines.
*/
#include <iostream>
using namespace std;

int main()
{
    // My first program in C++
    cout << "Hello, C++ World!" << endl;
    return 0;
}

// COMPILATION: g++ -o myHelloProgram HelloCWorld.cpp
// EXECUTION: ./myHelloProgram
```

Analyse du Code

Analysons le code ‘helloCWorld.cpp‘ ci-dessus :

- La première ligne est un commentaire, qui n’est pas considéré comme du code et sera ignoré par le compilateur.
- Le deuxième bloc représente un commentaire sur plusieurs lignes, tout comme dans le code précédent.
- La déclaration ‘#include <iostream>‘ est utilisée pour inclure la bibliothèque ‘iostream‘, qui fournit la fonctionnalité d’entrée/sortie en C++.
- En C++, nous utilisons l’instruction ‘cout‘ de la bibliothèque ‘iostream‘ pour afficher du texte. La ligne ‘cout « ”Hello, C++ World!” « endl;‘ affichera le texte ”Hello, C++ World!” dans la console. ‘endl‘ est utilisé pour insérer un caractère de nouvelle ligne après le texte.

Compilation et Exécution du Programme

- **Ouvrez un éditeur de texte** ou votre **IDE préféré** et **copiez-collez le code** `HelloCWorld.cpp` ci-dessus. Enregistrez le fichier dans un répertoire, supposons `myDirectory`, sous le nom de `HelloCWorld.cpp`.
- **Ouvrez une invite de commandes** ou un terminal et **naviguez vers le répertoire** où votre fichier est enregistré.
- Tapez `g++ -o myHelloProgram HelloCWorld.cpp` et **appuyez sur entrée** pour compiler votre code.
- **S'il n'y a pas d'erreurs dans votre code**, l'invite de commande passera à la ligne suivante, et un exécutable nommé `myHelloProgram` sera généré.
- Pour **exécuter le programme**, tapez `./myHelloProgram` et **appuyez sur entrée**.
- Vous verrez `Hello, C++ World!` imprimé à l'écran.

Compilation et Exécution du Programme

```
cd myDirectory
g++ -o myHelloProgram helloCWorld.cpp
./myHelloProgram
Hello, C++ World!
```

Syntaxe de base

- Cette section donne des détails sur la syntaxe de base du langage C++, y compris les jetons, les mots-clés, les identificateurs, etc.
- Nous avons vu une structure de base d'un programme C++ (`helloCWorld.cpp`), il sera donc facile de comprendre les éléments fondamentaux du langage C++.

Syntaxe de Base d'un Programme C++ (Partie 0)

En C++, un programme consiste en une collection d'instructions écrites dans une syntaxe spécifique. Voici quelques éléments clés de la syntaxe de base en C++ :

- **Variables** : Les variables sont utilisées pour stocker et manipuler des données en C++. Avant d'utiliser une variable, vous devez la déclarer en spécifiant son type et un nom unique. Par exemple, `int age;` déclare une variable entière appelée `age`.

Syntaxe de Base d'un Programme C++ (Partie 1)

- **Fonctions** : Un programme C++ commence par une fonction spéciale appelée fonction `main()`, qui sert de point d'entrée du programme. C'est là que l'exécution commence et se termine.
- **Instructions** : Les programmes C++ sont composés d'instructions qui effectuent des actions spécifiques. Chaque instruction se termine généralement par une point-virgule ; pour indiquer la fin de l'instruction.

Syntaxe de Base d'un Programme C++ (Partie 2)

- **Types de Données** : C++ fournit divers types de données pour représenter différents types de valeurs, tels que les entiers (`int`), les nombres à virgule flottante (`float`), les caractères (`char`), etc. Chaque type de données a une plage et un comportement spécifiques.

Syntaxe de Base d'un Programme C++ (Partie 3)

- **Commentaires** : Les commentaires en C++ servent à ajouter du texte explicatif ou descriptif dans le code. Ils sont ignorés par le compilateur et n'affectent pas l'exécution du programme. Les commentaires peuvent être sur une seule ligne (`//`) ou sur plusieurs lignes (`/* ... */`).
- **Structures de Contrôle** : C++ prend en charge des structures de contrôle telles que les instructions `if`, les boucles `for`, les boucles `while` et les instructions `switch` pour contrôler le flux d'exécution d'un programme en fonction de certaines conditions.

Comprendre ces composants de la syntaxe de base en C++ est important pour écrire un code bien structuré et facilement maintenable. Ils permettent d'ajouter de la clarté et de la flexibilité à vos programmes en incorporant des commentaires et en contrôlant le flux du programme.

Jetons en C++

- Un programme C++ est constitué de divers **jetons**.
- Un jeton peut être un **mot-clé**, un **identificateur**, une **constante**, une **chaîne littérale** ou un **symbole**.
- Les jetons sont les unités fondamentales du langage et sont utilisés pour construire des instructions et des expressions significatives en C++.

Jetons en C++

- Les jetons dans l'instruction C++ donnée sont :

```
1 cout
2 <<
3 "Hello, C++ World! \n"
4 <<
5 ;
```

Jetons en C++

- Les jetons ci-dessus se composent de :
  - Le mot-clé `cout`, qui est un objet C++ standard utilisé pour afficher des données.
  - L'opérateur <<, qui est l'opérateur d'insertion utilisé pour insérer des données dans le flux de sortie.
  - La chaîne littérale `Hello, C++ World! \n` qui représente le texte à imprimer.
  - Un autre opérateur << pour continuer l'insertion de données.
  - Le point-virgule `;` pour terminer l'instruction.

Points-virgules ; en C++

- En C++, un point-virgule ; est également un terminateur d'instruction. Chaque ligne de code doit se terminer par un point-virgule pour indiquer la fin d'une instruction.
- Considérons l'exemple suivant :

```
1                cout << "Hello, C++ World! \n" ;
2                return 0;
```

- Dans le code ci-dessus, nous avons deux instructions. La première instruction utilise l'objet `cout` pour afficher le texte `Hello, C++ World!` sur la console. La deuxième instruction est l'instruction `return`, qui renvoie la valeur 0 de la fonction `main`.

Commentaires en C++ - Sur une seule ligne

- Les commentaires sur une seule ligne en C++ commencent par // et s'étendent jusqu'à la fin de la ligne. Ils sont utilisés pour ajouter des informations explicatives ou descriptives au code.
- Exemple :

```
1 // Il s'agit d'un commentaire sur une seule ligne
```

Commentaires en C++ - Sur plusieurs lignes

- Les commentaires sur plusieurs lignes en C++ commencent par `/*` et se terminent par `*/`. Ils peuvent s'étendre sur plusieurs lignes et sont utilisés pour fournir des explications plus détaillées ou pour désactiver temporairement du code.
- Exemple :

