PROGRAMMATION II

Pr. OMARI Kamal

FACULTE POLYDISCIPLINAIRE D'OUARZAZATE

March 9, 2024

Chapitre 1 : Pointeurs et allocation dynamique

Objectifs

Ce cours vise à fournir une compréhension approfondie des pointeurs et de l'allocation dynamique de mémoire en langage C. Ces principaux objectifs sont :

- Comprendre le concept de pointeurs et leur utilisation dans la manipulation directe de la mémoire.
- Savoir allouer et libérer dynamiquement la mémoire à l'aide des fonctions `malloc()`, `calloc()`, `realloc()` et `free()`.
- Maîtriser les bonnes pratiques d'utilisation des pointeurs pour éviter les erreurs de segmentation et les fuites de mémoire.

Introduction

Les pointeurs sont l'une des caractéristiques les plus puissantes et délicates du langage C. Un pointeur est une variable qui contient l'adresse mémoire d'une autre variable. Ils permettent un accès direct à la mémoire, offrant ainsi une flexibilité considérable pour la manipulation des données.

Définition et utilisation des pointeurs

En C, un pointeur est une variable qui contient l'adresse mémoire d'une autre variable. On déclare un pointeur par l'instruction suivante :

type *nom_du_pointeur;

où type est le type de l'objet pointé et * est appelé opérateur d'indirection (déréférencement).

Définition et utilisation des pointeurs

Afin d'accéder indirectement à la valeur d'une variable à l'aide d'un pointeur, on utilise le code ci-dessous.

Exemple

```
int x = 10;
int *ptr;  // Déclaration d'un pointeur.
ptr = &x;  // Initialisation du pointeur avec l'adresse
de x.
printf("Valeur de x : %d", *ptr);  // Affiche la valeur
de x via le pointeur ptr.
```

Définition et utilisation des pointeurs

- * : Opérateur de déréférencement, permet d'accéder à la valeur pointée par un pointeur.
- & : Opérateur d'adresse, permet de récupérer l'adresse mémoire d'une variable.

Exemple

```
int x = 10; // Déclaration d'une variable entière x.
int *ptr = &x; // Déclaration d'un pointeur ptr pointant vers l'adresse mémoire de x.
printf("Adresse de x : %p", &x); // Affiche l'adresse mémoire de x.
printf("Valeur de x : %d", *ptr); // Affiche la valeur de x via le pointeur ptr.
```

Définition et utilisation des pointeurs

Qu'affiche le programme suivant :

```
#include<stdio.h>
void main()
{
int A = 2 , B = 3 , C=-1 , D;
int *ptr1,*ptr2;
ptr1 = &A;
ptr2 = &B;
C = *ptr1 + *ptr2;
ptr1 = &C;
++*ptr1;
D = *ptr1;
*ptr1 = *ptr2;
*ptr2 = D;
printf("A = %d B = %d C = %d D = %d", A, B, C, D);
}
```

Définition et utilisation des pointeurs

Correction:

La solution est A = 2, B = 6, C = 3, D = 6.

Arithmétique des pointeurs

L'arithmétique des pointeurs est une fonctionnalité puissante en langage C qui permet de manipuler les adresses mémoire des variables de manière efficace. Voici une explication de quelques opérations couramment utilisées en arithmétique des pointeurs :

- Incrémenter ou Décrémenter un pointeur.
- Additionner ou Soustraire un entier à un pointeur.
- La différence entre deux pointeurs.

Arithmétique des pointeurs

- Incrémenter ou Décrémenter un pointeur.

Lorsque vous incrémentez un pointeur en C avec l'opérateur `(++)`, le pointeur avance vers l'adresse mémoire suivante, augmentant ainsi sa valeur de la taille du type de données pointé. De même, lors de la décrémentation d'un pointeur avec l'opérateur `(--)`, le pointeur recule vers l'adresse mémoire précédente, diminuant sa valeur de la taille du type de données pointé.

Par exemple

```
Soit ptr un pointeur de type T.
Soit a la valeur de ptr (adresse contenue dans ptr).
Si on exécute ptr++, alors la nouvelle valeur de ptr
sera égale à : a + sizeof(T).
```

Arithmétique des pointeurs

Incrémenter un pointeur

```
#include<stdio.h>
void main()
{
int i=10;
int *ptr;
ptr=&i;
printf("la valeur de ptr avant l'incrémentation :
%d",ptr);
ptr++;
printf("la valeur de ptr après l'incrémentation :
%d",ptr);
}
```

Le même principe s'applique pour la décrémentation (`ptr--`).

Arithmétique des pointeurs

- Additionner ou Soustraire un entier à un pointeur.

