Mémoire (notions avancées)

Dans ce TP, nous allons voir comment les pointeurs nous permettent de réaliser des choses assez élaborées en C : une FAP comme dans un langage orienté objet avec les pointeurs sur fonctions, la gestion des fonctions à nombre variable d'arguments pour faire par exemple votre propre `printf`, et quelques considérations sur les tableaux multidimensionnels et leur allocation.

1 - Pointeurs sur fonction

Dans certains cas il semblerait utile de pouvoir passer en paramètre une fonction à un programme afin qu'il puisse changer son comportement sans que l'on ait besoin de changer directement son code source. Par exemple, dans le cas d'une fap, il serait utile de pouvoir changer la fonction de comparaison des priorités afin de pouvoir avoir une fap triée soit par ordre croissant, soit par ordre décroissant, sans avoir à changer le code qui maintient les structures de données.

En C, il est impossible de stocker une fonction proprement dite directement dans une variable. En revanche, il est possible de définir un pointeur sur une fonction. Le pointeur en question pointe alors sur le début de la fonction. On peut lui appliquer l'opérateur () qui a tout simplement pour résultat d'appeler la fonction en question avec pour paramètres les valeurs passées dans les ().

Declaration :

```
type_de_retour (*nom)(types_des_parametres);
```

Ceci a pour effet de déclarer un pointeur nommé `nom` sur une fonction retournant une valeur de type `type_de_retour` et prenant comme paramètres des valeurs dont les types sont indiqués par `types_des_parametres`. Remarque : il y a bien des parenthèses autour de (*fonction) qui indiquent que l'* s'applique au nom et désigne un pointeur sur fonction et non une fonction retournant type * si on ne les avait pas mises.

De façon générale on peut dire que les fonctions et pointeurs sur fonction se manipulent de la même manière :

- un nom de fonction est équivalent à un pointeur constant sur fonction (qu'on peut donc affecter directement à un pointeur sur fonction).
- on peut appliquer l'opérateur () à un pointeur sur fonction qui se comporte donc comme une fonction.

L'intérêt est tout simple :

- permet de paramétrer un programme générique en changeant certaines parties de son traitement, par ex : un tri générique auquel on passerait la fonction de comparaison (facile de trier par ordre croissant ou décroissant avec le même code).
- permet d'inclure à un type abstrait la façon dont il interprète ses propres données, ex : inclure dans le type fap la fonction de comparaison.

Exemple :

```
#include <stdio.h>

void echange(int *a, int *b)
{
  int tmp;

  tmp = *a;
  *a = *b;
  *b = tmp;
}

int main()
{
  void (*pointeur_fonction)(int *,int *);
  int a,b;

  a = 10;
  b = 20;
  pointeur_fonction = echange;
  pointeur_fonction(&a,&b);
  printf("%d %d\n",a,b);
  return 0;
}
```

Questions :

- réccupérez les fichiers essai_fap.c, fap.c et fap.h
- écrivez un `Makefile` permettant de compiler `essai_fap.c` en utilisant tous les fichiers sources
- modifiez le code de `fap.h` et `fap.c` de manière à pouvoir modifier l'ordre des priorités depuis `essai_fap.c`. Pour cela il faudra :
  - ajouter dans le type fap (dans `fap.h`) un pointeur sur une fonction de comparaison à utiliser pour comparer les priorités.
  - modifier le code de `fap.c` pour utiliser cette fonction au lieu des opérateurs de comparaison classiques.
  - modifier `creer_fap_vide` pour lui passer en paramètre (dans l'appel fait dans `essai_fap.c`) un pointeur sur la fonction de comparaison à utiliser pour cette fap.
  - utiliser tout ceci dans `essai_fap.c` en y écrivant une fonction de comparaison à passer à `creer_fap_vide` pour décider de l'ordre des priorités de la fap utilisée (essayez plusieurs ordres en changeant la fonction).
- essayez votre programme avec plusieurs fonctions différentes :
  - ordre croissant
  - ordre décroissant
  - fonction qui retourne 1
  - fonction qui retourne 0
  - fonction qui retourne 0 ou 1 de manière aléatoire

  reconnaissez vous certaines structures de données vues en algorithmique ?
- modifiez le programme principal pour gérer plusieurs types de fap en même temps : par exemple vous pouvez insérer les éléments dans deux faps ayant deux fonctions de tri différentes et afficher les éléments sortis de chacun des fap lors de l'extraction.
- consultez le manuel de `qsort` qui est un bon exemple de fonction paramétrée par une fonction de comparaison.

