Recueil d'exercices pour le cours de programmation en langage C en année 2 de la spécialité E2I

Dimitri Galayko

1 Rappel des outils élémentaires du langage C

Exo 1.1. Algorithmes et boucles

Écrire une fonction permettant de calculer la factorielle d'un nombre. Rappel : $n! = n \cdot (n-1) \cdot (n-2) \cdot ... \cdot 3 \cdot 2 \cdot 1$ et $0! = 1$ Calculer ensuite : 10! et 20!. Réflechissez aux choix des types de variables (long int ? long long ? double ? long double ?). A partir de quel valeur de n il on ne peut plus utiliser un type entier pour stoquer le résultat ?

Faites deux version de la fonction: celle utilisant la boucle *for* et celle utilisant une récursion.

Exo 1.2. Tableaux

Avant d'écrire les fonctions suivantes, vous déclarerez un tableau global d'entiers de taille N, et vous utiliserez ce tableau dans toutes les fonctions.

1) La fonction *int saisie (int n)* lit les valeurs saisies au clavier et les enregistre dans le tableau le nombre d'éléments et remplissant le tableau avec des valeurs entières entrées par l'utilisateur. n est le nombre maximum d'éléments que l'utilisateur peut saisir.

2) La fonction *int affiche (int n)* affiche m premières valeurs du tableau à l'écran.

3) La fonction *int aleatoire (int n)* remplissant un tableau d'entiers avec n valeurs pseudo-aléatoires obtenues avec la fonction rand().

4) Les fonctions *int indice_max(int n)* et *int indice_min(int n)* renvoyant respectivement l'indice de l'élément le plus grand et le plus petit d'un tableau d'entiers.

Exo 1.3. Calcul d'intégrale

Écrivez une fonction qui calcule l'intégrale de la fonction

```
double function(double x)
```

(que vous définirez à part), entre les limites (a, b) et avec un pas de dx :

```
double integrale(double a, double b, double x).
```

Pour calculer l'intégrale, vous utiliserez la méthode de trapèze (cf. fig. 1).

L'opération d'intégration correspond au calcul de la surface enfermée par le graphe de la fonction, l'axe des abscisses et les lignes $x = a$, $x = b$. La surface est positive si le graphe se trouve au-dessus de l'axe Ox, négative sinon.

Un des moyens de calculer la valeur de l'intégrale est de décomposer l'intervalle (a, b) en petites intervalles dx, et faire une hypothèse que le graphe est une droite sur cette intervalle: la surface enfermée est alors celle d'une trapèze de hauteurs dx et de longueurs des bases de $f(x)$ et $f(x + dx)$, qui vaut : $S(dx) = 0.5(f(x) + f(x + dx))dx$.

En additionnant les surfaces de ces trapèzes élémentaires, on obtient la valeur de l'intégrale.

