Pourquoi créer des fonctions ?

En C++, les fonctions sont utilisées pour regrouper des instructions liées en un seul bloc de code, qui peut être appelé à partir de n'importe quelle partie du programme. Les fonctions offrent plusieurs avantages, notamment:

- **Réduction de la redondance de code** : plutôt que d'écrire le même code à plusieurs reprises, on peut créer une fonction et l'appeler à chaque fois que cette fonctionnalité est nécessaire. Cela allège le code et permet de minimiser les erreurs de copier-coller.
- **Modularité**: Les fonctions permettent de diviser un programme complexe en parties plus petites et plus faciles à gérer. Chaque fonction peut être écrite séparément du reste du programme, sans avoir à penser au reste du code pendant l'écriture.
- **Abstraction**: Les fonctions fournissent une abstraction en permettant d'utiliser des fonctionnalités sans avoir à connaître les détails de leur implémentation. Par exemple, on peut utiliser des fonctions de bibliothèque sans se soucier de leur fonctionnement interne.
- **Réutilisabilité**: Une fois qu'une fonction est écrite, elle peut être appelée plusieurs fois dans le programme et même être partagée entre différents programmes, ce qui réduit la quantité de code qui doit être écrit et testé à chaque fois.

Différences entre les fonctions et les méthodes

Cette différence réside dans leur relation avec les objets dans la programmation orientée objet.

Une fonction :

- bloc de code indépendant qui peut être appelé par son nom et qui peut prendre des paramètres et renvoyer une valeur.
- Elles ne sont pas appelées sur des instances d'objets
- Les fonctions n'accèdent qu'aux données qui leur sont explicitement passées en tant que paramètres.

Une méthode :

- fonction associée à un objet ou à une classe.
- Elles sont appelées sur des instances d'objets.
- Elles ont accès aux données de l'objet sur lequel elles sont appelées, ainsi qu'aux autres méthodes de cet objet.

♦ Fonction sans retour

Voici un exemple très simple de programme en C++ qui utilise une fonction sans paramètre:

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

void printHello() {
    cout << "Hello, World!" << endl;
}

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

Dans cet exemple, une fonction appelée `printHello` qui ne prend aucun paramètre, est définie. Elle utilise la fonction `cout` pour afficher le message “Hello, World!” à l’écran.

Dans la fonction `main`, vous appelez la fonction `printHello` pour exécuter son code. Lorsque ce programme est exécuté, il affiche le message “Hello, World!” à l’écran.

♦ Fonction avec un paramètre et le retour d'une valeur

Voici un exemple simple de programme en C++ qui utilise une fonction qui retourne une valeur:

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

int square(int x) {
    return x * x;
}

int main() {
    int x = 5;
    int result = square(x);
    cout << "Le carré de " << x << " est " << result << endl;
    return 0;
}
```

Dans cet exemple, vous définissez une fonction appelée `square` qui prend un paramètre `x` de type `int` et retourne la valeur de `x` au carré. Cette fonction utilise l’opérateur `*` pour calculer le carré de `x` et l’instruction `return` pour renvoyer le résultat.

Dans la fonction `main`, vous déclarez une variable `x` et lui attribuez la valeur 5. Ensuite la fonction `square` avec `x` comme argument est appelée et vous stockez le résultat renvoyé dans une variable appelée `result`. Enfin, vous utilisez la fonction `cout` pour afficher le résultat à l'écran.

Lorsque ce programme est exécuté, il affiche le message suivant à l'écran:

