

Documentation Mini Projet Langage C

Caculli Giorgio, Jędrzej Tyranowski
Haute École de Louvain en Hainaut (HELHa)

14 décembre 2020

Résumé

Documentation pour le projet de Langage procédural sur les listes chaînées. Projet basé sur le concept d'un centre de formations.

Mots-clés : liste, chaînée, c, noeud, centre, formation, tête

Table des matières

1	Introduction	3
1.1	Le langage C	3
1.2	Fonctions générales utilisées	3
1.2.1	Qu'est-ce l'allocation dynamique de mémoire?	6
1.2.2	Qu'est-ce <code>calloc()</code> ?	6
1.2.3	Qu'est-ce <code>malloc()</code> ?	6
1.2.4	Différences entre <code>calloc()</code> et <code>malloc()</code>	6
2	Listes chaînées	7
2.1	Qu'est-ce une liste chaînée?	7
2.2	Création d'un nouveau nœud	7
2.3	Insertion d'un nœud dans une liste chaînée	7
2.4	Suppression d'un nœud d'une liste chaînée	8
2.5	Affichage d'une liste chaînée	8
3	Énoncé	9
4	Programme	10
4.1	Mode d'emploi	10
5	Code	11
5.1	Structures	11
5.2	Fonctions	13
5.2.1	Fonctions qui créent des nœuds	13
5.2.2	Fonctions qui affichent des listes chaînées	13
5.2.3	Fonctions qui ajoutent un nœud à une liste chaînée	13
5.2.4	Fonctions qui suppriment un nœud d'une liste chaînée	14
5.2.5	Fonctions qui servent de <code>getter()</code>	15
5.2.6	Fonctions qui affiches les différentes parties du menu interactif	15

1 Introduction

1.1 Le langage C

La langage de programmation utilisé lors du développement et la mise en œuvre du programme est le ANSI-C. Les différentes versions du langage disponibles lors du développement de ce programme sont :

- **ANSI-C** : La première version standardisée par le **American National Standard Institute**, abrégé en **ANSI** dans ce document, du langage C publiée en 1990.
- **C-99** : Révision de la version ANSI pour permettre aux développeurs d'utiliser les commentaires `//`, les booléens grâce à la librairie `<stdbool.h>`, la déclaration des `int` directement dans la boucle `for`, et d'autres modernisations de la syntaxe.
- **C-11** : Mise à jour du langage C pour permettre le support des `thread` afin de pouvoir faire du multi-threading.
- **C-17** : Révision de la version **C-11** qui n'ajoute aucune nouvelle fonctionnalité, mais corrige beaucoup bugs présents dans la version 11.

Dans les différentes applications que l'on a fait dans le cours de Langage procédural, la plupart des interactions que l'on a eu avec le langage C, notamment le fait de devoir déclarer un `int` avant une boucle `for`, ressemblaient fortement à l'ANSI-C. C'est pourquoi nous avons choisi d'utiliser cette version là.

1.2 Fonctions générales utilisées

Différentes fonctions ont été utilisées lors du développement de ce programme, ainsi que des MACRO pour permettre au compilateur de reconnaître le système d'exploitation dans lequel le logiciel compilé tournera. Voici une liste des fonctions clés utilisées :

- `fopen()` : Fonction qui sert à ouvrir un flux, plus précisément, un fichier.

Exemple :

```
1 /* Admettons que le fichier donnees.dat existe */
2 #include <stdio.h>
3 int main() {
4     FILE *fichier_in = fopen( "donnees.dat", "r" );
5     /* On ouvre le fichier donnees.dat en lecture */
6     FILE *fichier_out = fopen( "resultats.txt", "w" );
7     /* Si le fichier resultats.txt n'existe pas on le cree, s'il existe toute information
8        presente est ecrasee, puis on y accede en ecrute */
9     return 0;
10 }
```

- `fclose()` : Fonction qui sert à fermer tout flux ouvert.