Les opérations d'addition et de soustraction d'un entier à un pointeur déplacent le pointeur dans la mémoire.
L'addition avance le pointeur d'un nombre d'octets égal à l'entier multiplié par la taille du type pointé, tandis que la soustraction recule de la même manière.

Par exemple

```
Soit ptr un pointeur de type T.
Soit a la valeur de ptr (adresse contenue dans ptr).
Soit i un entier.
Si on exécute ptr+i alors la nouvelle valeur sera
égale à : a + i*sizeof(T).
```

Arithmétique des pointeurs

Additionner un entier à un pointeur.

```
#include<stdio.h>
void main()
{
int i=10;
int *ptr;
ptr=&i;
printf("la valeur de ptr avant l'incrémentation :
%d",ptr);
ptr+=2;
printf("la valeur de ptr après l'incrémentation :
%d",ptr);
}
```

Le même principe s'applique pour la soustraction.

Arithmétique des pointeurs

- La différence entre deux pointeurs.

La différence entre deux pointeurs permet de déterminer le nombre d'éléments entre les adresses mémoire pointées par ces deux pointeurs. C'est une opération essentielle lors de la manipulation de tableaux.

Par exemple

```
Soit ptr1 et ptr2 deux pointeurs de type T.
Soit a1 et a2 la valeur de ptr1 et ptr2.
Si on exécute ptr1-ptr2 on obtient (a1-a2)/sizeof(T).
```

N.B: Les opérateurs de comparaison sont également applicables aux pointeurs, à condition de comparer des pointeurs qui pointent vers des objets de même type.

Arithmétique des pointeurs

Qu'affiche le programme suivant Si &i=1230 ?

```
#include<stdio.h>
void main()
{
int i = 5;
int *p1, *p2;
p1 = &i;
p2 = p1 - 1;
printf("p1 = %d, p2 = %d, p1-p2 = %d",p1,p2,p1-p2);
}
```

Arithmétique des pointeurs

Correction:

La solution est p1 = 1230, p2 = 1226, p1-p2=1.

Fonctions malloc(), calloc(), realloc() et free()

L'allocation dynamique de mémoire en C est un processus qui permet de réserver de la mémoire à l'exécution d'un programme, plutôt qu'à la compilation. Cela permet de gérer dynamiquement la mémoire nécessaire à l'exécution d'un programme, en particulier lorsque la taille de la mémoire requise n'est connue qu'au moment de l'exécution.

Fonctions malloc(), calloc(), realloc() et free()

En langage C, l'allocation dynamique de mémoire se fait à l'aide de fonctions telles que `malloc(), calloc() et realloc()`, tandis que la libération de mémoire allouée dynamiquement se fait avec la fonction `free()`. Pour utiliser ces fonctions, il est nécessaire d'inclure la bibliothèque `stdlib.h`.

Fonctions malloc(), calloc(), realloc() et free()

- La fonction `malloc()`.

La fonction `malloc()` en langage C est utilisée pour allouer dynamiquement un bloc de mémoire de taille spécifiée en octets. Son prototype est défini dans la bibliothèque stdlib.h comme suit :

Prototype

```
void *malloc(size_t size);
```

Cette fonction prend un argument :

- size : la taille en octets.

La fonction `malloc()` prend en paramètre la taille en octets du bloc de mémoire à allouer et renvoie un pointeur vers le début de ce bloc. Si l'allocation échoue, malloc() renvoie NULL.

Fonctions malloc(), calloc(), realloc() et free()

- La fonction `malloc()`.

Exemple 1

```
#include <stdio.h>
#include <stdlib.h>
int main(){
size_t size = 10 * sizeof(int); // Taille en octets.
int *ptr;
// Allocation dynamique de mémoire
ptr = (int *)malloc(size);
if (ptr == NULL) {
printf("L'allocation dynamique de mémoire a échoué.\n");
return 1;
}
// Utilisation de la taille allouée dynamiquement...
// Libération de la mémoire allouée dynamiquement
return 0;
}
```

Fonctions malloc(), calloc(), realloc() et free()

- La fonction `malloc()`.

