INTRODUCTION AU C++

STRUCTURE DE BASE

EN-TÊTES

Les **en-têtes** sont utilisés pour inclure des **bibliothèques** et des **fichiers** nécessaires au bon fonctionnement du programme.

```
#include <iostream> // Pour utiliser cin et cout
#include <vector>   // Pour utiliser des vecteurs
#include "monFichier.h" // Pour inclure un fichier que vous avez créé
```

Les bibliothèques standard sont incluses avec `< >`, tandis que les fichiers créés par l'utilisateur sont inclus avec `" "`.

`#include`

La directive `#include` est utilisée pour inclure des **fichiers d'en-tête** dans le programme C++.

Syntaxe	Description
#include <>	Inclut un fichier d'en-tête du système
#include ""	Inclut un fichier d'en-tête de l'utilisateur

Exemple d'utilisation :

```
#include <iostream> // Inclut le fichier d'en-tête iostream du système
#include "mon_fichier.h" // Inclut le fichier d'en-tête mon_fichier.h créé par l'utilisateur
```

Les fichiers d'en-tête contiennent souvent des déclarations de fonctions et de classes, ainsi que des constantes, pour faciliter le partage d'informations entre différentes parties du code.

DÉCLARATION DES BIBLIOTHÈQUES

Les **bibliothèques standards** sont incluses grâce à la syntaxe suivante :

```
#include <iostream>
```

Les bibliothèques standards permettent d'utiliser certaines fonctionnalités prédéfinies. Le nom des bibliothèques se différencie en fonction de leur usage. Par exemple, `<iostream>` est utilisée pour gérer les entrées/sorties, `<fstream>` pour les opérations sur des fichiers.

NAMESPACE

Les **espaces de noms** fournissent un moyen d'éviter les **conflits de noms** dans les programmes.

Exemple:

```
namespace Espace1 {
  int fonction1(){
    return 1;
  }
}

namespace Espace2 {
  int fonction1(){
    return 2;
  }
}

int main() {
  int valeur1 = Espace1::fonction1();
  int valeur2 = Espace2::fonction1();
```

Les espaces de noms sont utilisés pour organiser et structurer le code, en séparant les fonctions et les variables ayant un but similaire ou appartenant à une même bibliothèque.

std

La plupart des fonctions et objets standard C++ sont définis dans l'espace de noms `std`.

Utilisation de `std` :

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

L'espace de noms `std` permet d'éviter les conflits de noms avec d'autres variables ou fonctions dans votre code. Il est préférable de toujours utiliser `std` pour accéder aux éléments de la bibliothèque standard.

CRÉATION DE NAMESPACES PERSONNALISÉS

Pour créer un espace de noms personnalisé :

```
namespace MonNamespace {
    int ma_variable;
    void ma_fonction();
}
```

Les **namespaces** permettent d'organiser le code en regroupant des variables, fonctions et classes associées, évitant ainsi les conflits de noms.

FONCTION MAIN

La fonction `main` est le **point d'entrée** de tout programme en **C++**.

```cpp
#include <iostream>

int main() {
  // Code du programme ici
  std::cout << "Bonjour, monde !" << std::endl;
  return 0;
}
```

Souligner l'importance de la fonction main comme point de départ de l'exécution du programme. Les erreurs de compilation sont souvent liées à l'absence ou à une mauvaise déclaration de la fonction main.

SYNTAXE

```
int main() {
    // Code
}
```

Le programme C++ doit contenir une fonction principale "main" qui est le point d'entrée du programme.

COMMENTAIRES

Les **commentaires** sont des moyens d'expliquer le **code** et sont ignorés par le **compilateur**.

- Commentaire sur une ligne :

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

- Commentaire multiligne :

```
/* Ceci est un commentaire
   sur plusieurs lignes */
```

COMMENTAIRES À UNE LIGNE (//)

Les commentaires à une ligne commencent par `//` et se terminent à la fin de la ligne.

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

Les commentaires sont essentiels pour expliquer le code à d'autres développeurs ou à soi-même. Pensez toujours à commenter votre code de manière claire et concise.

COMMENTAIRES MULTILIGNES (`/* ... */`)

Les commentaires multilignes commencent par `/*` et se terminent par `*/`.

```
/*
Ceci est un
commentaire multiligne
*/
```

TYPES DE DONNÉES

TYPES DE BASE

Les types de base en C++ incluent :

- `int` : **entiers**
- `float` : **nombres à virgule flottante** simple précision
- `double` : **nombres à virgule flottante** double précision
- `char` : **caractères**
- `bool` : **booléens** (vrai/faux)

Il est important de choisir le bon type de données pour leurs variables, en fonction de la taille et de la précision nécessaires.