Exemple :

```
1 /* Admettons que le fichier donnees.dat existe */
2 #include <stdio.h>
3 int main() {
4     FILE *fichier_in = fopen( "donnees.dat", "r" );
5     /* On ouvre le fichier donnees.dat en lecture */
6     FILE *fichier_out = fopen( "resultats.txt", "w" );
7     /* Si le fichier resultats.txt n'existe pas on le cree, s'il existe toute information
8        presente est ecrasee, puis on y accede en ecrute */
9     fclose( fichier_out );
10    fclose( fichier_in );
11    /* On ferme les fichiers lorsqu'on doit plus travailler avec eux */
12    return 0;
13 }
```

- `printf()` : Fonction qui sert à afficher une chaîne de caractères dans la console.

Exemple :

```
1 #include <stdio.h>
2 int main() {
3     printf( "Hello World!\n" ); /* Affiche "Hello World!" dans la console */
4     return 0;
5 }
```

- **fprintf()** : Fonction qui sert à écrire une chaîne de caractères vers un flux spécifique.

Exemple :

```
1 #include <stdio.h>
2 int main() {
3     File *fichier_sortie = fopen( "fichier_sortie.txt", "w" );
4     /* On cree et on ouvre un fichier nomme fichier_sortie.txt en ecriture */
5     fprintf( fichier_sortie, "Hello World!\n" );
6     /* Ecrit "Hello World!" dans le fichier fichier_sortie.txt mais pas dans la console */
7     fprintf( stdout, "Hello World!\n" );
8     /* Affiche "Hello World!" dans la console mais pas dans le fichier de sortie */
9     return 0;
10 }
```

- **scanf()** : Fonction qui sert à extrapoler une entrée du clavier et stocker les informations extrapolées dans les paramètres déclarés dans la fonction.

Exemple :

```
1 #include <stdio.h>
2 int main() {
3     char prenom[50];
4     int age;
5     printf( "Comment t'appelles-tu ? " );
6     scanf( "%s", prenom ); /* Admettons que l'utilisateur insere "Jedrzej" */
7     printf( "Quel age as-tu %s ? ", prenom );
8     scanf( "%d", &age ); /* Admettons que l'utilisateur insere "21" */
9     printf( "Salut %s, je vois que tu as %d ans!\n", prenom, age );
10    /* Affiche "Salut Jedrzej, je vois que tu as 21 ans!" dans la console */
11    return 0;
12 }
```

- **fscanf()** : Fonction qui sert à extrapoler des entrées à partir d'un flux spécifique en respectant une structure précise, et stocker les informations extrapolées dans des paramètres déclarés dans la fonction.

Exemple :

```
1 /* Contenu dans fichier_entree.txt */
2 /* Jedrzej 21 */
3 #include <stdio.h>
4 int main() {
5     char prenom[50];
6     int age;
7     FILE *fichier_in = fopen( "fichier_entree.txt", "r" );
8     /* On ouvre un fichier nomme fichier_entree.txt en lecture */
9     fscanf( fichier_in, "%s %d\n", prenom, &age );
10    /* On lit le contenu de fichier_entree.txt */
11    printf( "Salut %s, je vois que tu as %d ans!\n", prenom, age );
12    /* Affiche "Salut Jedrzej, je vois que tu as 21 ans!" dans la console */
13    return 0;
14 }
```

- **fgets()** : Fonction qui a le même principe que fscanf, mais garde les espaces.

Exemple :

```
1 /* Contenu dans fichier_entree.txt */
2 /* Caculli Giorgio 23 */
3 #include <stdio.h>
4 int main() {
5     char nom_prenom[15];
6     int age;
7     FILE *fichier_in = fopen( "fichier_entree.txt", "r" );
8     fgets( nom_prenom, 16, fichier_in );
9     /* Lit les 15 caracteres depuis le fichier de donnees fichier_entree.txt */
10    fscanf( fichier_in, "%d", &age );
11    /* Lit l'age depuis le fichier de donnees fichier_entree.txt */
12    printf( "%s %d\n", nom_prenom, age );
13    /* Affiche "Caculli Giorgio 23" dans la console */
14    return 0;
15 }
```

- **strcpy()** : Fonction qui sert à copier une chaîne de caractères vers une autres chaîne de caractères.

Exemple :

```

1 #include <stdio.h>
2 #include <string.h>
3 int main() {
4     char prenom[10];
5     char source[10] = "Giorgio";
6     strcpy( prenom, source );
7     printf( "%s\n", prenom ); /* Affiche "Giorgio" dans la console */
8     return 0;
9 }

```

- **sizeof()** : Fonction qui renvoie la quantité de mémoire (en bytes) qu'une entité va occuper dans la Random Access Memory, ou mémoire vive en français, abrégé en RAM dans ce document.