Exemple 2

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
int N;
int *ptr;
// Demande à l'utilisateur la taille.
printf("Entrez la taille : ");
scanf("%d", &N);
// Allocation dynamique de mémoire
ptr = (int *)malloc(N * sizeof(int));
if (ptr == NULL) {
printf("L'allocation dynamique de mémoire a échoué.\n");
return 1;
}
// Utilisation de la taille allouée dynamiquement...
// Libération de la mémoire allouée dynamiquement
return 0;
}
```

Fonctions malloc(), calloc(), realloc() et free()

- La fonction `calloc()`.

La fonction `calloc()` en langage C est utilisée pour allouer dynamiquement de la mémoire pour un tableau d'éléments et les initialise à zéro. Son prototype est défini dans la bibliothèque `stdlib.h` comme suit :

Prototype

```
void *calloc(size_t num_elements,size_t element_size);
```

Cette fonction prend deux arguments :

- num_elements : le nombre d'éléments à allouer.
- element_size : la taille en octets de chaque élément.

La fonction `calloc()` retourne un pointeur vers le début du bloc de mémoire alloué. Si l'allocation échoue, elle renvoie NULL.

Fonctions malloc(), calloc(), realloc() et free()

- La fonction `calloc()`.

Exemple

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
int N;
int *ptr;
// Demande à l'utilisateur la taille.
printf("Entrez la taille : ");
scanf("%d", &N);
// Allocation dynamique de mémoire
ptr = (int *)calloc(N , sizeof(int));
if (ptr == NULL) {
printf("L'allocation dynamique de mémoire a échoué.\n");
return 1;
}
// Utilisation de la taille allouée dynamiquement...
// Libération de la mémoire allouée dynamiquement
return 0;
}
```

Fonctions malloc(), calloc(), realloc() et free()

Exercice

Est-il possible d'écrire un programme en langage C qui réalise la même fonctionnalité que la fonction calloc(), mais en utilisant uniquement la fonction malloc() ? Si oui, veuillez écrire le programme correspondant.

Fonctions malloc(), calloc(), realloc() et free()

Correction:

```
#include <stdio.h>
#include <stdlib.h>
int main() {
int N;
int *ptr;
// Demande à l'utilisateur la taille.
printf("Entrez la taille : ");
scanf("%d", &N);
// Allocation dynamique de mémoire avec malloc()
ptr = (int *)malloc(N * sizeof(int));
if (ptr == NULL) {
printf("L'allocation dynamique de mémoire a échoué.");
return 1;
}// Initialisation de chaque élément à zéro
for (int i = 0; i < N; i++) {
*(ptr + i) = 0;
}
// Libération de la mémoire allouée dynamiquement
return 0;
}
```

Fonctions malloc(), calloc(), realloc() et free()

- La fonction `realloc()`.

La fonction `realloc()` en langage C est utilisée pour modifier la taille d'un bloc de mémoire déjà alloué dynamiquement. Son prototype est défini dans la bibliothèque `stdlib.h` comme suit :

Prototype

```
void *realloc(void *ptr, size_t size);
```

Cette fonction prend deux arguments :

- ptr : un pointeur vers le bloc de mémoire déjà alloué dynamiquement que vous souhaitez redimensionner. Si ptr est `NULL`, `realloc()` agira comme `malloc()`.
- size : la nouvelle taille en octets que vous souhaitez attribuer au bloc de mémoire. Si size est 0 et ptr n'est pas `NULL`, la fonction se comportera comme `free()` et libérera le bloc de mémoire.

Fonctions malloc(), calloc(), realloc() et free()

- La fonction `realloc()`.

Exemple

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
int *ptr;
// Allocation dynamique de mémoire
ptr = (int *)malloc(5 * sizeof(int));
if (ptr == NULL) {
printf("L'allocation dynamique de mémoire a échoué.\n");
return 1;
}
// Redimensionnement du bloc de mémoire à 10 entiers
ptr = (int *)realloc(ptr, 10 * sizeof(int));
if (ptr == NULL) {
printf("Le redimensionnement de la mémoire a échoué.");
return 1;
}
// Utilisation de la mémoire redimensionnée...
// Libération de la mémoire allouée dynamiquement
return 0;
}
```

Fonctions malloc(), calloc(), realloc() et free()

- La fonction `free()`.

La fonction `free()` en langage C est utilisée pour libérer la mémoire allouée dynamiquement à l'aide des fonctions telles que `malloc()`, `calloc()` ou `realloc()`. Son prototype est défini dans la bibliothèque `stdlib.h` comme suit :

Prototype

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

Cette fonction prend un argument :

- ptr : un pointeur vers le bloc de mémoire que vous souhaitez libérer.

Fonctions malloc(), calloc(), realloc() et free()

- La fonction `free()`.

Exemple

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
int N;
int *ptr;
// Demande à l'utilisateur la taille.
printf("Entrez la taille : ");
scanf("%d", &N);
// Allocation dynamique de mémoire
ptr = (int *)malloc(N * sizeof(int));
if (ptr == NULL) {
printf("L'allocation dynamique de mémoire a échoué.\n");
return 1;
}
// Utilisation de la taille allouée dynamiquement...
// Libération de la mémoire allouée dynamiquement
free(ptr);
return 0;
}
```