```
1  /*  Il s'agit d'un commentaire qui peut
2  s'étendre sur plusieurs lignes jusqu'à la
       fermeture
3  */
```

Identificateurs en C++ - Règles de dénomination

- En C++, un identificateur est un nom utilisé pour identifier des variables, des fonctions, des classes et d'autres entités.
- Les règles de dénomination pour les identificateurs en C++ sont les suivantes :
  - Un identificateur doit commencer par une lettre (a-z, A-Z) ou un trait de soulignement (_).
  - Après le premier caractère, un identificateur peut contenir des lettres, des chiffres (0-9) et des traits de soulignement.
  - C++ est sensible à la casse, donc "maVariable" et "mavariable" sont traités comme deux identificateurs différents.
- Les caractères spéciaux tels que @, $ et % ne sont pas autorisés dans les identificateurs.

Identificateurs en C++ - Exemples

- Exemples d'identificateurs valides en C++ :
  - Variable : **maVariable, age, num1**
  - Fonction : **calculateSum, displayMessage**
  - Class : **Student, Circme, Rectangle**
- Exemples d'identificateurs non valides en C++ :
  - **3numbers** (commence par un chiffre)
  - **my@variable** (contient le caractère spécial @)
  - **my-variable** (contient un tiret)

Mots-clés en C++

- Il existe des **mots réservés** qui ne doivent pas être utilisés comme constantes, variables ou tout autre nom d'identifiant. Les **mots réservés** sont réservés au langage lui-même. La liste non exhaustive est ci-dessous :

alignas	alignof	and	and_eq	asm	auto	bitand	bitor
bool	break	case	catch	char	char8_t	char16_t	char32_t
class	compl	concept	const	consteval	constexpr	constinit	const_cast
continue	co_await	co_return	co_yield	decltype	default	delete	do
double	dynamic_cast	else	enum	explicit	export	extern	false
float	for	friend	goto	if	inline	int	long
mutable	namespace	new	noexcept	not	not_eq	nullptr	operator
or	or_eq	private	protected	public	reflexpr	register	reinterpret_cast
requires	return	short	signed	sizeof	static	static_assert	static_cast
struct	switch	synchronized	template	this	thread_local	throw	true
try	typedef	typeid	typename	union	unsigned	using	virtual
void	volatile	wchar_t	while	xor	xor_eq		

Figure: Mots-clés réservés

Espaces blancs en C++

- Une ligne ne contenant que des espaces blancs, éventuellement dans (ou pas) un commentaire, est appelée une **ligne vide**, et un compilateur C++ l'**ignore totalement**.
- **Whitespace** est le terme utilisé en C++ pour décrire les **espaces**, les **tabulations**, les **caractères de nouvelle ligne** et les **commentaires**.
- L'espace blanc sépare une partie d'une instruction d'une autre.

Espaces blancs en C++

- L'espace blanc permet au compilateur de savoir où un élément dans une instruction, tel que `int`, se termine et où commence le suivant. Par conséquent, dans l'instruction suivante : `int mark;`
- Il doit y avoir au moins un caractère d'espace blanc (généralement un espace) entre `int` et l'identifiant `mark` pour que le compilateur puisse les distinguer. Sinon, le compilateur identifiera `intmark` qui peut être un identifiant en lui-même...

Variables en C++

Définition : Une **variable** est un nom donné à une zone de stockage que peut manipuler un programme.

- Chaque variable en C++ a :
  - **un type spécifique**, qui détermine la taille et la disposition de la mémoire de la variable ;
  - **l'étendue des valeurs** pouvant être stockées dans cette mémoire ;
  - et **l'ensemble des opérations** pouvant être effectuées sur la variable.

Variables en C++

- Le nom d'une variable en C++ doit suivre ces deux règles :
  - **Il doit commencer** par une lettre ou un trait de soulignement.
  - La variable peut être composée de valeurs alphanumériques et/ou du caractère de soulignement. Par exemple, _1, _myVar, a_var, var et var123_L sont tous des exemples valides.
  - Le langage de programmation C++ permet également la définition de divers autres types de variables tels que Enumération, Pointeur, Tableau, Structure, Union, etc. Cependant, dans cette discussion, nous nous concentrerons sur les variables de base.

  Remarque : C/C++ est sensible à la casse, donc les lettres majuscules et minuscules sont distinctes. Par exemple, `myVar` n'est **pas** la même que `myVar`.

Decl vs Def vs Init

- Dans les prochains cadres, nous mettrons en évidence la différence entre la **déclaration**, la **définition** et l'**initialisation** d'une variable, ***UN CONCEPT IMPORTANT*** dans les langages de programmation, et surtout dans ceux de C/C++.

Variables
Déclaration de variable en C/C++

Définition : La **déclaration** d'une variable est généralement une introduction à une nouvelle mémoire allouée et identifiée par un identificateur avec un type donné (prédéfini, de base ou défini par l'utilisateur).

- Les trois propriétés de la déclaration sont :
  - La création de l'espace mémoire se produit en même temps que la déclaration elle-même.
  - Les variables déclarées ***mais non définies ou initialisées*** peuvent contenir des valeurs indéterminées (valeur aléatoire du type donné).
  - Les variables ne ***PEUVENT PAS*** être utilisées avant la déclaration.

Variables
Déclaration de variable en C/C++

- Elle est faite comme suit :

```
type liste_de_variables;
```

- Le **type *doit*** être un **type de données C++ valide** comprenant : char, int, float, double, bool ou tout objet défini par l'utilisateur, etc.

- La **liste_de_variables** peut comprendre **un** ou **plusieurs** noms d'identificateurs séparés par des virgules. Quelques déclarations valides sont présentées ci-dessous :

Variables
Déclaration de variable en C/C++

```
// Ceci est un commentaire simple et sera ignoré par le compilateur
int a; // Déclaration de la variable a avec le type int
char b, c; // Déclaration de variables distinctes b et c avec le type char
float d, e, f; // Déclaration de variables distinctes d, e et f avec le type float
double _d; // Déclaration de la variable _d avec le type double.
```

Remarque : la même chose s'applique aux fonctions, on peut **déclarer** une fonction et la **définir** plus tard. Par exemple : void foo(); est une déclaration de la fonction foo. Et sans lui donner de définition, le code peut être compilé à condition que la fonction ne soit pas utilisée ailleurs dans votre programme. (ce qui est assez inutile...).

Variables
Définition de variable en C++

Définition : La **définition** d'une variable consiste à lui attribuer une valeur, généralement avec l'opérateur d'**affectation =**. C'est le fait d'attribuer une valeur ***valide*** à la variable précédemment déclarée.

```
int a, b; // Déclaration de variables distinctes a et b de type int
a = 1; // Définition de la variable a
b = a + 1; // Définition de la variable b comme la valeur de la variable a + 1 (=2 ici)
```

Variables
Initialisation de variable en C++

Définition : L'**initialisation** d'une variable consiste simplement à attribuer (définir) la valeur au moment de la déclaration.

Par exemple :

```
float f = 0.3;
```

signifie la déclaration d'une variable avec l'identificateur f, de type float, et lui attribue / définit la valeur 0.3.

Types de données en C++

En C++, les types de données font référence au fait que chaque variable ou fonction a son propre type, soit de base, générique (défini par l'utilisateur), et/ou des types dérivés. Le type d'une variable déterminera l'espace qu'elle doit occuper en mémoire et comment le motif binaire se forme. Ainsi, comme indiqué, il existe quatre types de variables en C++, classifiés comme suit :

- **Types de base :** types arithmétiques ; soit des **entiers** (nombres naturels et entiers en mathématiques) ou des **virgule flottante** (nombres réels).
- **Types énumérés :** arithmétiques, utilisés pour définir des variables pouvant prendre uniquement certaines **valeurs entières discrètes**.

Types de données en C++

- **Type void :** le type **void** indique qu'aucune valeur n'est assignée. (pensez au vide dans le monde réel)
- **Types dérivés :** Les types **dérivés** peuvent être des pointeurs, des tableaux, des structures, des classes (dans tous les langages de programmation orientée objet), des types union et des fonctions. (la liste n'est pas exhaustive).

Remarque : En C++, il existe deux types agrégés : les tableaux et les structures. Les types agrégés sont des collections de valeurs scalaires.

Catégorie	Type	Taille (bits)	Plage	Description
Types de base	bool	1	vrai/faux	Valeur booléenne
	char	8	-128 à 127	Caractère
	wchar_t	16 ou 32	Dépendant de la plate-forme	Caractère étendu
	int	32 ou 64	Dépendant de la plate-forme	Entier
	unsigned int	32 ou 64	0 à 4294967295	Entier non signé
	short	16	-32768 à 32767	Entier court

- Les types de base en C++ comprennent bool, char, wchar_t, int, unsigned int et short.
- Chaque type a une taille spécifique en bits et une plage de valeurs qu'il peut représenter.

Catégorie	Type	Taille (bits)	Plage	Description
Types de base	unsigned short	16	0 à 65535	Entier court non signé
	unsigned long	32 ou 64	0 à 18446744073709551615	Entier long non signé
	unsigned long long	64 ou plus	0 à 18446744073709551615	Entier long long non signé

- Les types de base non signés en C++ comprennent unsigned short, unsigned long et unsigned long long.
- Chaque type a une taille spécifique en bits et une plage de valeurs qu'il peut représenter.

Catégorie	Type	Taille (bits)	Plage	Description
Types dérivés	pointeurs	-	-	Stocke les adresses mémoire
	tableaux	-	-	Collection d'éléments du même type
	structures	-	-	Groupe de variables liées

- Les types dérivés en C++ comprennent les pointeurs, les tableaux et les structures.
- Les pointeurs stockent des adresses mémoire, les tableaux représentent des collections d'éléments du même type, et les structures regroupent des variables liées.

Catégorie	Type	Taille (bits)	Plage	Description
Types dérivés	classes	-	-	Plan de création d'objets
	unions	-	-	Stocke différents types dans le même espace mémoire
	fonctions	-	-	Blocs de code avec un nom

- Les types dérivés en C++ incluent également les classes, les unions et les fonctions.
- Les classes servent de plans de création d'objets, les unions stockent différents types dans le même espace mémoire, et les fonctions sont des blocs de code avec un nom.

Types de données

- Pour obtenir la taille exacte d'un type ou d'une variable sur un système d'exploitation particulier avec un compilateur particulier, vous pouvez utiliser l'opérateur sizeof. L'expression sizeof(type) donne la taille de stockage de l'objet ou du type en **octets**.

Remarque : soyez conscient du fait que l'opérateur sizeof renverra des valeurs légèrement différentes en fonction du système d'exploitation, du système d'exploitation, du compilateur, etc.

Types de données : Spécificateurs de format en C++ (Partie 1)

Les **spécificateurs de format** en C++ sont utilisés lors des opérations d'entrée et de sortie. Ils aident à spécifier le type de données dans une variable lors de la numérisation d'une variable à l'aide de `std::cin` ou lors de l'impression d'une variable à l'aide de `std::cout`.

En C++, les spécificateurs de format sont similaires à ceux en C, mais avec quelques différences et fonctionnalités supplémentaires. Ils sont utilisés avec les opérateurs d'extraction et d'insertion de flux (>> et <<) pour gérer différents types de données et options de formatage.

Types de données : Spécificateurs de format en C++ (Partie 2)

C++ fournit divers spécificateurs de format pour différents types de données et options de formatage. Certains spécificateurs de format couramment utilisés incluent :

- `%d` pour les entiers
- `%f` pour les nombres en virgule flottante
- `%c` pour les caractères
- `%s` pour les chaînes de caractères
- `%p` pour les pointeurs
- `%x` ou `%X` pour les valeurs hexadécimales

Types de données : Spécificateurs de format en C++ (Partie 3)

Voici quelques exemples d'utilisation des spécificateurs de format en C++ :

Listing 1: Exemple 1 : Affichage d'un nombre en virgule flottante avec 2 décimales

