Notion 1

Date de création	@10 décembre 2025 01:05
Matière	Structure des données / Langage C

Les Fonctions en C

La séance #8 est spécifiquement dédiée à la notion des **fonctions** et est considérée comme l'un des concepts les plus fondamentaux et importants du langage C, au même titre que les variables. Les fonctions servent principalement à **organiser et à structurer le code**.

1. Objectif principal : la factorisation

Le rôle central d'une fonction est de **factoriser** (regrouper) un ensemble d'instructions qui pourraient se répéter dans le programme.

- Si une tâche spécifique (comme réinitialiser la position d'un objet dans un jeu) doit être effectuée plusieurs fois, il est préférable de créer une seule fonction pour cette tâche au lieu de répéter les mêmes lignes de code partout.
- Ceci simplifie la **maintenance** : si les valeurs ou les paramètres de la tâche changent (par exemple, la position de la balle passe de 0 à 20), la modification n'a lieu qu'à un seul endroit — dans le corps de la fonction — et se répercute sur tout le programme.
- Une règle de programmation stipule qu'une fonction doit idéalement se concentrer sur **une seule chose** à la fois.

2. Structure et Définition des Fonctions

Une fonction est identifiée par plusieurs éléments :

- **Nom :** Elle doit porter un nom qui respecte les mêmes règles de nommage que les variables.

- **Type de retour :** Elle doit spécifier le type de valeur qu'elle renvoie (un entier, un flottant, etc.). Si elle ne retourne rien, le type de retour est `void`.
- **Paramètres :** Les informations ou variables qu'elle reçoit en entrée sont listées entre parenthèses. Chaque paramètre doit être précédé de son type (Exemple : `int pose_x`). Si aucun paramètre n'est nécessaire, on utilise habituellement `void` dans les parenthèses.

Des exemples de fonctions standards que vous utilisez déjà sont `printf` (pour l'affichage) et `scanf` (pour la saisie).

3. La Fonction `main` et les Prototypes

- La fonction `main` est la **seule fonction obligatoire** dans un programme C, car elle marque le point de départ de l'exécution pour le compilateur.
- Si vous placez l'implémentation (le corps du code) d'une fonction **après** la fonction `main`, vous devez obligatoirement déclarer un **prototype** en début de fichier.
- Le prototype agit comme un indicateur pour le compilateur, lui fournissant la signature de la fonction (type de retour, nom, types de paramètres) pour qu'elle puisse être reconnue lorsqu'elle est appelée.

4. Le Mot-clé `return` et l'Affectation

- Le mot-clé `return` est utilisé pour qu'une fonction renvoie une valeur.
- Dès qu'un `return` est exécuté, la fonction est **immédiatement quittée**, et la valeur renvoyée est copiée vers la variable qui a effectué l'appel.

Exemple simple de fonction en C

Voici un exemple de fonction qui calcule la somme de deux entiers

```
#include <stdio.h>

// Prototype de la fonction
int addition(int a, int b);

int main() {
    int resultat = addition(5, 3);
    printf("La somme de 5 et 3 est : %d
```

```
", resultat);
    return 0;
}

// Définition de la fonction
int addition(int a, int b) {
    return a + b;
}
```

Dans cet exemple :

- La fonction `addition` prend deux paramètres `int a` et `int b`.
- Elle retourne la somme de ces deux entiers.
- Dans la fonction `main`, on appelle `addition(5, 3)` et on stocke le résultat dans la variable `resultat`.
- Le résultat est ensuite affiché avec `printf`.

5. Portée et Durée de Vie des Variables (Concept Crucial)

C'est un point essentiel pour comprendre comment les fonctions manipulent les données :

- **Variables locales :** Toute variable déclarée à l'intérieur d'une fonction est **locale** à cette fonction et n'existe que dans son contexte d'exécution.
- **Durée de vie :** Une fois que la fonction est terminée (par la parenthèse fermante ou par un `return`), **toutes les variables locales sont détruites** et l'espace mémoire qu'elles occupaient est libéré.
- Même si deux fonctions utilisent des variables portant le même nom (par exemple `balle_x`), il s'agit de **deux variables distinctes** stockées à des adresses mémoire différentes.
- Le `return` permet d'affecter la valeur calculée d'une fonction à une variable du contexte appelant, mais cette valeur n'est qu'une **copie** de la valeur locale qui va être détruite.

