SR01 : Maitrise des systèmes informatiques

Devoir 1 : Montée en compétences C

GAJAN Antoine & ZHENG Cristina

Table des matières

Introduction

Ce TP a pour but de nous familiariser avec le langage C. Au cours de celui, les connaissances acquises lors des premiers cours magistraux et travaux dirigés seront mobilisées pour monter en compétences.

Composé de 3 exercices, il nous permettra de comprendre la priorité des opérateurs (exercice 1), les boucles (exercice 2) et la lecture de fichiers (exercice 3).

Dans chacun des exercices, nous veillerons à expliciter les choix effectués, les modifications de l'énoncé réalisées ainsi que proposer un code cohérent avec la consigne.

1 Exercice 1 : Compréhension de programmes

Dans cet exercice, il est question de comprendre les programmes afin de prédire le résultat obtenu. Pour cela, il nous semble bon de rappeler l'ordre des priorités des opérateurs en C grâce au tableau suivant.

Priorité	Opérateur
16	( ) [ ] . -> ++ -- (postfixé: i++ i--)
15	! ~ ++ -- (préfixé ++i --i) + - (unaire -3) * (indirection) & (adresse de) sizeof()
14	conversion
13	* (multiplication) / %
12	+ -
11	<< >>
10	< <= > >=
9	== !=
8	& (ET bit à bit)
7	^
6	|
5	&&
4	||
3	?:
2	= += -= *= /= %= <<= >>= &= ^= |=
1	,

FIGURE 1 – Priorités des opérateurs en C

1.1 Programme 1

```c
# include <stdio.h>
int main (){
    int A=20 , B =5;
    int C=!-- A /++! B;
    printf (" A=%d B=%d c=%d \n", A, B, C);
}
```

A la compilation du programme, on obtient l'erreur suivante :

```
error: lvalue required as increment operand
```

Cette erreur s'explique par la priorité des opérateurs. L'expression !–A/++!B est équivalent à (!(–A))/(++(!B)). Le problème est lié à (++(!B)). En effet, !B renvoie 0, qui est une rvalue (une valeur temporaire qui n'est pas persistante au-delà de l'expression dont elle est issue). Or l'opérateur ++ requiert une lvalue (expression qui permet d'accéder à une donnée, comme une variable définie). En effet, l'opérateur ++ va modifier la valeur de la lvalue en l'incrémentant et en modifiant donc sa valeur en mémoire. Ce qui n'est pas possible ici avec une rvalue.

1.2 Programme 2

```
# include <stdio.h>
int main (){
    int A=20 , B=5 , C= -10 , D =2;
    printf ("%d \n", A&& B ||! 0&& C ++&&! D ++) ;
    printf ("c=%d d=%d \n", C, D);
}
```

A l'exécution du programme, on obtient le résultat suivant :

```
1
c=-10 d=2
```

Pour le premier printf, la priorité des opérateurs donne l'expression suivante : (A&& B) || (((!0) && (C++)) && (!(D ++))). Cette expression, qui au premier abord peut sembler compliqué, ne l'est pas. En effet, l'expression est de la forme expression 1 ou expression2. Si expression1 est évaluée à vrai, le compilateur n'exécutera pas l'expression2. Ici, A&&B renvoie 1 (vrai) car 1 et B sont différents de 0. Donc le retour de cette expression donnera 1, l'expression2 n'étant pas exécutée. Ainsi, le premier printf renvoie 1.

En ce qui concerne le deuxième printf, les valeurs de C et D n'ont pas été modifiées au cours de l'exécution de la ligne précédente comme expliqué ci-dessus. Ainsi, on obtiendra C = -10 et D = 2.

1.3 Programme 3

```
# include <stdio.h>
int main (){
    int p [4]={1 , -2 ,3 ,4};
    int * q=p;
    printf ("c=%d\n", *++ q** q ++) ;
    printf ("c=%d \n" ,* q);
}
```

A l'exécution du programme, on obtient le résultat suivant :

```
c = 4
c = 3
```

L'explication de ce résultat est la suivante. Tout d'abord, on définit q comme un pointeur sur un entier, et plus précisément sur le premier élément de p qui contient la valeur 1.