TYPES DÉRIVÉS

Les types dérivés en C++ incluent :

- **Pointeurs**
- **Tableaux**

Type dérivé	Description
Pointeurs	Variables qui stockent l'adresse mémoire d'une valeur d'un type
Tableaux	Variables qui stockent une collection d'éléments d'un même type

Les types dérivés sont utilisés pour gérer des structures de données complexes et pour manipuler la mémoire d'un programme de manière plus efficace.

POINTEURS

SYNTAXE

Déclaration d'un pointeur :

```
int* ptr;
```

Initialisation d'un pointeur :

```
int a = 5;
ptr = &a;
```

Les pointeurs permettent de stocker l'adresse d'une variable, et d'accéder directement à la mémoire.

EXEMPLES D'UTILISATION

```
int a = 5;
int* ptr = &a;

cout << "Valeur pointée par ptr: " << *ptr << endl;
```

Cette slide illustre comment utiliser un pointeur pour stocker l'adresse d'une variable et accéder à la valeur de cette variable à travers le pointeur.

TABLEAUX

SYNTAXE

Déclaration d'un tableau :

```
int arr[5];
```

Initialisation d'un tableau :

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

La déclaration d'un tableau alloue simplement de la mémoire pour stocker les éléments, alors que l'initialisation d'un tableau permet de définir les valeurs des éléments lors de la création.

EXEMPLES D'UTILISATION

Accès aux éléments du tableau :

```
int somme = arr[0] + arr[1];
```

Ne pas oublier que l'indexation des tableaux en C++ commence à 0.

VARIABLES ET CONSTANTES

VARIABLES

DÉCLARATION

En C++, il est nécessaire de **déclarer** les variables avant leur utilisation, en spécifiant leur **type de données**.

```
int age;
float prix;
bool est_etudiant;
```

Il est également possible d'initialiser les variables au moment de leur déclaration, par exemple : `int age = 25;`.

INITIALISATION

Une variable peut être **initialisée** lors de sa déclaration ou ultérieurement dans le code.

```
int age = 25;
float prix = 19.99;
bool est_etudiant = true;
```

Il est préférable d'initialiser les variables dès que possible pour éviter des erreurs ou des comportements imprévisibles.

PORTÉE

La **portée** d'une variable est déterminée par la zone où elle est déclarée.

- **Variables locales** : déclarées dans un bloc, fonction ou méthode
- **Variables globales** : déclarées en dehors de toutes les fonctions

Les variables locales ont une durée de vie limitée et elles sont détruites lorsque le bloc, la fonction ou la méthode se termine. Les variables globales existent pendant toute la durée du programme. Évitez d'utiliser excessivement les variables globales, car elles peuvent rendre le code difficile à maintenir et à déboguer.

EXEMPLES DE DÉCLARATION ET INITIALISATION DE VARIABLES

```
int age = 25;
float taille = 1.75f;
double pi = 3.14159;
char lettre = 'A';
bool estVrai = true;
```

Il est important de connaître les différents types de variables pour choisir le plus approprié en fonction des besoins. Par exemple, `float` et `double` sont utilisés pour les nombres à virgule, mais `double` permet une meilleure précision.

AFFECTATION DE VALEURS

```
age = 30;
taille = 1.80f;
pi = 3.14;
lettre = 'B';
estVrai = false;
```

Type	Variable	Exemple de valeur
int	age	30
float	taille	1.80f
double	pi	3.14
char	lettre	'B'
bool	estVrai	false

Les affectations de valeurs aux variables dépendent du **type de données**. Assurez-vous que les valeurs assignées correspondent au type de la variable.

CONSTANTES

DÉCLARATION

Une **constante** est une valeur fixe qui ne peut pas être modifiée pendant l'exécution du programme. Pour déclarer une constante en C++, utilisez la mot clé `const`.

```
const int ANNEE_NAISSANCE = 1995;
```

Les constantes permettent de sécuriser les programmes en s'assurant que certaines valeurs ne peuvent pas être modifiées accidentellement.

INITIALISATION

Les **constantes** doivent être initialisées lors de leur **déclaration**.

```
const int ANNEE_NAISSANCE = 1995;
const float PI = 3.14159;
const bool MAJORITE = true;
```

PORTÉE

Comme les **variables**, la portée des **constantes** dépend du lieu de leur déclaration.