Il existe une exception à la destruction automatique des variables : l'utilisation du mot-clé `static` à la déclaration permet à une variable de conserver sa valeur lors des appels ultérieurs de la fonction.

La Programmation Modulaire

La modularité est une notion de programmation intermédiaire cruciale qui vise à **organiser et structurer le code** en le divisant en plusieurs fichiers distincts, souvent appelés "modules" ou "bibliothèques".

Cette approche permet la **factorisation** du code et l'intégration d'ensembles de fonctions prêtes à l'emploi, que ce soit des fonctions standard ou des fonctions développées par vous-même.

Structure des fichiers modulaires

Un module de programmation modulaire est généralement composé de deux types de fichiers associés :

1. **Le fichier d'en-tête (extension `.h`)**
    - Il contient les **prototypes** des fonctions, c'est-à-dire la signature complète de la fonction (son type de retour, son nom et le type de ses paramètres), suivie d'un point-virgule.
    - Il est crucial d'y inclure des **directives de préprocesseur** (comme `#ifndef`, `#define`, et `#endif`) pour assurer la sécurité et éviter les problèmes d'inclusion infinie (où deux fichiers essaient de s'inclure mutuellement).
    - Le fichier `.h` peut également contenir l'inclusion d'autres fichiers d'en-tête si nécessaire, bien que par convention on préfère limiter les inclusions à celles indispensables et les placer dans les fichiers source.
2. **Le fichier source (extension `.c`)**
    - Il contient l'**implémentation** des fonctions dont le prototype est défini dans le fichier `.h` associé (qui porte généralement le même nom, par exemple `player.c` et `player.h`).
    - Le fichier source doit **inclure son propre fichier d'en-tête** (ainsi que les bibliothèques standard nécessaires, comme `stdio.h`).

Utilisation et Compilation

Pour utiliser les fonctions d'un module (par exemple, un module `player`) dans votre programme principal (`main.c`) :

- Vous devez inclure le fichier d'en-tête du module dans votre fichier principal en utilisant des **doubles guillemets** (ex : `#include "player.h"`) pour

indiquer qu'il s'agit d'un fichier propre au projet et non d'une bibliothèque standard (qui utilise des chevrons `<>`).

- La **compilation** nécessite que tous les fichiers source (`.c`) soient compilés ensemble (par exemple, en utilisant la commande `gcc *.c -o prog` pour compiler tous les fichiers `.c` du répertoire).

La modularité est la base de l'utilisation de **bibliothèques externes** comme la SDL, que vous utiliserez plus tard pour la programmation 2D.

Portée des variables et mot-clé `static`

La modularité est étroitement liée à la gestion de la **portée des variables** :

- **Variables statiques locales :** Une variable déclarée avec le mot-clé `static` à l'intérieur d'une fonction n'est **pas détruite** à la fin de cette fonction, contrairement aux variables locales standard. Elle conserve sa valeur lors des appels suivants.
- **Variables globales :** Déclarées en dehors de toute fonction, elles sont théoriquement disponibles partout dans le programme. Cependant, les variables globales sont **fortement déconseillées** car elles peuvent être modifiées par n'importe quelle partie du programme, ce qui peut rendre le débogage difficile.
- **Fonctions statiques :** Le mot-clé `static` peut être placé devant une fonction pour la rendre **accessible uniquement** dans le fichier `.c` où elle a été implémentée.

La dixième séance du cours est entièrement dédiée au **préprocesseur** et à ses **directives**, bien que le sujet soit principalement théorique et serve de base essentielle pour aborder les notions futures, comme les tableaux et les structures de données.

Le rôle du préprocesseur

Le préprocesseur est un **sous-programme** qui s'exécute **avant tout** et de manière **prioritaire** à l'étape de la compilation.

