Cours : Les pointeurs en C

Introduction : Pourquoi apprendre les pointeurs ?

Imaginez que votre programme est comme une grande bibliothèque, et que chaque variable est un livre. Les pointeurs sont comme les index de cette bibliothèque : ils vous disent **où** se trouvent les livres, plutôt que de vous donner directement leur contenu. Avec les pointeurs, vous pouvez :

1. Manipuler directement les données dans la mémoire de l'ordinateur.
2. Écrire des fonctions plus puissantes (comme modifier des variables en dehors de la fonction).
3. Gérer efficacement des structures de données comme les tableaux, les listes chaînées, etc.
4. Contrôler l'allocation dynamique de mémoire.

1. Les bases des pointeurs

1.1 Qu'est-ce qu'un pointeur ?

Un pointeur est une variable spéciale qui contient l'**adresse mémoire** d'une autre variable.

- **Adresse mémoire** : chaque variable dans votre programme est stockée quelque part dans la mémoire de l'ordinateur, et cette position a une adresse unique.
- **Déréférencement** : si vous avez l'adresse mémoire d'une variable, vous pouvez accéder à sa valeur.

1.2 Déclaration d'un pointeur

```
int *p;
```

- `int` : le type de la variable pointée (ici un entier).
- `*` : indique que `p` est un pointeur.
- `p` : contient l'adresse d'une variable de type `int`.

2. Exemple progressif : Manipuler les pointeurs

Étape 1 : Variables et adresses mémoire

```
#include <stdio.h>

int main() {
    int a = 42;

    printf("Valeur de a : %d\n", a);
    printf("Adresse de a : %p\n", &a);

    return 0;
}
```

Explications :

- `&a` : donne l'adresse mémoire de `a`.
- `%p` : format pour afficher une adresse mémoire.

Étape 2 : Déclarer et initialiser un pointeur

```
#include <stdio.h>

int main() {
    int a = 42;
    int *p = &a;  // 'p' contient l'adresse de 'a'

    printf("Adresse stockée dans p : %p\n", p);
    printf("Valeur pointée par p : %d\n", *p);

    return 0;
}
```

- `int *p = &a;` : Le pointeur `p` contient l'adresse de la variable `a`.
- `*p` : Accède à la valeur stockée à l'adresse mémoire pointée.

3. Applications pratiques des pointeurs

3.1 Modifier une variable à l'aide d'une fonction

```
#include <stdio.h>

void modifierValeur(int *p) {
    *p = 99;  // Change la valeur à l'adresse pointée
}

int main() {
    int a = 42;
    printf("Avant modification : %d\n", a);

    modifierValeur(&a);  // Passe l'adresse de 'a' à la fonction

    printf("Après modification : %d\n", a);

    return 0;
}
```

Explications :

- La fonction reçoit un pointeur et utilise `*p` pour modifier la valeur de la variable originale.

3.2 Tableaux et pointeurs

Les noms des tableaux sont en réalité des pointeurs constants.

```
#include <stdio.h>

int main() {
    int tableau[5] = {10, 20, 30, 40, 50};
    int *p = tableau;  // 'p' pointe sur le premier élément du tableau

    for (int i = 0; i < 5; i++) {
        printf("Valeur : %d, Adresse : %p\n", *(p + i), (p + i));
    }

    return 0;
}
```

- `p + i` : pointe sur le i-ème élément du tableau.
- `*(p + i)` : accède à la valeur de cet élément.

4. Gestion dynamique de mémoire avec `malloc` et `free`

La **gestion dynamique de la mémoire** en programmation, particulièrement en langage C, permet d'allouer et de libérer de la mémoire pendant l'exécution d'un programme. Cela est essentiel pour manipuler des structures de données dont la taille n'est pas connue à la compilation, comme des tableaux de taille variable, des listes chaînées, des arbres, etc. Les fonctions principales pour gérer la mémoire dynamiquement en C sont malloc et free.

1. Allocation de mémoire avec malloc

La fonction malloc (pour **memory allocation**) permet de réserver un bloc de mémoire de taille spécifiée dans le **tas** (ou **heap**), une région de la mémoire utilisée pour l'allocation dynamique.

Syntaxe :

void* malloc(size_t taille);

- **taille** : La taille en octets du bloc de mémoire à allouer.
- **Valeur de retour** : Un pointeur de type void* (pointeur générique) vers le début du bloc alloué. Si l'allocation échoue (par exemple, si la mémoire est insuffisante), malloc retourne NULL.

