

PRÉSENTATION DE C++

HISTORIQUE DE C++

Le C++ a été développé par Bjarne Stroustrup en 1979. Il est une extension du langage C. Le C++ a été standardisé en 1998 par l'ISO. Les versions majeures incluent C++98, C++03, C++11, C++14, C++17, et C++20. C++ est utilisé pour la programmation système, les jeux vidéo, et les applications haute performance.

CARACTÉRISTIQUES PRINCIPALES DE C++

- Langage compilé
- Support de la programmation orientée objet (POO)
- Gestion manuelle de la mémoire
- Support des templates et de la programmation générique
- Compatibilité avec le C
- Exécution rapide et efficace
- Large bibliothèque standard (STL)

APPLICATIONS COURANTES DE C++

- Développement de systèmes d'exploitation
- Jeux vidéo et moteurs de jeux
- Applications financières et de trading
- Logiciels embarqués
- Simulations scientifiques et techniques
- Applications de bureau et GUI
- Bases de données et serveurs

COMPARAISON AVEC D'AUTRES LANGAGES

Langage	Caractéristiques principales
C	Performant, bas niveau, peu de fonctionnalités de POO
Java	POO, gestion automatique de la mémoire, portable
Python	Langage interprété, syntaxe simple, moins performant
C#	POO, gestion automatique de la mémoire, intégré à .NET
Rust	Sécurité de la mémoire, moderne, concurrentiel

PHILOSOPHIE DE CONCEPTION DE C++

- Efficacité et performance
- Flexibilité et contrôle
- Compatibilité avec le C
- Support de multiples paradigmes de programmation
- Équilibre entre abstraction et performance
- Évolutivité et extensibilité
- Priorité à la sécurité et à la robustesse du code

ÉVOLUTION DE C++

- C++98 : Première norme ISO
- C++03 : Corrections mineures et clarifications
- C++11 : Améliorations majeures (lambda, auto, smart pointers)
- C++14 : Améliorations mineures et nouvelles fonctionnalités
- C++17 : Nouvelles fonctionnalités (std::optional, std::variant)
- C++20 : Concepts, coroutines, ranges, modules
- C++23 : Améliorations continues et nouvelles fonctionnalités

LE C++ DANS L'ENVIRONNEMENT QT

- Qt est un framework C++ pour le développement d'applications GUI
- Supporte le développement multiplateforme (Windows, macOS, Linux)
- Utilise des signaux et des slots pour la communication entre objets
- Offre des widgets et des composants graphiques avancés
- Intégré avec des outils comme Qt Creator pour le développement
- Utilisé dans des applications comme KDE, Autodesk Maya, et VirtualBox
- Supporte également le développement mobile (Android, iOS)

UTILISATION DE MAKEFILE

DÉFINITION

Un Makefile est un fichier utilisé par l'outil `make`. Il automatise le processus de compilation. Il contient des instructions pour compiler et lier un programme. Il est couramment utilisé dans les projets C++.

POURQUOI UTILISER UN MAKEFILE

Automatisation de la compilation. Gérer les dépendances entre fichiers. Faciliter le processus de construction du projet. Améliorer l'efficacité et réduire les erreurs humaines.

STRUCTURE D'UN MAKEFILE

Un Makefile se compose de cibles (targets), dépendances et commandes. Format de base :

```
target: dependencies  
        command
```

Chaque ligne de commande doit commencer par une tabulation.

VARIABLES DANS UN MAKEFILE

Les variables simplifient la gestion des chemins et options. Déclaration de variable :

```
CC = g++
CFLAGS = -Wall -g
```

Utilisation :

```
$(CC) $(CFLAGS) -o output file.cpp
```

CIBLES (TARGETS)

Une cible est un fichier ou une action. Elle peut être un fichier exécutable ou une étape intermédiaire.

Exemple :

```
all: myprogram
```

La cible "all" dépend de "myprogram".

DÉPENDANCES

Les dépendances spécifient les fichiers nécessaires pour créer une cible. Exemple :

```
myprogram: main.o utils.o
```

Ici, "myprogram" dépend de "main.o" et "utils.o".

COMMANDES

Les commandes sont les actions à exécuter pour créer une cible. Elles doivent être précédées d'une tabulation. Exemple :

```
main.o: main.cpp  
g++ -c main.cpp
```

Compile "main.cpp" en "main.o".