Son rôle est d'opérer des changements, des inclusions ou des paramétrages sur le code source en suivant des instructions spécifiques que sont les directives de préprocesseur. Une fois que le préprocesseur a terminé son

travail (*pre-processing*), l'étape de la compilation peut commencer pour créer l'exécutable binaire.

Le préprocesseur ne participe pas à l'étape de compilation elle-même, mais prépare le code source pour celle-ci.

Les directives de préprocesseur

Les directives sont des lignes spéciales dans le code source qui commencent par le symbole **dièse (`#`)**. La séance en présente trois types principaux :

1. `#include`

C'est la directive que vous utilisez depuis le début de la formation, elle permet d'**inclure des fichiers**.

- **Fichiers natifs/standards :** L'inclusion de fichiers d'en-tête natifs (intégrés au langage), comme `stdio.h` (Standard Input Output), se fait en utilisant des **chevrons** (`<>`). L'inclusion de `stdio.h` est nécessaire, par exemple, pour pouvoir utiliser des fonctions standards comme `printf` et `scanf`. Le préprocesseur inclut ce code dans votre programme avant la compilation.
- **Fichiers personnels :** L'inclusion de fichiers d'en-tête que vous avez créés (comme ceux utilisés en programmation modulaire) se fait en utilisant des **doubles guillemets** (`""`).

2. `#define`

Cette directive est utilisée pour **définir quelque chose**, notamment une **constante**.

- **Définition de constantes :** Le `#define` permet d'associer un nom à une valeur, par exemple pour définir la TVA à 20 ou 25. L'avantage est que si cette valeur doit changer, vous n'avez qu'à la modifier à un seul endroit (la ligne du `#define`), et le changement sera répercuté dans tout le code source. Le préprocesseur effectue simplement un **remplacement textuel** de l'ancien contenu (le nom de la constante) par le nouveau contenu (sa valeur) partout dans le code avant la compilation.
- **Définition d'expressions ou de code :** Le `#define` est plus puissant que la simple définition de valeur. Il peut remplacer une partie de code par une autre, y compris des fonctions complètes (comme remplacer le mot `afficher` par `printf`) ou des expressions mathématiques complexes.

- **Conventions :** Il est habituel de nommer les constantes de préprocesseur en **majuscules**. Elles peuvent également commencer et se terminer par deux tirets du huit (`__`), bien que cela soit surtout une convention ou une habitude.
- **Utilité :** Cette directive est notamment utilisée pour définir la taille d'un tableau, ce qui permet de modifier la dimension à un seul endroit, y compris dans les boucles qui parcourent ce tableau.

3. Directives conditionnelles (`#ifndef` , `#define` , `#endif`)

Ces directives sont spécifiquement utilisées dans les fichiers d'en-tête (fichiers `.h`) pour assurer la **protection** de la bibliothèque et **éviter les inclusions infinies**.

- **Logique :** Elles permettent de vérifier si une constante spécifique a déjà été définie (`#ifndef` pour "if not defined").
  - Si la constante n'est **pas encore définie**, le préprocesseur la **définit** (`#define`).
  - Le contenu du fichier `.h` (les prototypes de fonctions) est ensuite inclus.
  - L'ensemble est fermé par un `#endif`.
- **Mécanisme :** Si le compilateur rencontre à nouveau une inclusion du même fichier `.h`, il voit que la constante est déjà définie. La condition `#ifndef` est donc fausse, et il n'inclut pas le contenu dupliqué, brisant ainsi le risque de boucle.

Constantes prédéfinies

Le langage C inclut également des constantes que vous pouvez afficher via le préprocesseur pour obtenir des informations sur l'exécution ou le fichier :

Constante	Signification
`__FILE__`	Le nom du fichier source.
`__LINE__`	Le numéro de ligne.
`__DATE__`	La date de compilation.
`__TIME__`	L'heure de compilation.

Le préprocesseur est donc l'étape qui assure la flexibilité et la modularité de votre code avant que le compilateur ne commence son travail de traduction en binaire.

