Notion reunis

Date de création	@10 décembre 2025 01:20
Matière	Structure des données / Langage C

Voici un récapitulatif des séances C#15 à C#20, couvrant des notions avancées en langage C, notamment l'allocation dynamique et les structures de données (listes, piles, files, arbres).

C#15 : Allocation Dynamique de Mémoire

Cette séance est consacrée à l'**allocation dynamique de mémoire**, permettant de réserver des zones en mémoire de manière flexible selon les besoins du programme, ce qui était impossible avec les tableaux statiques vus précédemment.

Concept et Utilité

L'allocation dynamique est essentielle lorsque l'on travaille avec des données dont la **taille n'est pas connue à l'avance**. Elle permet de créer, par exemple, un tableau dont la dimension dépend de la saisie de l'utilisateur.

Fonctions Clés

L'allocation dynamique repose sur l'inclusion du fichier d'en-tête `stdlib.h`. Les fonctions principales sont :

1. `malloc` **(Memory Allocation)** : Alloue un espace mémoire en octets,.
2. `size of` : Fonction utilisée conjointement avec `malloc` pour déterminer la taille exacte en octets du type de donnée à allouer,. On multiplie souvent ce résultat par le nombre d'éléments souhaités (ex: `nombre_de_joueurs * sizeof(int)`).
3. `free` : Fonction **obligatoire** qui libère l'espace mémoire alloué par `malloc` ou `calloc`. Si l'espace n'est pas libéré, cela entraîne des **fuites de mémoire**.
4. `calloc` **(Clear Allocation)** : Fonction similaire à `malloc`, mais qui a l'avantage d'**initialiser toutes les zones allouées à zéro** par défaut,.
5. `realloc` **(Reallocation)** : Permet de modifier la taille d'un bloc de mémoire déjà alloué.

Il est crucial de toujours vérifier si l'allocation a réussi (si le pointeur retourné n'est pas `NULL`) avant de continuer le programme,.

C#16 : Les Fichiers

C'est la dernière séance abordant les notions de base du langage C, se concentrant sur la gestion des fichiers (lecture et écriture) en **mode texte**,.

Ouverture et Fermeture

Pour manipuler un fichier, il faut déclarer un pointeur de type `FILE` (une structure interne au langage) :

- `F open` : Ouvre un fichier en spécifiant le nom et le **mode d'ouverture** (lecture seule `r`, écriture seule `w`, ajout en fin de fichier `a`, etc.),,.
- `F close` : **Ferme un fichier ouvert**, ce qui est une étape obligatoire pour éviter les fuites de mémoire, tout comme la libération de la mémoire.

Lecture des Données

Les fonctions de lecture incluent :

- `F get c` : Lit un seul **caractère** du fichier.
- `F get s` : Lit une **ligne complète** sous forme de chaîne de caractères. Cette fonction est plus sécurisée, car elle permet de spécifier la taille maximale de la chaîne à lire.
- `F scanf` : Lit du **texte formaté** (en utilisant `%s`, `%d`, etc.), ce qui est utile pour stocker des données (nombres, chaînes) dans différentes variables,.

Écriture des Données

Les fonctions d'écriture suivent une logique similaire :

- `F put c` : Écrit un **caractère** dans le fichier.
- `F put s` : Écrit une **ligne de texte** (une chaîne de caractères).
- `F printf` : Écrit du **texte formaté** dans le fichier (similaire à `printf`, mais avec le fichier spécifié en premier paramètre),.

Fonctions Diverses

- `F seek` et `F tel` : Permettent de manipuler la **position du curseur** dans le fichier.

- `rename` et `remove` : Permettent de renommer ou de supprimer un fichier,.

C#17 : Les Piles (Stacks)

C'est la première séance sur les **structures de données avancées**.

Règle de Structure (LIFO)

La pile (Stack) suit la règle **LIFO** (*Last In, First Out*) : le dernier élément inséré est le premier à être retiré. Cela est illustré par l'analogie d'une pile d'assiettes.