Gestion des erreurs lors de l'allocation

Lors de l'allocation dynamique de mémoire en C à l'aide de fonctions telles que malloc(), calloc() ou realloc(), il est essentiel de gérer les erreurs qui peuvent survenir si la mémoire demandée n'est pas disponible. Cela garantit un comportement robuste de votre programme et évite les plantages inattendus. Voici quelques bonnes pratiques pour gérer les erreurs lors de l'allocation de mémoire :

Gestion des erreurs lors de l'allocation

- Vérifiez si la mémoire a été allouée avec succès :

Après chaque appel à une fonction d'allocation de mémoire, vérifiez si le pointeur retourné est NULL. Cela indique que l'allocation a échoué.

Exemple

```
ptr = (int *)malloc(N * sizeof(int));
if (ptr == NULL) {
printf("L'allocation dynamique de mémoire a échoué.\n");
return 1;
}
```

Gestion des erreurs lors de l'allocation

- Utilisez calloc() si vous avez besoin d'initialiser à zéro :

Si vous avez besoin que la mémoire allouée soit initialisée à zéro, utilisez plutôt la fonction calloc(). Elle garantit que la mémoire allouée est initialisée à zéro.

Exemple

```
ptr = (int *)calloc(N, sizeof(int));
if (ptr == NULL) {
printf("L'allocation dynamique de mémoire a échoué.");
return 1;
}
```

Gestion des erreurs lors de l'allocation

- Gérez les erreurs de redimensionnement avec realloc() :

Lorsque vous utilisez realloc() pour redimensionner un bloc de mémoire, assurez-vous de vérifier si le pointeur retourné est NULL pour détecter les erreurs de redimensionnement.

Exemple

```
ptr = (int *)realloc(ptr, N * sizeof(int));
if (ptr == NULL) {
printf("Le redimensionnement de la mémoire a échoué.");
return 1;
}
```

Gestion des erreurs lors de l'allocation

- Libérez la mémoire correctement :

Assurez-vous de libérer la mémoire allouée dynamiquement avec free() une fois que vous avez fini de l'utiliser. Cela évite les fuites de mémoire.

Exemple

```
free(ptr);
```

N.B: En suivant ces bonnes pratiques, vous pouvez rendre votre code plus robuste et éviter les problèmes liés à la gestion de la mémoire dynamique en C.

Pointeur et tableau à une dimension

- Un tableau à une dimension est un pointeur sur un ensemble d'éléments de même type contigües en mémoire.
- Le tableau T est un pointeur constant représentant l'adresse de la première case du tableau, T = &T[0].
- L'adresse de la case d'indice i est T+i, car &T[i] = T+i.
- Le contenu de la case d'indice i est *(T+i), car T[i] = *(T+i).

Pointeur et tableau à une dimension

Exercice

Ecrire un programme en C qui réalise les opérations suivantes :
1) La lecture d'un tableau.
2) L'affichage du tableau.
3) L'affichage des adresses des éléments.
En utilisant la déclaration suivante : int tab[10], n=5, i;

Pointeur et tableau à une dimension

Correction:

```
#include <stdio.h>
int main() {
int tab[10], n = 5, i;

// Lecture du tableau
printf("Entrez les %d éléments du tableau :\n", n);
for (i = 0; i < n; i++) {
printf("Element %d : ", i + 1);
scanf("%d", tab + i);
}
// Affichage du tableau
printf("\nLe tableau que vous avez saisi est :\n");
for (i = 0; i < n; i++) {
printf("%d ", *(tab + i));
}
// Affichage des adresses des éléments
printf("des éléments du tableau :");
for (i = 0; i < n; i++) {
printf("&tab[%d] = %p", i, (tab + i));
}
return 0;
}
```

Accès aux éléments d'un tableau à une dimension par un pointeur

Accéder aux éléments d'un tableau à une dimension par un pointeur est une pratique courante en langage C. Voici comment cela fonctionne :

Exemple

```
int tab[] = 10, 20, 30, 40, 50;
```

Nous pouvons créer un pointeur qui pointe vers le premier élément de ce tableau :

Exemple

```
int *ptr = tab;
```

Saisie des éléments d'un tableau à une dimension par un pointeur

- Saisie des éléments d'un tableau à une dimension par un pointeur.

