FONCTIONS EN C++

INTRODUCTION AUX FONCTIONS

DÉFINITION

Une fonction est un **bloc de code réutilisable** qui effectue une tâche spécifique. Elle prend un ensemble d'entrées (**arguments**), exécute des opérations et retourne éventuellement une valeur (**type de retour**).

```cpp
#include <iostream>

// Déclaration de la fonction
int addition(int a, int b) {
  int somme = a + b;
  return somme;
}

int main() {
  int x = 5;
  int y = 7;

  // Appel de la fonction
  int resultat = addition(x, y);
  std::cout << "Le résultat de l'addition est : " << resultat << std::endl;
```

Avant d'utiliser une fonction, il est nécessaire de la déclarer, en précisant son type de retour, ses arguments et le nom de la fonction.

STRUCTURE D'UNE FONCTION

```
[Type de retour] [nom_de_la_fonction] ([paramètres])
{
    // Corps de la fonction
}
```

Exemple :

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

DÉCLARATION ET DÉFINITION

PROTOTYPES DE FONCTION

SYNTAXE

Un **prototype de fonction** est une déclaration qui précède la définition. Il indique le **type de retour**, le **nom** et les **types des arguments**.

```
type_retour nom_fonction(type1 arg1, type2 arg2, ...);
```

Les prototypes de fonction permettent au compilateur de vérifier si les fonctions sont appelées avec les bons arguments et les bons types.

IMPORTANCE DES PROTOTYPES

- Ils précisent l'**interface** de la fonction.
- Ils permettent la vérification des **types** lors de l'appel de la fonction.
- Ils autorisent la définition des fonctions **après** leur utilisation dans le code.

Les prototypes permettent également d'éviter les erreurs de compilation et les problèmes de portée des fonctions.

CORPS DE LA FONCTION

SYNTAXE

La définition d'une **fonction** suit la déclaration du **prototype** et inclut le corps de la fonction.

```
type_retour nom_fonction(type1 arg1, type2 arg2, ...) {
    // Corps de la fonction
}
```

Les fonctions en C++ doivent être déclarées avec le type de retour et les types d'arguments pour permettre la vérification des appels de fonctions.

EXEMPLES D'UTILISATION

Prototype de la fonction d'addition :

```cpp
int addition(int a, int b);
```

Définition de la fonction d'addition :

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

Dans le `main` :

```cpp
#include <iostream>

int main() {
    int x = 5;
    int y = 3;
    int resultat = addition(x, y);
    std::cout << "Le résultat est : " << resultat << std::endl;
    return 0;
}
```

TYPES DE RETOUR

TYPES DE RETOUR

Les fonctions en C++ peuvent retourner des valeurs de différents types : `void`, **types primitifs**, **objets** et **classes**.

Exemple :

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

class Personne {
  public:
    string nom;
    int age;
};

Personne creerPersonne(string nom, int age) {
  Personne p;
  p.nom = nom;
  p.age = age;
  return p;
```

La valeur de retour d'une fonction est spécifiée par son type de retour dans la signature de la fonction. Il est important de bien choisir le type de retour adapté.

void

Une fonction **void** ne retourne aucune valeur.

```
void afficherMessage() {
  cout << "Bonjour!" << endl;
}

int main() {
  afficherMessage();
  return 0;
}
```

Les fonctions **void** peuvent être utilisées pour effectuer des actions sans retourner de valeur.

SYNTAXE

```
void maFonction() {
    // instructions
}
```

EXEMPLES D'UTILISATION

```cpp
void afficherMessage() {
    std::cout << "Hello, world!" << std::endl;
}

int main() {
    afficherMessage(); // Appel de la fonction afficherMessage
    return 0;
}
```

TYPES PRIMITIFS

Les fonctions peuvent retourner des valeurs de types **primitifs** tels que `int`, `float`, `double`, `char`, `bool`, etc.

Type	Description
int	Entier
float	Flottant simple précision
double	Flottant double précision
char	Caractère
bool	Booléen (vrai ou faux)

SYNTAXE

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

Cette fonction ajoute deux entiers et renvoie le résultat de l'opération.

EXEMPLES D'UTILISATION

```
int main() {
    int somme = addition(3, 4); // Appel de la fonction addition
    std::cout << "La somme est : " << somme << std::endl;
    return 0;
}
```

OBJETS ET CLASSES

Les **fonctions** peuvent également retourner des **objets** et des instances de **classe**.