Exemple :

```

1 #include <stdio.h>
2 int main() {
3     printf( "Taille de char: %lu\n", sizeof( char ) );
4     /* Affiche 1 dans la console */
5     printf( "Taille de int: %lu\n", sizeof( int ) );
6     /* Affiche 4 dans la console */
7     printf( "Taille de long: %lu\n", sizeof( long ) );
8     /* Affiche 8 dans la console */
9     printf( "Taille de int[4]: %lu\n", sizeof( int[4] ) );
10    /* Affiche 16 dans la console, soit 4*4=16 */
11    return 0;
12 }

```

- **memcpy()** : Fonction qui copie n octets depuis une zone mémoire vers une autre zone mémoire.

Exemple :

```

1 #include <stdio.h>
2 #include <string.h>
3 int main() {
4     int src[] = {1, 2, 3};
5     size_t n = sizeof(src) / sizeof(src[0])
6     /* En sachant que la taille d'un int = 4 bytes, et qu'il y a 3 int dans src, on
       obtient 4 * 3 = 12 bytes utilisés. On divise la taille totale du vecteur, soit 12,
       par la taille du premier element du vecteur, soit 4, donc 12 / 4 = 3 elements dans
       le vecteur */
7     int dest[n]; /* On initialise le vecteur dest avec la meme taille de src, ici 3 */
8     memcpy( dest, src, sizeof(dest) ); /* On copie les informations de src vers dest */
9     int i;
10    for( i = 0; i < n; i++ ) {
11        /* On parcourt tous les elements dans dest, du premier (indice 0) jusqu'au troisieme
           (indice 2) */
12        printf( "%d ", dest[i] ); /* Affiche 1 2 3 dans la console */
13    }
14    return 0;
15 }

```

- **getchar()** : Fonction qui sert à lire un caractère depuis un flux.

Exemple :

```

1 #include <stdio.h>
2 int main() {
3     char lettre;
4     printf( "Inserez une lettre: " ); /* Admettons que l'utilisateur insere "a" */
5     char c = getchar();
6     printf( "Lettre inseree : %c\n", c );
7     /* Affiche "a" dans la console */
8 }

```

- **system()** : Fonction qui permet de lancer l'exécution d'une commande sur le système d'exploitation hôte.

Exemple :

```

1 #include <stdlib.h>
2 int main() {
3     char *commande = "dir";
4     system( commande );
5     /* Execute la commande "dir", fonction qui sert a afficher ce qui est present dans le
       repertoire */
6     return 0;
7 }

```

— `calloc()` : Fonction qui permet de faire de l'allocation dynamique de mémoire.

Exemple :

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 int main() {
4     int *pointer;
5     int i, n;
6     printf( "Nombres d'elements a ajouter: " );
7     scanf( "%d", &n ); /* Admettons que l'utilisateur insere 3 */
8     pointer = ( int * ) calloc( n, sizeof( int ) );
9     for( i = 0; i < n; i++ )
10    {
11        printf( "Entrez le numero N.%d ", i + 1 );
12        scanf( "%d", &pointer[i] ); /* Admettons que l'utilisateur insere 1 2 et 3 */
13    }
14    for( i = 0; i < n; i++ )
15    {
16        printf( "%d ", pointer[i] );
17        /* Affiche 1 2 3 dans la console */
18    }
19    return 0;
20 }
```

1.2.1 Qu'est-ce l'allocation dynamique de mémoire ?

Une allocation dynamique de mémoire est le processus d'allouer de la mémoire lors de l'exécution d'un programme. Il existe quatre fonctions en C qui peuvent être utilisées pour allouer et libérer de la mémoire : `calloc()`, `free()`, `realloc()` et `malloc()`. Ces fonctions sont accessibles lors de l'introduction du header `<stdlib.h>` dans le code.

1.2.2 Qu'est-ce `calloc()` ?

`calloc()` est une fonction qui renvoie un pointeur vers un espace mémoire suffisamment libre pour stocker un tableau au nombre d'objets indéterminé et à la taille spécifiée. Si c'est pas le cas, la fonction renverra `NULL`. Le stockage est initialisé à zéro.

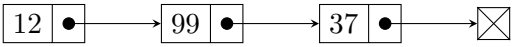
1.2.3 Qu'est-ce `malloc()` ?

`malloc()` est une fonction qui renvoie un pointeur vers un espace mémoire non initialisé. Si l'allocation n'est pas possible, la fonction renverra `NULL`. Si l'espace affecté par l'allocation est saturé, les résultats ne seront pas définis.

1.2.4 Différences entre `calloc()` et `malloc()`

<code>malloc()</code>	<code>calloc()</code>
<code>void * malloc(size_t n);</code>	<code>void * calloc(size_t n, size_t size);</code>
<code>malloc()</code> prend un argument (le nombre d'octets)	<code>calloc()</code> prend deux arguments (le nombre de blocs et la taille de chaque bloc)
<code>malloc()</code> est plus rapide que <code>calloc()</code>	<code>calloc()</code> prends plus de temps que <code>malloc()</code> car la mémoire doit être initialisée à zéro

2 Listes chaînées

Voici une représentation d'une liste chaînée : 

2.1 Qu'est-ce une liste chaînée ?

Une liste chaînée est une séquence de structures de données, qui sont liées par des nœuds. Chaque nœud contient une connexion vers un autre nœud.

Voici à quoi ressemble une structure pour une liste chaînée linéaire en C :