```
1    double valeur = 3.14159;
2    std::cout << "La␣valeur␣est␣:␣" << std::fixed << std::
        setprecision(2) << valeur;
```

Listing 2: Exemple 2 : Formatage d'une chaîne avec une largeur spécifiée

```
1    std::string nom = "John";
2    std::cout << std::setw(10) << std::left << nom;
```

Manipulations de Format en C++ - Partie 1

Lors de la manipulation de la sortie standard en C++, les manipulations de format offrent un contrôle précis sur la manière dont les données sont présentées. Ces manipulations sont comparables aux spécificateurs de format utilisés dans d'autres langages, tels que le style `%f` en C.

- `std::fixed` : Spécifie une représentation à virgule fixe pour les nombres à virgule flottante. Cela garantit une précision déterminée au niveau des décimales.
- `std::setprecision(n)` : Définit la précision des chiffres après la virgule lors de l'affichage des nombres à virgule flottante, où `n` représente le nombre de décimales souhaité.

Manipulations de Format en C++ - Partie 2

- `std::setw(n)` : Spécifie la largeur du champ lors de l'affichage de données. Utile pour aligner et formater les données, notamment dans le cas des chaînes de caractères.

Ces manipulations offrent une flexibilité similaire aux spécificateurs de format dans d'autres langages, permettant aux développeurs de contrôler la mise en forme des données à la sortie standard de manière précise et intuitive.

Types de données : le type void

Le **type void** est un concept important, un type intégré, et spécifie qu'aucune valeur n'est disponible. Il est utilisé dans trois situations particulières :

- **Le type de retour d'une fonction est de type void**. La fonction retourne comme void. Il existe diverses fonctions en C++ qui ne renvoient aucune valeur, d'où le **type void**. Exemple :

```
void draw_line() {
        std::cout « "- – — - – – — — – – — – -" « std::endl;
}
```

Types de données : le type void (Partie 1)

- **Arguments de fonction comme void :** Les fonctions en C++ prenant une liste d'arguments comme void sont simplement des fonctions avec des paramètres inconnus, tandis qu'en C++, `f(void)` est exactement la même chose que `f()`.

Types de données : le type void (Partie 2)

- **Pointeurs vers void :** Un pointeur de type **void *** représente l'adresse d'un objet, mais **pas son type**. Par exemple, l'opérateur `new` en C++ peut allouer de la mémoire et renvoyer un pointeur vers void qui peut être **casté** vers n'importe quel type de données. Par exemple :

```
void *ptr = new int(10); // Alloue de la mémoire pour un entier
int *intPtr = static_cast<int*>(ptr); // Cast du pointeur void vers un pointeur int
delete intPtr; // Libère la mémoire
```

- **Remarque :** Le type void est un concept important en C++ pour la manipulation des zones mémoire, des pointeurs et des conversions. Le seul objet qui peut être déclaré avec le spécificateur de type void est un pointeur.

Types de données entiers

En C++, les types de données entiers représentent des nombres entiers sans partie fractionnaire.

```cpp
#include <iostream>

int main() {
    int age = 25;
    unsigned int population = 1000000;
    long long largeNumber = 123456789012345;
    std::cout << "Âge : " << age << std::endl;
    std::cout << "Population : " << population << std::endl;
    std::cout << "Grand Nombre : " << largeNumber << std::endl;

    return 0;
}
```

Types de données à virgule flottante

En C++, les types de données à virgule flottante représentent des nombres avec des parties décimales.

```cpp
#include <iostream>

int main() {
    float distance = 5.6f;
    double pi = 3.14159265359;
    long double veryPreciseValue = 123456789012345.678;
    std::cout << "Distance : " << distance << std::endl;
    std::cout << "Pi : " << pi << std::endl;
    std::cout << "Valeur Très Précise : " << veryPreciseValue <<
        std::endl;

    return 0;
}
```

Calcul arithmétique et affichage des résultats

En C++, vous pouvez effectuer des calculs arithmétiques et afficher les résultats.

```cpp
#include <iostream>

int main() {
    int a = 10;
    double b = 4.5;
    // Calcul et affichage du résultat
    double result = a * b / 2.0;
    std::cout << "Le résultat du calcul est : " << result << std::
        endl;

    return 0;
}
```

Calcul arithmétique et affichage des résultats (suite)

Vous pouvez également utiliser des parenthèses pour définir l'ordre des opérations.

```
#include <iostream>

int main() {
    int x = 5;
    double y = 2.5;
    // Utilisation des parenthèses pour définir l'ordre des
       opérations
    double result = (x + 2) * y;
    std::cout << "Le résultat du calcul est : " << result << std::
       endl;

```

Mélange des types entiers et à virgule flottante

En C++, vous pouvez mélanger des types entiers et à virgule flottante dans les calculs et les affectations.

```cpp
#include <iostream>

int main() {
    int a = 5;
    double b = 2.5;
    // Mélange des types dans le calcul
    double result = a + b;
    std::cout << "Le résultat du calcul est : " << result << std::
        endl;

    return 0;
}
```

Mélange des types (suite)

Assurez-vous de comprendre comment les conversions implicites fonctionnent dans les expressions.

```cpp
#include <iostream>

int main() {
    int x = 3;
    double y = 4.7;
    // Conversion implicite dans l'expression
    double result = x + y;
    std::cout << "Le résultat du calcul est : " << result << std::
        endl;

    return 0;
}
```

Utilisation des références pour l'efficacité

En C++, l'utilisation de références peut améliorer l'efficacité en évitant la copie d'objets.

```cpp
#include <iostream>

int main() {
    int original = 42;
    // Utilisation d'une référence
    int& ref = original;
    std::cout << "La valeur de la référence est : " << ref << std
        ::endl;

    return 0;
}
```

Utilisation des constantes pour la sécurité

En C++, l'utilisation de constantes peut renforcer la sécurité en empêchant la modification accidentelle des valeurs.