```
Le carré de 5 est 25
```

Il est important de se rappeler que lorsqu'on transmet des valeurs de variables à une fonction, ces valeurs sont copiées dans de nouvelles cases mémoire. Par conséquent, la fonction `square()` ne modifie pas les variables déclarées dans la fonction `main()`, mais travaille uniquement avec ses propres cases mémoire.

Ce n'est que lors de l'exécution de l'instruction `return` que les variables de `main()` sont modifiées, c'est-à-dire ici la variable `result`. La variable `x` reste inchangée lors de l'appel à la fonction.

♦ Portée des variables (scope) dans une fonction

Dans des fonctions, on peut utiliser le même nom pour nommer une variable.

Par contre, une variable dans une fonction est considérée comme locale et est utilisée uniquement dans le corps de la fonction.

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

int square(int x) {
    int result = x * x;
    return result;
}

int main() {
    int x = 5;
    int result = square(x);
    cout << "Le carré de " << x << " est " << result << endl;
    return 0;
}
```

Dans cet exemple, nous avons défini une fonction appelée `square` qui prend un paramètre `x` de type `int` et retourne la valeur de `x` au carré. Cette fonction utilise l'opérateur `*` pour calculer le carré de `x` et stocker le résultat dans une variable locale appelée `result`. La fonction renvoie ensuite la valeur de `result` à l'aide de l'instruction `return`.

Dans la fonction `main`, nous déclarons deux variables: `x` et `result`. La variable `x` est initialisée à 5, puis nous appelons la fonction `square` avec `x` comme argument et stockons le résultat renvoyé dans la variable `result`. Enfin, nous utilisons la fonction `cout` pour afficher le résultat à l'écran.

Bien que les variables `result` dans la fonction `square` et dans la fonction `main` aient le même nom, ce sont des variables distinctes stockées dans des cases mémoire différentes. La modification de l'une n'affecte pas l'autre.

♦ Fonction multi argument

! Une fonction peut demander plusieurs arguments !

Le nombre maximum d'arguments que l'on peut passer à une fonction en C++ est défini par l'implémentation. Selon la norme C++, une implémentation doit être capable de prendre en charge au moins 256 arguments pour une fonction. Cependant, il est important de noter que ce nombre est seulement une recommandation et que certaines implémentations peuvent supporter plus ou moins d'arguments.

En pratique, il est rare d'avoir besoin de passer un grand nombre d'arguments à une fonction. Si vous vous trouvez dans une situation où vous avez besoin de passer un grand nombre d'arguments à une fonction, il peut être judicieux de revoir la conception de votre programme pour voir s'il existe des moyens de réduire le nombre d'arguments nécessaires.

Voici un exemple simple de programme en C++ qui utilise une fonction avec deux arguments:

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

int add(int x, int y) {
    return x + y;
}

int main() {
    int a = 5;
    int b = 3;
    int result = add(a, b);
    cout << a << " + " << b << " = " << result << endl;
    return 0;
}
```

Dans cet exemple, nous avons défini une fonction appelée `add` qui prend deux paramètres `x` et `y` de type `int` et retourne la somme de ces deux nombres. Cette fonction utilise l'opérateur `+` pour calculer la somme de `x` et `y`, puis renvoie le résultat à l'aide de l'instruction `return`.

Dans la fonction `main`, nous déclarons trois variables : `a`, `b` et `result`. Les variables `a` et `b` sont initialisées à 5 et 3, respectivement. Nous appelons ensuite la fonction `add` avec `a` et `b` comme arguments et stockons le résultat renvoyé dans la variable `result`. Enfin, nous utilisons la fonction `cout` pour afficher le résultat à l'écran.

Lorsque ce programme est exécuté, il affiche le message suivant à l'écran:

```
5 + 3 = 8
```

♦ Passage par valeur

Le passage par valeur est une méthode de passage de paramètres à une fonction dans laquelle la valeur d'un argument est copiée dans un paramètre formel de la fonction. Cela signifie que toute modification apportée au paramètre formel dans la fonction n'affectera pas l'argument original.

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

void increment(int x) {
    x++;
    cout << "Valeur de x dans la fonction: " << x << endl;
}

int main() {
    int a = 5;
    cout << "Valeur de a avant l'appel de la fonction: " << a << endl;
    increment(a);
    cout << "Valeur de a après l'appel de la fonction: " << a << endl;
    return 0;
}
```

Dans cet exemple, nous avons défini une fonction `increment` qui prend un paramètre `x` de type `int`. Cette fonction incrémente la valeur de `x` et affiche sa nouvelle valeur à l'écran. Dans la fonction `main`, nous déclarons une variable `a` et lui affectons la valeur `5`. Nous appelons ensuite la fonction `increment` en lui passant `a` comme argument.