```
class Personnage {
    public:
        std::string nom;
        int age;
};

Personnage creerPersonnage(std::string nom, int age) {
    Personnage p;
    p.nom = nom;
    p.age = age;
    return p;
}

int main() {
    Personnage perso = creerPersonnage("Arthur", 26);
```

Dans cet exemple, la fonction creerPersonnage crée une instance de la classe Personnage et la retourne. On peut alors utiliser l'instance créée dans le main pour accéder aux membres de la classe.

SYNTAXE

```
class MaClasse {
  public:
    int x;
};

MaClasse creerInstance() {
    MaClasse obj;
    obj.x = 42;
    return obj;
}
```

Dans cet exemple, nous créons une classe `MaClasse` avec un membre public `x` et une fonction `creerInstance()` qui crée une instance de cette classe et initialise `x` à 42.

EXEMPLES D'UTILISATION

```
int main() {
    MaClasse monObjet = creerInstance(); // Appel de la fonction creerInstance
    std::cout << "Valeur de x : " << monObjet.x << std::endl;
    return 0;
}
```

PASSAGE DES ARGUMENTS

PASSAGE DES ARGUMENTS

Les **arguments** sont des valeurs fournies à une fonction lors de son appel. Ils permettent de généraliser et réutiliser une fonction pour différentes tâches.

```
#include <iostream>

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

int main() {
  afficherMessage("Bonjour !");
  afficherMessage("Comment ça va ?");
  return 0;
}
```

Le passage des arguments peut se faire par valeur, par référence ou par pointeur. Dans cet exemple, le passage se fait par valeur.

PASSAGE PAR VALEUR

Le **passage par valeur** crée une copie de la valeur passée dans la fonction. Les modifications effectuées sur la copie n'affectent pas l'original.

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

void modifier(int x) {
  x = x + 1;
}

int main() {
  int value = 5;
  modifier(value);
  cout << "La valeur après la modification est: " << value << endl;
}
```

Dans cet exemple, bien que la fonction modifier() augmente la valeur de x, le changement n'affecte pas la variable "value" dans le main() car la valeur est passée par copie.

SYNTAXE

```
void fonction(Type argument) { /*...*/ }
```

Type : Le type de donnée de l'argument (int, float, etc.)

fonction : Le nom de la fonction

argument : Le nom de l'argument

EXEMPLES D'UTILISATION

```
void increment(int x) {
  x++;
}
int a = 5;
increment(a); // a reste 5
```

Dans cet exemple, la fonction `increment` fait partie des passages par valeur et ne modifie pas la valeur de la variable dans le contexte appelant.

PASSAGE PAR RÉFÉRENCE

Le passage par **référence** permet à la fonction de modifier directement la valeur de l'argument plutôt que de travailler sur une copie.

```cpp
#include <iostream>

void doubler(int &a) {
  a *= 2;
}

int main() {
  int nombre = 5;
  doubler(nombre);
  std::cout << "Nombre doublé : " << nombre << std::endl;
  return 0;
}
```

Avant l'appel de doubler	Après l'appel de doubler
nombre = 5	nombre = 10

La variable 'nombre' est directement modifiée grâce au passage par référence, contrairement au passage par valeur qui ne modifie pas la variable d'origine.

SYNTAXE

```
void fonction(Type &argument) { /*...*/ }
```

Cette syntaxe est pour définir une **fonction** en C++ qui accepte un **paramètre** par référence. Cette technique permet de modifier directement la valeur de l'**argument** passé à la fonction.

EXEMPLES D'UTILISATION

```
void increment(int &x) {
  x++;
}
int a = 5;
increment(a); // a vaut maintenant 6
```

Note : Dans cet exemple, la fonction `increment` prend un **référence** à un entier en argument et l'**incremente**. Cette approche permet de **modifier directement** la valeur de `a` dans la mémoire sans avoir besoin de retourner une nouvelle valeur.