```cpp
#include <iostream>

int main() {
    const int constantValue = 10;
    // Tentative de modification de la constante (erreur de compilation)
    // constantValue = 20;

    std::cout << "La valeur de la constante est : " << constantValue << std::endl;

    return 0;
}
```

Constantes et littéraux

Définition : Les constantes font référence à des valeurs fixes qu'un programme ne peut pas modifier pendant son exécution. Ces valeurs fixes sont appelées littéraux.

- Les constantes peuvent être de n'importe quel type de données de base :
  - une constante **entière**;
  - une constante **flottante**;
  - une constante **caractère**;
  - une **chaîne** littérale;
  - une constante d'**énumération**.

Systèmes de numération

Nous distinguons quatre systèmes de numération couramment utilisés dans les protocoles de communication quotidiens. Nous les détaillons comme suit :

- Le **système binaire**, qui fonctionne sur la base **2**;
- Le **système octal**, qui fonctionne sur la base **8**;
- Le **système décimal**, qui fonctionne sur la base **10**;
- Le **système hexadécimal**, qui fonctionne sur la base **16**.

Systèmes de numération

Voici une image illustrant l'équivalence entre les systèmes :

Decimal	Binary	Octal	Hexadecimal
0	0000	0	0
1	0001	1	1
2	0010	2	2
3	0011	3	3
4	0100	4	4
5	0101	5	5
6	0110	6	6
7	0111	7	7
8	1000	10	8
9	1001	11	9
10	1010	12	A
11	1011	13	B
12	1100	14	C
13	1101	15	D
14	1110	16	E
15	1111	17	F

Figure: Systèmes de numération

Constantes et littéraux
Littéraux entiers

Un littéral entier peut être une constante *décimale*, *octale* ou *hexadécimale*. Un préfixe spécifie la base ou le radix :

- **0x** ou **0X** pour **l'hexadécimal** (base **16**);
- **0** pour **l'octal** (base **8**);
- **rien** pour le **décimal** (base **10**).

Un exemple de littéraux entiers :

Constantes et littéraux
Littéraux flottants

Un littéral en virgule flottante est composé de :

- une partie entière ;
- un point décimal ;
- une partie fractionnaire ;
- une partie exposant.

Les littéraux en virgule flottante peuvent être représentés soit sous forme **décimale** soit sous forme **exponentielle**.

Constantes et littéraux
Littéraux flottants

- **Sous forme décimale** : doit inclure le point décimal, l'exposant ou les deux ;
- **Sous forme exponentielle** : doit inclure la partie entière, la partie fractionnaire ou les deux. Tandis que l'exposant signé est introduit par la minuscule ou majuscule **e** (**e** ou **E**).

Quelques exemples du site Web d'IBM :

floating-point constant	Value
5.3876e4	53,876
4e-11	0.00000000004
1e+5	100000
7.321E-3	0.007321
3.2E+4	32000
0.5e-6	0.0000005
0.45	0.45
6.e10	60000000000

Constantes et littéraux
Constantes de caractère

Définition : Les **littéraux de caractères** sont inclus entre guillemets simples et peuvent être stockés dans une simple variable de type char.

- Un littéral de caractère peut être un caractère simple (par exemple, char a = 'a' ;) ;
- une séquence d'échappement '\t' (nous aborderons cela plus tard) ;
- un caractère universel (nous aborderons cela plus tard).

Constantes et littéraux
Constantes de caractère

Certains caractères en C, lorsqu'ils sont précédés d'un antislash, ont une signification spéciale. Une liste non exhaustive est ci-dessous :

Escape sequence	Meaning
\\	\ character
\'	' character
\"	" character
\?	? character
\a	Alert or bell
\b	Backspace
\f	Form feed
\n	Newline
\r	Carriage return
\t	Horizontal tab
\v	Vertical tab

Figure: Séquences d'échappement

Constantes et littéraux
Littéraux de chaîne

Définition : Les littéraux ou constantes de chaîne sont inclus entre guillemets doubles "".
Une chaîne est simplement un tableau de littéraux de caractères : caractères simples, séquences d'échappement et/ou caractères universels.

Constantes et littéraux
Littéraux de chaîne

- Nous pouvons vouloir diviser une longue chaîne en plusieurs en utilisant des littéraux en les séparant par des espaces. Un exemple peut être le suivant :

```
"Bonjour le monde !"
"Bonjour \
monde !"
"Bonjour " "l" "e monde !"
```

Constantes et littéraux
Définir des constantes

Il existe deux façons de définir des constantes en C et C++ :

- avec la directive de préprocesseur **#define**;
- avec le mot-clé **const**. Voici les deux façons :

- #define my_Identifier my_Value (par exemple #define PI 3.14)
- const type my_Identifier = my_Value; (par exemple const float PI = 3.14...)

Remarque : pas de point-virgule nécessaire pour les constantes du préprocesseur. **Note :** il est recommandé / une bonne pratique de définir les constantes en MAJUSCULES.

Classes de stockage en C++

Définition : En langage de programmation C++, une **Classe de stockage** définit la portée (visibilité) et la durée de vie des variables et des fonctions dans un programme C++. Ces spécificateurs (aussi appelés classes de stockage) précèdent le type qu'ils modifient.

C++ a également quatre classes de stockage principales : **auto**, **extern**, **static** et **register**. La classe de stockage pour une variable est spécifiée comme suit :

```
classeDeStockage typeDeVariable nomDeVariable;
```

Ces classes de stockage fournissent différents comportements et

Déclaration et **Définition** de Fonction

Une **Déclaration de Fonction**, comme indiqué ci-dessus, a la signature :

```
type_de_retour nom_de_ma_fonction(arguments si présents);
```

Alors qu'une **Définition de Fonction** est donnée par :

```
type_de_retour nom_de_ma_fonction(arguments si présents) {
        instructions_de_fonction;
}
```

Déclaration et Définition de Fonction en C++

Un exemple complet est le suivant :

```
// Déclaration de Fonction
int ma_fonction();
int main() {
        /*
       Appel de fonction. La Définition peut venir :
       - juste après la Déclaration dans le même programme source,
       - ou faire partie d'un autre programme source qui doit être compilé avant que nous utilisions la fonction.
       */
       // Appel de Fonction
       ma_fonction();
}
// Définition de Fonction
int ma_fonction() return (-123);
// La fonction retourne simplement la valeur entière -123. Les parenthèses sont ici pour des raisons de précision.
```

Notion de récursivité : fonctions récursives

Définitions :

- Un algorithme est appelé **récursif** s'il est défini par lui-même.
- De la même manière, une fonction est appelée **récursive** si elle se référence elle-même.

Rvalues et Lvalues en C++ (Partie 1)

En C++, les expressions peuvent être catégorisées comme **rvalues** ou **lvalues**.

- Les **lvalues** représentent des objets avec un emplacement mémoire spécifique, tels que des variables ou des objets nommés.

  ```
  int x = 5; // Ici, x est une lvalue.
  ```

- Les **rvalues** font référence à des objets temporaires qui n'ont pas d'emplacement mémoire spécifique.