```
1 typedef struct noeud
2 {
3     int donnee;           /* L'information que l'on souhaite stocker dans le noeud */
4     struct noeud *suivant; /* Le lien vers le prochain noeud */
5 } noeud;
6 noeud *tete = NULL;      /* Le point de démarrage d'une liste chaînée */
```

Contrairement à un vecteur, qui lui ressemblerait à ça :

```
1 int donnee[10];
```

Les deux vont très bien stocker des `int` que l'on a nommé `donnee`, cependant, un vecteur possède une taille fixe, il ne saura stocker que 10 éléments maximum dans notre cas. Tandis que dans le cas d'une liste chaînée, tant qu'il y a de l'espace en mémoire, il saura stocker un nombre indéfini de nœuds, et par conséquent, un nombre indéfini de `int donnee`.

2.2 Création d'un nouveau nœud

Afin de pouvoir créer un nœud que l'on stockera dans notre liste chaînée, il faudra d'abord lui allouer une taille dans la mémoire. Toujours en se basant par le bout de code vu précédemment, et le bout de code vu dans le chapitre précédent sur l'allocation de mémoire, on obtiendrait le code suivant :

```
1 noeud *creer_noeud( int data )
2 {
3     noeud *temporaire = ( noeud * ) calloc( 1, sizeof( noeud ) );
4     /* On crée un noeud temporaire ou l'on stockera les informations que l'on souhaite */
5     temporaire->data = data; /* La donnée que l'on souhaite stocker */
6     temporaire->suivant = NULL; /* Lien vers le prochain noeud s'il existe */
7     return tmp; /* On retourne le noeud qui sera stocké dans la liste chaînée */
8 }
9 noeud *treinte_sept = creer_noeud( 37 );
10 free( treinte_sept );
```

Le langage C ne fait pas de **Garbage Collection**, ce qui revient à dire que toute manipulation de mémoire doit être manipulée manuellement. Donc, lorsqu'on a besoin de libérer de la mémoire de données que l'on peut écraser, on utilise la fonction `free()`.

Dans ce cas-ci, le nœud est créé mais il n'est pas encore stocké dans la liste chaînée. On verra dans la section suivante comment on peut stocker ce nouveau nœud dans la liste chaînée.

2.3 Insertion d'un nœud dans une liste chaînée

À l'état actuel de choses, on peut supposer que la liste chaînée soit vide : 

Une procédure d'ajout dans une liste chaînée pourrait être la suivante :

1. On initialise le nœud temporaire que l'on ajoutera à la base de données.
2. On associe `data` au pointeur `data` présent dans la structure `noeud`.
3. On initialise le prochain nœud de la liste `*suivant` à `NULL`.
4. Si la tête `*tete` de la liste est `NULL`, alors la tête devient le nouveau nœud. On arrête la fonction d'ajout là.
5. Sinon, on fait une copie de la tête dans le nœud `*suivant` que l'on avait initialisé à `NULL`.
6. On déclare la tête comme étant le nœud temporaire que l'on a initialisé.

Dans une fonction, cette procédure ressemblerait à ça :