RÉFÉRENCES CONSTANTES (`const`)

Les références constantes garantissent que la fonction ne pourra pas modifier l'argument passé par **référence**.

```
void maFonction(const int &monEntier) {
  // Ici, monEntier ne peut pas être modifié.
}
```

Avantages	Désavantages
Protège l'intégrité des données	Ne permet pas de modifier l'argument
Compatibilité avec les objets temporaires	

Les références constantes sont utiles pour s'assurer qu'une fonction n'altère pas l'état d'un objet ou d'une variable passée en argument, tout en évitant la copie inutile de données.

SYNTAXE

```
void fonction(const Type &argument) { /*...*/ }
```

Cette syntaxe permet de définir une fonction qui prend un argument constant par référence.

EXEMPLES D'UTILISATION

```
void afficher(const std::string &texte) {
  std::cout << texte;
}
std::string message = "Bonjour";
afficher(message); // Affiche "Bonjour"
```

FONCTIONS INLINE

INTRODUCTION AUX FONCTIONS INLINE

Les fonctions **inline** sont une optimisation du compilateur pour réduire le coût des appels de fonctions en remplaçant l'appel par le code de la fonction elle-même.

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

int main() {
  int resultat = somme(2, 3);
  return 0;
}
```

L'utilisation des fonctions inline peut améliorer les performances, mais augmente également la taille du code compilé.

AVANTAGES ET INCONVÉNIENTS

AVANTAGES

- **Réduit le coût** des appels de fonctions
- Peut **améliorer la performance** du programme, notamment lorsque les fonctions sont courtes

Le coût des appels de fonction est réduit grâce à l'inlining qui permet d'éviter les appels répétitifs de petites fonctions, ce qui améliore la performance globale du programme.

INCONVÉNIENTS

- Augmente la **taille du code**
- Peut causer une **mauvaise utilisation** de l'instruction `inline`

Dans certaines situations, l'utilisation abusive de l'instruction `inline` peut détériorer les performances du programme.

SYNTAXE

Pour déclarer une fonction **inline**, il suffit d'ajouter le mot-clé `inline` devant la déclaration de la fonction.

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

Les fonctions inline sont utilisées pour améliorer les performances en évitant les appels de fonction lorsque le code de la fonction est court et simple.

EXEMPLES D'UTILISATION

EXEMPLE 1 : FONCTION INLINE

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

inline int multiply(int a, int b) {
  return a * b;
}

int main() {
  cout << "Result: " << multiply(4, 5) << endl;
  return 0;
}
```

Cet exemple illustre une fonction "inline" qui permet d'accélérer l'exécution du programme en évitant les appels de fonction.

EXEMPLE 2

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

inline double divide(double a, double b) {
  return a / b;
}

int main() {
  cout << "Result: " << divide(10.0, 3.0) << endl;
  return 0;
}
```

Dans ces exemples, l'appel aux fonctions **multiply** et **divide** est remplacé par le code des fonctions elles-mêmes.

Les fonctions inline permettent d'accélérer l'exécution en substituant l'appel de la fonction par le code de la fonction elle-même. Ceci peut cependant augmenter la taille du code.

RÉCURSIVITÉ

DÉFINITION

La **récursivité** est une technique de programmation où une fonction s'appelle elle-même. Elle démarre généralement avec un **cas de base** et travaille avec un problème plus petit à chaque appel récursif.

Exemple en C++ :

```cpp
int factorielle(int n) {
  if (n == 0) {
    return 1; // Cas de base
  } else {
    return n * factorielle(n - 1); // Appel récursif
  }
}
```

N'oubliez pas de mentionner que la récursivité doit être utilisée avec prudence, car elle peut entraîner des problèmes de performance et de mémoire si elle n'est pas conçue correctement.

AVANTAGES ET INCONVÉNIENTS

- **Avantages** :
  - Code plus **concis** et plus **facile à comprendre**
  - Peut être plus **élégant** que les solutions itératives
- **Inconvénients** :
  - Peut entraîner une **explosion de la pile d'appel** (stack overflow)
  - **Performances inférieures** à celles des solutions itératives

EXEMPLES DE FONCTIONS RÉCURSIVES

FACTORIELLE

La fonction factorielle illustre un exemple classique de **récursivité** :

```
int factorielle(int n) {
    if (n == 0)
        return 1;
    else
        return n * factorielle(n - 1);
}
```

La récursivité peut être plus lente et consommer plus de mémoire que les approches itératives.

SUITE DE FIBONACCI

La **suite de Fibonacci** est un autre exemple de **récursivité** :