Lorsque nous exécutons ce code, nous pouvons voir que la valeur de `a` ne change pas après l'appel de la fonction `increment`. Cela est dû au fait que le passage par valeur crée une copie de l'argument `a` dans le paramètre formel `x` de la fonction. Toute modification apportée à `x` dans la fonction n'affecte donc pas la valeur originale de `a`.

En résumé, le passage par valeur permet de passer des arguments à une fonction sans affecter les valeurs originales des arguments. Cela peut être utile pour éviter les effets de bord indésirables dans un programme.

Le passage par valeur est une méthode de passage de paramètres à une fonction dans laquelle la valeur d'un argument est copiée dans un paramètre formel de la fonction. Cela signifie que toute modification apportée au paramètre formel dans la fonction n'affectera pas l'argument original.

Voici un schéma qui illustre le passage par valeur:

```
+-----------------+----------------+
| Adresse mémoire | Valeur         |
+-----------------+----------------+
| 0x00000000      |                |
| 0x00000001      |                |
| 0x00000002      |                |
| 0x00000003      |                |
| ...             |                |
| 0x0000XXXX      | a: 5           |
| ...             |                |
| 0x0000YYYY      | x: 5 (copie)   |
| ...             |                |
+-----------------+----------------+

Fonction increment:
    x = 5 (copie de a)
    x++
    // La valeur de a reste inchangée

+-----------------+----------------+
| Adresse mémoire | Valeur         |
+-----------------+----------------+
| 0x00000000      |                |
| 0x00000001      |                |
| 0x00000002      |                |
| 0x00000003      |                |
| ...             |                |
| 0x0000XXXX      | a: 5           |
| ...             |                |
| 0x0000YYYY      | x: 6           |
| ...             |                |
+-----------------+----------------+
```

Dans cet exemple, nous avons déclaré une variable `a` et lui avons affecté la valeur `5`. Cette variable est stockée dans la RAM à une adresse mémoire spécifique, que nous avons représentée par `0x0000XXXX`. Nous avons également défini une fonction `increment` qui prend un paramètre `x` de type `int`. Cette fonction incrémente la valeur de `x`, mais comme `x` est une copie de `a`, la valeur de `a` ne change pas.

Lorsque nous appelons la fonction `increment` en lui passant `a` comme argument, le paramètre formel `x` reçoit une copie de la valeur de `a`. Cela signifie que toute modification apportée à `x` dans la fonction n'affectera pas la valeur originale de `a`. Dans notre exemple, lorsque nous incrémentons `x`, sa valeur passe de `5` à `6`, mais la valeur de `a` reste inchangée.

Le passage par valeur permet de passer des arguments à une fonction en créant une copie de ces arguments. Cela permet d'éviter les effets de bord indésirables en empêchant les modifications apportées aux paramètres formels dans la fonction d'affecter les arguments originaux.

♦ Passage d'argument par référence

Le passage par référence en C++ permet de passer une référence à une variable à une fonction, plutôt que de passer une copie de la valeur de la variable. Cela signifie que toute modification apportée à la variable à l'intérieur de la fonction affectera directement la variable originale, plutôt que de ne modifier qu'une copie locale.

Pour passer une variable par référence en C++, on utilise l'opérateur `&` lors de la déclaration du paramètre dans la fonction. Par exemple, pour passer une variable `x` de type `int` par référence à une fonction `f`, on déclarerait le paramètre comme suit:

```
void f(int& x) {
    // ...
}
```

À l'intérieur de la fonction, on peut accéder et modifier directement la variable `x` passée en argument. Par exemple, si on appelle la fonction `f` avec une variable `a` comme argument, toute modification apportée à `x` dans la fonction affectera directement la variable `a`.