```

1 int ajouter_noeud( noeud *tete, int data )
2 {
3     noeud *temporaire = creer_noeud( data );
4     /* On cree le noeud temporaire qui stocke la donnee */
5     if( tete == NULL )
6     {
7         tete = temporaire;
8         return 1;
9     }
10    temporaire->suitant = tete;
11    tete = temporaire;
12    return 1;
13 }
14 noeud *tete = NULL;
15 ajouter_noeud( tete, 37 );

```

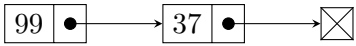
Si la fonction a eu du succès, la liste chaînée ressemblera à ça : 

Admettons que l'on souhaite ajouter une autre donnée à notre liste chaînée, on refait appel à la fonction :

```

16 ajouter_noeud( tete, 99 );

```

Si la fonction a eu du succès, la liste chaînée ressemblera à ça : 

2.4 Suppression d'un nœud d'une liste chaînée

2.5 Affichage d'une liste chaînée

3 Énoncé

4 Programme

4.1 Mode d'emploi

5 Code

5.1 Structures

Voici les différentes structures qui ont été utilisées dans la conception du programme :

```
30 typedef struct personne
31 {
32     int id;
33     char nom[25];
34     char prenom[25];
35     int formateur;
36     int nb_formationen;
37     int formations[30];
38     int nb_jours_indisponible;
39     int jours_indisponible[7];
40     int reduction;
41     int val_reduction;
42 } personne;
```

Cette structure sert à stocker les informations qui composent une personne quelconque : étudiant ou formateur.

Voici ce que représente chaque partie de la structure :

- `int id` : L'identifiant unique de la personne.
- `char nom[25]` : Le nom de la personne (25 caractères maximum).
- `char prenom[25]` : Le prénom de la personne (25 caractères maximum).
- `int formateur` : 1 si la personne est un formateur, 0 si la personne est un étudiant.
- `int nb_formationen` : Le nombre de formations auquel la personne participera.
- `int formations[30]` : Vecteur qui stockera les identifiants des différentes formations auquel la personne participera (On suppose dans une année, une personne ne peut participer qu'à 30 formations maximum).
- `int nb_jours_indisponible` : Si la personne est un formateur, il se peut qu'il/elle ait des jours d'indisponibilité, cette variable va stocker le nombre de jours où cette personne est indisponible (maximum 7).
- `int jours_indisponibles[7]` : Le vecteur qui stockera les jours auquel le formateur ne sera pas disponible (1 - lundi, 2 - mardi, etc...).
- `int reduction` : Si la personne est un étudiant, il se peut qu'il ait une réduction sur son minerval, 1 s'il a droit à une réduction, 0 si pas.
- `int val_reduction` : Le pourcentage de réduction auquel un étudiant a droit.

```
51 typedef struct noeud_db_personne
52 {
53     personne *p;
54     struct noeud_db_personne *next;
55 } noeud_db_personne;
```

Cette structure sert à devenir les différents nœuds qui seront stockés dans la base de donnée, soit la structure `db_personne`. Voici ce qui représente chaque partie de la structure :

- `personne *p` : Le pointeur de la personne qui sera stocké dans ce nœud lors de sa création.
- `struct noeud_db_personne *next` : qui contiendra le tête lors qu'on créera un nouveau nœud, sinon NULL.

```
63 typedef struct db_personne
64 {
65     noeud_db_personne *head;
66 } db_personne;
```

Cette structure sert à contenir tous les différents nœuds `noeud_db_personne`. C'est à partir de cette structure que l'on stockera les différentes nœuds qui eux-mêmes stockeront leurs personnes respectives.

- `noeud_db_personne *head` : La tête de la liste chaînée qui stockera toutes les personnes.

```
73 typedef struct noeud_formation
74 {
75     personne *p;
76     struct noeud_formation *next;
77 } noeud_formation;
```

Cette structure va stocker les différentes personnes qui participeront à une formation spécifique.

— `personne *p` : La personne qui participera à la formation.

— `struct noeud_formation *next` : Le nœud de pour la prochaine personne qui sera stockée.