- **Constantes locales** : déclarées dans un **bloc**, **fonction** ou **méthode**
- **Constantes globales** : déclarées en dehors de toutes les fonctions

Les constantes locales sont accessibles uniquement à l'intérieur du bloc ou de la fonction tandis que les constantes globales sont accessibles dans tout le programme.

OPÉRATEURS

OPÉRATEURS ARITHMÉTIQUES

Les opérateurs arithmétiques permettent d'effectuer des **opérations mathématiques** sur les variables.

Opération	Syntaxe
Addition	a + b
Soustraction	a - b
Multiplication	a * b
Division	a / b
Modulo	a % b

La division entière et la division réelle sont différentes en C++.

OPÉRATEURS DE COMPARAISON

Les opérateurs de comparaison permettent de **comparer** des valeurs entre elles et retourner un **booléen**.

Opération	Syntaxe
Égalité	a == b
Inégalité	a != b
Supériorité	a > b
Infériorité	a < b
Supérieur / égal	a >= b
Inférieur / égal	a <= b

Ces opérateurs sont fondamentaux en programmation pour le contrôle de flux et les conditions. Rappeler les bonnes pratiques, comme éviter les doubles égalités (==) pour comparer des objets complexes ou utiliser les fonctions de comparaison personnalisées selon les types de données utilisés.

OPÉRATEURS LOGIQUES

Les **opérateurs logiques** permettent de manipuler les **valeurs booléennes** en combinant des conditions.

Opération	Syntaxe
AND	a && b
OR	a
NOT	!a

Ces opérateurs sont fondamentaux dans les structures de contrôle telles que les instructions "if", "else if" et "else" ainsi que les boucles "while" et "for".

EXEMPLE D'UTILISATION DES OPÉRATEURS

```
int a = 10;
int b = 20;
int c;

c = a + b; // Addition
c = a - b; // Soustraction
c = a * b; // Multiplication
c = a / b; // Division
c = a % b; // Modulo

bool comp1 = a == b; // Égalité
bool comp2 = a != b; // Inégalité
bool comp3 = a > b;  // Supériorité
bool comp4 = a < b;  // Infériorité
bool comp5 = a >= b; // Supérieur / égal
```

Vous pouvez utiliser ces opérateurs pour créer des expressions complexes et contrôler le flux d'exécution de votre programme avec des instructions conditionnelles et des boucles.

INSTRUCTIONS CONDITIONNELLES

IF

`if` est une instruction **conditionnelle** qui exécute des instructions en fonction de la vérification d'une condition.

```
if (condition) {
  // Instructions exécutées si la condition est vraie
}
```

- La condition doit être un booléen (`true` ou `false`).
- Les instructions à l'intérieur du bloc seront exécutées si la condition est évaluée à `true`.
- Si la condition est évaluée à `false`, le bloc ne sera pas exécuté.

Il est possible d'ajouter un bloc `else` après un bloc `if` pour exécuter du code lorsque la condition est fausse.

SYNTAXE

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

Opérateurs habituellement utilisés dans les conditions :

- Comparaison : ==, !=, <, >, <=, >=
- Logiques : && (ET), || (OU), ! (NON)

Il est possible d'utiliser des conditions plus complexes en combinant ces opérateurs et en utilisant des parenthèses pour préciser les priorités.

EXEMPLES D'UTILISATION

```cpp
int a = 5;
int b = 10;

if (a < b) {
    // Cette instruction sera exécutée car a est inférieur à b
    std::cout << "a est inférieur à b." << std::endl;
}
```

Les structures conditionnelles, comme le **if**, permettent d'exécuter du code en fonction des conditions spécifiées. Dans cet exemple, le code à l'intérieur du bloc est exécuté si la condition (`a < b`) est vraie.

ELSE

`else` est une clause complémentaire à `if` pour exécuter des **instructions** lorsque la **condition** du `if` est fausse.

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

Il est possible d'utiliser des structures else if pour tester plusieurs conditions successivement.

SYNTAXE

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

Les conditions peuvent être des expressions booléennes, des comparaisons ou des combinaisons de celles-ci à l'aide d'opérateurs logiques.

EXEMPLES D'UTILISATION

```cpp
int a = 5;
int b = 10;

if (a > b) {
    // Cette instruction NE sera PAS exécutée car a n'est pas supérieur à b
    std::cout << "a est supérieur à b." << std::endl;
} else {
    // Cette instruction sera exécutée car a n'est pas supérieur à b
    std::cout << "a n'est pas supérieur à b." << std::endl;
}
```

Cet exemple illustre l'utilisation des instructions **if-else** pour tester des conditions et afficher des messages en fonction du résultat de ces conditions.