  ```
  int sum = 2 + 3; // Ici, 2 + 3 est une rvalue.
  ```

- Comprendre la distinction entre rvalues et lvalues est crucial pour la programmation en C++.

Rvalues et Lvalues en C++ (Partie 2)

En C++, les rvalues et lvalues ont des comportements différents.
Lvalues :

- Elles ont un emplacement mémoire spécifique.
- Elles peuvent être assignées et utilisées comme références.

Rvalues :

- Elles n'ont pas d'emplacement mémoire spécifique.
- Elles ne peuvent pas être assignées ou utilisées directement comme références.

Comprendre ces concepts est essentiel pour une programmation C++ efficace.

Rvalues et Lvalues en C++ (Partie 3)

En plus des rvalues et lvalues de base, C++ introduit les **références d'rvalue** et la **sémantique de déplacement**.
Référence d'rvalue :

- Une référence d'rvalue peut se lier à une rvalue et potentiellement la modifier.

Sémantique de déplacement :

- La sémantique de déplacement optimise le transfert de ressources entre les objets, améliorant l'efficacité.

En exploitant ces concepts, C++ offre des outils puissants pour une programmation efficace.

Rvalues et Lvalues en C++ (Résumé)

- Les **lvalues** représentent des objets avec un emplacement mémoire spécifique et peuvent être assignées.
- Les **rvalues** font référence à des objets temporaires sans emplacement mémoire spécifique.
- C++ introduit les **références d'rvalue** et la **sémantique de déplacement** pour optimiser la manipulation d'objets temporaires.
- Les références d'rvalue permettent la liaison et la modification potentielles des rvalues.
- La sémantique de déplacement facilite le transfert efficace de ressources entre les objets.

Plus d'exemples sur Lvalues et Rvalues en C++

- Poursuite avec des exemples de lvalues et rvalues :

  - Exemple 1 : Passage d'une lvalue en tant qu'argument de fonction
    - `int carre(int x) { return x * x; }`
    - Ici, `x` est une lvalue qui est passée en argument à la fonction `carre`.
  - Exemple 2 : Utilisation d'une référence d'rvalue
    - `int&& referenceRvalue = 10;`
    - Ici, `referenceRvalue` est une référence d'rvalue qui peut se lier à une rvalue.

Passage des Arguments : Par Valeur ou Par Référence

En C++, le choix entre le passage des arguments par valeur ou par référence a un impact significatif sur le fonctionnement des fonctions et la manipulation des données.

Exemple : swap(x, y) et swap(&x, &y)

- **Par Valeur :**
  - Passe la valeur réelle de l'argument.
  - Les modifications effectuées à l'intérieur de la fonction n'affectent pas la variable d'origine.
- **Par Référence :**
  - Passe l'adresse mémoire de l'argument.
  - Les modifications effectuées à l'intérieur de la fonction affectent directement la variable d'origine.

Visibilité, Durée et Valeur Initiale des Variables Temporaires Locales et des Paramètres

En C++, la visibilité, la durée de vie et la valeur initiale des variables temporaires locales et des paramètres sont des aspects essentiels à comprendre pour assurer un comportement correct des programmes.

Exemple : swap(x, y) et swap(&x, &y)

- **Visibilité :**
  - Définit où une variable peut être utilisée dans le code.
  - Les variables locales sont généralement visibles uniquement dans le bloc où elles sont déclarées.
- **Durée de Vie :**
  - Indique la période pendant laquelle une variable existe en mémoire.
  - Les variables locales ont une durée de vie limitée à leur bloc d'exécution.
- **Valeur Initiale :**
  - Définit la valeur par défaut d'une variable lors de sa création.
  - Les variables locales peuvent avoir une valeur initiale explicite ou dépendre du contexte.

Portées

Définition : Une **portée** dans n'importe quel langage de programmation est la zone où une fonction ou une variable est visible et accessible. En dehors de cette même portée, la fonction ou la variable n'est pas accessible.

Il existe trois régions où les variables peuvent être déclarées en C++ :

- À l'intérieur d'une fonction ou d'un bloc, appelé **variables locales**;
- En dehors de toutes les fonctions, appelé **variables globales**;
- Dans la définition des paramètres de fonction, appelé **paramètres formels**.

Portées

Voici trois exemples de code source illustrant des variables locales, globales et des paramètres formels :

- **Exemple de variables locales;**
- **Exemple de variables globales;**
- **Exemple de paramètres formels.**

Portées : comportement de l'initialisation des variables locales et globales

Lorsqu'une **variable locale** est **déclarée**, elle n'est ***pas initialisée*** par le compilateur. En revanche, les **variables globales** sont **automatiquement initialisées** lorsqu'elles sont déclarées :

Type de données	Valeur par défaut lors de la déclaration
int	0
char	'\0'
float	0.0 ou simplement 0
double	0.0 ou simplement 0
Pointeur	NULL

Note importante : Il est fortement recommandé d'initialiser correctement les variables. Sinon, la variable non initialisée aura une valeur indésirable déjà présente à son emplacement mémoire, ce qui peut entraîner un comportement et/ou des résultats inattendus.

Opérateurs

Définition : En mathématiques et en programmation informatique, un **opérateur** est un symbole ou un caractère qui indique au compilateur d'effectuer des processus mathématiques ou logiques spécifiques.

Chaque langage de programmation a ses propres opérateurs intégrés, cependant, les opérateurs présentés ici sont communs à de nombreux langages de programmation. C fournit ces opérateurs intégrés : opérateurs **arithmétiques**, opérateurs **relationnels**, opérateurs **logiques**, opérateurs **bit à bit**, opérateurs **d'assignation** et opérateurs **divers** (Misc).

Opérateurs
Table de vérité

Un concept important lié à la manipulation des opérateurs est la table de vérité. Supposons que nous avons deux propositions (P) et (Q). Alors :

P	Q	P ET Q	P OU Q	P XOR Q	NON P	NON Q
0	0	0	0	0	1	1
0	1	0	1	1	1	0
1	0	0	1	1	0	1
1	1	1	1	0	0	0

Opérateurs
Opérateurs Arithmétiques

Le tableau suivant explicite les opérateurs arithmétiques intégrés en C.
Cet exemple montre ces opérateurs arithmétiques dans un programme C.

Opérateur	Signification
$+$	Ajoute la valeur de l'opérande du côté droit (RHS) à la valeur de l'opérande du côté gauche (LHS) (a+b)
$-$	Soustrait la valeur de l'opérande du côté droit (RHS) à la valeur de l'opérande du côté gauche (LHS) (a-b)
$*$	Multiplie les valeurs des opérandes du côté gauche (LHS) et du côté droit (RHS) ensemble (a*b)
$/$	Divise la valeur de l'opérande du côté gauche (LHS) par la valeur de l'opérande du côté droit (RHS) (a/b)
$\%$	Le modulo renvoie le reste d'une division euclidienne de la valeur de l'opérande du côté gauche (LHS) par la valeur de l'opérande
$++$	Incrémente une valeur entière de un (++a **pré-incrément**) ou (a++ **post-incrément**)
$--$	Décrémente une valeur entière de un (–a **pré-décrément**) ou (a– **post-décrément**)

Opérateurs
Opérateurs Relationnels

Le tableau suivant explicite les opérateurs relationnels intégrés en C.
Cet exemple montre ces opérateurs relationnels dans un programme C.

Opérateur	Signification
$==$	Évalué à vrai seulement si les deux opérandes sont égales
$!=$	Évalué à vrai seulement si les deux opérandes ne sont pas égales
$>$	Évalué à vrai seulement si la valeur de l'opérande du côté gauche (LHS) est strictement supérieure à celle du côté droit (RHS)
$<$	Évalué à vrai seulement si la valeur de l'opérande du côté droit (RHS) est strictement supérieure à celle du côté gauche (LHS)
$>=$	Évalué à vrai seulement si la valeur de l'opérande du côté gauche (LHS) est strictement supérieure ou égale à celle du côté droit (R
$<=$	Évalué à vrai seulement si la valeur de l'opérande du côté droit (RHS) est strictement supérieure ou égale à celle du côté gauche (L

Opérateurs
Opérateurs Logiques

Le tableau suivant explicite les opérateurs logiques intégrés en C.
Cet exemple montre ces opérateurs logiques dans un programme C.

Opérateur	Signification
&&	Opérateur ET logique. La condition est vraie seulement si les deux opérandes ne sont pas nulles
||	Opérateur OU logique. La condition est vraie seulement si l'une des opérandes n'est pas nulle
!	Opérateur NON logique. Il inverse l'état logique. Si la condition est vraie, alors l'opérateur NON logique la rendra fausse, et vice ve

Opérateurs
Opérateurs Bit à Bit

Le tableau suivant explicite les opérateurs bit à bit intégrés en C.
Cet exemple montre ces opérateurs bit à bit dans un programme C.

Opérateur	Signification
&	Opérateur ET binaire, évalue à VRAI seulement si les deux bits sont VRAIS
|	Opérateur OU binaire, évalue à VRAI seulement si l'un des bits est VRAI
^	Opérateur OU exclusif binaire, évalue à VRAI seulement si un bit est VRAI dans un opérande mais pas da
~	Opérateur NON binaire unaire (ou opérateur de complément). L'opérateur de complément inverse simplement les bits
«	Opérateur de décalage binaire à gauche. La valeur des opérandes du côté gauche est déplacée vers la gauche par le nombre de bit
»	Opérateur de décalage binaire à droite. La valeur des opérandes du côté gauche est déplacée vers la droite par le nombre de bits

Opérateurs
Opérateurs d'**Assignation**

Le tableau suivant explicite les opérateurs d'assignation intégrés en C. **Cet exemple** montre ces opérateurs d'assignation dans un programme C.

Opérateur	Signification
=	Opérateur d'assignation. Il assigne les valeurs des opérandes du côté droit (RHS) aux opérandes du côté
+=	Opérateur d'assignation ET d'addition. Il ajoute l'opérande du côté droit (RHS) à l'opérande du côté gauche (LHS) et assigne le résu
-=	Opérateur d'assignation ET de soustraction. Il soustrait l'opérande du côté droit (RHS) à l'opérande du côté gauche (LHS) et assigne le
*=	Opérateur d'assignation ET de multiplication. Il multiplie l'opérande du côté droit (RHS) à l'opérande du côté gauche (LHS) et assigne le
/=	Opérateur d'assignation ET de division. Il divise l'opérande du côté gauche (LHS) par l'opérande du côté droit (RHS) et assigne le rés
%=	Opérateur d'assignation ET de modulo. Il assigne le reste de la division euclidienne de l'opérande du côté gauche (LHS) par l'opérande du cô
«=	Opérateur d'assignation ET de décalage à gauche.
»=	Opérateur d'assignation ET de décalage à droite.
&=	Opérateur d'assignation ET logique.
^=	Opérateur d'assignation ET de OU exclusif logique.
|=	Opérateur d'assignation ET de OU inclusif logique.

Opérateurs
Opérateurs Divers

Le tableau suivant explicite les opérateurs divers intégrés en C.
Cet exemple montre ces opérateurs divers dans un programme C.

Opérateur	Signification
sizeof()	Il renvoie simplement la taille d'une variable en octets.
&	Il renvoie l'adresse d'une variable.
*	Opérateur de pointeur.
? :	Opérateur ternaire. Expression conditionnelle.

Opérateurs
Précédence et **priorité** en C++

La **précédence des opérateurs** décide du regroupement des termes dans une expression. Elle affecte la manière dont une expression est évaluée. Certains opérateurs ont une précédence plus élevée que d'autres.

- **La règle est** : dans une expression, les opérateurs de précédence plus élevée seront évalués en premier.
- Dans le tableau ci-dessous, les opérateurs ayant la plus haute précédence apparaissent en haut, tandis que ceux ayant la plus basse précédence apparaissent en bas.

Opérateurs
Précédence et **priorité** en C++

Opérateur	Classification	Associativité
() [] -> . ++ –	Postfix	De gauche à droite
+ - ! ~ ++ – (type)* & sizeof	Unaire	De droite à gauche
*/ %	Multiplicatif	De gauche à droite
+-	Additif	De gauche à droite
« »	Décalage	De gauche à droite
< <= > >=	Relationnel	De gauche à droite
== !=	Égalité	De gauche à droite
&	ET binaire	De gauche à droite
^	OU exclusif binaire	De gauche à droite
|	OU binaire	De gauche à droite
&&	ET logique	De gauche à droite
||	OU logique	De gauche à droite
?:	Conditionnel	De droite à gauche
= += -= *= /= %= »= «= &= ^= |=	Assignation	De droite à gauche
,	Virgule	De gauche à droite

Structures conditionnelles alternatives et répétitives & Boucles

- En algorithmique et en programmation, nous appelons **structure conditionnelle** l'ensemble d'instructions qui testent si une condition est vraie ou non. Il en existe trois :
  - La structure conditionnelle **if** :