```
94 typedef struct formation
95 {
96     int id;
97     char nom[40];
98     float prix;
99     int nb_jours;
100    int jours[7];
101    float heures[24];
102    float durees[10];
103    int nb_prerequis;
104    int prerequis[10];
105    noeud_formation *head;
106 } formation;
```

Cette structure sert à stocker toutes les informations qui composent une formations. Voici ce que chaque partie représente :

- `int id` : L'identifiant unique de la formation.
- `char nom[40]` : Le nom de la formation (40 caractères maximum).
- `float prix` : Le coût de la formation.
- `int nb_jours` : Le nombre de jours par semaine où cette formation à cours.
- `int jours[7]` : Vecteur contenant les jours où la formation a cours.
- `float heures[24]` : Le nombre d'heures du début de la formation.
- `float durees[10]` : Les différentes durées du cours lors de la semaine.
- `int nb_prerequis` : Le nombre de prérequis pour avoir accès à cette formation.
- `int prerequis[10]` : Vecteur contenant les identifiants des formations qui seraient des prérequis.
- `noeud_formation *head` : Étant donné qu'une formation stocke des personnes, elle-même est une liste chaînée qui stockera un nombre indéterminé de participants.

```
115 typedef struct noeud_db_formation
116 {
117     formation *f;
118     struct noeud_db_formation *next;
119 } noeud_db_formation;
```

Cette structure suit la même logique que la structure `noeud_db_personne`. Elle sert à stocker les différentes formations, qui eux-mêmes stockeront les personnes à leurs tour.

— `formation *f` : La formation qui sera stockée dans la base de données.

— `struct noeud_db_formation *next` : La prochaine formation qui sera stockée dans la base de données. NULL si pas de prochaine formation.

```
127 typedef struct db_formation
128 {
129     noeud_db_formation *head;
130 } db_formation;
```

Cette structure aussi suit la même logique que la structure `db_formation`. Elle sert de tête pour la la liste chaînée et c'est à partir de cette structure-ci que l'on démarrera les différentes interactions avec la base de données des formations.

— `noeud_db_formation *head` : La tête de la liste chaînée qui stockera les différentes formations.

Pourquoi avons-nous choisi cette approche? Principalement car on ne voulait pas stocker les différentes personnes et les différentes formations dans des vecteurs. On ne voulait pas qu'il y ait un nombre prédéfini de personnes et un nombre prédéfini de formations, on s'est donc fiés aux listes chaînées. Et c'est pourquoi maintenant il est possible de stocker autant de formations que la RAM nous permet ainsi que de stocker autant de formations que la RAM nous permet.

5.2 Fonctions

5.2.1 Fonctions qui créent des nœuds

Voici les différentes fonctions que l'on a du créer pour le bon fonctionnement du programme :

```
144 personne *creer_personne( char nom[], char prenom[], int formateur );
```

Cette fonction sert à créer un pointeur qui permettra d'initialiser les différentes informations présentes dans la structure **personne**. Lors de l'initialisation d'une personne, on n'aura besoin que du nom de famille de la personne, son prenom et s'il/elle est un formateur ou pas. Le reste des informations est manipulé par la suite lors des différentes interactions.

```
168 db_personne *creer_db_personne();
```

Cette fonction sert à initialiser le pointer **noeud_db_personne *head** dans la structure **db_personne** à NULL, afin que l'on puisse commencer à faire des manipulations avec cette structure.

```
303 formation *creer_formation( char nom[], float prix );
```

Cette fonction sert à créer un pointeur qui permettra d'initialiser les différentes informations présentes dans la structure **formation**. Lors de l'initialisation d'une formation, on n'aura besoin que du nom de la formation et de son prix. Le reste des informations est manipulé par la suite lors des différentes interactions.

```
442 db_formation *creer_db_formation();
```

Cette fonction sert à initialiser le pointer **noeud_db_formation *head** dans la structure **db_formation** à NULL, afin que l'on puisse commencer à faire des manipulations avec cette structure.

5.2.2 Fonctions qui affichent des listes chaînées

```
157 void afficher_personne( personne *p );
```

Cette fonction sert à afficher les informations de base qui caractérisent une **personne**. De manière générale, son identifiant, son nom de famille, son prénom et s'il est formateur ou étudiant.