D'après les priorités des opérateurs, l'expression *++ q** q ++ équivaut à (*(++ q)) * (*(q ++)). L'expression ++q est exécutée en première car elle contient l'opérateur avec la priorité la plus élevée. ++q modifie la valeur de q en l'incrémentant immédiatement car c'est une pré-incrémentation. q pointe donc sur le 2ème élément du tableau. Quant à l'expression q++, elle incrémentera q après le retour car c'est une post-incrémentation. La valeur pointée par q est alors -2. On a donc (*(++ q)) et (*(q ++)) qui valent toutes deux -2. (-2) * (-2) = 4 d'où le résultat obtenu. Après ce premier printf, la post-incrémentation est effectuée. q pointe alors sur la 3ème case du tableau. Pour le deuxième printf, il affiche la valeur pointée par q. La valeur contenue dans la 3ème case étant 3, le deuxième printf affichera 3.

1.4 Programme 4

```
# include <stdio.h>
int main (){
    int p [4]={1, -2 ,3 ,4};
    int * q=p;
    int d=* q&* q ++|* q ++;
    printf ("d=%d\n", d);
    printf ("q=%d \n" ,* q);
}
```

Ce programme renvoie l'affichage suivant :

```
d=-1
q=3
```

Comme pour le programme 3, on définit on définit q comme un pointeur sur un entier, et plus précisément sur le premier élément de p qui contient la valeur 1.

Ensuite on a * q&* q ++|* q ++ qui, avec la priorité des opérateurs, peut s'écrire ((*q)&(*(q++)))|(*(q++)). L'expression q++ est en post-incrémentation, donc a (*q)&(*(q++)) = 1&1 = 1, & étant un "ET" en logique binaire. Le "OU" en logique binaire | compte comme une nouvelle opération, q est donc incrémenté une fois par le premier q++ avant d'exécuter le "OU". q pointe donc le deuxième élément de p. On se retrouve avec d = 1|(-2) = -1.

En effet :
$(1)_{10} = (0001)_2$
$(-2)_{10} = (1110)_2$ (complément à deux : $(2)_{10} = (0010)_2$, donc $(-2)_{10} = (1101+1)_2 = (1110)_2$).
$(0001)_2|(1110)_2 = (1111)_2$, avec le complément à deux, on trouve $(-1)_{10}$.
On a finalement la post-incrémentation du dernier q++ qui s'effectue, *q pointe la 3e valeur de p, *q = 3 à la fin.

1.5 Programme 5

```
# include <stdio.h>
int main (){
    int a=-8 , b =3;
    int c= ++a&&-- b ? b --: a ++;
    printf ("a=%d b=%d c=%d\n",a, b, c);
}
```

Ce programme renvoie l'affichage suivant :

```
a=-7 b=1 c=2
```

La valeur de c est donnée selon le raisonnement suivant. Si ++a et –b différent de 0, alors c = b–, sinon c = a++. ++a et –b sont des pré-incrémentations. On aura donc a = -7 et b = 2. a et b étant non nuls, on a c = b–. b– étant une post-décrémentation, on aura finalement c = 2. Puis la post-décrémentation est effectuée. Donc b = 1. Ainsi, on a bien a = -7, b = 1 et c = 2.

1.6 Programme 6

```
# include <stdio.h>
int main (){
    int a=-8 , b =3;
    a > >=2^b;
    printf ("a=%d\n",a);
}
```

Ce programme retourne :

```
a=-4
```

En effet, a >>= 2^b s'écrit aussi a >>= (2^b avec la priorité des opérateurs. On a 2^3 = $(0000\ 0010)_2$^$(00000011)_2$ = $(0001)_2$ = 1.

On a donc l'expression a >>= 1. Cela signifie que a = a >> 1. Autrement dit, on doit décaler a de 1 bit vers la droite.

On sait que a = -8.