EXEMPLE DE MAKEFILE SIMPLE

```
CC = g++
CFLAGS = -Wall -g

all: myprogram

myprogram: main.o utils.o
    $(CC) $(CFLAGS) -o myprogram main.o utils.o

main.o: main.cpp
    $(CC) $(CFLAGS) -c main.cpp

utils.o: utils.cpp
    $(CC) $(CFLAGS) -c utils.cpp

clean:
```

MAKEFILE POUR PROJETS MULTI-FICHIERS

Utilisation de variables pour les fichiers sources et objets :

```
CC = g++
CFLAGS = -Wall -g
SRCS = main.cpp utils.cpp
OBJS = $(SRCS:.cpp=.o)

all: myprogram

myprogram: $(OBJS)
    $(CC) $(CFLAGS) -o myprogram $(OBJS)

.cpp.o:
    $(CC) $(CFLAGS) -c $<

clean:
    rm -f $(OBJS) myprogram
```

MAIN, ARGC ET ARGV

DEFINITION

La fonction `main` est le point d'entrée d'un programme C++. Elle est automatiquement appelée au démarrage du programme. Les paramètres `argc` et `argv` permettent de gérer les arguments passés en ligne de commande.

STRUCTURE DE LA FONCTION MAIN

```
int main(int argc, char *argv[]) {  
    // Code du programme  
    return 0;  
}
```

`argc` est un entier qui représente le nombre d'arguments. `argv` est un tableau de chaînes de caractères représentant les arguments.

ROLE DE ARGC

`argc` (argument count) est un entier. Il indique le nombre total d'arguments passés au programme. `argc` inclut le nom du programme comme premier argument. Par exemple, si le programme est appelé avec `./prog arg1 arg2`, alors `argc` vaut 3.

ROLE DE ARGV

`argv` (argument vector) est un tableau de chaînes de caractères. Chaque élément du tableau contient un argument passé en ligne de commande. `argv[0]` est toujours le nom du programme. Les autres éléments (`argv[1]`, `argv[2]`, etc.) contiennent les arguments supplémentaires.

EXEMPLE DE PROGRAMME AVEC ARGC ET ARGV

```
#include <iostream>

int main(int argc, char *argv[]) {
    std::cout << "Nombre d'arguments: " << argc << std::endl;
    for (int i = 0; i < argc; ++i) {
        std::cout << "Argument " << i << ": " << argv[i] << std::endl;
    }
    return 0;
}
```

GESTION DES ARGUMENTS EN LIGNE DE COMMANDE

- Les arguments sont passés au programme lors de son exécution.
- `argc` permet de savoir combien d'arguments ont été passés.
- `argv` permet d'accéder aux valeurs des arguments.
- Utilisez des boucles pour traiter les arguments.
- Vérifiez toujours `argc` avant d'accéder à `argv` pour éviter les erreurs.

LES VARIABLES ET TYPES DE DONNÉES

DEFINITION

Les variables sont des emplacements de mémoire nommés. Elles stockent des valeurs qui peuvent être modifiées pendant l'exécution du programme. Les types de données définissent le type de valeur qu'une variable peut contenir. Les variables permettent de manipuler et de stocker des informations.

DECLARATION DE VARIABLE

La déclaration de variable en C++ suit la syntaxe suivante :

```
type nom_de_variable;
```

Exemple :

```
int age;  
double salaire;
```

TYPES DE DONNÉES DE BASE

Les types de données de base en C++ incluent :

- `int` : entier
- `float` : nombre à virgule flottante simple précision
- `double` : nombre à virgule flottante double précision
- `char` : caractère
- `bool` : booléen (vrai ou faux)

INITIALISATION DE VARIABLE

L'initialisation de variable consiste à attribuer une valeur initiale lors de la déclaration :

```
int age = 30;
double salaire = 45000.50;
char initiale = 'A';
bool estEtudiant = true;
```

CONVERSION DE TYPES

La conversion de types change le type de données d'une variable. En C++, il existe deux types de conversion :

1. Conversion implicite (automatique)
2. Conversion explicite (cast)

Exemple de conversion explicite :

```
int a = 10;
double b = static_cast<double>(a);
```

CONSTANTES

Les constantes sont des variables dont la valeur ne peut pas être modifiée après l'initialisation. Elles sont déclarées avec le mot-clé **const** :

```
const int MAX_AGE = 100;  
const double PI = 3.14159;
```

VARIABLES LOCALES ET GLOBALES

Les variables locales sont déclarées à l'intérieur d'une fonction. Elles sont accessibles uniquement dans cette fonction.

Les variables globales sont déclarées en dehors de toutes les fonctions. Elles sont accessibles par toutes les fonctions du programme.