Notions SI et SEULEMENT SI

Les directives conditionnelles du préprocesseur permettent d'inclure ou d'exclure du code en fonction de conditions préalables, ce qui est essentiel pour écrire du code portable et adaptatif.

- `#if` : Permet d'inclure du code si une condition est vraie.
- `#ifdef` : Inclut le code si une macro est définie.
- `#ifndef` : Inclut le code si une macro n'est pas définie.
- `#else` : Partie alternative si la condition précédente est fausse.
- `#elif` : "else if" pour chaîner plusieurs conditions.
- `#endif` : Termine la directive conditionnelle.

Ces directives sont souvent utilisées pour gérer différentes configurations, plateformes ou versions de bibliothèques, permettant ainsi d'activer ou de désactiver certaines parties du code selon les besoins.

Par exemple, on peut écrire du code qui ne sera compilé que sous Windows, ou uniquement si une certaine fonctionnalité est activée, en utilisant ces directives.

Cela permet d'améliorer la portabilité et la maintenance du code, en évitant les duplications et en centralisant les conditions d'inclusion.Le préprocesseur est donc l'étape qui assure la flexibilité et la modularité de votre code avant que le compilateur ne commence son travail de traduction en binaire.

Les pointeurs

Les pointeurs sont une notion fondamentale et puissante du langage C. Ils permettent de manipuler directement les adresses mémoire, ce qui ouvre la voie à des fonctionnalités avancées et une meilleure gestion des données.

1. Concept de base

Un pointeur est une variable qui contient l'adresse mémoire d'une autre variable. Contrairement à une variable classique qui stocke une valeur, un pointeur stocke l'emplacement où cette valeur est située en mémoire.

- **Variable simple** : stocke une valeur (ex: `int nombre = 100;`)
- **Pointeur** : stocke l'adresse d'une variable (ex: `int *ptr = &nombre;`)

2. Syntaxe et manipulation

Notation	Description	Exemple
Valeur de la variable	Accès à la valeur stockée	`nombre`
Adresse de la variable	Adresse mémoire de la variable	`&nombre`
Déclaration d'un pointeur	Variable qui contient une adresse	`int *ptr;`
Initialisation d'un pointeur	Affectation d'une adresse à un pointeur	`ptr = &nombre;`
Valeur pointée	Valeur de la variable pointée	`*ptr`
Adresse du pointeur	Adresse mémoire du pointeur lui-même	`&ptr`

3. Affichage des adresses

Pour afficher une adresse mémoire avec `printf`, il faut utiliser le spécificateur `%p` et non `%d`. Le format `%p` affiche l'adresse en hexadécimal, ce qui est la représentation standard des adresses mémoire.

Exemple sans cast `(void*)` (possible mais non recommandé) :

```
int nombre = 42;
int *ptr = &nombre;
printf("Adresse de nombre : %p
", &nombre);
printf("Adresse stockée dans ptr : %p
", ptr);
```

Exemple recommandé avec cast `(void*)` :

```
int nombre = 42;
int *ptr = &nombre;
printf("Adresse de nombre : %p
", (void*)&nombre);
```

```
printf("Adresse stockée dans ptr : %p
", (void*)ptr);
```

> Remarque : Le cast (void*) convertit l'adresse en pointeur générique. C'est la pratique recommandée avec %p pour garantir la portabilité et éviter les avertissements. Sans ce cast, l'affichage est possible mais moins sûr et peut générer des avertissements.

4. Utilité des pointeurs

- Permettent de modifier directement les variables passées en paramètre à une fonction (passage par adresse).
- Évitent la copie de grandes structures de données, améliorant ainsi la performance.
- Sont essentiels pour la gestion dynamique de la mémoire.

5. Bonnes pratiques

- Initialiser les pointeurs à `NULL` lorsqu'ils ne pointent vers rien.
- Toujours vérifier qu'un pointeur n'est pas `NULL` avant de l'utiliser.
- Manipuler les pointeurs avec précaution pour éviter les erreurs mémoire (segmentation fault, fuite mémoire, etc.).