D'après la complémentation à deux, comme $8_{10} = (00001000)_2$ on a $-8_{10} = (11110111)_2 + 1 = (11111000)_2$.

Comme -8 est signé, en décalant de 1 bit vers la droite, on va insérer un bit de poids fort de valeur 1. Ainsi (1111 1000) >> 1 donnera $(1111\ 1100)_2$.

En utilisant à nouveau la complémentation à deux, on obtient que la valeur absolue du nombre précédent est $(0000\ 0011)_2 + 1 = (00000100)_2$.

Ceci est le code de la valeur 4. Donc la valeur de a recherchée est -4.

2 Exercice 2 : Notes

2.1 Lecture de N notes et mémorisation

Il s'agit d'écrire un programme qui lit les notes de N étudiants de l'UTC dans un devoir de l'UV SR01 et les mémorise dans un tableau POINTS de dimension N. Pour cela, nous avons décidé de réaliser une boucle "for" à N itérations. A chaque itération, on demandera à l'utilisateur la note en s'assurant qu'elle soit dans l'intervalle [0, 60]. Ce point n'est pas explicité dans le sujet, mais les questions suivantes laissent penser que l'on travaille avec des nombres entiers entre 0 et 60. C'est pourquoi nous avons décidé de traiter le problème de cette manière.

```
// Fonction qui demande a l'utilisateur N notes et les stocke dans un
    tableau
int* lecture(int N)
{
    // Allocation dynammique du tableau points
    int *points =  malloc(sizeof(int) * N);
    printf("\nEntrez des notes entre 0 et 60.");
    for (int i = 0; i < N; i++)
    {
        printf("\nNote n %d : ", i+1);
        scanf("%d", &points[i]);
        // Tant que la note n'est pas dans l'intervalle
        while (points[i] < 0 || points[i] > 60)
        {
            // On affiche erreur et on redemande
            printf("\nErreur : la note doit etre comprise entre 0 et
    60.\nNote n %d : ", i + 1);
            scanf("%d", &points[i]);
        }
    }
    return points;
}
```

Lors de la création de cette fonction, nous avons été confronté à un problème : la création du tableau "points". En effet, il aurait été peut-être plus aisé de définir un entier TAILLE-MAX qui aurait indiqué le nombre maximal de notes que l'utilisateur peut rentrer. Toutefois, cela n'était pas optimal car si l'utilisateur ne souhaite rentrer qu'une seule note et que le tableau points est défini sur 100 cases, cela consommerait une place mémoire importante. C'est pourquoi nous avons choisi de pratiquer l'allocation dynamique, en créant un tableau "sur mesure" adapté au nombre N fixé en paramètre.

L'inconvénient de cette pratique est qu'il ne faudra pas oublier de supprimer dynamiquement, avec free, le tableau. Sinon, cela pourrait générer une fuite mémoire, ce qui n'est pas l'effet recherché. En effet, l'allocation dynamique nous a permis une meilleure gestion mémoire, il serait dommage de la gâcher.

2.2 Répartition des notes

A partir du tableau points des étudiants réalisés ci-dessus, on veut réaliser un tableau NOTES de dimension 7 qui est composé de la façon suivante :

— NOTES[6] contient le nombre de notes 60
— NOTES[5] contient le nombre de notes de 50 à 59
— NOTES[4] contient le nombre de notes de 40 à 49
— ...
— NOTES[0] contient le nombre de notes de 0 à 9

Pour répondre à cette question, nous allons d'abord allouer dynamiquement un tableau de taille 7, que nous initialiserons avec la valeur 0.

Ensuite, pour chaque valeur du tableau, nous regarderons à quel intervalle la valeur appartient. On incrémentera alors la case de notes adaptées.

Si la note vaut 24, alors elle doit appartenir à NOTES[2]. On remarquera donc que l'indice de NOTES associées à la note lue est l'indice des dizaines. En C, pour obtenir le chiffre des dizaines, il faut tout simplement diviser la note par 10 et la stocker dans un entier.