PORTÉE DES VARIABLES

La portée d'une variable détermine où elle peut être utilisée dans le code. Les variables locales ont une portée limitée à la fonction où elles sont déclarées. Les variables globales ont une portée qui s'étend à tout le programme.

LES OPÉRATEURS

DÉFINITION

Les opérateurs en C++ sont des symboles qui indiquent au compilateur d'effectuer des opérations spécifiques. Ils sont utilisés pour manipuler des variables et des valeurs. Les opérateurs peuvent être classés en différentes catégories selon leur fonction.

OPÉRATEURS ARITHMÉTIQUES

Opérateur	Description	Exemple
+	Addition	$a + b$
-	Soustraction	$a - b$
*	Multiplication	$a * b$
/	Division	a / b
%	Modulo	$a \% b$

OPÉRATEURS DE COMPARAISON

Opérateur	Description	Exemple
<code>==</code>	Égal à	<code>a == b</code>
<code>!=</code>	Différent de	<code>a != b</code>
<code>></code>	Supérieur à	<code>a > b</code>
<code><</code>	Inférieur à	<code>a < b</code>
<code>>=</code>	Supérieur ou égal à	<code>a >= b</code>
<code><=</code>	Inférieur ou égal à	<code>a <= b</code>

OPÉRATEURS LOGIQUES

Opérateur	Description	Exemple
&&	ET logique	a && b
'		'
!	NON logique	! a

OPÉRATEURS D'AFFECTION

Opérateur	Description	Exemple
=	Affectation simple	a = b
+=	Addition et affectation	a += b
-=	Soustraction et affectation	a -= b
*=	Multiplication et affectation	a *= b
/=	Division et affectation	a /= b
%=	Modulo et affectation	a %= b

OPÉRATEURS BIT À BIT

Opérateur	Description	Exemple
&	ET bit à bit	a & b
'	'	OU bit à bit
^	OU exclusif bit à bit	a ^ b
~	NON bit à bit	~a
<<	Décalage à gauche	a << b
>>	Décalage à droite	a >> b

OPÉRATEURS D'INCRÉMENT ET DE DÉCRÉMENT

Opérateur	Description	Exemple
<code>++</code>	Incrément (pré/post)	<code>++a</code> ou <code>a++</code>
<code>--</code>	Décrément (pré/post)	<code>--a</code> ou <code>a--</code>

PRIORITÉ DES OPÉRATEURS

La priorité des opérateurs détermine l'ordre dans lequel les opérations sont effectuées. Les opérateurs avec une priorité plus élevée sont évalués avant ceux avec une priorité plus basse. Les parenthèses () peuvent être utilisées pour modifier l'ordre d'évaluation.

EXEMPLES D'UTILISATION

```
int a = 5, b = 10;
int sum = a + b; // Addition
bool isEqual = (a == b); // Comparaison
bool result = (a < b) && (b > 0); // Opérateurs logiques
a += 2; // Opérateur d'affectation
int bitwiseAnd = a & b; // Opérateur bit à bit
a++; // Incrément
```

LES INSTRUCTIONS CONDITIONNELLES

DEFINITION

Les instructions conditionnelles permettent de prendre des décisions dans le code. Elles exécutent des blocs de code en fonction de conditions spécifiées. Les principales instructions conditionnelles en C++ sont **if**, **if-else**, **if-else if-else** et **switch**.

IF STATEMENT

L'instruction `if` exécute un bloc de code si une condition est vraie. Syntaxe de base :

```
if (condition) {  
    // code à exécuter si la condition est vraie  
}
```

IF-ELSE STATEMENT

L'instruction `if-else` exécute un bloc de code si une condition est vraie et un autre bloc si elle est fausse.

Syntaxe de base :

```
if (condition) {  
    // code à exécuter si la condition est vraie  
} else {  
    // code à exécuter si la condition est fausse  
}
```

IF-ELSE IF-ELSE STATEMENT

L'instruction `if-else if-else` permet de tester plusieurs conditions. Syntaxe de base :

```
if (condition1) {  
    // code à exécuter si condition1 est vraie  
} else if (condition2) {  
    // code à exécuter si condition2 est vraie  
} else {  
    // code à exécuter si aucune condition n'est vraie  
}
```

NESTED IF STATEMENTS

Les instructions `if` peuvent être imbriquées pour tester des conditions multiples. Syntaxe de base :

```
if (condition1) {  
    if (condition2) {  
        // code à exécuter si les deux conditions sont vraies  
    }  
}
```