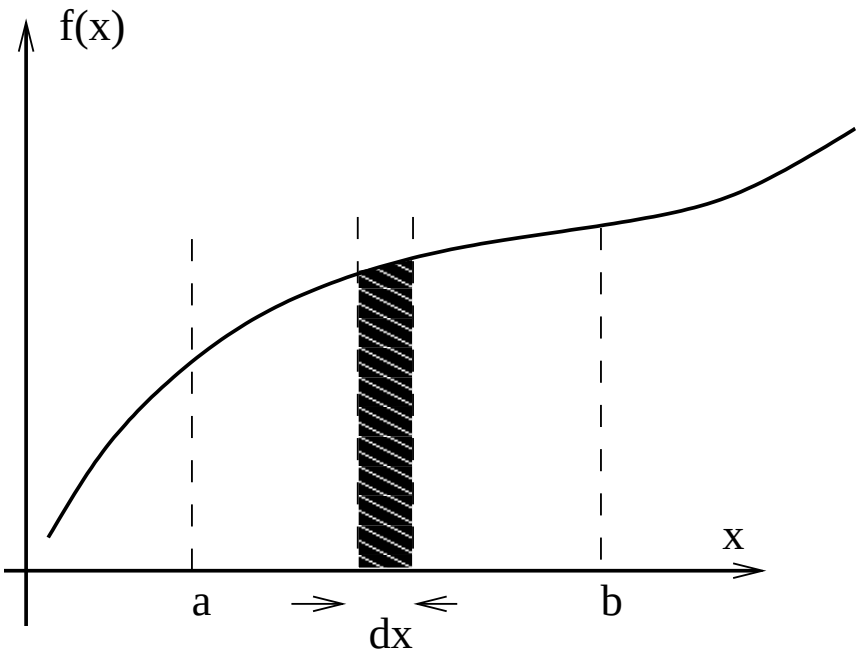

Figure 1:

Exo 1.4. Dichotomie.

Écrivez une fonction qui résout une équation algébrique

$$cos(x) - 0.3 = 0. \tag{1}$$

par dichotomie sur l'intervalle (0,3.14).

Vous utiliserez une fonction C pour calculer $f(x) = cos(x) - 0.3$:

```
double f(double x)
{
......
}
```

On s'arrêtera lorsque l'intervalle de dichotomie sera égal au paramètre $err = 10^{-10}$ qui sera défini au début du programme.

Vous utiliserez le boucle *while*. Vous ferez afficher le résultat.

Exo 1.5. Dichotomie : application au calcul du nombre π.

Vous utiliserez l'algorithme réalisé pour calculer la valeur de π avec une précision de 10 chiffres après la virgule. Pour cela on cherchera la solution de l'équation $sin(x) = 0$ en spécifiant l'intervalle initial comme (1.5, 4.5) (dans cet intervalle l'équation est sensée avoir une seule racine égale à π).

Pour afficher la valeur obtenue avec 10 chiffres après la virgule, vous utiliserez la fonction printf.

2 Syntaxe du langage C: pointeurs, tableaux, chaînes de caractères

Exo 2.1. Types élémentaires du langage C

1) Écrivez un programme qui affiche sur l'écran la taille en octets de types short, int, long, long long, float, double et long double, pointeur vers entier.

2) Écrivez un programme qui affiche les valeurs maximales des types non-signés char, short, long, long long. Modifiez le programme pour que soient affichées les limites supérieures des types signés.

Faites attention au format des nombres en appelant la fonction printf.

Exo 2.2. Objets du langage C

Déclarez les objets suivants :
– Tableau de 3 éléments de pointeurs vers des tableau de 6 éléments de type int;
– pointeur vers un tableau de 5 éléments flottants;
– tableau de 5 pointeurs vers les éléments de type char;
Quel objets sont déclarés :
int (*a[5])[9];
char *a[5][9];
Pourquoi la déclaration suivante est incorrecte :
char a[][];
Laquelle des deux déclarations est correcte ?
char a[4][];
char a[][4];

Exo 2.3. Déclaration et utilisation des tableaux, pointeurs et indirections

1) Considérez les déclarations suivantes et expliquez quel est l'objet défini par chaque ligne.

```
int *pi1;

double a;

double b;

double *pd1=&b;

int *ac={1, 2, 3};

int bc[]={1, 4, 10, 11};

int *pi2=ac+2;

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

int  *pi3=ad[1];
```

2) On continue le programme commencé dans la question précédente.

Analyser la suite du code et commentez. Détectez les cas où une erreur ou un comportement incorrect se produit.

```
*pi1=1;

*pi1=&a;

pi1=&a;

pi1[0]=2;

printf("%d\n", a);

pi1[1]=3;

bc[3]=-25;

printf("%d\n", pi2[-1]);

printf("%d\n", pi2[3]);

printf("%d\n", pi3[1]);

printf("%d\n", *(pi3+2));
```

Exo 2.4. Les expressions du langage C

Donnez le résultat des opérations, en considérant la priorité des opérateurs :

```
int i=0;
int tab={0,1,2,3}
i<25\&&i!=0+15|3*-tab[i|2]

int i=0;
int tab={0,1,2,3}
*&tab[2]<<1**tab;
```

Exo 2.5. Opérateurs "bit-à-bit" (1)

Écrivez la fonction qui retourne la valeur du bit numéro n du mot d'entrée x:

```
int valeur_bit(int x, int n){
...
}
```

Les bits sont numérotés à partir de 0, en commençant par le bit du poids le plus faible.