Enfin, on retournera ce tableau notes, qui contient la répartition des notes.

```
// Fonction de repartition en intervalles de dizaines
int* repartition(int* points, int taille)
{
    int *notes = malloc(sizeof(int)*7);
    // Initialisation du tableau
    for (int i = 0; i < 7; i++) notes[i] = 0;
    // Mise a jour du tableau de notes pour chaque note dans points
    for (int i = 0; i < taille; i++)
    {
        int intervalle_appartenance = points[i] / 10;
        // Incrementation du nombre de notes dans l'intervalle voulu
        notes[intervalle_appartenance] ++;
    }
    return notes;
}
```

2.3 Graphique en nuage de points

Il s'agit ici d'établir un graphique en nuage de points représentant le tableau notes. On utilisera le symbole 'o' pour représenter le point dans le graphique et affichez le domaine des notes en dessous du graphique. On souhaite obtenir l'affichage ci-dessous :

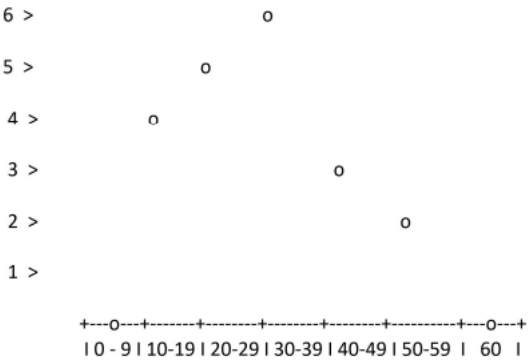

FIGURE 2 – Graphique en nuage de points

Cette fonction, qui peut sembler compliqué au premier abord, n'est en fait qu'une imbrication de boucles "for". Il faudra donc commencer par trouver le nombre de lignes nécessaires MAXN. Pour cela, nous avons écrit la fonction suivante, qui retourne le maximum d'un tableau passé en paramètre :

```
// Fonction qui retourne le max du tableau d'une taille donnee
int max(int* tab, int taille)
{
    int indice_max = 0;
    for (int i = 0; i < taille; i++)
    {
        // Si la valeur actuelle est superieure a la valeur max, on
   change le max
        if (tab[i] > tab[indice_max])
        {
            indice_max = i;
        }
    }
    // Retourne le max
    return tab[indice_max];
}
```

Ensuite, pour chaque valeur x possible entre MAXN et 1, il faudra savoir s'il faut afficher un "o" ou non pour chaque intervalle de notes. Nous modéliserons un intervalle de notes par une chaine de longueur 8. Pour savoir s'il faut afficher le "o" ou non, il faut regarder si la valeur NOTES[i] est égal à x.

Enfin, il faudra afficher l'axe des abscisses. Pour cela, deux cas sont à distinguer. Soit, il faut afficher un "o" car aucune valeur n'est présente dans l'intervalle de NOTES (on aura donc "+—o—+"), soit il faudra afficher l'axe classique (on aura donc "+———-+"). Ainsi, on obtient le code suivant :

```
// Fonction qui affiche en nuage de points le tableau
void nuage_points(int* tab, int taille)
{
    int maxn = max(tab, taille);
```

```
    // Affichage des ordonnees
    for (int i = maxn; i > 0; i--)
    {
        printf("%d >\t", i);
        for (int j = 0; j < taille; j++)
        {
            // Affichage du o si necessaire
            if (tab[j] == i)
            {
                printf("    o   ");
            }
            // Sinon affichage espace blanc
            else
            {
                printf("        ");
            }
        }
        printf("\n\n");
    }
    printf("  \t");
    // Affichage de l'axe des abscisses
    for (int i = 0; i < 7; i++)
    {
        if (tab[i] == 0)
        {
            printf("+---o---");
        }
        else
        {
            printf("+--------");
        }
        // Si c'est les notes 60, on ajoute le + final de l'affichage
   puis retour a la ligne
        if (i == 6) printf("+\n");
    }
    printf("  \t");
    // Affichage des valeurs en abscisse
    for (int i = 0; i < 7; i++)
    {
        if (i == 0)
        {
            printf("| %d - %d ", i*10, i*10+9);
        }
        else if (i == 6)
        {
            printf("|  60   |\n");
        }
        else
        {
            printf("|%d - %d", i*10, i*10+9);
        }
    }
}
```