SWITCH STATEMENT

L'instruction `switch` sélectionne un bloc de code à exécuter parmi plusieurs options. Syntaxe de base :

```
switch (expression) {  
    case valeur1:  
        // code à exécuter si expression == valeur1  
        break;  
    case valeur2:  
        // code à exécuter si expression == valeur2  
        break;  
    // autres cas  
    default:  
        // code à exécuter si aucune valeur ne correspond  
}
```

SYNTAXE

Les instructions conditionnelles en C++ utilisent les opérateurs de comparaison :

- `==` : égal à
- `!=` : différent de
- `>` : plus grand que
- `<` : plus petit que
- `>=` : plus grand ou égal à
- `<=` : plus petit ou égal à

LES BOUCLES

DÉFINITION

Les boucles permettent de répéter une série d'instructions plusieurs fois. Elles sont utilisées pour automatiser les tâches répétitives. Il existe plusieurs types de boucles en C++. Les boucles peuvent être conditionnées ou basées sur un compteur. Elles améliorent l'efficacité et la lisibilité du code.

TYPES DE BOUCLES

- Boucle while
- Boucle for
- Boucle do-while

BOUCLE WHILE

La boucle `while` répète une série d'instructions tant qu'une condition est vraie. Syntaxe :

```
while (condition) {  
    // instructions  
}
```

BOUCLE FOR

La boucle **for** est utilisée pour un nombre déterminé de répétitions. Syntaxe :

```
for (initialisation; condition; incrément) {  
    // instructions  
}
```

BOUCLE DO-WHILE

La boucle `do-while` exécute les instructions au moins une fois avant de vérifier la condition. Syntaxe :

```
do {  
    // instructions  
} while (condition);
```

COMPARAISON DES BOUCLES

Type de boucle	Condition vérifiée	Utilisation principale
while	Avant la boucle	Répétitions avec condition
for	Avant chaque itération	Répétitions avec compteur
do-while	Après la boucle	Au moins une exécution

EXEMPLE DE BOUCLE WHILE

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


UTILISATION DES BOUCLES IMBRIQUÉES

Les boucles imbriquées permettent de répéter des instructions dans une autre boucle. Exemple :

```
for (int i = 0; i < 3; i++) {  
    for (int j = 0; j < 3; j++) {  
        std::cout << i << "," << j << std::endl;  
    }  
}
```

LES FONCTIONS

DÉFINITION

Une fonction est un bloc de code réutilisable qui exécute une tâche spécifique. Les fonctions permettent de structurer le code en modules plus petits et plus gérables. Elles peuvent prendre des paramètres et retourner une valeur. Les fonctions sont définies une fois et peuvent être appelées plusieurs fois.

DÉCLARATION ET DÉFINITION

La déclaration d'une fonction informe le compilateur de son existence. La définition d'une fonction fournit le corps de la fonction. Syntaxe de déclaration :

```
retour nomFonction(type param1, type param2);
```

Syntaxe de définition :

```
retour nomFonction(type param1, type param2) {  
    // corps de la fonction  
}
```

PARAMÈTRES ET ARGUMENTS

Les paramètres sont des variables définies dans la déclaration de la fonction. Les arguments sont les valeurs réelles passées à la fonction lors de son appel. Exemple de fonction avec paramètres :

```
void afficherMessage(string message) {  
    cout << message << endl;  
}
```

Appel de la fonction avec argument :

```
afficherMessage("Bonjour!");
```

VALEUR DE RETOUR

La valeur de retour est le résultat qu'une fonction renvoie à son appelant. Le type de retour est spécifié dans la déclaration de la fonction. Exemple de fonction avec valeur de retour :

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

Appel de la fonction et utilisation de la valeur de retour :

```
int resultat = addition(3, 4);
```

APPEL DE FONCTION

Pour appeler une fonction, écrivez son nom suivi de parenthèses contenant les arguments. Exemple d'appel de fonction :

```
nomFonction(argument1, argument2);
```

Les arguments doivent correspondre aux types et à l'ordre des paramètres.

SURCHARGE DE FONCTIONS

La surcharge permet de définir plusieurs fonctions avec le même nom mais des paramètres différents.
Exemple de surcharge :

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

Le compilateur choisit la fonction appropriée en fonction des types des arguments.

FONCTIONS RÉCURSIVES

Une fonction récursive s'appelle elle-même. Exemple de fonction récursive pour calculer la factorielle :

```
int factorielle(int n) {
    if (n <= 1) return 1;
    return n * factorielle(n - 1);
}
```

Les fonctions récursives doivent avoir une condition d'arrêt pour éviter les boucles infinies.