Fonctions Clés

L'implémentation d'une pile implique des fonctions spécifiques :

- `push stack` : Fonction pour **ajouter un élément** (empiler) au sommet de la pile,.
- `pop stack` : Fonction pour **retirer le dernier élément** (dépiler), c'est-à-dire celui au sommet de la pile.
- `top stack` : Renvoie la valeur de l'élément au **sommet** de la pile,.
- `stack len` : Calcule la **longueur** (ou hauteur) de la pile,.
- `clear stack` : Fonction de nettoyage qui utilise souvent la **récursivité** ou un bouclage de `pop stack` pour libérer la mémoire de tous les éléments,.

L'implémentation des piles utilise intensivement les **pointeurs** et l'**allocation dynamique**.

C#18 : Les Files (Queues)

Cette structure de données est opposée à la pile.

Règle de Structure (FIFO)

La File (Queue) suit la règle **FIFO** (*First In, First Out*) : le premier élément inséré est le premier à être retiré. Cela est illustré par une file d'attente.

Implémentation

L'implémentation des files est faite différemment que pour les piles dans cette séance :

- Elle utilise des **variables statiques globales** (`first` , `last` , `nb_element`) pour accéder rapidement à la tête et à la queue de la file,.

- L'utilisation de variables statiques permet de ne pas avoir à passer la file en paramètre de chaque fonction.

Fonctions Clés

- `push queue` : Fonction pour **ajouter un élément** à la **queue** de la file,,.
- `pop queue` : Fonction pour **retirer le premier élément** (la **tête**) de la file,,.
- `queue first` / `queue last` : Renvoient la valeur du premier ou du dernier élément.
- `clear queue` : Nettoie la file en utilisant une boucle de `pop queue`.

C#19 : Les Listes Simples

Les listes chaînées sont la structure la plus flexible.

Règle de Structure

Contrairement aux piles et aux files, la liste simple **ne respecte pas d'ordre particulier** pour l'insertion ou la suppression,. Les données sont chaînées par des pointeurs `next`.

Fonctions d'Insertion

- `push front list` : Ajoute un élément au **début** de la liste.
- `push back list` : Ajoute un élément à la **fin** de la liste. L'implémentation est plus complexe pour cette fonction car elle nécessite un **parcours** de toute la liste pour se positionner à la fin,.

Fonctions de Suppression

- `pop front list` : Retire le premier élément,.
- `pop back list` : Retire le dernier élément. Cette fonction est plus complexe car elle nécessite l'utilisation d'un pointeur de "sauvegarde" pour ne pas perdre le maillon précédent avant de rompre le chaînage et libérer la mémoire,.

C#20 : Les Listes Doublement Chaînées

Cette structure améliore la liste simple en introduisant un deuxième pointeur.

Structure

Chaque maillon (`node`) de la liste doublement chaînée possède deux pointeurs :

1. `next` : Pointe vers l'élément **suivant**,.

2. `back` (ou `prev`) : Pointe vers l'élément **précédent**,.

La liste utilise également une structure `D list` qui stocke la **longueur** (`len`) et les pointeurs vers le **début** (`begin`) et la **fin** (`end`) de la liste, accélérant ainsi les traitements,.

Avantages

L'avantage principal est de pouvoir **parcourir la liste dans les deux sens** (du début à la fin et inversement),. Cela rend les opérations d'insertion/suppression en tête ou en queue presque aussi rapides l'une que l'autre, contrairement à la liste simple où le retrait en queue exigeait un parcours complet.

Fonctions Clés

Les fonctions d'insertion (`push front`/`push back`) et de suppression (`pop front`/`pop back`) existent, mais l'implémentation est ajustée pour gérer et maintenir les deux chaînages (`next` et `back`),,. Le nettoyage (`clear list`) est simplifié grâce à la fonction `pop front` ou `pop back`,.