Exemple :

```
#include <stdio.h>
#include <stdlib.h>

int main() {
    int *tableau;
    int n = 5; // Taille du tableau

    // Allocation dynamique d'un tableau de 5 entiers
    tableau = (int*)malloc(n * sizeof(int));

    if (tableau == NULL) {
        printf("Échec de l'allocation de mémoire\n");
        return 1;
    }

    // Utilisation du tableau
    for (int i = 0; i < n; i++) {
        tableau[i] = i * 2;
    }

    // Affichage des valeurs
    for (int i = 0; i < n; i++) {
        printf("%d ", tableau[i]);
    }

    // Libération de la mémoire
    free(tableau);

    return 0;
}
```

Explication :

- malloc(n * sizeof(int)) alloue de la mémoire pour un tableau de n entiers.
- sizeof(int) donne la taille en octets d'un entier sur la machine.
- Le pointeur retourné par malloc est casté en int* pour correspondre au type de données stockées.

2. Libération de mémoire avec free

La fonction free permet de libérer la mémoire précédemment allouée avec malloc (ou d'autres fonctions similaires comme calloc ou realloc). Cela rend la mémoire disponible pour d'autres allocations.

Syntaxe :

```
void free(void* ptr);
```

- **ptr** : Un pointeur vers le bloc de mémoire à libérer. Ce pointeur doit avoir été retourné par malloc, calloc, ou realloc.
- **Attention** : Après avoir appelé free, le pointeur ne doit plus être utilisé, car il pointe vers une zone de mémoire invalide.

Exemple :

```
free(tableau); // Libère la mémoire allouée pour le tableau
```

3. Bonnes pratiques

1. **Vérifier le retour de malloc** :
   Toujours vérifier si malloc a retourné NULL avant d'utiliser la mémoire allouée. Cela évite des erreurs de segmentation.

```
int *ptr = (int*)malloc(100 * sizeof(int));
if (ptr == NULL) {
   printf("Échec de l'allocation de mémoire\n");
   exit(1); // Quitter le programme en cas d'échec
}
```

2. **Libérer toute la mémoire allouée** :
   Pour éviter les **fuites de mémoire**, assurez-vous de libérer toute la mémoire allouée dynamiquement lorsque vous n'en avez plus besoin.
3. **Ne pas libérer deux fois la même mémoire** :
   Libérer un pointeur déjà libéré ou non alloué entraîne un comportement indéfini.

```
free(ptr); // Correct
free(ptr); // Erreur : double libération
```

4. **Ne pas utiliser de pointeurs après free** :
   Après avoir appelé free, le pointeur ne doit plus être utilisé.

```
free(ptr);
ptr = NULL; // Bonne pratique : assigner NULL après libération
```

5. **Utiliser calloc pour l'allocation initialisée** :
   Si vous avez besoin d'une mémoire initialisée à zéro, utilisez calloc au lieu de malloc.

```
int *ptr = (int*)calloc(100, sizeof(int)); // Alloue et initialise à zéro
```

6. **Utiliser realloc pour redimensionner la mémoire** :
   Si vous devez redimensionner un bloc de mémoire déjà alloué, utilisez realloc.

```
ptr = (int*)realloc(ptr, 200 * sizeof(int)); // Redimensionne à 200 entiers
```

4. Différence entre malloc et calloc

- **malloc** : Alloue un bloc de mémoire sans l'initialiser. Le contenu de la mémoire est indéfini.
- **calloc** : Alloue un bloc de mémoire et l'initialise à zéro. Utile pour les tableaux et les structures.

5. Exemple complet

```
#include <stdio.h>
#include <stdlib.h>

int main() {
   int *tableau;
   int n = 10;

   // Allocation dynamique
   tableau = (int*)malloc(n * sizeof(int));
   if (tableau == NULL) {
      printf("Échec de l'allocation de mémoire\n");
      return 1;
```

```
    }

    // Remplissage du tableau
    for (int i = 0; i < n; i++) {
        tableau[i] = i + 1;
    }

    // Affichage du tableau
    printf("Tableau : ");
    for (int i = 0; i < n; i++) {
        printf("%d ", tableau[i]);
    }
    printf("\n");

    // Libération de la mémoire
    free(tableau);
    tableau = NULL; // Bonne pratique

    return 0;
}
```

6. Conclusion

La gestion dynamique de la mémoire avec malloc et free est un outil puissant en C, mais elle nécessite une attention particulière pour éviter les erreurs courantes comme les fuites de mémoire, les accès invalides, ou les double libérations. En suivant les bonnes pratiques, vous pouvez écrire des programmes efficaces et robustes.