Les Tableaux

La douzième séance de la formation en langage C est consacrée aux **tableaux**, une notion considérée comme la **suite logique** de la séance sur les pointeurs. La compréhension des tableaux est jugée essentielle, car ils seront utilisés **constamment** dans les vidéos qui suivent.

1. Définition et Structure en Mémoire

Un tableau est une structure qui permet de stocker un ensemble d'éléments.

- **Type de données unique :** Un tableau ne peut contenir qu'un seul type de donnée (par exemple, seulement des entiers ou seulement des flottants).

- **Espace Contigu :** Contrairement aux variables isolées, les éléments d'un tableau sont stockés de manière **contiguë** en mémoire. Cela signifie qu'ils se suivent immédiatement au niveau des adresses, et le système doit trouver un **bloc complet** de mémoire libre pour l'allouer.

2. Déclaration et Initialisation

Pour déclarer un tableau, on doit spécifier son type, son nom et le **nombre de cases** qu'il doit contenir entre crochets.

Type d'Initialisation	Description
Par défaut (Déclaration simple)	Le tableau est rempli de **valeurs aléatoires**.
Initialisation explicite	Les valeurs sont listées entre accolades (ex: `int tableau[] = {16, 4, -5, 22, 1888};`).
Initialisation à zéro	Pour garantir l'absence de valeurs aléatoires, il est courant d'initialiser toutes les cases à zéro en utilisant `{0}`.
Cas spécial	Si vous initialisez avec une seule valeur non nulle (ex: `{4}`), **seule la première case** prend cette valeur, le reste étant initialisé à zéro.

Il est recommandé d'utiliser une **directive de préprocesseur (`#define`)** pour définir la taille du tableau. Cela permet de modifier la dimension du tableau (y compris dans les boucles de parcours) à un seul endroit.

3. Indexation et Accès aux Éléments

- **Indice de départ :** La première case d'un tableau a toujours l'indice **0**.
- **Plage d'indices :** Pour un tableau de taille X, les indices vont de 0 à $X-1$. L'élément d'indice X est donc le $(X+1)^{ème}$ élément.
- **Accès :** L'accès ou la modification d'un élément se fait via la syntaxe `tableau[indice]`.

Pour afficher ou manipuler les éléments, on utilise généralement une **boucle `for`** qui est particulièrement adaptée pour le **parcours** de ces structures.

4. Le Lien Essentiel entre Tableaux et Pointeurs

Le nom d'un tableau est fondamentalement un pointeur.

- **Le nom est l'adresse :** Le nom du tableau sans crochets représente l'**adresse du tableau** (l'adresse de son premier élément).
- **Syntaxe simplifiée :** La notation d'accès standard `tableau[X]` est une version simplifiée et plus lisible de la notation par pointeur `(tableau + X)`. Cette notation simplifiée est privilégiée pour éviter les **erreurs de syntaxe** (oublis de parenthèses ou d'étoiles).
- **Passage aux fonctions :** Quand un tableau est passé à une fonction, le programme passe l'adresse (le pointeur) et **non une copie** des données. Par conséquent, toute modification effectuée dans la fonction est appliquée au **tableau original**.
    - Lors de l'appel de fonction, vous passez simplement le nom du tableau sans esperluette (`&`), car le nom est déjà considéré comme l'adresse (pointeur).
    - Pour qu'une fonction puisse travailler avec un tableau, elle doit recevoir obligatoirement **le pointeur** (le nom) et **la taille** du tableau.
- **Retourner un tableau :** Pour qu'une fonction puisse retourner un tableau, celui-ci doit être déclaré avec le mot-clé `static` à l'intérieur de la fonction. Sans cela, le tableau, étant une variable locale, serait **détruit** à la fin de la fonction, et l'adresse retournée pointerait vers une zone mémoire libérée.

5. Tableaux à Deux Dimensions (Bonus)

Il est possible de créer des tableaux à deux dimensions pour représenter des structures matricielles (comme un damier).