ELSE IF

`else if` permet de tester plusieurs **conditions** en cascade. Les instructions correspondant à la première condition vraie seront exécutées.

```cpp
#include <iostream>

int main() {
  int nombre = 5;

  if (nombre > 10) {
    std::cout << "Le nombre est supérieur à 10" << std::endl;
  } else if (nombre > 5) {
    std::cout << "Le nombre est supérieur à 5 et inférieur ou égal à 10" << std::endl;
  } else if (nombre == 5) {
    std::cout << "Le nombre est égal à 5" << std::endl;
  } else {
    std::cout << "Le nombre est inférieur à 5" << std::endl;
  }
```

Utilisez les else if pour éviter les problèmes d'imbrication et d'indentation, améliorant la lisibilité du code.

SYNTAXE

```
if (condition1) {
    // Instructions à exécuter si la **condition1** est vraie
} else if (condition2) {
    // Instructions à exécuter si la **condition1** est fausse mais la **condition2** est vraie
} else {
    // Instructions à exécuter si toutes les **conditions précédentes** sont fausses
}
```

Les conditions doivent être des expressions booléennes, évaluées à vrai (true) ou faux (false).

EXEMPLES D'UTILISATION

```cpp
int a = 5;
int b = 10;

if (a > b) {
    // Cette instruction NE sera PAS exécutée car a n'est pas supérieur à b
    std::cout << "a est supérieur à b." << std::endl;
} else if (a == b) {
    // Cette instruction NE sera PAS exécutée car a n'est pas égal à b
    std::cout << "a est égal à b." << std::endl;
} else {
    // Cette instruction sera exécutée car toutes les conditions précédentes sont fausses
    std::cout << "a est inférieur à b." << std::endl;
}
```

Cet exemple illustre les instructions conditionnelles en C++, en utilisant "if", "else if" et "else" pour comparer deux variables.

BOUCLES

while

Les boucles `while` permettent d'exécuter plusieurs fois une série d'instructions **tant qu'une condition est vraie**.

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

int main() {

  int i = 0;

  while (i < 5) {
    cout << "i = " << i << endl;
    i++;
  }

  return 0;
}
```

N'oubliez pas d'incrémenter le compteur dans la boucle pour éviter une boucle infinie.

SYNTAXE

```
while (condition) {
    // Instructions à exécuter
}
```

N'oubliez pas d'incorporer une modification de la condition à l'intérieur de la boucle pour éviter les boucles infinies.

EXEMPLES D'UTILISATION

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

Cet exemple montre une boucle **while** simple qui affiche les nombres de 0 à 4.

do/while

Les boucles `do/while` ressemblent aux boucles `while`, mais la **condition** est vérifiée **après** l'exécution des instructions au moins une fois.

```cpp
#include <iostream>

int main() {
    int x = 0;
    do {
        std::cout << "x vaut: " << x << std::endl;
        x++;
    } while (x < 5);

    return 0;
}
```

Insistez sur le fait que la boucle `do/while` est utile lorsque l'on veut exécuter un bloc d'instructions au moins une fois, même si la condition est fausse dès le départ.

SYNTAXE

```
do {
    // Instructions à exécuter
} while (condition);
```

La boucle do-while est une boucle similaire à la boucle while, mais la condition est testée à la fin du bloc d'instructions, garantissant ainsi au moins une exécution.

EXEMPLES D'UTILISATION

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

La boucle do-while s'exécute au moins une fois puisque la condition est vérifiée après l'exécution du bloc de code.

`for`

Les boucles `for` permettent de répéter une série d'instructions un nombre précis de fois en utilisant une **variable de compteur**.

```
for (int i = 0; i < 10; i++) {
  cout << i << endl;
}
```

Type de boucle	Description
for	Répète une série d'instructions un nombre précis de fois en utilisant une variable de compteur.

- `int` i: déclaration de la variable de compteur
- `i < 10`: condition pour répéter la boucle
- `i++`: incrémentation de la variable de compteur à chaque itération

La boucle `for` est idéale pour parcourir des conteneurs tels que des tableaux ou d'autres structures de données linéaires.

SYNTAXE

```
for (initialisation; condition; mise à jour) {
    // Instructions à exécuter
}
```

La boucle for est idéale pour les situations où nous devons exécuter un ensemble spécifique d'instructions un certain nombre de fois, selon la condition donnée.

EXEMPLES D'UTILISATION

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

Boucle for : La boucle for est utilisée pour exécuter un bloc de code un certain nombre de fois.

- Initiation : `int i = 0`
- Condition : `i < 5`
- Iteration : `i++`

CONTRÔLE DE FLUX AVANCÉ

break

Disponible pour les boucles `for`, `while` et `do-while`, permet d'**interrompre** une boucle **prématurément**.