2.4 Graphique en bâtons

Le principe est similaire, on souhaite obtenir le résultat suivant :

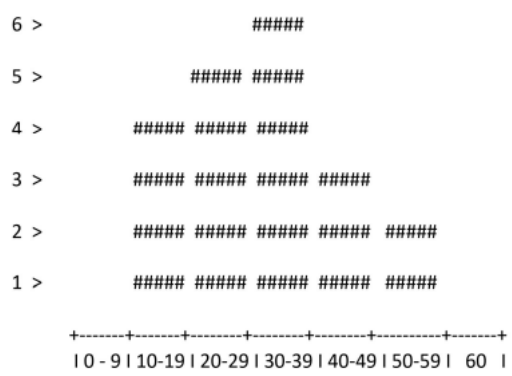

FIGURE 3 – Graphique en batons

Au lieu d'écrire un "o", on souhaite écrire une chaine constituée de "#". La seule différence est que l'on souhaite écrire une ligne de "#" sur chaque hauteur inférieure à la valeur de NOTE[i]. Le principe reste le même, c'est pourquoi on peut réutiliser notre code pour obtenir le résultat escompté. Le code de la fonction étant long, il ne sera pas rédigé ici, mais vous pourrez le trouver dans le fichier ex2.c. Son prototype est :

```
void graphique_batons(int* tab, int taille);
```

2.5 Fonctions de service

Dans la même lignée que pour le maximum, nous avons codé des fonctions retournant le minimum et la moyenne d'un tableau. Ces fonctions sont présentes sous les prototypes suivants dans le fichier ex2.h :

```
int min(int* tab, int taille)
int max(int* tab, int taille)
double moyenne(int* tab, int taille)
```

3 Exercice 3 : Gestion de restaurants

3.1 Structure restaurant

On souhaite créer une structure Restaurant en C contenant les champs suivants : nom du restaurant, adresse du restaurant, position du restaurant et sa spécialité.

A première vue, on pourrait être tenté de proposer une structure naïve comme suit :

```
typedef struct Restaurant{
    char nom[TAILLE_MAX];
    char adresse[TAILLE_MAX];
    double x, y; // Position
    char *specialite[TAILLE_MAX];
}Restaurant;
```

Cependant, en analysant la structure du fichier restau.txt proposé et les fonctions à réaliser, cette structure ne serait pas adaptée. En effet, comment pourrait-on savoir rapidement combien de spécialités ont été inscrites ? De plus, les attributs x et y sont peu parlants pour quelqu'un n'ayant pas lu le sujet.

C'est pourquoi nous avons décidé de modifier la structure demandée, en créant des structures annexes (Position et Specialite) pour gérer respectivement la position et les spécialités du restaurant.

On obtient donc le code suivant pour les structures :

```
#define NB_RESTAU_MAX 100
#define TAILLE_MAX 50

typedef struct Position{
    double x;
    double y;
}Position;

typedef struct Specialite
{
    char noms[5][TAILLE_MAX]; // un restaurant propose au maximum 5
    specialites
    int nb;
}Specialite;

typedef struct Restaurant{
    char nom[TAILLE_MAX];
    char adresse[TAILLE_MAX];
    Position coordonnees;
    Specialite spe;
}Restaurant;
```

3.2 Lire un fichier de restaurants

On souhaite à présent lire un fichier contenant des restaurants.