- **Déclaration :** Elle utilise un double crochet, le premier pour le nombre de lignes et le second pour le nombre de colonnes.

```
int tableau[3][4]; // Déclare un tableau de 3 lignes et 4 colonnes
```

- **Initialisation :**

```
int tableau[2][3] = {
    {1, 2, 3},
    {4, 5, 6}
};
```

- **Parcours :** L'affichage ou la manipulation nécessite l'utilisation de **deux boucles** `for` **imbriquées**.

```
for (int i = 0; i < 2; i++) {
    for (int j = 0; j < 3; j++) {
        printf("%d ", tableau[i][j]);
    }
    printf("
");
}
```

Ce code affiche tous les éléments du tableau en parcourant chaque ligne puis chaque colonne.

Les Chaînes de Caractères en C

La manipulation des chaînes de caractères est un aspect fondamental et incontournable de la programmation en langage C. Contrairement à d'autres langages, le C ne possède pas un type de données spécifique pour les chaînes, qui sont en réalité des tableaux de caractères terminés par un caractère nul (`'\0'`). Cette page présente les concepts clés, les méthodes de déclaration, les opérations d'entrée/sortie, ainsi que les fonctions principales de la bibliothèque standard `string.h` pour manipuler les chaînes.

1. Déclaration et Initialisation des Chaînes de Caractères

Nature des Chaînes en C

En C, une chaîne de caractères est stockée sous forme d'un tableau contigu de caractères. Le dernier caractère est toujours le caractère nul (`'\0'`), qui indique la fin de la chaîne.

Méthodes de Déclaration

- **Méthode explicite (caractère par caractère) :**

```
char chaine[] = {'H', 'e', 'l', 'l', 'o', '\0'};
```

- **Méthode simplifiée (recommandée) :**

  ```
  char chaine[] = "Hello";
  ```

  Le compilateur ajoute automatiquement le caractère nul.

- **Chaîne constante via pointeur :**

  ```
  const char *ptr = "Hello";
  ```

  Cette chaîne ne peut pas être modifiée.

Importance du Caractère Nul ( )

Le caractère nul est essentiel pour signaler la fin de la chaîne. Sans lui, les fonctions de manipulation risquent de lire au-delà de la mémoire allouée, causant des erreurs.

2. Affichage et Lecture des Chaînes

Affichage

- Avec `printf` :

  ```
  printf("%s", chaine);
  ```

- Avec `puts` :

  ```
  puts(chaine);
  ```

  `puts` ajoute automatiquement un retour à la ligne.

Lecture

- Avec `scanf` :

  ```
  scanf("%s", chaine);
  ```

  Limite : ne lit pas les espaces, s'arrête au premier espace.

- Avec `gets` (non sécurisée mais lit les espaces) :

```
gets(chaine);
```

3. Fonctions de Manipulation de Chaînes (`string.h`)

Pour utiliser ces fonctions, inclure la bibliothèque :

```
#include <string.h>
```

Fonction	Description	Détails importants
`strlen`	Retourne la longueur de la chaîne	Ne compte pas le caractère nul (`\0`)
`strcpy`	Copie une chaîne source vers une destination	Copie aussi le caractère nul
`strncpy`	Copie un nombre spécifié de caractères	
`strcat`	Concatène une chaîne à la fin d'une autre	
`strncat`	Concatène un nombre spécifié de caractères	
`strcmp`	Compare deux chaînes	Retourne 0 si identiques, sinon selon ordre ASCII
`strchr`	Recherche la première occurrence d'un caractère	Renvoie un pointeur ou NULL
`strrchr`	Recherche la dernière occurrence d'un caractère	
`strstr`	Recherche une sous-chaîne	
`strcspn`	Retourne la position du premier caractère dans la chaîne qui correspond à un caractère donné	Utile pour trouver la première occurrence d'un caractère parmi un ensemble
`sprintf`	Écrit du contenu formaté dans une chaîne	

Fonctions additionnelles (non standard mais courantes)

- `strrev` : inverse la chaîne
- `strlwr` : convertit en minuscules

- `strupr` : convertit en majuscules

---

4. Remarques sur la Lecture avec `scanf` et les Espaces

La fonction `scanf` avec `%s` s'arrête dès qu'elle rencontre un espace, ce qui empêche la lecture de chaînes contenant des espaces. Pour contourner cette limitation, on peut utiliser la syntaxe suivante :

```
scanf("%\n", chaine);
```

Cette expression lit toute la ligne jusqu'au retour à la ligne, incluant les espaces.