```
 if (condition) {
      mes_instructions;
}
```

Structures conditionnelles alternatives et répétitives & Boucles

- **Suite :**
  - La structure conditionnelle **if ...else** :

```
 if (condition) {
      mes_instructions;
}
else {
      mes_autres_instructions;
}
```

Structures conditionnelles alternatives et répétitives & Boucles

- **Suite :**
  - La structure conditionnelle **if ...elif ...else** :

```
 if (condition_1) {
       mes_instructions_1;
}
elif (condition_2) {
       mes_instructions_2;
}
else {
       mes_autres_instructions;
}
```

Structures conditionnelles alternatives et répétitives & Boucles

- De plus, une **structure conditionnelle itérative** permet d'exécuter plusieurs fois (itérations) la même série d'instructions. L'instruction **while** (tant que) exécute les blocs d'instructions tant que la condition de while est vraie. Le même principe s'applique à la boucle **for** (pour), les deux structures sont détaillées ci-dessous :
  - La structure conditionnelle itérative **while** :

```
while (condition)
     mes_instructions;
```

Boucle For en C++ (basée sur l'index)

- La boucle `for` en C++ peut être utilisée de manière indexée pour itérer sur une plage de valeurs ou d'éléments d'un tableau.
- Voici un exemple d'utilisation d'une boucle `for` de manière indexée :

```
for (int i = 0; i < n; i++) {
    // Accéder à l'élément à l'index i
    // Effectuer des opérations
}
```

- Dans cette approche, un compteur de boucle `i` est initialisé à l'index de départ, et la boucle continue tant que `i < n` est faux, où `n` représente la taille ou la longueur de la plage ou du tableau.

Boucle For en C++ (basée sur l'index)

- Le compteur de boucle est incrémenté ou décrémenté à la fin de chaque itération.
- Vous pouvez utiliser le compteur de boucle `i` pour accéder à des éléments à différents indices et effectuer des opérations dans le corps de la boucle.
- Cette approche est utile lors de la manipulation d'éléments spécifiques d'un tableau ou d'une plage définie de valeurs.

Exemples de boucles for en C++ (basées sur l'index)

- Exemple 1 :

```
for (int i = 0; i < 5; i++) {
        std::cout << "Itération : " << i << std::endl;
}
```

- Exemple 2 :

```
int numbers[] = {1, 2, 3, 4, 5};
int longueur = sizeof(numbers) / sizeof(numbers[0]);
for (int i = 0; i < longueur; i++) {
        std::cout << "Nombre : " << numbers[i] << std::endl;
}
```

Boucle For en C++ (basée sur les éléments)

- La boucle `for` en C++ peut également être utilisée de manière basée sur les éléments pour itérer directement sur les éléments d'un conteneur ou d'une séquence basée sur une plage.
- Voici un exemple d'utilisation d'une boucle `for` de manière basée sur les éléments :

```
for (const auto& element : container) {
    // Accéder à l'élément
    // Effectuer des opérations
}
```

- Dans cette approche, la boucle itère sur chaque élément du `container` (tel qu'un tableau, un vecteur ou toute séquence basée sur une plage).

Boucle For en C++ (basée sur les éléments)

- La variable de boucle `element` représente l'élément actuel traité à chaque itération.
- Vous pouvez accéder directement et travailler avec les éléments dans le corps de la boucle sans avoir besoin d'un indice explicite.
- Cette approche est plus pratique lorsque vous souhaitez itérer sur tous les éléments d'un conteneur ou d'une séquence sans gérer le compteur de boucle.

Exemples de boucles for basées sur les éléments

- Exemple 1 :

```
std::vector<int> numbers = {1, 2, 3, 4, 5};
for (const auto& num : numbers) {
        std::cout << "Nombre : " << num << std::endl;
}
```

- Exemple 2 :

```
std::string str = "Bonjour le monde !";
for (const auto& ch : str) {
        std::cout << "Caractère : " << ch << std::endl;
}
```

Structures conditionnelles alternatives et répétitives & Boucles

- **Suite** :
  - La structure conditionnelle itérative **do ...while** :

```
do {
      mes_instructions;
} while(my_condition);
```

- **Note** : une différence réside entre la boucle **while** et la boucle **do while** : dans la boucle **while**, la condition de vérification s'exécute avant la première itération de la boucle, tandis que **do while** vérifie la condition **après** l'exécution des instructions à l'intérieur de la boucle.

Tableaux

- Un tableau est une collection d'éléments de données du même type (généralement) qui satisfait les conditions suivantes :
  - Une collection de données qui ont le **même type** (bool, int, double, chaîne de caractères...)
  - Les éléments sont stockés **de manière contiguë** en mémoire (figure 5)
  - La **taille du tableau** doit être connue lors de la déclaration du tableau.
  - En considérant un tableau de taille *N*, l'**indice du premier élément** du tableau est 0, tandis que l'**indice du dernier élément** est $N - 1$. (dans la plupart des langages de programmation : C/C++, Python, etc.)
  - Les éléments d'un tableau sont **accessibles** *via* leurs **indice**.

Example of array representation
in memory of an array of ints

Memory Location	400	404	408	412	416	420
Index	0	1	2	3	4	5
Data	18	10	13	12	19	9

Figure: Exemple de tableau

Tableaux

- Avantages des tableaux :
  - **Optimisation du code :** la complexité pour ajouter ou supprimer un élément à la fin du tableau, l'accès à un élément *via* son indice, ajouter un élément ou le supprimer est en $O(1)$. La recherche **séquentielle** (linéaire) d'un élément dans un tableau a une complexité de $O(n)$.
  - **Facilité d'accès :** accès direct à un élément *via* son indice (par exemple : monTableau[3]), itération sur les éléments d'un tableau à l'aide d'une boucle, tri simplifié.
- Inconvénients des tableaux :
  - La contrainte de devoir déclarer sa taille avant le **temps de compilation** (ce qu'on appelle un **tableau statique**, nous y reviendrons), et que la taille du tableau ne peut pas augmenter pendant l'exécution.
  - Insérer et supprimer des éléments peut être coûteux car les éléments doivent être gérés en fonction de la nouvelle zone mémoire allouée (Exemple : vouloir ajouter un élément au milieu d'un tableau).

Tableaux

- Déclaration d'un tableau en C/C++ :
  - **typename monTableau**$[n]$: crée un tableau de taille *n* (*n* éléments) de type **typename** avec des valeurs aléatoires. **Exemple : int a**$[3]$ produit :