Chaque ligne du fichier se présente comme suit : Restaurant ; adresse ; coordonee ; specialite ;

Par exemple, on a : Edern ; 6, rue Arsene Houssaye - Paris 8eme ;(x=1.5, y=44.8) ; Cuisine gastronomique ;

Ce qui donne le formalisme de représentation suivant : Nom ; Adresse ; (x= ?, y= ?) ;Spe1, Spe2,... ;

On veut récupérer dans le tableau restaurants toutes les informations concernant chaque restaurant inséré dans le fichier nommé chemin. La fonction lire-restaurant retournera en fin de lecture le nombre de restaurants lu à partir du fichier.

Pour résoudre ce problème, on pourra ouvrir le fichier avec la fonction fopen(). Puis, on pourra parcourir le fichier ligne par ligne grâce à la fonction fgets(). On veillera à étudier les lignes qui sont non vides, c'est-à-dire celles qui contiennent un restaurant. A chaque restaurant rencontré, on incrémentera un compteur de 1 afin de le retourner à la fin de la fonction.

On séparera dans chaque ligne les différentes parties. Il faudra pour cela couper le texte à chaque " ;" rencontré. Cela pourra être fait avec strtok(). On pourra alors ensuite créer un objet de type restaurant que l'on ajoutera au tableau. Enfin, on retournera le nombre de restaurants lus dans le fichier.

Le code de la fonction étant long, il ne sera pas rédigé ici. Toutefois, vous pourrez le trouver dans le fichier ex3.c. Son prototype est :

```
int lire_restaurant(char* chemin, Restaurant restaurants[NB_RESTAU_MAX
    ]);
```

Le fichier proposé dans l'énoncé comporte 21 restaurants par exemple.

3.3 Insérer un restaurant

On veut à présent ajouter un restaurant au fichier. Pour cela, nous allons ouvrir le fichier en écriture avec fopen(). Ensuite, nous écrirons avec fprintf() selon le formalisme évoqué ci-dessus. Ainsi, on obtient le code suivant :

```
void inserer_restaurant(char* chemin, Restaurant restaurant)
{
    // Ouverture du fichier en lecture
```

```
    FILE* fichier = fopen(chemin, "a");
    // Si l'ouverture a fonctionne
    if (fichier != NULL)
    {
        // Ecriture dans le fichier
        fprintf(fichier, "\n\n%s; %s; (x=%f, y=%f); {", restaurant.nom,
    restaurant.adresse, restaurant.coordonnees.x, restaurant.
    coordonnees.y);
        // Ajout des specialites
        for (int i = 0; i < restaurant.spe.nb; i++)
        {
            if (i == restaurant.spe.nb - 1) fprintf(fichier, "%s",
    restaurant.spe.noms[i]);
            else fprintf(fichier, "%s, ", restaurant.spe.noms[i]);
        }
        fprintf(fichier, "};");
    }
    // Fermeture du fichier
    fclose(fichier);
}
```

3.4 Chercher un restaurant à proximité

On veut obtenir l'ensemble des restaurants se situant dans un rayon donné par rapport à la position de l'utilisateur.

Pour cela, il nous a semblé intéressant de définir deux fonctions de service : une qui calcule la distance entre un utilisateur et un restaurant, l'autre qui indique si le restaurant est dans le rayon donné.

Pour rappel, une distance à vol d'oiseau entre un point A(x, y) et B(x', y') se calcule comme suit : $\sqrt{(x-x')^2 + (y-y')^2}$

On obtient donc les fonctions de service suivantes :

```
// Fonction qui calcule la distance entre un point et un restaurant
double calcul_distance(double x, double y, Restaurant r)
{
    return sqrt((x-r.coordonnees.x)*(x-r.coordonnees.x) + (y - r.
    coordonnees.y)*(y - r.coordonnees.y));
}
```

```
// Fonction qui renvoie si un restaurant est dans le rayon demande par
    l'utilisateur
int est_dans_le_rayon(double x, double y, Restaurant r, double rayon)
{
    // Retourne 1 si on est dans le rayon
    if (calcul_distance(x, y, r) <= rayon)
    {
        return 1;
    }
    // Retourne 0 sinon
```

```
    else
    {
        return 0;
    }
}
```

Enfin, pour connaitre l'ensemble des restaurants dans le rayon, il ne reste plus qu'à itérer sur les restaurants de la liste. On regarde s'ils sont ou non dans le rayon imposé. Si c'est le cas, alors on les ajoute à la liste results.