Voici un exemple simple qui illustre le passage par référence en C++:

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

void increment(int& x) {
    x++;
}

int main() {
    int a = 5;
    cout << "Avant l'incrémentation: " << a << endl;
    increment(a);
    cout << "Après l'incrémentation: " << a << endl;
    return 0;
}
```

Dans cet exemple, nous avons défini une fonction `increment` qui prend un paramètre `x` passé par référence. Cette fonction utilise l'opérateur d'incrémentation `++` pour incrémenter la valeur de `x`.

Dans la fonction `main`, nous déclarons une variable `a` et lui attribuons la valeur 5. Nous appelons ensuite la fonction `increment` avec `a` comme argument. Étant donné que `x` est passé par référence, l'incrémentation de `x` dans la fonction affecte directement la variable `a`. Lorsque ce programme est exécuté, il affiche les messages suivants à l'écran:

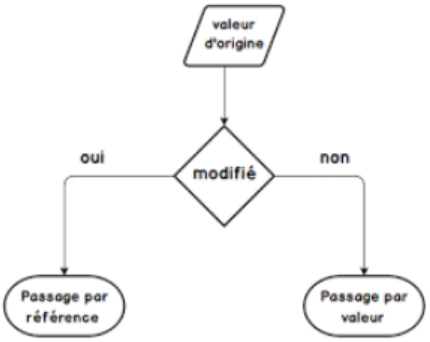
```
Avant l'incrémentation: 5
Après l'incrémentation: 6
+-----------------+----------------+
| Adresse mémoire | Valeur         |
+-----------------+----------------+
| 0x00000000      |                |
| 0x00000001      |                |
| 0x00000002      |                |
| 0x00000003      |                |
| ...             |                |
| 0x000012FE      | a: 5           |
| ...             |                |
+-----------------+----------------+

Fonction increment:
    x -> 0x000012FE (référence à a)
    x++
    // La valeur de a est maintenant 6

+-----------------+----------------+
| Adresse mémoire | Valeur         |
+-----------------+----------------+
| 0x00000000      |                |
| 0x00000001      |                |
```

```
| 0x00000002      |                |
| 0x00000003      |                |
| ...             |                |
| 0x000012FE      | a: 6           |
| ...             |                |
+-----------------+----------------+
```

Dans cet exemple, nous avons déclaré une variable `a` et lui avons affecté la valeur `5`. Cette variable est stockée dans la RAM à une adresse mémoire spécifique, que nous avons représentée par `0x0000XXXX`. Nous avons également défini une fonction `increment` qui prend un paramètre `x` de type `int&` (référence à un entier). Cette fonction incrémente la valeur de `x` et, comme `x` est une référence à `a`, la valeur de `a` est également incrémentée.

Lorsque nous appelons la fonction `increment` en lui passant `a` comme argument, le paramètre formel `x` reçoit une référence à `a`. Cela signifie que toute modification apportée à `x` dans la fonction affectera également la valeur de `a`. Dans notre exemple, lorsque nous incrémentons `x`, la valeur de `a` passe de `5` à `6`.

En résumé, le passage par référence permet de passer des arguments à une fonction en créant une référence à ces arguments. Cela permet aux modifications apportées aux paramètres formels dans la fonction d'affecter les arguments originaux.



En C++, il est possible de créer une fonction qui prend un argument par valeur et un autre par référence. Voici un exemple simple de programme en C++ qui utilise une fonction avec un argument passé par valeur et un autre passé par référence:

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

void swap(int x, int& y) {
    int temp = x;
    x = y;
    y = temp;
}

int main() {
    int a = 5;
    int b = 3;
    cout << "Avant l'échange: a = " << a << ", b = " << b << endl;
    swap(a, b);
    cout << "Après l'échange: a = " << a << ", b = " << b << endl;
    return 0;
}
```

Dans cet exemple, nous avons défini une fonction appelée `swap` qui prend deux paramètres : `x` et `y`. Le premier paramètre, `x`, est passé par valeur, ce qui signifie qu'une copie de la valeur de `x` est créée et passée à la fonction. Le deuxième paramètre, `y`, est passé par référence, ce qui signifie que la fonction reçoit une référence à la variable `y` plutôt qu'une copie de sa valeur.