```
int fibonacci(int n) {
    if (n <= 1)
        return n;
    else
        return fibonacci(n - 1) + fibonacci(n - 2);
}
```

FONCTIONS DE LA BIBLIOTHÈQUE STANDARD

TYPES DE FONCTIONS

La **bibliothèque standard** de C++ fournit un ensemble de **fonctions prédéfinies** pour faciliter les tâches courantes. Elles comprennent :

- **Entrée/Sortie** (cin, cout)
- **Mathématiques** (pow, sqrt)
- **Manipulation des chaînes de caractères** (strcpy, strcat)
- **Gestion du temps** (ctime, time)

ENTRÉE/SORTIE (CIN, COUT)

SYNTAXE

Les fonctions d'entrée/Sortie **`cin`** et **`cout`** sont définies dans le fichier d'en-tête `<iostream>`.

```cpp
#include <iostream>

int main() {
    int num;
    std::cout << "Entrez un nombre:";
    std::cin >> num;
    std::cout << "Vous avez entré: " << num;
}
```

MATHÉMATIQUES (POW, SQRT)

SYNTAXE

- Pour utiliser les **fonctions mathématiques**, on inclut `<cmath>`.
- `pow(base, exposant)` : élève la **base** à la **puissance** de l'exposant.
- `sqrt(x)` : renvoie la **racine carrée** de x .

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

int main() {
    double base = 3.0, exposant = 4.0;
    std::cout << "3^4 = " << pow(base, exposant) << std::endl;
    std::cout << "sqrt(9) = " << sqrt(9) << std::endl;
}
```

La bibliothèque `<cmath>` fournit également d'autres fonctions mathématiques telles que `sin()`, `cos()`, `tan()` et `ceil()`.

MANIPULATION DES CHAÎNES DE CARACTÈRES (STRCPY, STRCAT)

SYNTAXE

- Inclure `<cstring>` pour utiliser des **fonctions de manipulation de chaînes**.
- `strcpy(destination, source)` : copie la chaîne source dans la chaîne destination.
- `strcat(destination, source)` : ajoute la chaîne source à la chaîne destination.

```cpp
#include <iostream>
#include <cstring>

int main() {
    char s1[20] = "Hello";
    char s2[20] = " World";

    strcat(s1, s2);
    std::cout << "s1: " << s1 << std::endl;

    strcpy(s2, s1);
    std::cout << "s2: " << s2 << std::endl;
}
```

Attention à ne pas dépasser la taille allouée pour les chaînes lors de l'utilisation de strcpy et strcat.

GESTION DU TEMPS (CTIME, TIME)

SYNTAXE

- Inclure **<ctime>** pour utiliser des fonctions liées au temps.
- `time(nullptr)` : renvoie l'heure actuelle en secondes depuis 00:00:00, 1 janvier 1970 (**epoch UNIX**).
- `ctime(&time)` : renvoie une représentation sous forme de chaîne de caractères de l'heure.

```cpp
#include <iostream>
#include <ctime>

int main() {
    time_t now = time(nullptr);
    std::cout << "Timestamp actuel: " << now << std::endl;
    std::cout << "Heure actuelle: " << ctime(&now);
}
```

FONCTIONS À PRÉDICAT

DÉFINITION ET UTILISATION

Les **fonctions à prédicat** sont des fonctions qui renvoient un **booléen** en fonction du résultat d'une condition spécifiée.

Exemple : Vérifier si un nombre est pair

```
bool estPair(int x) {
  return (x % 2) == 0;
}
```

Ces fonctions sont utiles pour déterminer si un objet satisfait une condition spécifique.

Autre exemple : Trouver l'élément qui satisfait la condition

```cpp
#include <algorithm>
#include <vector>
#include <iostream>

bool estPair(int x) {
  return (x % 2) == 0;
}

int main() {
  std::vector<int> vec{1, 5, 7, 8, 11, 14};

  auto iter = std::find_if(vec.begin(), vec.end(), estPair);

  if (iter != vec.end()) {
    std::cout << "Le premier nombre pair trouvé est : " << *iter << std::endl;
```

Les **algorithme de la STL** sont très utiles pour manipuler et parcourir des conteneurs. Ils permettent d'effectuer des opérations standard de manière très simple.