Exo 2.6. Opérateurs "bit-à-bit" (2)

Soit un système dans lequel la transmission de données se fait selon une interface série. La taille d'un mot est 2 octets (16 bits), sur lesquels le premier et le dernier bit sont toujours nuls (bits "stop" et "start"), l'avant-dernier bit est le bit de parité ("parité"). Ainsi, la donnée est codée sur 13 bits. Le bit de parité est défini de la manière suivante : il vaut 1 si dans le mot de la donnée le nombre de bits égaux à 1 est paire ou nul, 0 sinon.

Écrivez une fonction qui prend en argument la donnée sur 13 bits à transmettre, en format *unsigned short*, et qui retourne une valeur de type *short* (2 octets) prête pour l'envoi, i.e., avec les bits "stop", "start" et "parité". Pour écrire ce programme, vous utiliserez la fonction valeur_bit écrite précédemment.

3 Utilisation de base des pointeurs

Exo 3.1. Fonction *swap* pour le type *int*

Écrivez une fonction qui échange les valeurs des 2 variables de type int. Expliquez pourquoi il n'est pas possible de réaliser cette fonction en langage C sans employer le mécanisme de pointeurs.

```
void swap(int *a, int *b)
```

Exo 3.2. Fonction calculant la moyenne d'une suite de nombres entiers

Écrivez une fonction qui calcule la moyenne des valeurs entières contenues dans un tableau passé en argument. Pourquoi cette fonction aura besoin de deux arguments ? Si le premier argument est le pointeur vers le tableau contenant la suite des valeurs, quel est le second ?

Exo 3.3. Traitement élémentaire d'une chaîne de caractères

Une chaîne de caractère est définie de la façon suivante :

```
char str[100];
printf("Entrez une phrase pas très longue : ");
scanf("%s", str);
```

On souhaite analyser la chaine caractère par caractère. Par exemple, on souhaite connaître le nombre d'occurrences de la lettre 'o'.

On peut utiliser le parcours du tableau d'une manière élégante :

```
i=0;
char c; // variable intermédiaire
int counter=0; // compteur d'occurrences

while(c=str[i++])
if (c=='o') counter++;
printf("Le nombre d'occurrences
 de 'o' est %d", counter);
```

Analysez et commentez le code. Expliquez le fonctionnement de la boucle *while*, notamment celui de la condition de poursuite du cycle.

Exo 3.4.

Testez le programme suivant et en déduisez l'intérêt de la fonction "fonction_inconnue":

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

void fonction_inconnue(char * str1, char * str2){
  int i=0;
  int j=0;
  while(str2[i]!='\0'){
     if (str2[i]!=' ')
          str1[j++]=str2[i];
      i++;
  }
  str1[j]='\0';
```

```
  return;
}

main(){

  // le test de la fonction :

   char *s2="que fait cette fonction ?";
   char  s1[100];

   // on appelle la fonction
   fonction_inconnue(s1, s2);
   // on affiche le contenu de s1
   printf("%s\n", s1);
return;
}
```

Modifiez le programme pour que la fonction inconnue (dont le rôle vous connaissez maintenant) insère deux espaces à la place de chacun des espaces dans la nouvelle chaîne.

Exo 3.5. Trouvez l'erreur

Soit *tab* est un tableau de 10 entiers. Expliquez pourquoi le programme suivant génère une erreur :