Exemple 1

```
#include <stdio.h>
int main() {
int tab[5];
int *ptr = tab; // Pointeur vers le premier élément
du tableau
// Saisie des éléments du tableau à l'aide du
pointeur
printf("Saisie des éléments du tableau à l'aide du
pointeur:\n");
for (int i = 0; i < 5; i++) {
printf("Entrez un entier pour la case %d: ", i);
scanf("%d", ptr + i); // Saisie de la valeur à
l'index i
}return 0;
}
```

Exemple 2

```
#include <stdio.h>
int main() {
int tab[5];
int *ptr = tab; // Pointeur vers le premier élément
du tableau
// Saisie des éléments du tableau à l'aide du
pointeur
printf("Saisie des éléments du tableau à l'aide du
pointeur:\n");
for (ptr = tab; ptr < tab + 5; ptr++) {
printf("Entrez un entier pour la case %d: ", i);
scanf("%d", ptr);
}return 0;
}
```

Accès aux éléments d'un tableau à une dimension par un pointeur

- Accès aux éléments d'un tableau à une dimension par un pointeur.

Exemple 1

```
#include <stdio.h>
int main() {
int i, tab[5] = {10, 20, 30, 40, 50};
int *ptr = tab; // Pointeur vers le premier élément
du tableau
// Accès aux éléments du tableau à l'aide du
pointeur
printf("Accès aux éléments du tableau à l'aide du
pointeur:\n");
for (i = 0; i < 5; i++) {
printf("%d ", *(ptr + i)); // *(ptr + i) accède à
l'élément de l'index i
}return 0;
}
```

Exemple 2

```
#include <stdio.h>
int main() {
int tab[5] = {10, 20, 30, 40, 50};
int *ptr = tab; // Pointeur vers le premier élément
du tableau
// Accès aux éléments du tableau à l'aide du pointeur
printf("Accès aux éléments du tableau à l'aide du
pointeur:\n");
for (ptr = tab; ptr < tab + 5; ptr++) {
printf("%d ", *ptr);
}return 0;
}
```

Pointeur et tableau à une dimension

Exercice

Soit P un pointeur qui pointe sur un tableau A :

```
int A [] = {12,23,34,45,56,67,78,89,90}
int *P;
P=A;
```

Quelles valeurs ou adresses fournissent ces expressions :

1) `*P+2`
2) `*(P+2)`
3) `&A[4]-3`
4) `A+3`
5) `&A[7]-P`
6) `P+(*P-10)`
7) `*(P+*(P+8)-A[7])`

Pointeur et tableau à une dimension

Correction

1) `*P+2 = 14`
2) `*(P+2) = 34`
3) `&A[4]-3= &A[1]`
4) `A+3 = &A[3]`
5) `&A[7]-P = 7` *(la distance entre A[7] et P)*
6) `P+(*P-10) = &A[2]`
7) `*(P++*(P+8)-A[7]) = 23`

Pointeur et tableau à une dimension

Qu'affiche le programme suivant ?

```
#include<stdio.h>
void main()
{
int a[] = {0, 1, 2, 3, 4};
int i, *p ;

// Première boucle
for (p = &a[0]; p <= &a[4]; p++)
printf("%d", *p);

// Deuxième boucle
for (p = a, i=0; p+i <= a+4; p++, i++) printf("%d", *(p+i));

// Troisième boucle
for (p = a+4; p >= a; p--) printf("%d", *p);

// Quatrième boucle
for (p = a+4; p >= a; p--)
printf("%d", a[p-a]);
}
```

Pointeur et tableau à une dimension

Exercice

Ecrire un programme qui réalise les opérations suivantes :
1) La lecture d'un tableau.
2) L'affichage du tableau.
En utilisant la déclaration suivante : int T[100]; int n;
int *P; P=T;

Pointeur et tableau à une dimension

Correction

```
#include <stdio.h>
int main() {
int T[100];
int n;
int *P;
P = T;

// Lecture du tableau
printf("Entrez la taille du tableau : ");
scanf("%d", &n);
printf("Entrez les éléments du tableau :\n");
for (P = T; P < T + n; P++) {
scanf("%d", P);
}

// Affichage du tableau
printf("Le tableau est :\n");
for (P = T; P < T + n; P++) {
printf("%d ", *P);
}
return 0;
}
```

Allocation Dynamique de Tableaux Unidimensionnels

Les tableaux unidimensionnels avec allocation dynamique de mémoire permettent de créer des tableaux dont la taille peut être déterminée à l'exécution du programme.

Allocation Dynamique de Tableaux Unidimensionnels

Exemple

```
#include <stdlib.h>
#include <stdio.h>
void main()
{int n;
int *tab;
...
tab = (int*)malloc(n * sizeof(int));
//on peut utiliser calloc en cas d'initialisation à 0
...
free(tab);
}
```

Les éléments de tab sont manipulés avec l'opérateur d'indexation `[]`, exactement comme pour les tableaux.