```
257 void afficher_db_personne( db_personne *db );
```

Cette fonction parcourt l'entièreté de la base de données **db_personne *db**. Tant que la tête n'est pas NULL, on affichera les informations que l'on souhaite afficher de chaque personne présente dans la base de données.

```
408 void afficher_formation( formation *f );
```

Cette fonction sert à afficher les informations de base qui caractérisent une **formation**. De manière générale, son identifiant, son nom, son prix, ainsi que les personnes qui y participent.

```
461 void afficher_db_formation( db_formation *dbf );
```

Cette fonction parcourt l'entièreté de la base de données **db_formation *dbf**. Tant que la tête n'est pas NULL, on affichera les informations que l'on souhaite afficher de chaque formation présente dans la base de données.

5.2.3 Fonctions qui ajoutent un nœud à une liste chaînée

```
187 void ajouter_db_personne( db_personne *db, personne *p );
```

Cette fonction sert à initialiser un pointeur **noeud_db_personne *ndb** qui stockera **personne *p** dans la base de données **db_personne *db**. Ici, l'ajout dans la liste chaînée a lieu par le mécanisme suivant :

1. On initialise le noeud temporaire que l'on ajoutera à la base de données.
2. On associe **p** au pointeur **p** présent dans la structure **noeud_db_personne**.
3. On initialise le prochain nœud de la liste ***next** à NULL.
4. Si la tête ***head** de la base de donnée est NULL, alors la tête devient le nouveau nœud. On arrête la fonction d'ajout là.

5. Sinon, on fait une copie de la tête dans le nœud ***next** que l'on avait initialisé à NULL.
6. On déclare la tête comme étant le nœud temporaire que l'on a initialisé.

```
322 int ajouter_formation( formation *f, personne *p );
```

Cette fonction sert à initialiser un pointeur **noeud_formation *nf** qui stockera **personne *p** qui participera dans **formation *f**. Ici, l'ajout dans la liste chaînée à lieu par le mécanisme suivant :

1. On initialise le noeud temporaire que l'on ajoutera dans la formation.
2. On associe **p** au pointeur **p** présent dans la structure **noeud_formation**.
3. On initialise le prochain nœud de la liste ***next** à NULL.
4. Si la tête ***head** de la formation est NULL, alors la tête devient le nouveau nœud. On arrête la fonction d'ajout là.
5. Sinon, on fait une copie de la tête dans le nœud ***next** que l'on avait initialisé à NULL.
6. On déclare la tête comme étant le nœud temporaire que l'on a initialisé.

```
559 void ajouter_db_formation( db_formation *db, formation *f );
```

Cette fonction sert à initialiser un pointeur **noeud_db_formation *ndb** qui stockera **formation *f** dans la base de données **db_formation *db**. Ici, l'ajout dans la liste chaînée à lieu par le mécanisme suivant :

1. On initialise le noeud temporaire que l'on ajoutera à la base de données.
2. On associe **f** au pointeur **f** présent dans la structure **noeud_db_formation**.
3. On initialise le prochain nœud de la liste ***next** à NULL.
4. Si la tête ***head** de la base de donnée est NULL, alors la tête devient le nouveau nœud. On arrête la fonction d'ajout là.
5. Sinon, on fait une copie de la tête dans le nœud ***next** que l'on avait initialisé à NULL.
6. On déclare la tête comme étant le nœud temporaire que l'on a initialisé.

5.2.4 Fonctions qui suppriment un nœud d'une liste chaînée

```
216 int supprimer_db_personne( db_personne *dbp, int id );
```

Cette fonction sert à supprimer une personne de la base de données à partir de son identifiant. La démarche faite dans cette fonction est la suivant :

1. On vérifie que la tête de la base de données **dbp->head** ne soit pas NULL, si oui, on arrête la fonction.
2. On vérifie si le premier élément de la liste correspond à l'id de la personne que l'on souhaite supprimer.
3. Si oui :
 - (a) On crée un nœud temporaire qui stockera la personne suivante dans la liste.
 - (b) On libère l'espace mémoire occupé par head avec la fonction **free(dbp->head)**.
 - (c) On attribue à **dbp->head** le nœud temporaire que l'on avait créé.
 - (d) On arrête la fonction.
4. Sinon, on parcourt l'entièreté de la liste jusqu'au moment où l'on trouve la personne qui a le même id que l'id en paramètre.
5. Si on le trouve, on pivote l'élément qui suit vers l'élément que l'on vient de supprimer.
6. On arrête la fonction, si réussite, on obtient 1, si pas, on obtient 0.