```
int i;
for (i=0; tab[i]!=234 && i<=9 ; i++)
{ /* corps de boucle*/}
```

4 Allocation dynamique de mémoire

Ces exercices ont pour but d'introduire la fonction *malloc* qui permet une allocation dynamique de mémoire et indirectement, permet de créer des tableaux à taille définie lors de l'exécution (et non pas de compilation).

*void * malloc(unsigned int N)*

Cette fonction réserve un bloc mémoire de N octets, et fournit l'adresse non-typée vers le début du bloc. S'il n'y a pas de mémoire libre, il retourne le pointeur $NULL$ (zéro).

Pour libérer le bloc de mémoire réservée par malloc:

*free(void *pt).*

Ici *pt* est le pointeur vers le bloc qui avait été réservé par *malloc*.

Pour connaître la taille d'un type de donnée: l'opérateur C *sizeof* qui retourne le nombre d'octets occupés par une variable ou un type:

```
sizeof(int); // retourne la taille du type int

sizeof(int *); // retourne la taille du type pointeur vers entier

char tab[100];

sizeof(tab); //  retourne 100 (la taille du tableau en octet
```

Voici l'exemple typique d'utilisation de la fonction *malloc*:

```
int *a; // créer un pointeur de type int

a=(int*) malloc(sizeof(int)*100); // réserver une espace pour 100 entiers
```

```
// maintenant on peut traiter a comme si c'était un tableau de
//100 éléments entiers

.....// les commandes manipulant le tableau a

free(a); // libération de l'espace réservé
         // pour a après l'utilisation
```

Exo 4.1. Utilisation de la fonction malloc()

Soit un tableau contenant des informations sur les élèves d'une école (dans cet exemple, le nom, le prénom et le code postal du domicile). Écrire un programme qui gère la saisie d'un tel tableau, c'est-à-dire, qui demande à l'utilisateur de saisir des données des élèves au clavier, et qui enregistre l'information saisie dans le tableau.

Indications :

1) On définira une structure

```
struct eleve {
     char * nom;
     char * prenom;
     int zip;
};
```

Notez que contrairement à la stratégie que nous avons utilisée l'année passée, on ne définit pas de tableau "char nom[100]", où 100 est la taille maximale du nom. On définit un pointeur vers une chaîne ; l'espace pour la chaîne devra être réservé à l'aide de la fonction *malloc*. (Quel est l'avantage de cette manière de procéder ?)

2) On ne définit pas une taille maximale du tableau, contrairement à ce qu'il a été fait l'année dernière pour un exercice similaire. On utilisera l'allocation dynamique de mémoire pour étendre le tableau.

Exo 4.2. Équation quadratique

1) Écrivez une fonction qui prend en argument les trois coefficients d'une équation quadratique et qui retourne ses racines réelles.

Le problème : on connaît pas d'avance le nombre des racines – il dépend du signe du déterminant. Ainsi, la fonction doit retourner un nombre d'arguments variable. Pour cela, vous écrirez une fonction qui retournera un tableau des doubles, dont la premier élément donnera le nombre des racines, et les autres éléments, éventuellement, les valeurs des racines. Ainsi, selon le cas, vous aurez un tableau d'un, deux ou trois éléments.

Le prototype de la fonction s'écrit :

```
double * racines(double a, double b, double c)
```

2) Le défaut de cette fonction est que le nombre d'argument, qui est entier par nature, est stocké dans une case de tableau de type *double*. Une version plus sophistiquée du programme pourrait retourner le pointeur vers un tableau de 2 pointeurs, dont le premier pointerait vers une case de type *int*, le seconde vers une case de type "tableau de *double*" contenant la ou les racines. Écrivez cette version de la fonction.

3) Proposer une modification de la fonction de la recherche des racines d'équation quadratique de sorte à pouvoir retourner les racines complexes dans le cas où le déterminant est négatif.

5 Chaînes de caractères

Exo 5.1. Lecture d'une chaîne de caractères du clavier

Pour faire ces exercices, il faudra utiliser la technique d'allocation dynamique de mémoire (les fonctions *malloc* et *free*).

1) Écrivez la fonction qui permet de saisir une chaîne de caractère au clavier. Vous utiliserez la fonction *scanf*. Voici la signature de la fonction:

```
char * input_str(int N);
```

ici N est le nombre maximal de caractère de la chaîne que l'on souhaite recevoir.
Cette fonction :

- réservera un espace de mémoire nécessaire pour stocker une chaîne de N caractères ;
- À l'aide de la fonction *scanf* elle lira la chaîne de caractère de l'utilisateur, et l'enregistrera dans cet espace de mémoire ;
- Elle déterminera la taille occupée par la chaîne dans la mémoire (inférieur à N) ;
- Elle réservera une autre espace de mémoire nécessaire pour stocker la chaîne rentrée ;
- Elle copiera la chaîne dans cet espace de mémoire ;
- Elle libérera le premier espace de mémoire ;
- Elle retournera le pointeur sur le deuxième espace.

2) (*) Quel problème peut on avoir lors de l'utilisation de cette fonction ? Réfléchissez à une fonction qui ne serait pas contrainte par la longueur de la chaîne saisie.

Exo 5.2.

Écrivez les fonctions suivantes:

```
int is_prefix(char * string1, char * string 2)
```

Cette fonction retourne 1 si string1 est une chaîne préfixe de la chaîne string2, 0 sinon.

```
int str_length(char* s)
```

Cette fonction retourne la longueur de la chaîne de caractères *s* passée en argument;

```
void str_copy(char* str1, char * str2)
```

Cette fonction copie la chaine *str1* vers la chaîne *str2*.

```
int str_substring(char* str1, char * str2)
```

Cette fonction retourne 1 si la chaîne *str1* est une sous-chaine de *str2*.

```
char * concatenate\_string(char *str1, char *str2)
```

Cette fonction prend en argument deux chaînes de caractère et retourne une chaîne composée de ces deux chaînes concaténées.

Exo 5.3. Tableaux de chaînes de caractère

Note sur les tableaux des chaînes de caractères

Un tableau de chaînes de caractères est un tableau de pointeurs de type char. En effet, une chaîne de caractères est un tableau de caractères. Ainsi, lorsque l'on déclare

```
char str[10]="hello";
```

on déclare un pointeur de type char (str), on réserve une zone de mémoire de taille de 10 octets, et on initialise le pointeur str par l'adresse de début de cette zone. En même temps, les 5 premières cases du tableau sont initialisées par la chaîne "hello", et la sixième par le caractère de valeur 0.

Lorsque on souhaite déclarer un tableau de chaîne de caractères, on doit déclarer un tableau de pointeurs de type char, chaque pointeur pointant vers le début de la chaîne correspondante. Ainsi la ligne

```
char str[][10]={"hello", "here", "Jussieu", "ABCdaire"};
```

doit être lue de la manière suivante : "str est un tableau de taille inconnue, dont chaque élément est un tableau de 10 éléments de type char.

Ainsi, un tel tableau peut contenir un nombre indéfini de chaînes de caractères, de 9 caractère maximum chacune (pourquoi 9 et pas 10 ?).

Pour utiliser une des chaînes de ce tableau, il suffit de s'y référer ainsi : $str[n]$ pour la $(n+1)$ème chaîne.

La définition ci-dessus crée un tableau de 4 chaînes, et réserve de la place juste pour 4 chaînes de 9 caractères chacune. Cela n'est pas efficace pour les raisons évidentes: il se peut qu'à l'étape de l'écriture de programme, on n'ait pas d'information sur la taille du tableau.

Pour créer une structure plus efficace, on utilise la fonction malloc qui permet d'allouer dynamiquement de l'espace.

Ainsi, pour créer un tableau de chaînes de caractères contenant des chaînes de longueurs variables, on procède de la manière suivante.

– On déclare un tableau de pointeurs de type char sans l'initialiser :