Afin d'avoir un code portable, nous avons ajouté 2 arguments à la fonction de recherche : un tableau de restaurants (qui est la liste à traiter) et le nombre d'élements dans cette liste.

Ainsi, la fonction possède le prototype suivant et est disponible dans ex3.c :

```
int cherche_restaurant(double x, double y, double rayon_recherche,
    Restaurant liste_depart[NB_RESTAU_MAX], int nb_restau, Restaurant
    results[NB_RESTAU_MAX]);
```

Le type de retour est int afin de conserver la taille du tableau results. Cette taille nous sera utile afin d'afficher une liste des restaurants de results dans la fonction main().

Pour les afficher comme il est demandé, nous définirons des fonctions de service, qui affichent les restaurants, avec ou sans leur distance par rapport à un point (disponible dans la dernière section de cet exercice).

3.5 Chercher un restaurant par spécialité

On souhaite afficher à l'écran l'ensemble des restaurants qui possèdent une spécialité donnée et qui se trouvent dans un rayon spécifié. Cette fonction peut être codée avec la même logique que précédemment. Il faudra itérer sur chacun des restaurants, puis sur chacune des spécialités, afin de voir si elles correspondent ou non avec les spécialités souhaitées par l'utilisateur. On veillera à ce que, si deux spécialités correspondent à un même restaurant, alors le restaurant n'apparaît qu'une fois dans la liste résultat. On obtient donc le code suivant :

```
int cherche_par_specialite(double x, double y, Specialite spec,
    Restaurant liste_depart[NB_RESTAU_MAX], int nb_restau, Restaurant
    results[NB_RESTAU_MAX])
{
    // Variable qui contient l'indice du restaurant actuel dans results
    int indice_restau_result = 0;

    // Variables pour parcourir les specialites
    int est_ajoute = 0;
```

```
    // Parcours pour chaque restaurant
    for (int i = 0; i < nb_restau; i++)
    {
        est_ajoute = 0;
        // Pour chaque specialite du restaurant
        for (int j = 0; j < liste_depart[i].spe.nb && !est_ajoute; j++)
        {
            // Pour chaque spe en parametre
            for (int k = 0; k < spec.nb && !est_ajoute; k++)
            {
                // Si la specialite est correcte
                if (strcmp(liste_depart[i].spe.noms[j], spec.noms[k])
   == 0)
                {

                    // Ajout du restaurant a la liste des resultats
                    est_ajoute = 1;
                    results[indice_restau_result] = liste_depart[i];
                    indice_restau_result ++;
                }
            }
        }
    }
    // Tri du tableau
    tri(results, indice_restau_result, x, y);
    // Retourne le nombre de restaurants
    return indice_restau_result;
}
```

De même, nous avons modifié le prototype de la fonction en modifiant le type de retour (renvoi de la taille du tableau) et les arguments (ajout du tableau de départ avec sa taille).

3.6 Tri des restaurants

Il nous faut à présent trier le tableau pour l'afficher à l'écran comme demandé dans la consigne. Pour cela, on pourra effectuer un tri par sélection et utiliser la fonction de service calcul-distance() pour comparer les éléments. A chaque itération i, on cherchera donc le i-ème restaurant le plus proche, puis on l'échangera avec tab[i]. Ainsi, on obtient le code suivant :

```
void tri(Restaurant liste[NB_RESTAU_MAX], int nb, double x, double y)
{
    // Variable qui contient l'indice du minimum
    int indice_min;
    // Variable de copie temporaire lors du switch
    Restaurant temp;
    for (int i = 0; i < nb - 1; i++)
    {
        indice_min = i;
        for (int j = i + 1; j < nb; j++)
        {
```

```
            // Si la distance est inferieure
            if (calcul_distance(x, y, liste[indice_min]) >
    calcul_distance(x, y, liste[j]))
            {
                // On change l'indice_min
                indice_min = j;
            }
        }
        if (indice_min != i)
        {
            // Echanger les elements
            temp = liste[i];
            liste[i] = liste[indice_min];
            liste[indice_min] = temp;
        }
    }
}
```

On peut donc appeler cette fonction dans les fonctions de recherche précédemment évoquées afin d'améliorer l'expérience utilisateur. En effet, il est probable que l'utilisateur souhaite connaître en priorité les restaurants les plus proches, qui seront donc plus susceptibles de l'intéresser.