La fonction `swap` utilise une variable temporaire pour échanger les valeurs de `x` et `y`. Étant donné que `x` est passé par valeur, les modifications apportées à `x` dans la fonction n'affectent pas la variable `a` dans la fonction `main`. En revanche, comme `y` est passé par référence, les modifications apportées à `y` dans la fonction affectent directement la variable `b` dans la fonction `main`.

Lorsque ce programme est exécuté, il affiche les messages suivants à l'écran:

```
Avant l'échange: a = 5, b = 3
Après l'échange: a = 5, b = 5
```

Exercice pratique :

Corrigez cette fonction qui fonctionne mal pour que les données des deux variables soient échangées.

♦ Passage par référence constante

Le passage par référence constante avec un texte (chaîne de caractères) plutôt qu'un entier:

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

void print(const string& text) {
    cout << "Texte dans la fonction: " << text << endl;
    // text = "Nouveau texte"; // Erreur: text est une référence constante
}

int main() {
    string str = "Ne modifier pas ce texte";
    cout << "Texte avant l'appel de la fonction: " << str << endl;
    print(str);
    cout << "Texte après l'appel de la fonction: " << str << endl;
    return 0;
}
```

Dans cet exemple, nous avons défini une fonction `print` qui prend un paramètre `text` de type `const string&` (référence constante à une chaîne de caractères). Cette fonction affiche le texte à l'écran, mais ne peut pas le modifier car `text` est une référence constante. Dans la fonction `main`, nous déclarons une variable `str` de type `string` et lui affectons la valeur `"Bonjour"`. Nous appelons ensuite la fonction `print` en lui passant `str` comme argument.

Lorsque nous exécutons ce code, nous pouvons voir que le texte stocké dans `str` ne change pas après l'appel de la fonction `print`. Cela est dû au fait que le passage par référence constante crée une référence constante à l'argument `str` dans le paramètre formel `text` de la fonction. Comme `text` est une référence constante, il ne peut pas être modifié dans la fonction.

En résumé, cet exemple montre comment utiliser le passage par référence constante avec un texte (chaîne de caractères) en C++. Cette méthode permet d'accéder au texte original dans la fonction sans pouvoir le modifier.

Résumé

- Les fonctions en C++ sont des blocs de code qui contiennent des instructions et remplissent un rôle spécifique dans un programme. Chaque programme doit avoir au moins une fonction, appelée `main()`, qui est exécutée au démarrage du programme.

- Il est recommandé de diviser son programme en plusieurs fonctions, chacune ayant un rôle bien défini, pour améliorer l'organisation et la lisibilité du code. Les fonctions ont toutes la même structure, avec un type de retour, un nom, des arguments et un corps contenant les instructions à exécuter.
- Les fonctions peuvent être appelées plusieurs fois au cours de l'exécution d'un programme et peuvent recevoir des arguments en entrée pour traiter des données spécifiques. Elles peuvent également renvoyer un résultat en sortie grâce à l'instruction `return`.
- Les fonctions peuvent recevoir des références en argument pour modifier directement une valeur stockée en mémoire. Cela permet d'effectuer des modifications sur les données d'origine sans avoir à créer de copie.
- Les fonctions en C++ sont des outils puissants pour structurer et organiser un programme en regroupant des instructions dans des blocs de code ayant un rôle spécifique. Elles permettent de manipuler et de traiter des données de manière flexible et efficace.