5. Note sur la Sécurité des Fonctions `gets` et `puts`

Les fonctions `gets` et `puts` sont considérées comme non sécurisées :

- `gets` ne contrôle pas la taille de la chaîne lue, ce qui peut provoquer des dépassements de tampon (buffer overflow).
- `puts` est généralement sûre pour afficher, mais `fputs` est une alternative plus flexible.

Alternatives sécurisées : `fgets` et `fputs`

- `fgets` permet de lire une chaîne en limitant le nombre de caractères lus, évitant ainsi les dépassements :

  ```
  fgets(chaine, taille_max, stdin);
  ```

- `fputs` écrit une chaîne sur une sortie donnée (par exemple, stdout) :

  ```
  fputs(chaine, stdout);
  ```

Déclaration et Initialisation pour `fgets` et `fputs`

Pour utiliser ces fonctions, il faut déclarer un tableau de caractères suffisamment grand :

```
char chaine[100]; // taille adaptée selon le besoin
```

Puis utiliser `fgets` pour la lecture et `fputs` pour l'écriture.

En résumé, la gestion des chaînes en C repose sur la compréhension des tableaux de caractères terminés par `'\0'` et l'utilisation des fonctions de la bibliothèque standard pour manipuler efficacement ces chaînes, tout en tenant compte des bonnes pratiques de sécurité pour la lecture et l'écriture.

Structure et Types

La séance aborde trois concepts principaux : les structures (`struct`), les énumérations (`enum`), et les unions (`union`), ainsi que le mot-clé `typedef`.

1. Les Structures (`struct`)

Une structure sert à **rassembler un ensemble de données** sous une seule entité, ce qui permet de manipuler ces données de manière plus pratique.

Définition de la structure

- **Mot-clé :** On utilise le mot-clé `struct` suivi d'un nom (par exemple, `player`).
- **Contenu :** À l'intérieur des accolades, on définit les **champs** de la structure, qui sont en fait des variables (comme un `char name[]` pour le nom, un `int` pour les points de vie `HP`, etc.).
- **Emplacement :** Les structures sont généralement définies **au-dessus de la fonction** `main` ou, dans le cadre de la programmation modulaire, dans les **fichiers d'en-tête** (fichiers `.h`).

Déclaration et Initialisation

- **Déclaration :** Pour créer une variable du type de la structure (un joueur P1), on utilise la syntaxe `struct Nom_Structure Nom_Variable` (Exemple : `struct Player P1`).
- **Initialisation :** On peut initialiser les champs dès la déclaration en utilisant des **accolades** `{}` pour lister les valeurs dans l'ordre, séparées par des virgules.

Accès et Modification des données

- **Accès :** Pour accéder à un champ, on utilise un **point** (Exemple : `P1.HP`).
- **Modification :** La modification se fait champ par champ. Pour les variables simples (`int`, `float`), on utilise l'opérateur d'affectation (`P1.HP = 50`). Pour les chaînes de caractères (qui sont des tableaux), il est obligatoire d'utiliser une **fonction de copie** (`strcpy`) et non l'opérateur `=`.

Structures et Pointeurs

Lorsque l'on travaille avec des **pointeurs vers des structures** (par exemple, en passant une structure à une fonction par adresse), la syntaxe d'accès aux champs change :

- **Syntaxe longue :** Il faut utiliser des parenthèses, l'étoile et le point (Exemple : `(*ptr_structure).champ`).
- **Syntaxe raccourcie (recommandée) :** Elle est remplacée par le symbole **flèche** (`>`) pour plus de lisibilité et éviter les erreurs de syntaxe (Exemple : `ptr_structure->champ`).

Tableaux de Structures

Il est tout à fait possible de créer des **tableaux** dont chaque case contient une structure (Exemple : un tableau de 5 joueurs `Player tableau_joueur`).