PORTÉE DES VARIABLES

La portée d'une variable détermine où elle peut être utilisée dans le code. Les variables locales sont définies à l'intérieur d'une fonction et ne sont accessibles qu'à l'intérieur de cette fonction. Les variables globales sont définies en dehors de toutes les fonctions et sont accessibles partout dans le programme.

AVANTAGES

- Réutilisabilité du code
- Meilleure organisation et lisibilité
- Facilite le débogage et la maintenance

INCONVÉNIENTS

- Peut augmenter la complexité si mal utilisé
- Risque de surcharge excessive
- Les fonctions récursives peuvent consommer beaucoup de mémoire

LES TABLEAUX

DÉFINITION

Les tableaux en C++ sont des structures de données qui permettent de stocker plusieurs valeurs du même type. Ils sont utilisés pour gérer des collections de données de manière efficace. Chaque élément d'un tableau est accessible via un indice.

DÉCLARATION

La déclaration d'un tableau en C++ se fait comme suit :

```
type nomTableau[taille];
```

Par exemple :

```
int monTableau[10];
```

INITIALISATION

Un tableau peut être initialisé lors de sa déclaration :

```
int monTableau[5] = {1, 2, 3, 4, 5};
```

Si la taille est omise, elle est déterminée par le nombre d'éléments fournis :

```
int monTableau[] = {1, 2, 3, 4, 5};
```

ACCÈS AUX ÉLÉMENTS

Les éléments d'un tableau sont accessibles via leur indice, en commençant par 0 :

```
int premierElement = monTableau[0];
int deuxièmeElement = monTableau[1];
```

TAILLE FIXE

La taille d'un tableau en C++ est fixe et doit être connue à la compilation. Elle ne peut pas être modifiée après la déclaration :

```
const int TAILLE = 10;  
int monTableau[TAILLE];
```

BOUCLE AVEC TABLEAUX

Les tableaux sont souvent parcourus à l'aide de boucles :

```
for (int i = 0; i < 5; i++) {  
    cout << monTableau[i] << endl;  
}
```

TABLEAUX MULTIDIMENSIONNELS

Les tableaux peuvent avoir plusieurs dimensions :

```
int tableau2D[3][4];
```

L'accès aux éléments se fait de manière similaire :

```
int element = tableau2D[1][2];
```

LIMITES ET PRÉCAUTIONS

- Les indices de tableau commencent à 0.
- Accéder à un indice hors limites provoque un comportement indéfini.
- La taille des tableaux doit être connue à la compilation.

TABLEAUX VS VECTEURS (STD::VECTOR)

Caractéristique	Tableaux	std::vector
Taille	Fixe	Dynamique
Allocation mémoire	Statique	Dynamique
Flexibilité	Moins flexible	Plus flexible
Utilisation	Bas niveau	Haut niveau
Gestion de la taille	Manuelle	Automatique

LES POINTEURS

DÉFINITION

Un pointeur est une variable qui stocke l'adresse mémoire d'une autre variable. Les pointeurs permettent de manipuler directement la mémoire. Ils sont utilisés pour l'allocation dynamique de mémoire et la manipulation de tableaux et de chaînes de caractères.

DÉCLARATION DE POINTEUR

La syntaxe pour déclarer un pointeur est la suivante :

```
type *nom_du_pointeur;
```

Exemple :

```
int *pointeur;
```

OPÉRATEUR D'ADRESSE (&)

L'opérateur d'adresse (&) est utilisé pour obtenir l'adresse d'une variable. Exemple :

```
int var = 10;  
int *pointeur = &var;
```

OPÉRATEUR DE DÉRÉFÉRENCEMENT (*)

L'opérateur de déréférencement (*) est utilisé pour accéder à la valeur stockée à l'adresse pointée par le pointeur. Exemple :

```
int var = 10;  
int *pointeur = &var;  
int valeur = *pointeur; // valeur est 10
```

POINTEURS ET TABLEAUX

Les pointeurs peuvent être utilisés pour parcourir les éléments d'un tableau. Exemple :

```
int tableau[3] = {1, 2, 3};  
int *pointeur = tableau;
```

POINTEURS ET FONCTIONS

Les pointeurs peuvent être passés aux fonctions pour modifier les valeurs des arguments. Exemple :

```
void increment(int *p) {  
    (*p)++;  
}
```

POINTEURS ET MÉMOIRE DYNAMIQUE

Les pointeurs sont utilisés pour allouer et libérer de la mémoire dynamique. Exemple :

```
int *p = new int;  
delete p;
```

POINTEURS NULLS

Un pointeur null est un pointeur qui ne pointe vers aucune adresse valide. Il est initialisé avec la valeur `nullptr`.

```
int *pointeur = nullptr;
```