```
361 int supprimer_personne_de_formation( formation *f, int id );
```

Cette fonction sert à supprimer une personne de la fonction à partir de son identifiant. La démarche faite dans cette fonction est la suivant :

1. On vérifie que la tête de la fonction `f->head` ne soit pas `NULL`, si oui, on arrête la fonction.
2. On vérifie si le premier élément de la liste correspond à l'id de la personne que l'on souhaite supprimer.
3. Si oui :
 - (a) On crée un nœud temporaire qui stockera la personne suivante dans la liste.
 - (b) On libère l'espace mémoire occupé par `head` avec la fonction `free(f->head)`.
 - (c) On attribue à `f->head` le nœud temporaire que l'on avait créé.
 - (d) On arrête la fonction.
4. Sinon, on parcourt l'entière de la liste jusqu'au moment où l'on trouve la personne qui a le même id que l'id en paramètre.
5. Si on le trouve, on pivote l'élément qui suit vers l'élément que l'on vient de supprimer.
6. On arrête la fonction, si réussite, on obtient 1, si pas, on obtient 0.

```
490 int supprimer_db_formation( db_formation *dbf, int id );
```

Cette fonction sert à supprimer une formation de la base de données à partir de son identifiant. La démarche faite dans cette fonction est la suivante :

1. On vérifie que la tête de la base de données `dbf->head` ne soit pas `NULL`, si oui, on arrête la fonction.
2. On vérifie si le premier élément de la liste correspond à l'id de la formation que l'on souhaite supprimer.
3. Si oui :
 - (a) On crée un nœud temporaire qui stockera la formation suivante dans la liste.
 - (b) On libère l'espace mémoire occupé par `head` avec la fonction `free(dbf->head)`.
 - (c) On attribue à `dbf->head` le nœud temporaire que l'on avait créé.
 - (d) On arrête la fonction.
4. Sinon, on parcourt l'entière de la liste jusqu'au moment où l'on trouve la formation qui a le même id que l'id en paramètre.
5. Si on le trouve, on pivote l'élément qui suit vers l'élément que l'on vient de supprimer.
6. On arrête la fonction, si réussite, on obtient 1, si pas, on obtient 0.

5.2.5 Fonctions qui servent de `getter()`

```
275 personne *get_personne( db_personne *db, char nom[], char prenom[], int formateur );
```

Cette fonction renvoie `NULL` si une personne d'un nom spécifique, d'un prénom spécifique, et s'il est formateur ou étudiant n'existe pas dans la base de données `db_personne *db`. Sinon, la fonction retourne la personne trouvée.

```
532 formation *get_formation( db_formation *dbf, char nom_formation[] );
```

Cette fonction renvoie `NULL` si une formation avec un nom spécifique n'existe pas dans la base de données `db_formation *dbf`. Sinon, la fonction retourne la formation trouvée.

5.2.6 Fonctions qui affiches les différentes parties du menu interactif

les fonctions ci-dessous ne servent qu'à afficher les différentes parties du menu interactif. Elle ne font aucune manipulation particulière autre qu'afficher du texte.

```
1 void menu_creer_formation( db_formation *f );
```

```
1 void menu_creer_personne( db_personne *p );
```

```
1 int menu_creer( db_formation *f, db_personne *p );
```

```
1 void menu_ajouter_formation( db_formation *f, db_personne *p );  
  
1 void menu_supprimer_personne( db_formation *dbf, db_personne *dbp );  
  
1 void menu_supprimer_formation( db_formation *dbf, db_personne *dbp );  
  
1 int menu_supprimer_personne_de_formation( db_formation *dbf );  
  
1 int menu_supprimer( db_formation *dbf, db_personne *dbp );  
  
1 int menu_affichage( db_formation *f, db_personne *p );  
  
1 int menu( db_formation *f, db_personne *p );  
  
1 int main( void );
```


Glossaire

ANSI American National Standard Institute. 3

RAM Random Access Memory. 4, 11