```
char *str[100];
```

La taille du tableau correspond au nombre maximal des chaînes de caractères que l'on souhaite stocker.

– Pour chaque chaîne que l'on souhaite stocker (par exemple, la chaîne s), on réserve de l'espace mémoire, et on affecte l'élément de tableau avec le numéro correspondant i :

```
char s[]="first string".
str[i]=(char*) malloc(sizeof(char)*strlen(s));
```

Ensuite, on copie la chaîne de caractères que l'on souhaite stocker vers le bloc de mémoire réservé:

```
strcpy(str[2], s);
```

1) On déclare un tableau de chaînes de caractère comme suit :
char tab_str[][LENGTH_MAX]={"stalker", "writer", "scientist", "zona"};
Expliquez comment les données sont stoquées dans la mémoire.

2) Écrivez une fonction affichant les chaînes de caractères du tableau des chaînes de caractère passé en argument. La fonction a deux arguments: le pointeur vers le tableau des chaînes de caractère et le nombre des éléments de ce tableau.

3) Écrivez une fonction affichant le contenu de *toutes les cases* du tableau de chaînes de caractère. On affichera "\0" si un élément de tableau contient la valeur 0.

Les deux fonctions prendront le tableau en argument. Aucune variable globale ne doit être utilisée, sauf les constantes désignant les dimensions du tableau.

6 Les listes chaînées

Le problème des tableaux en langage C, est qu'il n'est pas toujours facile d'en manipuler.

Par exemple, si on a créé un tableau de taille 100 et si on souhaite à l'étape de l'exécution augmenter sa taille de 1 élément, on doit faire les opérations suivantes :

– créer un bloc de mémoire capable de stocker 101 éléments de tableau,

– copier les 100 éléments existants dans cette zone,

– écrire le 101ème élément.

Si le tableau initial contient un grand nombre d'éléments, ce mode de fonctionnement n'est pas efficace.

De même, si jamais il est nécessaire d'insérer un élément au milieu d'un tableau, il peut être nécessaire de déplacer un très grand nombre d'éléments.

Ainsi, il existe une autre structure de liste (tableau) qui permet de réaliser ces opérations avec beaucoup de facilité. Il s'agit de listes chaînées.

Soit une structure qui contient :

– un champ de donnée (l'information qui est stockée dans l'élément)

– un pointeur vers l'élément suivant

– un pointeur vers l'élément précédent

Par exemple, une liste chaînée d'entier se définit à l'aide de la structure suivante :

```
struct element{
int data;
struct element * suiv;
struct element * prec;
};
```

Ainsi, les listes chaînées ne font pas du tout recours aux mécanismes de tableau du C. Leur avantage principal est que les éléments de la liste peuvent se trouver dans les zones de mémoire séparées; ils ne sont pas obligées de se suivre. Ici nous proposons quelques exercices permettant de découvrir la manipulation de listes chaînées.

Le but de cette suite d'exercice est d'écrire une bibliothèque de fonctions gérant une liste chaînée contenant une base de données sur les élèves. Pour chaque élève, les informations suivantes doivent être stockées :

– numéro d'étudiant;

– nom d'étudiant

– sa moyenne

Cette fonction retourne le pointeur vers le premier élément de la liste.

La liste ainsi créée doit être triée par le numéro d'étudiant.

Cette bibliothèque contiendra les fonctions suivantes :

```
struct element * saisir_element();
```

Cette fonction crée un élément (à l'aide de la fonction *malloc* - pourquoi ?)de la liste et remplit ses champs par les valeurs saisies par l'opérateur au clavier. Les champs *suiv* et *prec* sont initialisées à *NULL*. En ce qui concerne le champ "nom", vous réserverez exactement l'espace mémoire nécessaire pour stocker le nom à l'aide de la fonction *malloc*.

Si l'utilisateur saisit -1 pour le numéro d'étudiant, cette fonction doit retourner *NULL*, sans rien faire d'autre. Cela signifiera la fin de la saisie.

