**PRÉNOM : ABOUBACAR**

**NOM : SYLLA**

**THÈME : POINTEUR ET ALLOCATION**

**1-POINTEUR**

**En algorithmique, un pointeur est une variable qui stocke l’adresse mémoire d’une autre variable. Imaginez un pointeur comme une étiquette collée sur une boîte. L’étiquette indique où trouver la boîte, et non le contenu de la boîte elle-même.**

**Pourquoi utiliser des pointeurs ?**

**Les pointeurs offrent plusieurs avantages :**

**Gestion efficace de la mémoire : Les pointeurs permettent d’accéder directement aux données en mémoire, sans avoir à copier la totalité des données. Cela est particulièrement utile pour les grands tableaux et les structures de données complexes.**

**-Création de structures de données dynamiques : Les pointeurs permettent de créer des structures de données dynamiques, comme les listes chaînées et les arbres, qui peuvent s’adapter à la taille des données au fur et à mesure que l’application évolue.**

**-Passage de paramètres par référence : Les pointeurs permettent de modifier des données dans une autre fonction en passant l’adresse de la variable en paramètre.**

**-Concepts clés :**

**-Adresse mémoire : Chaque variable est stockée à une adresse mémoire unique.**

**-Déclaration d’un pointeur :Un pointeur est déclaré en utilisant un astérisque (\*) suivi du type de données pointé. Par exemple : `int \*ptr ;` déclare un pointeur nommé `ptr` qui pointe vers un entier.**

**-Affectation d’un pointeur :Un pointeur est affecté à une adresse mémoire en utilisant l’opérateur d’adressage (&). Par exemple : `ptr = &variable ;` affecte le pointeur `ptr` à l’adresse de la variable `variable`.**

**-Accès à la valeur pointée :La valeur pointée par un pointeur est accessible en utilisant l’opérateur de déréférencement (\*). Par exemple : `valeur = \*ptr ;` affecte la valeur pointée par `ptr` à la variable `valeur`.**

**Exemple en pseudo-code :**

**Pseudocode**

**// Déclarer un tableau d’entiers**

**Entier[] tableau = [1, 2, 3, 4, 5]**

**// Déclarer un pointeur vers un entier**

**Pointeur vers entier ptr**

**// Affecter le pointeur au début du tableau**

**Ptr = tableau[0]**

**// Afficher la valeur pointée par le pointeur**

**Afficher \*ptr // affiche 1**

**// Avancer le pointeur au prochain élément du tableau**

**Ptr = ptr + 1**

**// Afficher la valeur pointée par le pointeur**

**Afficher \*ptr // affiche 2**

**```**

**Dans cet exemple, le pointeur `ptr` est initialisé à l’adresse du premier élément du tableau `tableau`. Ensuite, on accède à la valeur pointée par `ptr` en utilisant l’opérateur `\*`. En déplaçant `ptr` d’une position mémoire (en utilisant `ptr + 1`), on accède au deuxième élément du tableau.**

**-Conclusion :**

**Les pointeurs sont un concept puissant en algorithmique. Ils permettent de manipuler efficacement la mémoire et de créer des structures de données dynamiques. Comprendre les pointeurs est essentiel pour écrire des algorithmes performants et flexibles.**

**2- ALLOCATION**

**L’allocation dynamique de mémoire est un concept crucial en programmation qui permet aux programmes d’ajuster leur taille mémoire au fur et à mesure de leurs besoins. Contrairement à l’allocation statique, où la taille de la mémoire est définie à la compilation, l’allocation dynamique permet de réserver et de libérer de la mémoire pendant l’exécution du programme.**

**Pourquoi l’allocation dynamique ?**

**-Flexibilité : L’allocation dynamique permet de créer des structures de données de taille variable, s’adaptant aux besoins en constante évolution du programme.**

**-Optimisation de la mémoire : L’allocation dynamique permet d’utiliser la mémoire uniquement lorsque nécessaire, réduisant ainsi les gaspillages et améliorant les performances.**

**-Gestion efficace des données : L’allocation dynamique permet de gérer efficacement des structures de données complexes, telles que les listes chaînées et les arbres, où la taille est inconnue à l’avance.**

**-Fonctionnement de l’allocation dynamique :**

**1. Demande de mémoire : Le programme utilise des fonctions spécifiques pour demander de la mémoire au système d’exploitation, spécifiant la quantité de mémoire requise.**

**2. Allocation de mémoire : Le système d’exploitation alloue un bloc de mémoire contiguë de la taille demandée au programme.**

**3. Utilisation de la mémoire : Le programme peut utiliser la mémoire allouée pour stocker des données, créer des variables ou construire des structures de données.**

**4.Libération de mémoire : Lorsque le programme a terminé d’utiliser la mémoire allouée, il doit la libérer pour éviter les fuites de mémoire. Cela permet de rendre la mémoire disponible pour d’autres processus.**

**-Fonctionnalités clés de l’allocation dynamique :**

**-malloc()` :Alloue un bloc de mémoire de la taille spécifiée.**

**-calloc()` :Alloue un bloc de mémoire de la taille spécifiée, initialisé à zéro.**

**-`realloc()` :Modifie la taille d’un bloc de mémoire déjà alloué.**

**-free()` : Libère un bloc de mémoire alloué précédemment.**

**Exemple en pseudo-code :**

**Pseudocode**

**// Allouer un tableau de 10 entiers**

**Pointeur vers entier tableau = malloc(10 \* taille(entier))**

**// Vérifier si l’allocation a réussi**

**Si tableau == NULL alors**

**Afficher « Erreur : allocation mémoire échouée »**

**Sinon**

**// Initialiser les éléments du tableau**

**Pour i de 0 à 9**

**Tableau[i] = i + 1**

**Fin pour**

**// Afficher les éléments du tableau**

**Pour i de 0 à 9**

**Afficher tableau[i]**

**Fin pour**

**// Libérer la mémoire allouée**

**Free(tableau)**

**Fin si**

**Dans cet exemple, `malloc()` est utilisé pour allouer un bloc de mémoire pour un tableau de 10 entiers. Le code vérifie ensuite si l’allocation a réussi avant de l’utiliser. Finalement, `free()` est utilisé pour libérer la mémoire allouée.**

**Conclusion :**

**L’allocation dynamique est un outil essentiel pour la gestion efficace de la mémoire en programmation. En utilisant les fonctions appropriées pour demander, utiliser et libérer la mémoire, les programmeurs peuvent créer des programmes flexibles, performants et résistants aux erreurs de mémoire.**