POINTEURS CONSTANTS

Un pointeur constant ne peut pas être modifié pour pointer vers une autre adresse. Exemple :

```
int var = 10;  
int *const pointeur = &var;
```

AVANTAGES ET INCONVÉNIENTS

Avantages :

- Accès direct à la mémoire
- Manipulation efficace des tableaux et chaînes

Inconvénients :

- Complexité accrue
- Risque de fuites de mémoire et erreurs de segmentation

LES CHAÎNES DE CARACTÈRES

DÉFINITION

Les chaînes de caractères en C++ sont des séquences de caractères. Elles peuvent être manipulées comme des tableaux de caractères. Elles se terminent par un caractère nul \0. Elles peuvent aussi être gérées via la classe `std::string` de la STL. Les chaînes de caractères permettent de manipuler du texte.

DÉCLARATION ET INITIALISATION

En C++ on peut déclarer des chaînes de caractères de deux façons :

1. Utiliser un tableau de caractères :

```
char str[] = "Hello";
```

2. Utiliser la classe std::string :

```
std::string str = "Hello";
```

OPÉRATIONS DE BASE

Les opérations de base sur les chaînes de caractères incluent :

- Accès aux caractères individuels
- Modification des caractères
- Mesure de la longueur de la chaîne
- Copie de chaînes

CONCATENATION

La concaténation permet de combiner deux chaînes :

1. Avec `std::string` :

```
std::string str1 = "Hello, ";
std::string str2 = "World!";
std::string result = str1 + str2;
```

2. Avec `char` :

```
char str1[20] = "Hello, ";
char str2[] = "World!";
strcat(str1, str2);
```

COMPARAISON

Les chaînes de caractères peuvent être comparées de plusieurs façons :

1. Avec `std::string` :

```
std::string str1 = "Hello";
std::string str2 = "World";
bool result = (str1 == str2);
```

2. Avec `char` :

```
char str1[] = "Hello";
char str2[] = "World";
int result = strcmp(str1, str2)
```

MÉTHODES COURANTES

Quelques méthodes courantes de `std::string` :

- `length()` : Retourne la longueur de la chaîne
- `substr()` : Retourne un sous-ensemble de la chaîne
- `find()` : Trouve la position d'une sous-chaîne
- `replace()` : Remplace une partie de la chaîne
- `append()` : Ajoute une chaîne à la fin

CONVERSION ENTRE CHAÎNES ET AUTRES TYPES

La conversion entre chaînes et autres types peut se faire via des fonctions :

- `std::stoi` : Convertit une chaîne en entier
- `std::stof` : Convertit une chaîne en float
- `std::to_string` : Convertit un entier/float en chaîne

CHAÎNES DE CARACTÈRES ET POINTEURS

Les chaînes de caractères peuvent être manipulées avec des pointeurs :

```
char *str = "Hello";
while (*str != '\0') {
    std::cout << *str;
    str++;
}
```

Les pointeurs permettent de parcourir et modifier les chaînes.

GESTION DE LA MÉMOIRE

En C++, la gestion de la mémoire pour les chaînes se fait de deux façons :

1. Avec `char` :

```
char *str = new char[20];
// Ne pas oublier de libérer la mémoire
delete[] str;
```

2. Avec `std::string` : La gestion est automatique.

AVANTAGES :

- Flexibilité dans la manipulation des chaînes
- `std::string` simplifie la gestion de la mémoire

INCONVÉNIENTS :

- Les chaînes de type **char** nécessitent une gestion manuelle de la mémoire
- Les opérations peuvent être moins intuitives avec **char**

LES STRUCTURES

DÉFINITION

Une structure en C++ est une collection de variables de différents types. Elle permet de regrouper des données sous un même nom. Les structures facilitent l'organisation et la gestion des données. Elles sont définies à l'aide du mot-clé **struct**. Les structures sont similaires aux classes, mais par défaut, leurs membres sont publics.