Allocation Dynamique de Tableaux Unidimensionnels

Exemple

```
#include <stdio.h>
#include <stdlib.h>
void main() {
    int *p,*ptr;
    int n;
    printf("Entrez la taille du tableau : ");
    scanf("%d", &n);
    ptr = (int*)malloc(n * sizeof(int));
    if (ptr == NULL) {
        printf("Erreur d'allocation de mémoire.\n");
        return 1;
    }
    printf("Entrez les valeurs pour le tableau :\n");
    for (p = ptr; p < ptr + n; p++) {
        printf("Valeur : ");
        scanf("%d", p);
    }
    printf("Le tableau est : ");
    for (p = ptr; p < ptr + n; p++) {
        printf("%d ", *p);
    }
    printf("\n");
    free(ptr);
}
```

Allocation Dynamique de Tableaux Unidimensionnels

Exercice

Écrire un programme en langage C qui permet à l'utilisateur de gérer dynamiquement un tableau unidimensionnel d'entiers. Le programme doit réaliser les opérations suivantes :

1) Demander à l'utilisateur de saisir la taille initiale du tableau.
2) Allouer dynamiquement de la mémoire pour le tableau en utilisant la fonction malloc() avec la taille saisie.
3) Si l'allocation de mémoire échoue, afficher un message d'erreur et quitter le programme.
4) Demander à l'utilisateur de saisir les valeurs pour chaque élément du tableau.
5) Afficher les éléments du tableau.
6) Demander à l'utilisateur de saisir une nouvelle taille pour le tableau.
7) Utiliser la fonction realloc() pour redimensionner le tableau en fonction de la nouvelle taille saisie.
8) Si la réallocation de mémoire échoue, afficher un message d'erreur et libérer la mémoire allouée pour le tableau original avant de quitter le programme.
9) Libérer la mémoire allouée dynamiquement pour le tableau avant de quitter le programme.

Votre programme doit être capable de gérer efficacement l'allocation dynamique de mémoire et la manipulation des tableaux unidimensionnels en suivant les étapes ci-dessus.

Allocation Dynamique de Tableaux Unidimensionnels

Code C - Partie 1

```
#include <stdio.h>
#include <stdlib.h>
int main() {
int taille_initiale;
printf("Entrez la taille initiale du tableau : ");
scanf("%d", &taille_initiale);
int *tableau = (int*)malloc(taille_initiale *
sizeof(int));
if (tableau == NULL) {
printf("Erreur d'allocation de mémoire pour le
tableau initial.\n");
return 1;
}
printf("Entrez les valeurs pour chaque élément du
tableau :\n");
for (int i = 0; i < taille_initiale; i++) {
scanf("%d", &tableau[i]);
}
printf("Les éléments du tableau sont :\n");
for (int i = 0; i < taille_initiale; i++) {
printf("%d ", tableau[i]);
}
printf("\n");
int nouvelle_taille;
printf("Entrez la nouvelle taille du tableau : ");
scanf("%d", &nouvelle_taille);
```

Code C - Partie 2

```
int *nouveau_tableau = (int*)realloc(tableau,
nouvelle_taille * sizeof(int));
if (nouveau_tableau == NULL) {
printf("Erreur de réallocation de mémoire pour le
nouveau tableau.\n");
free(tableau); // Libérer la mémoire allouée pour le
tableau initial
return 1;
}
tableau = nouveau_tableau;
printf("Entrez les nouvelles valeurs pour les
éléments supplémentaires du tableau :\n");
for (int i = taille_initiale; i < nouvelle_taille;
i++) {
scanf("%d", &tableau[i]);
}
printf("Les éléments du tableau après
redimensionnement sont :\n");
for (int i = 0; i < nouvelle_taille; i++) {
printf("%d ", tableau[i]);
}
free(tableau);
return 0;
}
```

Pointeur et tableau à deux dimensions

- Un tableau à deux dimensions est par définition, un tableau de tableaux. Il s'agit donc d'un pointeur vers un pointeur.
- Considérons le tableau à deux dimensions défini par : int T[M][N];
- T c'est un pointeur constant qui représente l'adresse de la première case du tableau &T[0][0].
- L'adresse de la case d'indice i est &T[i][j], c'est aussi T[i]+j ou bien *(T+i)+j.
- Le contenu de la case d'indice i est T[i][j], c'est aussi *(*(T+i)+j) ou bien *(T[i]+j).