2. Le Mot-clé `typedef` (Définition de type)

Le mot-clé `typedef` permet de créer un **alias** ou un **nouveau type** à partir d'un type existant ou d'une structure, ce qui simplifie la syntaxe de déclaration.

- **Rôle :** Il permet de remplacer une écriture longue (comme `struct Player`) par un nom plus simple (comme `Player`).
- **Utilisation :** En ajoutant `typedef` devant la définition de la structure (ou de l'énumération), on peut ensuite utiliser le nouveau nom (`Player`) comme un type de variable de base (`int` ou `float`).

3. Les Énumérations (`enum`)

Une énumération est utilisée pour **énumérer une liste de valeurs possibles** pour un type de données, souvent pour faciliter la lecture du code.

- **Rôle :** Elle crée des **constantes** (souvent en majuscules) qui peuvent être utilisées à la place de chiffres.

- **Valeurs :** Par défaut, le premier champ de l'énumération vaut **0**, le suivant **1**, et ainsi de suite. On peut attribuer des valeurs explicites aux champs si nécessaire (Exemple : `Mazda = 40`).
- **Type sous-jacent :** Une énumération est stockée en mémoire comme un **entier** (`int`), car chaque champ correspond à une valeur numérique.

4. Les Unions (`union`)

Une union, comme une structure, permet de regrouper plusieurs types de données, mais avec une différence essentielle concernant la mémoire.

- **Rôle :** Tous les membres d'une union **partagent le même emplacement mémoire**. Seul l'un des membres peut être actif à la fois.
- **Utilité :** L'union va réserver en mémoire l'espace nécessaire pour stocker le **plus grand** des types de données déclarés en son sein. Cela permet d'économiser de la mémoire si l'on sait que seule l'une des variables sera utilisée à la fois (Exemple : stocker soit un `int`, soit un `float`, mais en utilisant le même emplacement).

5. Exemples combinés de structures, enum, typedef et union

Exemple 1 : Combinaison de `struct`, `enum` et `typedef`

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

// Définition d'une énumération pour les classes de joueur
typedef enum {
    GUERRIER,
    MAGE,
    VOLEUR
} Classe;

// Définition d'une structure Player avec typedef
typedef struct {
    char nom[50];
    int pointsDeVie;
```

```
    Classe classe;
} Player;

int main() {
    Player joueur1;

    // Initialisation
    strcpy(joueur1.nom, "Arthur");
    joueur1.pointsDeVie = 100;
    joueur1.classe = GUERRIER;

    // Affichage
    printf("Nom: %s
", joueur1.nom);
    printf("Points de vie: %d
", joueur1.pointsDeVie);
    printf("Classe: %d
", joueur1.classe); // Affiche 0 pour GUERRIER

    return 0;
}
```

Exemple 2 : Utilisation d'une union dans une structure

```
#include <stdio.h>

// Définition d'une union pour stocker différents types de données
union Valeur {
    int entier;
    float reel;
    char caractere;
};

// Structure contenant une union
struct Donnee {
    int type; // 0 = int, 1 = float, 2 = char
    union Valeur valeur;
};
```

```
int main() {
    struct Donnee d;

    d.type = 0; // On stocke un entier
    d.valeur.entier = 42;

    if (d.type == 0) {
        printf("Entier: %d
", d.valeur.entier);
    }

    d.type = 1; // On stocke un float
    d.valeur.reel = 3.14f;

    if (d.type == 1) {
        printf("Réel: %f
", d.valeur.reel);
    }

    return 0;
}
```

Exemple 3 : Tableau de structures avec enum et typedef

```
#include <stdio.h>

typedef enum {
    ROUGE,
    VERT,
    BLEU
} Couleur;

typedef struct {
    char nom[20];
    Couleur couleurPreferee;
} Personne;
```

```
int main() {
    Personne groupe[3] = {
        {"Alice", ROUGE},
        {"Bob", VERT},
        {"Charlie", BLEU}
    };

    for (int i = 0; i < 3; i++) {
        printf("Nom: %s, Couleur préférée: %d
", groupe[i].nom, groupe[i].couleurPreferee);
    }

    return 0;
}
```