```
for (int i = 0; i < 10; i++) {
  if (i == 5) {
    break;
  }
  cout << i << endl;
}
```

Cet exemple montre une boucle `for` qui s'arrête lorsque la variable `i` atteint la valeur 5. Pensez à faire des démonstrations avec `while` et `do-while` également.

SYNTAXE

```
for (int i = 0; i < 10; ++i) {
  if (i == 5) {
    break;
  }
}
```

N'oubliez pas de mentionner que `break` permet d'interrompre la boucle immédiatement lorsque la condition spécifiée est remplie.

EXEMPLES D'UTILISATION

```
while (true) {
  cout << "Entrez un nombre:";
  int nb;
  cin >> nb;
  if (nb == -1) {
    break;
  }
}
```

Cet exemple montre une boucle **while** qui se termine lorsque l'utilisateur entre -1.

IMPACT SUR LES BOUCLES

`break` sort de la **boucle en cours**, mais n'affecte pas les **boucles englobantes**.

Exemple :

```
for (int i = 0; i < 5; ++i) {
  for (int j = 0; j < 5; ++j) {
    if (j == 2) {
      break; // sort de la boucle j uniquement
    }
    cout << "i: " << i << ", j: " << j << endl;
  }
}
```

Utiliser `break` avec précaution pour éviter les erreurs logiques et les boucles infinies.

`continue`

Disponible pour les boucles **`for`**, **`while`** et **`do-while`**. Interrompt l'itération en cours et passe à la suivante.

```
for (int i = 0; i < 10; ++i) {
  if (i % 2 == 0) {
    continue;
  }
  cout << "Nombre impair: " << i << endl;
}
```

`continue` permet de sauter une itération de la boucle et de poursuivre avec l'itération suivante, cela aide à éviter les blocs de code inutiles et rend le code plus lisible.

SYNTAXE

```
for (int i = 0; i < 10; ++i) {
  if (i % 2 == 0) {
    continue;
  }
}
```

Dans cet exemple, le mot-clé **continue** est utilisé pour sauter toutes les itérations où la variable **i** est un nombre pair.

EXEMPLES D'UTILISATION

```
while (!file.eof()) {
  string line;
  getline(file, line);

  if (line.empty()) {
    continue; // ne traite pas les lignes vides
  }

  // traitement de la ligne
}
```

Cet exemple illustre la lecture d'un fichier et le traitement des lignes, en ignorant les lignes vides.

IMPACT SUR LES BOUCLES

`continue` n'affecte que l'**itération en cours** de la boucle et ne sort pas de la boucle. Les **itérations suivantes** sont encore exécutées.

Le mot-clé `continue` peut être utile pour sauter rapidement une itération lorsqu'une certaine condition est rencontrée.

FONCTIONS

DÉCLARATION ET DÉFINITION

SYNTAXE

Une fonction est déclarée avec le **type de retour**, le **nom de la fonction** et les **paramètres** entre parenthèses.

```
type_de_retour nom_fonction(paramètre_1, paramètre_2)
{
    // Corps de la fonction
}
```

Le type de retour peut être void s'il n'y a pas de valeur à retourner. Les paramètres doivent également être précédés de leur type.

PARAMÈTRES

Les **paramètres** sont déclarés avec leur **type** et leur **nom**.

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

Les paramètres servent à passer des valeurs à une fonction pour effectuer des opérations. Ils sont également appelés "arguments" lorsqu'ils sont passés à la fonction.

VALEUR DE RETOUR

La valeur de retour est renvoyée avec l'instruction `return`.

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

`return` permet de sortir de la fonction et retourner une valeur au point d'appel de la fonction. Il est important de s'assurer que la valeur retournée est bien du type déclaré pour la fonction.

APPEL DE FONCTIONS

SYNTAXE

Pour appeler une fonction, utilisez son nom suivi des **arguments** entre parenthèses.

```
int resultat = somme(5, 3);
```

Assurez-vous que la fonction que vous appelez est correctement définie et déclarée.

PASSAGE DE PARAMÈTRES

Les **arguments** sont passés aux fonctions selon leur **déclaration** et leur **ordre**.

```
int a = 5;
int b = 3;
int resultat = somme(a, b);
```

Les valeurs des variables sont passées en paramètre dans la fonction, et non pas les variables elles-mêmes.