3.7 Fonction main

A présent, pour que le code compile et soit agréable pour l'utilisateur, il faut définir une fonction main.

Cette fonction main affichera les différentes options envisageables (lire un fichier, écrire dans un fichier, restaurants à proximité, tri par spécialité, quitter le programme). L'utilisateur devra alors choisir parmi les options proposées. On veillera à vérifier la cohérence des données rentrées. Ensuite, pour chacune des options envisageables, on veillera à demander les informations nécessaires au bon fonctionnement. Par exemple, pour insérer un restaurant, il faudra demander le nom, l'adresse, la position et les spécialités. Pour les fonctions de recherche, il faudra demander la position de l'utilisateur et les spécialités qu'il souhaite. Enfin, dans chacun des cas, on affichera les restaurants afin d'améliorer l'expérience utilisateur.

3.8 Fonctions de service

Nous avons créé les fonctions de service suivantes pour nous faciliter la tâche. Elles nous permettent entre autres de faire des affichages de restaurant.

Ainsi, on a réalisé les fonctions suivantes :

```
// Fonction qui affiche un restaurant
void affiche_restaurant(Restaurant restaurant)
```

```
{
    // Affichage
    printf("\n\nNom : %s\nAdresse : %s\nCoordonnees :\n\t- x=%f\n\t- y
   =%f\nSpecialite(s):", restaurant.nom, restaurant.adresse,
    restaurant.coordonnees.x, restaurant.coordonnees.y);
    // Affichage des specialites
    for (int j = 0; j < restaurant.spe.nb; j++)
    {
        printf("\n\t- %s", restaurant.spe.noms[j]);
    }
}
```

```
// Fonction qui affiche les restaurants d'une liste de restaurants sans
    leur distance par rapport a un point
void affiche_restaurants(Restaurant liste[NB_RESTAU_MAX], int nb)
{
    // Iteration sur les restaurants
    for (int i = 0; i < nb; i++)
    {
        affiche_restaurant(liste[i]);
    }
    printf("\n");
}
```

```
// Fonction qui affiche les restaurants d'une liste de restaurants avec
    leur distance a un point (x, y)
void affiche_restaurants_distance(Restaurant liste[NB_RESTAU_MAX], int
   nb, double x, double y)
{
    // Iteration sur les restaurants
    for (int i = 0; i < nb; i++)
    {
        affiche_restaurant(liste[i]);
        printf("\nDistance : %lf\n", calcul_distance(x, y, liste[i]));
    }
}
```

Egalement, nous avons créé deux fonctions qui permettent de vider le buffer et d'enlever le retour-chariot d'une chaine de caractères. Ces fonctions sont indispensables au bon fonctionnement de notre code. En effet, vider le buffer est une bonne pratique pour éviter les erreurs en cas de maladresse de la part de l'utilisateur. Quant à la fonction qui enlève le retour chariot, elle est nécessaire pour obtenir des données cohérentes avec la fonction fgets(). fgets() laisse par défaut le retour-chariot entré par l'utilisateur. Il nous faut donc le supprimer pour éviter d'avoir des retours à la ligne dans le fichier restau.txt ou lors d'affichages dans la console.

Ainsi, les prototypes de ces deux fonctions sont :

```
void vider_buffer();
void enlever_newline(char* texte);
```

Conclusion

Pour conclure, ce premier devoir nous a permis de nous familiariser avec le langage C. Il nous a permis au travers de ces exercices de comprendre et manipuler les fonctions de bases du langage.