SYNTAXE

La syntaxe de définition d'une structure est la suivante :

```
struct NomStructure {  
    type_membre1 nom_membre1;  
    type_membre2 nom_membre2;  
    // autres membres  
};
```

DÉCLARATION DE STRUCTURE

Pour déclarer une structure, utilisez la syntaxe suivante :

```
struct Personne {  
    std::string nom;  
    int age;  
};
```

ACCÈS AUX MEMBRES

Pour accéder aux membres d'une structure, utilisez l'opérateur . :

```
Personne p;  
p.nom = "Alice";  
p.age = 30;
```

INITIALISATION

Vous pouvez initialiser une structure lors de sa déclaration :

```
Personne p = {"Alice", 30};
```

STRUCTURES IMBRIQUÉES

Les structures peuvent contenir d'autres structures :

```
struct Adresse {  
    std::string rue;  
    std::string ville;  
};  
  
struct Personne {  
    std::string nom;  
    int age;  
    Adresse adresse;  
};
```

UTILISATION AVEC FONCTIONS

Les structures peuvent être passées à des fonctions :

```
void afficherPersonne(Personne p) {
    std::cout << p.nom << ", " << p.age << " ans" << std::endl;
}
```

AVANTAGES ET INCONVÉNIENTS

Avantages :

- Organisation des données
- Regroupement logique des variables

Inconvénients :

- Pas de méthodes (comme les classes)
- Membres publics par défaut

LES CLASSES ET OBJETS

DÉFINITION

Une classe est un modèle pour créer des objets. Elle définit des propriétés (membres) et des comportements (méthodes). Les objets sont des instances de classes. Les classes permettent de regrouper des données et des fonctions.

MEMBRES DE CLASSE

Les membres de classe sont les variables définies dans une classe. Ils peuvent être :

- Publics : accessibles depuis l'extérieur de la classe.
- Privés : accessibles uniquement depuis la classe elle-même.
- Protégés : accessibles depuis la classe et ses dérivées.

MÉTHODES DE CLASSE

Les méthodes de classe sont des fonctions définies dans une classe. Elles peuvent manipuler les membres de la classe. Les méthodes peuvent être publiques, privées ou protégées. Les méthodes spéciales incluent les constructeurs et destructeurs.

ACCÈS AUX MEMBRES

Les membres publics sont accessibles directement :

```
objet.membre_public;
```

Les membres privés et protégés sont accessibles via des méthodes publiques :

```
objet.methode_publique();
```

EXEMPLE DE CLASSE

```
class Person {
public:
    string name;
    int age;

    void display() {
        cout << "Name: " << name << ", Age: " << age << endl;
    }
};
```

OBJETS ET INSTANCIATION

Pour créer un objet d'une classe :

```
Person person1;  
person1.name = "Alice";  
person1.age = 30;  
person1.display();
```

Chaque objet a ses propres copies des membres de la classe.

LES CONSTRUCTEURS ET DESTRUCTEURS

DÉFINITION DES CONSTRUCTEURS

Un constructeur est une fonction membre spéciale d'une classe. Il est appelé automatiquement lors de la création d'un objet de cette classe. Son rôle principal est d'initialiser les objets. Il porte le même nom que la classe. Il ne retourne pas de valeur, pas même `void`. Il peut avoir des paramètres pour initialiser les objets avec des valeurs spécifiques.

SYNTAXE DES CONSTRUCTEURS

La syntaxe d'un constructeur en C++ est la suivante :

```
class NomDeClasse {
public:
    NomDeClasse() {
        // Corps du constructeur
    }
};
```

TYPES DE CONSTRUCTEURS

Il existe plusieurs types de constructeurs :

- Constructeur par défaut
- Constructeur avec paramètres
- Constructeur de copie
- Constructeur de déplacement

CONSTRUCTEUR PAR DÉFAUT

Un constructeur par défaut est un constructeur sans paramètres. Il est utilisé pour initialiser les objets avec des valeurs par défaut. Si aucun constructeur n'est défini, le compilateur génère un constructeur par défaut.

```
class Exemple {  
public:  
    Exemple() {  
        // Initialisation par défaut  
    }  
};
```

CONSTRUCTEUR AVEC PARAMÈTRES

Un constructeur avec paramètres permet d'initialiser les objets avec des valeurs spécifiques. Il prend des arguments pour initialiser les membres de la classe.

```
class Exemple {  
public:  
    Exemple(int x, int y) {  
        // Initialisation avec paramètres  
    }  
};
```

DÉFINITION DES DESTRUCTEURS

Un destructeur est une fonction membre spéciale d'une classe. Il est appelé automatiquement lors de la destruction d'un objet de cette classe. Son rôle principal est de libérer les ressources allouées. Il porte le même nom que la classe, précédé d'un tilde (~). Il ne prend pas de paramètres et ne retourne pas de valeur.