```
struct element * inserer_debut(struct element * liste, struct element *pelement);
```

Cette fonction insère l'élément donné par le pointeur *pelement* au début de la liste. La fonction retourne le pointeur vers la liste modifiée.

```
struct element * inserer_en_place(struct element * liste, struct element *pelement);
```

Cette fonction, la plus complexe, permet d'insérer l'élément donné par le pointeur *pelement* à sa place *dans une liste triée*. La liste initiale est donnée par le pointeur liste. Pour écrire cette fonction, il faut choisir le champs sur lequel on effectue le tri: par exemple, par le numéro d'étudiant.

```
struct element * supprimer_element(struct element *pelement);
```

Cette fonction détruit l'élément pointé par le pointeur *pelement*. La fonction retourne le pointeur vers le début de la liste modifiée.

```
struct element * inserer_fin(struct element * liste, struct element *pelement);
```

Cette fonction, insère l'élément donné par le pointeur *pelement* à la fin de la liste. La fonction retourne le pointeur vers la liste modifiée.

```
void detruire_liste(struct element *liste);
```

Cette fonction détruit la liste. En fait, dans la mesure où les éléments de la liste seront créés à l'aide de la fonction *malloc*, il faut absolument prévoir la possibilité de libérer la mémoire réservée. Il faut penser à libérer la mémoire occupée par les noms d'élèves et par les éléments mêmes de la liste.

```
void afficher_liste(struct element *liste);
```

Cette fonction permet d'afficher les éléments de la liste.

```
int longueur_liste(struct element *liste);
```

Cette fonction affiche la longueur de la liste.

Pour les deux dernières fonctions, écrivez une version basées sur les boucles, et une basée sur la récursivité. Vous ajouterez le suffixe *_recursive* à leurs noms.

Vous écrirez ces fonctions dans le fichier liste.c, accompagné du fichier liste.h. La fonction main sera écrite dans un fichier main.c. Vous compilerez le tout avec la commande:

```
gcc liste.c main.c
```

Pour vous aider, nous proposons ici la listing de la fonction main():

```
main (){
// construction de la liste :
struct element * liste=NULL, *pelement;

// création de la liste
while(1){ // cycle infini
// saisie d'un élément de la liste
pelement=saisir_element();

if (pelement == NULL)  // on ne sort de la cycle que si l'utilisateur a saisi -1
                       // pour le numéro d'étudiant
        break;
else
        liste=inserer_en_place(liste, pelement); // sinon on insère l'élément saisi à sa place
}

// on affiche la liste
afficher_liste(liste);

printf("La liste contient %d elements. \n" longueur_liste(liste));

// on detruit la liste
detruire_liste(liste);
return;
}
```

Exo 6.1. Exercice supplémentaire: traitement des listes

1) Écrivez une fonction qui retire d'une liste donnée les personnes ayant un numéro d'étudiant pair.

2) Écrivez une fonction qui fusionne deux listes.

7 Lecture de fichiers

Une saisie manuelle des bases de données est fastidieuse. Nous souhaitons maintenant avoir une possibilité d'enregistrer une liste chaînée dans un fichier (tableau) et de lire une liste à partir d'un fichier. On utilisera les fonctions manipulant les listes chaînées réalisées dans le chapitre précédent.

Nous écrivons 2 fonctions:

3) Lecture.