```
Indice : 0 | adresse : 0x7ffee253c4c0 | valeur : -497826512
Indice : 1 | adresse : 0x7ffee253c4c4 | valeur : 32766
Indice : 2 | adresse : 0x7ffee253c4c8 | valeur : 225226960
```

  - **typename monTableau**$[n] = \{v_1, ..., v_n\}$: crée un tableau de taille *n* de type **typename** avec des valeurs initialisées à v_1 jusqu'à v_n. **Exemple :** **int a**$[3] = \{1, 4, 9\}$ crée un tableau de 3 éléments initialisés respectivement à 1, 4 et 9.

Tableaux

- Déclaration d'un tableau en C/C++ :
  - **typename monTableau**$[n] = \{v, ..., v\}$: crée un tableau de taille *n* de type **typename** avec toutes les valeurs initialisées à *v*. **Exemple :**
    int a$[5] = \{1, 1, 1, 1, 1\}$ crée un tableau de 5 éléments, tous initialisés à 1.

```
Indice : 0 | adresse : 0x7ffeea95b510 | valeur : 1
Indice : 1 | adresse : 0x7ffeea95b514 | valeur : 1
Indice : 2 | adresse : 0x7ffeea95b518 | valeur : 1
Indice : 3 | adresse : 0x7ffeea95b51c | valeur : 1
Indice : 4 | adresse : 0x7ffeea95b520 | valeur : 1
```

Tableaux

- Déclaration d'un tableau en C/C++ :
  - **typename monTableau**$[n] = \{\}$: crée un tableau de taille *n* de type **typename** avec toutes les valeurs initialisées à 0. **Exemple : int a**$[4] = \{\}$ crée un tableau de 4 éléments, tous initialisés à 0.

```
Indice : 0 | adresse : 0x7ffee3257510 | valeur : 0
Indice : 1 | adresse : 0x7ffee3257514 | valeur : 0
Indice : 2 | adresse : 0x7ffee3257518 | valeur : 0
Indice : 3 | adresse : 0x7ffee325751c | valeur : 0
```

- **typename monTableau**$[n] = \{v\}$: crée un tableau de taille *n* de type **typename** avec la première valeur initialisée à *v*, le reste à 0.

Introduction Pointeurs - Static vs Dynamic

- Il existe deux principaux types d'allocation de mémoire :

Définition : L'Allocation statique de mémoire au temps de compilation (allocation de mémoire au temps de compilation), se produit lors de la déclaration d'une variable, le compilateur alloue automatiquement un emplacement mémoire pour celle-ci.

Définition : L'Allocation dynamique de mémoire au temps d'exécution (également appelée Allocation de mémoire au temps d'exécution). Elle se produit lorsque la mémoire peut être allouée pour des variables de données après le début de l'exécution du programme.

Introduction Pointeurs - Compiletime vs Runtime

Définition : Le **temps de compilation (compile time)** est la période pendant laquelle le code source du programme (tel que C, C++, C#, Java ou Python) est converti en code binaire.

Définition : Le **temps d'exécution (run time)** est la période pendant laquelle un programme s'exécute et survient généralement après le temps de compilation.

Introduction Pointeurs - Notion d'adresse & d'allocation mémoire

Définition : Une **adresse mémoire** ou **adresse** est l'emplacement où une variable est stockée sur l'ordinateur. Une adresse est toujours un entier non négatif. L'adresse d'une variable peut être obtenue à l'aide de l'opérateur esperluette (&) qui représente l'adresse de la variable.

Définition : L'**allocation de mémoire** est le processus de réservation (virtuelle ou physique) d'un **espace informatique** ou d'une **emplacement** pour l'exécution d'un programme informatique.

Pointeurs en C++ (Partie 1)

Définition : Les pointeurs, ou variables de pointeur, sont des variables spéciales en C++ qui sont utilisées pour stocker des adresses. Un pointeur est une variable qui contient l'adresse mémoire d'une autre variable en tant que valeur.

Pointeurs en C++ (Partie 2)

On doit distinguer deux types principaux de définition d'un pointeur en C++ :

- **Définir un pointeur à la pile** : En C++, lors de la déclaration d'un pointeur qui pointe vers une variable locale ou une fonction, il est stocké à la pile. La pile est une région de mémoire utilisée pour les variables locales et les informations sur l'appel de fonctions. Vous pouvez en apprendre davantage sur la manière dont les appels de fonctions se résolvent dans un programme C++, y compris le traitement des variables locales et des pointeurs vers des variables locales, dans cet article.

Pointeurs en C++ (Partie 3)

- **Définir un pointeur à la heap** : En C++, vous pouvez déclarer un pointeur de manière dynamique en utilisant l'opérateur `new` ou d'autres fonctions intégrées telles que `malloc()`, `calloc()` ou `realloc()` fournies par la bibliothèque standard. Cela vous permet d'allouer dynamiquement de la mémoire sur le tas. Le tas est une région de mémoire utilisée pour l'allocation dynamique de mémoire.

Pointeurs - Déclaration

Supposons que nous ayons la déclaration d'une variable de type **typename** et de nom **var** :

```
// Déclaration :
typename var;
```

Ici, nous déclarons un pointeur de type **typename *** et de nom **pvar** :

```
// Déclaration :
typename * pvar;
```

Pointeurs - Détails

- Quelques notes importantes :
  - **pvar** est simplement une convention afin d'être cohérent concernant la dénomination de nos variables. Ici, nous commençons le nom **pvar** avec la lettre p pour indiquer qu'il s'agit d'un pointeur, pointant sur la variable **var**. Ainsi, **pvar** est le nom de notre pointeur.
  - **typename** doit être l'un des types intégrés : char, int, float, double.

Note importante : Définissons maintenant le pointeur **pvar** pour pointer sur la variable **var** :

```
// Définition :
pvar = &var;
```

Pointeurs - Notions fondamentales

- Un fait important à propos des pointeurs est :
  - ***pvar** se réfère à la valeur de **var**.
  - **pvar** se réfère à l'adresse de la variable **var** : (**&var**).
  - **&pvar** se réfère à l'adresse du pointeur (qui est simplement une variable possédant une adresse).

Pointeurs - Exemple

Expliquons maintenant avec un exemple concret comment cela se fait :

```
 // Déclaration d'une variable f
float f;
// Initialisation de f à 1.2
f = 1.2;
// Déclaration du pointeur pf
float * pf;
// Positionnement du pointeur pf à l'adresse de f
pf = &f;
// Maintenant, pf pointe vers f.
```

Pointeurs - Exemple (suite)

On peut définir directement un pointeur comme suit :

```
 // Déclaration d'une variable f
float f;
// Initialisation de f à 1.2
f = 1.2;
// Définition du pointeur pf
// Déclaration du pointeur pf et positionnement à l'adresse
de f
float * pf = &f;
// Maintenant, pf pointe vers f.
```

Un exemple plus concret est donné dans cet exemple : **pointersExample.c**

Arithmétique des Pointeurs - Partie 1

- L'arithmétique des pointeurs est une fonctionnalité en C++ qui permet d'effectuer des opérations arithmétiques sur les pointeurs.
- Lorsque vous effectuez des opérations arithmétiques sur des pointeurs, elles sont échelonnées en fonction de la taille du type de données auquel ils pointent.
- L'incrémentation d'un pointeur de un le déplace vers la prochaine adresse mémoire du type de données correspondant.

Arithmétique des Pointeurs - Partie 2

- De même, la décrémentation d'un pointeur de un le déplace vers l'adresse mémoire précédente.
- L'arithmétique des pointeurs est particulièrement utile lors de la manipulation d'arrays et de l'itération sur des éléments à l'aide de pointeurs.
- Cependant, il est important de faire preuve de prudence avec l'arithmétique des pointeurs pour éviter d'accéder à des emplacements mémoire invalides ou de provoquer un comportement indéfini.

Exemple d'Arithmétique des Pointeurs - Partie 1

- Considérons un exemple d'arithmétique des pointeurs et d'arrays.
- Supposons que nous ayons un tableau d'entiers `arr` avec 5 éléments.
- Nous pouvons définir un pointeur `ptr` qui pointe vers le premier élément du tableau.

Exemple d'Arithmétique des Pointeurs - Partie 2

- En utilisant l'arithmétique des pointeurs, nous pouvons itérer sur le tableau et afficher ses éléments de la manière suivante :