SYNTAXE DES DESTRUCTEURS

La syntaxe d'un destructeur en C++ est la suivante :

```
class NomDeClasse {
public:
    ~NomDeClasse() {
        // Corps du destructeur
    }
};
```

RÔLE DES DESTRUCTEURS

Le destructeur est utilisé pour :

- Libérer la mémoire allouée dynamiquement
- Fermer les fichiers ouverts
- Libérer les autres ressources système
- Effectuer des tâches de nettoyage avant la destruction de l'objet

LES ESPACES DE NOMS

DÉFINITION

Un espace de noms (namespace) en C++ est un conteneur qui permet de regrouper des identificateurs. Il aide à organiser le code et à éviter les conflits de noms. Un espace de noms peut contenir des variables, des fonctions, des classes, etc. Il est défini par le mot-clé `namespace`.

UTILITÉ DES ESPACES DE NOMS

- Évite les conflits de noms entre différentes parties du code.
- Organise le code en regroupant les éléments logiquement liés.
- Facilite la maintenance et la lisibilité du code.
- Permet de réutiliser des noms courants sans conflit.

SYNTAXE DE BASE

La syntaxe de base pour déclarer un espace de noms est :

```
namespace NomEspace {  
    // Déclarations  
}
```

DÉCLARATION D'UN ESPACE DE NOMS

Exemple de déclaration d'un espace de noms en C++ :

```
namespace MonEspace {
    int x = 42;
    void afficher() {
        std::cout << x << std::endl;
    }
}
```

UTILISATION DE L'OPÉRATEUR '::'

L'opérateur de résolution de portée :: est utilisé pour accéder aux membres d'un espace de noms.

```
MonEspace::afficher(); // Appelle la fonction afficher de MonEspace
```

ESPACES DE NOMS IMBRIQUÉS

Les espaces de noms peuvent être imbriqués pour une organisation hiérarchique.

```
namespace EspaceA {  
    namespace EspaceB {  
        int y = 10;  
    }  
}
```

Pour accéder à y :

```
int val = EspaceA::EspaceB::y;
```

DIRECTIVES 'USING'

La directive `using` permet d'éviter d'utiliser l'opérateur `::` à chaque fois.

```
using namespace MonEspace;  
afficher(); // Appelle afficher() sans MonEspace::
```

Attention : l'utilisation excessive peut entraîner des conflits de noms.

EXEMPLES PRATIQUES

Déclaration et utilisation d'espaces de noms :

```
#include <iostream>

namespace EspaceMath {
    int addition(int a, int b) {
        return a + b;
    }

    int main() {
        int resultat = EspaceMath::addition(5, 3);
        std::cout << "Resultat: " << resultat << std::endl;
        return 0;
    }
}
```

LES ENTRÉES/SORTIES EN C++

DÉFINITION

Les entrées/sorties en C++ permettent d'interagir avec l'utilisateur. Elles sont gérées par la bibliothèque standard iostream. Les flux d'entrée/sortie incluent cin, cout, cerr, et clog. cin est utilisé pour les entrées standard. cout est utilisé pour les sorties standard. cerr est utilisé pour les messages d'erreur. clog est utilisé pour les messages de log.

FLUX D'ENTRÉE ET DE SORTIE

Les flux d'entrée et de sortie sont des objets de la bibliothèque iostream. cin : flux d'entrée standard (clavier). cout : flux de sortie standard (écran). cerr : flux de sortie pour les erreurs (écran). clog : flux de sortie pour les logs (écran).

UTILISATION DE CIN

cin est utilisé pour lire des entrées depuis le clavier. Syntaxe de base :

```
int age;  
std::cin >> age;
```

L'opérateur >> extrait les données du flux cin.

UTILISATION DE COUT

cout est utilisé pour afficher des sorties à l'écran. Syntaxe de base :

```
std::cout << "Hello, World!";
```

L'opérateur << insère des données dans le flux cout.

FORMATAGE DE LA SORTIE

Le formatage de la sortie peut être contrôlé avec des manipulateurs. Exemples courants :

```
std::cout << std::setw(10) << std::setprecision(2) << std::fixed;
```

setw : largeur du champ. setprecision : précision des nombres flottants. fixed : notation fixe pour les nombres flottants.

ENTRÉES/SORTIES AVEC DES CHAÎNES DE CARACTÈRES

Les flux stringstream permettent des opérations d'entrée/sortie avec des chaînes. Exemple :

```
#include <sstream>
std::stringstream ss;
ss << "123";
int num;
ss >> num;
```

GESTION DES ERREURS D'ENTRÉE/SORTIE

Les erreurs d'entrée/sortie peuvent être détectées avec les méthodes fail() et bad(). Exemple :

```
if (std::cin.fail()) {
    std::cout << "Erreur d'entrée";
}
```

fail() : erreur de format. bad() : erreur matérielle.