```
struct element *read_list(char *filename)
```

Cette fonction lit le fichier *filename*, en extrait les informations et les enregistre dans une liste chaînée. Le fichier doit être composé des lignes ayant le format suivant :

```
NOM_ETUDIANT NUMERO_ETUDIANT MOYENNE
```

4) Écriture

```
void write_list(char* filename, struct element *liste)
```

Cette fonction écrit la liste *liste* dans le fichier *filename*, en créant dans le fichier les lignes selon le même format.

Ces deux fonctions sont écrites dans un fichier file_liste.c qui est compilé en même temps que les fichiers main.c et liste.c à l'aide de la commande

```
gcc liste.c main.c file_liste.c
```

N'oubliez pas de déclarer toutes les fonctions créées dans le fichier liste.h à l'aide du mot clé *extern*.

8 Arbres

Un arbre est une structure de donnée organisée d'une manière hiérarchique. Elle est composée des sommets (nœuds) et des liens reliant les sommets. Un arbre est un cas particulier d'un graphe orienté. Chaque nœud d'un arbre possède jusqu'à N liens descendants vers les nœuds appelés fils. A la racine d'un arbre se trouve un seul nœud appelé "nœud racine". Ainsi, chaque nœud, à part le nœud racine, possède un et un seul parent (le lien ascendant). Les nœud ne possédant pas de fils s'appellent feuilles.

L'intérêt des arbres en informatique se manifeste partout où il s'agit d'une organisation hiérarchique des données. Ainsi, un arbre informatique est l'ensemble des nœuds avec ses liens, chaque nœud possédant une certaine information.

Une application classique faisant appel aux arbres est donnée par la représentation d'expression arithmétiques.

La structure permettant de réaliser des arbres d'expressions arithmétiques est donnée ci-dessous:

```
struct noeud{
          struct noeud *fg;
          struct noeud *fd;
          int type;
          int value
};
```

Ici, *fg* et *fd* sont des pointeurs vers les fils gauches et droits, *type* est le type du nœud (0 pour une opérande, 1 pour "+", 2 pour "-", etc..), value est la valeur de l'opérande le cas échéant.

Vous devez créer les fonctions suivantes :

```
struct noeud *create_noeud(struct noeud *fg, struct noeud *fd, int
type, int value)
```

Cette fonction crée un nœud,

```
void afficher_expression(struct noeud *n)
```

Cette fonction permet d'afficher l'expression contenue dans l'arbre sous forme préfixe.

```
float calculer_expression(struct noeud *n)
```

Cette fonction calcule la valeur de l'expression. Attention, les opérandes de l'expression sont entiers, mais le résultat peut être à virgule flottante (ex., 1/2).

La fonction *main* permettant de tester vos fonction peut ressembler à cela:

```
main(){
  // création d'un arbre pour l'expression -1+2*(3-4)/5
//                            +
//                         -      /
//                       1     *   5
//                            2  -
//                              3 4


 struct noeud *n1, *n2, *n3, *n4, *n5, *n6, *n7, *n8, *n9, *n10;
 n1=create_noeud(NULL,NULL, 0, 1); // 1
 n2=create_noeud(NULL,NULL, 0, 2); // 2
 n3=create_noeud(NULL,NULL, 0, 3); // 3
 n4=create_noeud(NULL,NULL, 0, 4); // 4
 n5=create_noeud(NULL,NULL, 0, 5); // 5
 n6=create_noeud(n3,n4, 2, 0); // 3-4
 n7=create_noeud(n2,n6, 3, 0); // 2*(3-4)
 n8=create_noeud(n7,n5, 4, 0); // (2*(3-4))/5
 n9=create_noeud(n1,NULL,5,0); // -1
 n10=create_noeud(n9,n8 ,1,0); // -1+((2*(3-4))/5)
 afficher_expression(n10);
 printf("=%f \n",calculer_expression(n10));
}
```

9 Interface avec l'utilisateur: lecture de la ligne commande

Écrivez un programme qui calcule la résistance équivalente de deux résistances connectées en parallèle ou en série. L'objectif de cet exercice est de créer une interface utilisateur en ligne de commande.

L'utilisation du programme est le suivant :

```
resistance <ser|par> <value 1> <value2>
```

On donne en ligne de commande le type de la connexion et les valeurs des résistances en ohms. Le programme doit détecter les erreurs de syntaxe.

Pour réaliser cet exercice, vous avez besoin de la fonction *strtod* de la bibliothèque *stdio.h*