```
int* ptr = arr;
for (int i = 0; i < 5; i++) {
  cout << *(ptr + i) << " ";
}
```

- Cet extrait de code montre comment nous pouvons accéder aux éléments du tableau en utilisant l'arithmétique des pointeurs, en incrémentant le pointeur de la taille du type de données (entier dans ce cas) à chaque itération.

Pointeurs et Arrays - Partie 1

- En C++, les arrays et les pointeurs ont une relation étroite.
- Dans la plupart des cas, le nom d'un array se comporte comme un pointeur constant qui pointe vers le premier élément du tableau.

Pointeurs et Arrays - Partie 2

- Vous pouvez utiliser des pointeurs pour accéder aux éléments d'un tableau en utilisant l'arithmétique des pointeurs.
- Par exemple, étant donné un tableau `arr`, vous pouvez accéder à ses éléments en utilisant un pointeur comme `*(arr + i)`, où `i` est l'indice.

Pointeurs et Arrays - Partie 3

- Les pointeurs offrent une manière d'itérer sur les arrays, de manipuler leurs éléments et d'effectuer diverses opérations efficacement.
- Cependant, il est important de manipuler avec soin les pointeurs et les arrays pour éviter d'accéder à des zones mémoire hors limites ou de provoquer un comportement indéfini.

Pointeurs et Structures - Partie 1

- Les pointeurs peuvent également être utilisés avec des structures en C++.
- Les pointeurs vers des structures permettent l'allocation dynamique et la manipulation d'objets de structure.

Pointeurs et Structures - Partie 2

- Vous pouvez allouer dynamiquement de la mémoire pour une structure en utilisant l'opérateur `new` ou d'autres fonctions appropriées d'allocation de mémoire.
- L'accès aux membres d'une structure à l'aide d'un pointeur se fait à l'aide de l'opérateur de flèche (->).

Pointeurs et Structures - Partie 3

- Par exemple, si `ptr` est un pointeur vers une structure `st`, vous pouvez accéder à une variable membre `member` comme `ptr->member`.

Pointeurs et Structures - Partie 4

- Les pointeurs vers des structures sont particulièrement utiles lors de la manipulation de données allouées dynamiquement ou lors du passage efficace de grandes structures à des fonctions.

Introduction aux Smart Pointers en C++

- En C++, la gestion manuelle de la mémoire peut être délicate et sujette à des erreurs telles que les fuites de mémoire et les pointeurs invalides.
- Les smart pointers sont une fonctionnalité introduite dans le langage C++ pour simplifier la gestion de la mémoire et améliorer la sécurité des programmes.
- Les smart pointers sont des objets qui agissent comme des pointeurs classiques, mais avec des fonctionnalités supplémentaires de gestion automatique de la mémoire.
- Ils sont définis dans l'en-tête '<memory>' et offrent trois principaux types : 'std::unique_ptr', 'std::shared_ptr', et 'std::weak_ptr'.

Les Trois Principaux Types de Smart Pointers

- **std::unique_ptr** : Possède la propriété exclusive de la ressource pointée. La ressource est automatiquement libérée lorsque le ‘std::unique_ptr‘ est détruit.
- **std::shared_ptr** : Peut partager la propriété de la ressource avec d’autres ‘std::shared_ptr‘. La ressource est libérée lorsque le dernier ‘std::shared_ptr‘ pointant vers elle est détruit.
- **std::weak_ptr** : Comme ‘std::shared_ptr‘, mais ne contribue pas au comptage des références. Utilisé pour éviter les références circulaires dans les structures de données partagées.

Smart Pointers en C++ - std::unique_ptr

```cpp
#include <memory>

int main() {
    std::unique_ptr<int> pInt = std::make_unique<int>(42);
    // Utilisation de pInt comme un pointeur normal
    // La mémoire est automatiquement libérée à la sortie du bloc
} // La mémoire est libérée ici
```

Smart Pointers en C++ - std::shared_ptr

```cpp
#include <memory>

class MyClass {
public:
    void DoSomething() {
        // Fonctionnalités de la classe
    }
};

int main() {
    std::shared_ptr<MyClass> sharedPtr = std::make_shared<MyClass
        >();
    sharedPtr->DoSomething();
    // La mémoire est partagée entre tous les shared_ptr
} // La mémoire est libérée lorsque le dernier shared_ptr est
    détruit
```

Smart Pointers en C++ - std::weak_ptr

```cpp
#include <memory>

std::weak_ptr<int> weakPtr;

void SomeFunction() {
    std::shared_ptr<int> sharedPtr = std::make_shared<int>(10);
    weakPtr = sharedPtr;
}

// weakPtr peut être vérifié avant utilisation avec lock()
```

Gestion de la Mémoire et Lien avec le C

- En C++, l'utilisation de smart pointers (comme std::unique_ptr et std::shared_ptr) simplifie la gestion de la mémoire.
- Ces smart pointers libèrent automatiquement la mémoire lorsqu'ils ne sont plus nécessaires, évitant ainsi les fuites de mémoire.
- En C, la gestion de la mémoire repose généralement sur les fonctions malloc() et free(), nécessitant une gestion manuelle et propice aux erreurs.
- L'utilisation de smart pointers en C++ simplifie grandement cette tâche et améliore la sécurité et la robustesse des programmes.

Declaration of Arrays in C++

```cpp
#include <iostream>

int main() {
    // Declaration of a static array of 5 integers
    int staticArray[5];

    // Initialization of the elements in the array
    staticArray[0] = 10;
    staticArray[1] = 20;
    staticArray[2] = 30;
    staticArray[3] = 40;
    staticArray[4] = 50;

    // Access and display elements in the array
    for (int i = 0; i < 5; ++i) {
        std::cout << staticArray[i] << " ";
```

Using Pointers with Arrays in C++

```cpp
#include <iostream>

int main() {
    // Declaration of a dynamic array of 5 integers
    int* dynamicArray = new int[5];

    // Initialization of the elements in the array
    for (int i = 0; i < 5; ++i) {
        dynamicArray[i] = (i + 1) * 10;
    }

    // Access and display elements in the array using a pointer
    for (int i = 0; i < 5; ++i) {
        std::cout << *(dynamicArray + i) << "␣";
    }

```

Chaînes de caractères

Définition : Une **chaîne de caractères** en C est un tableau de caractères terminé par un caractère nul.

- La définition suivante crée une chaîne de caractères nommée **myString** qui se compose du mot ”Programming”.
- Afin de contenir le caractère terminateur nul à la fin du tableau, la taille du tableau de caractères qui contient la chaîne **doit être** d’un caractère de plus que le nombre de caractères dans le mot ”Programming”.

```
1  char myString[12] = {'P','r','o','g','r','a','m','m','i','n','g','\0'};
```

Chaînes de caractères

- On peut déclarer la même chaîne en suivant les règles d'initialisation des tableaux (il s'agira d'une **chaîne littérale**) et sans écrire le caractère terminateur nul. L'exemple suivant met en évidence comment faire :

```
1  char myString[] = "Programming";
```

Voici la représentation mémoire du tableau de caractères "Programming" précédemment défini :

Figure: Représentation d'une chaîne de caractères en mémoire

Chaînes de caractères

- Le fichier d'en-tête **#include<string.h>** contient plusieurs fonctions pouvant être utilisées avec des chaînes de caractères et des littéraux de chaînes. Voici une liste non exhaustive :

  - **strcpy(s1, s2);** Copie la chaîne s2 dans la chaîne s1.
  - **strcat(s1, s2);** Concatène la chaîne s2 à la fin de la chaîne s1.
  - **strlen(s);** Retourne la longueur de la chaîne s.
  - **strcmp(s1, s2);** Retourne 0 si s1 et s2 contiennent les mêmes caractères ; un nombre négatif si s1<s2 et un nombre positif si s1>s2. Plus de détails peuvent être trouvés dans **ce lien**.
  - **strchr(s, c);** Retourne un pointeur vers la première occurrence du caractère c dans la chaîne s.
  - **strstr(s1, s2);** Retourne un pointeur vers la première occurrence de la chaîne s2 dans la chaîne s1.