Pointeur et tableau à deux dimensions

Pour créer un pointeur P vers un tableau à deux dimensions T[M][N] de type int, il suffit d'utiliser la syntaxe suivante :

- int *P=T; ou bien *P=T[0][0];

La manipulation du tableau T peut se faire en utilisant le pointeur P avec la syntaxe suivante :

- Adresse: P+i*C+j : L'adresse de l'élément T[i][j].
- Valeur: *(P+i*C+j) : La valeur de l'élément T[i][j].

Pointeur et tableau à deux dimensions

Exercice

Écrivez un programme en langage C qui permet à l'utilisateur de saisir les éléments d'une matrice de dimensions L lignes et C colonnes, puis d'afficher cette matrice.

Pointeur et tableau à une dimension

Correction

```
#include <stdio.h>
#define MAX_LIGNES 100
#define MAX_COLONNES 100
void main() {
int matrice[MAX_LIGNES][MAX_COLONNES];
int i ,j, L, C;
int *p=matrice;
printf("Entrez le nombre de lignes de la matrice : ");
scanf("%d", &L);
printf("Entrez le nombre de colonnes de la matrice : ");
scanf("%d", &C);
printf("Entrez les éléments de la matrice :\n");
for (i = 0; i < L; i++) {
for (j = 0; j < C; j++) {
printf("Element [%d][%d] : ", i, j);
scanf("%d", P + i * C + j);
}
}
for (i = 0; i < L; i++) {
for (j = 0; j < C; j++) {
printf("%d ", *(P + i * C + j));
}
printf("\n");
}
}
```

Allocation Dynamique de Tableaux à Deux Dimensions

L'allocation dynamique de tableaux à deux dimensions en langage C permet de créer des matrices dont la taille peut être déterminée pendant l'exécution du programme. Contrairement aux tableaux statiques dont la taille doit être connue à la compilation, les tableaux dynamiques offrent une plus grande flexibilité en permettant à l'utilisateur de spécifier la taille à l'exécution.

Allocation Dynamique de Tableaux à Deux Dimensions

Exemple

```
#include <stdio.h>
#include <stdlib.h>
void main()
{
int L, C;
int **tab;
tab = (int**)malloc(L * sizeof(int*));
for (i = 0; i < L; i++)
tab[i] = (int*)malloc(C * sizeof(int));
....
for (i = 0; i < L; i++)
free(tab[i]);
free(tab);
}
```

Pointeur et tableau à deux dimensions

Exercice

Écrivez un programme en langage C qui permet à l'utilisateur de saisir les éléments d'une matrice de dimensions L lignes et C colonnes, puis d'afficher cette matrice. Assurez-vous d'utiliser l'allocation dynamique de mémoire pour créer la matrice.

Allocation Dynamique de Tableaux Unidimensionnels

Code C - Partie 1

```
#include <stdio.h>
#include <stdlib.h>
int main() {
int i, j, L, C;

printf("Entrez le nombre de lignes de la matrice:");
scanf("%d", &L);
printf("Entrez le nombre de colonnes de la
matrice:");
scanf("%d", &C);

int **matrice = (int **)malloc(L * sizeof(int *));
for (i = 0; i < L; i++) {
matrice[i] = (int *)malloc(C * sizeof(int));
}

printf("Entrez les éléments de la matrice :\n");
for (i = 0; i < L; i++) {
for (j = 0; j < C; j++) {
printf("Element [%d][%d] : ", i, j);
scanf("%d", &matrice[i][j]);
}
}
```

Code C - Partie 2

```
printf("\nMatrice :\n");
for (i = 0; i < L; i++) {
for (j = 0; j < C; j++) {
printf("%d ", matrice[i][j]);
}
printf("\n");
}

for (i = 0; i < L; i++) {
free(matrice[i]);
}
free(matrice);

return 0;
}
```

Fuites de mémoire

La fuite de mémoire est un problème courant dans les programmes écrits en langage C en raison de la gestion manuelle de la mémoire. Voici quelques exemples de causes de fuite de mémoire, les conséquences de ces fuites, et des solutions pour les éviter.

Fuites de mémoire

- Oubli de libération de mémoire :

L'allocation de mémoire dynamique avec malloc() ou calloc() doit être suivie d'une libération avec free(). L'oubli de cette libération peut entraîner une fuite de mémoire. Par exemple :

Exemple

```
// Allocation de mémoire pour un tableau d'entiers
int *ptr = (int*)malloc(sizeof(int) * 100);
// Oubli de la libération de mémoire
// free(ptr);
```

Fuites de mémoire

- Références croisées :

Lorsque plusieurs pointeurs référencent la même zone mémoire et que l'un d'eux est libéré, mais pas les autres, cela peut conduire à une fuite de mémoire. Par exemple :

Exemple

```
int *ptr1 = (int*)malloc(sizeof(int));
int *ptr2 = ptr1;
// Libération d'une seule des références
free(ptr1);
```

Fuites de mémoire

- Boucles infinies :

Si une boucle alloue continuellement de la mémoire sans jamais la libérer, une fuite de mémoire se produira. Par exemple :

Exemple

```
// Boucle qui alloue de la mémoire sans libération
while (1) {
int *ptr = (int*)malloc(sizeof(int));
}
```

Fuites de mémoire

- Conséquences de la fuite de mémoire en C :

1) **Épuisement des ressources système** : Les allocations non libérées peuvent épuiser progressivement la mémoire disponible sur le système, entraînant une dégradation des performances ou même un crash du programme.

2) **Instabilité du programme** : Les fuites de mémoire peuvent entraîner des comportements imprévisibles et des plantages du programme, car la mémoire n'est pas gérée correctement.

Fuites de mémoire

- Solutions pour éviter les fuites de mémoire en C :

1) **Libération appropriée de la mémoire** : Assurez-vous de libérer toute mémoire allouée dynamiquement à l'aide de la fonction free() une fois qu'elle n'est plus nécessaire.

2) **Évitez les références croisées** : Assurez-vous qu'une zone mémoire est libérée uniquement lorsque toutes les références à cette zone ont été libérées.

3) **Analyse statique et dynamique** : Utilisez des outils d'analyse statique et dynamique pour détecter les fuites de mémoire potentielles et les corriger.

Accès invalide à la mémoire

En programmation en langage C, la gestion de la mémoire est une tâche cruciale. Des erreurs dans la manipulation de la mémoire peuvent conduire à des comportements indéterminés, voire à des pannes de programme. L'un des problèmes les plus courants est l'accès invalide à la mémoire, qui survient lorsque le programme tente d'accéder à une zone de mémoire qui ne lui est pas allouée. Cela peut se produire pour diverses raisons, telles que l'accès à une mémoire non initialisée, l'utilisation de pointeurs incorrects ou l'écriture en dehors des limites d'un tableau.

Types d'Accès Invalide à la Mémoire

- Dé-référencement de pointeurs non initialisés :

Lorsqu'un pointeur est utilisé sans avoir été correctement initialisé avec une adresse de mémoire valide, tout dé-référencement de ce pointeur entraînera un accès invalide à la mémoire.

Exemple

```
int *ptr;
*ptr = 10; // Accès invalide à la mémoire car ptr n'est pas initialisé
```

Types d'Accès Invalide à la Mémoire

- Dépassement de limites de tableau :

L'accès à un élément d'un tableau en dehors de ses limites définies peut entraîner un accès invalide à la mémoire. Cela peut conduire à des résultats imprévisibles ou à un plantage du programme.

Exemple

```
int arr[5];
arr[5] = 10; // Accès invalide à la mémoire car l'index dépasse les
limites du tableau
```

Types d'Accès Invalide à la Mémoire

- Pointeurs sauvages (Wild Pointers) :

Les pointeurs qui pointent vers des zones de mémoire libres ou désallouées peuvent conduire à des accès invalides à la mémoire. Cela peut se produire si le programme tente d'accéder à la mémoire via un pointeur qui a été désalloué ou qui pointe vers une adresse incorrecte.

Exemple

```
int *ptr = malloc(sizeof(int));
free(ptr);
*ptr = 10; // Accès invalide à la mémoire car ptr pointe maintenant vers
une zone mémoire libérée
```

Conséquences de l'Accès Invalide à la Mémoire

Les conséquences de l'accès invalide à la mémoire peuvent varier en fonction de divers facteurs tels que l'architecture du processeur, le système d'exploitation et le comportement spécifique du programme. Cependant, certaines conséquences courantes incluent :

- Plantage du programme (segmentation fault ou violation de protection).
- Comportement imprévisible du programme.
- Corruption de données.
- Vulnérabilités de sécurité (comme les débordements de tampon).

Prévention

Pour prévenir les accès invalides à la mémoire, il est essentiel de suivre de bonnes pratiques de programmation, telles que :

- Initialiser correctement les pointeurs avant de les utiliser.
- Éviter l'utilisation de pointeurs sauvages en désallouant les pointeurs lorsque leur utilisation n'est plus nécessaire.
- Utiliser des outils de débogage et d'analyse statique pour détecter les erreurs de mémoire pendant la phase de développement.
- Vérifier les limites des tableaux avant d'accéder à leurs éléments.