2 - Fonctions à nombre variable d'arguments

Il est possible en C de déclarer des fonctions dont le nombre d'arguments n'est pas fixé à l'avance (à l'image des fonctions printf et scanf). Une telle fonction est déclarée avec une première partie de ses arguments fixe (qui doit être non vide, nous comprendrons ensuite pourquoi), suivie de trois points (...). On récupère alors les arguments à l'aide de macros de la bibliothèque standard et contenues dans `stdarg.h` :

- `void va_start(va_list ap, last);`
  doit être appelée en premier, avant de reccupérer tout argument. Les paramètres sont une variable de type va_list (argument pointer) et le nom du dernier argument de la partie fixe.
- `type va_arg(va_list ap, type);`
  doit être appelée pour reccupérer chacun des arguments. les paramètres sont l'argument pointer initialisé avec va_start et le type de l'argument.
- `void va_end(va_list ap);`
  doit être appelée à la fin, une fois tous les arguments reccupéres.

Note : lors de l'utilisation de ces macros, c'est au programmeur de déterminer combien il y a d'arguments et quel est leur type (le compilateur ne fournit aucune aide pour cela). C'est aussi à ceci que sert la première partie fixe dans les arguments.

Exemple : une fonction qui affiche les entiers passés en paramètres (attention au piège).

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

void essai(int nombre, ...)
{
  int i;
  va_list ap;

  va_start(ap,nombre);
  for (i=0;i<nombre;i++)
      printf("%d\n", va_arg(ap,int));
  va_end(ap);
}

int main()
{
  essai(4,5,6,7,9);
  return 0;
}
```

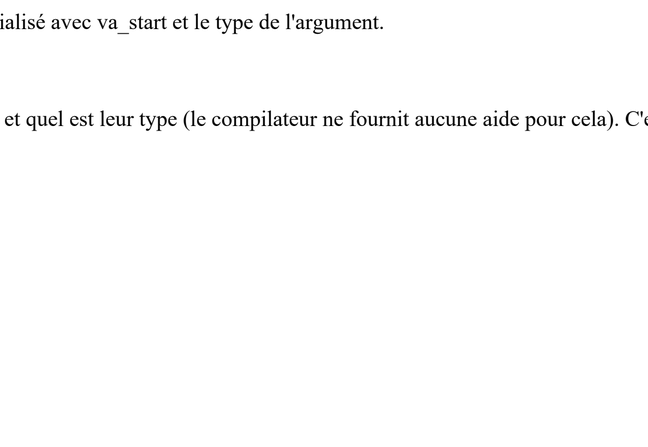
Question :

- écrire une fonction qui renvoie le minimum des nombres flottants qui lui sont passés en arguments.

3 - Remarques sur l'organisation des tableaux multidimensionels en mémoire

Rappel : En C, l'allocation dynamique d'un tableau multidimensionnel se fait en deux étapes.

- allocation d'un tableau de pointeurs dont la taille correspond au nombre de lignes souhaité
- allocation de chacune des lignes en utilisant le tableau de pointeurs précédent pour stocker leur adresse



Structure d'un tableau bidimensionnel dynamique

ce schéma correspond au code suivant :

```
  int **t;
  int i,j;

  t = (int **) malloc(sizeof(int *)*5);
  for (i=0;i<5;i++)
      t[i] = malloc(sizeof(int)*3);
  for (i=0;i<5;i++)
      for (j=0;j<3;j++)
          t[i][j] = j;
  for (i=0;i<5;i++)
      for (j=0;j<3;j++)
          printf("%d ",t[i][j]);

  /* Affichage : 0 1 2 0 1 2 0 1 2 0 1 2 0 1 2 */
```

L'accès au tableau se fait comme d'habitude en C, en utilisant l'opérateur [] pour préciser l'indice dans chacune des dimensions. La seule difference avec un tableau alloué de manière statique est le type des objets manipulés : dans le cas dynamique ce sera un tableau de pointeurs, dans le cas statique ce sera un tableau de tableaux (les tableaux sont réellement stockés les uns après les autres en mémoire).

Une méthode pour allouer un tableau multidimensionel dynamique de manière contigue en mémoire, c'est-à-dire de la même manière que le compilateur pour les tableaux déclarés statiquement, consiste à allouer la bonne taille avec un malloc et à se débrouiller pour ranger ça dans un pointeur sur tableau (de taille fixe). Le tableau ainsi alloué est alors effectivement stocké de manière contigüe, sans pointeurs intermédiaires, comme les matrices linéaires du TP sur la mémoire. La différence est qu'il s'utilise comme tout autre tableau en C, inutile d'écrire des fonctions spéciales d'accès chargées de traduire des indices en deux dimensions en indice en une dimension. L'exemple suivant illustre cette méthode pour un tableau de 5 lignes et 3 colonnes :

```
  /* notons le parenthesage qui permet de designer un pointeur sur tableau
     et non un tableau de pointeurs */
  int (*pa)[3];
  int *pb;
  int i,j;

  pa = (int (*)[3]) malloc(sizeof(int [3])*5);
  pb = (int *) pa;
  for (i=0;i<5;i++)
      for (j=0;j<3;j++)
          pa[i][j] = j;
  /* en itérant en mémoire avec pb, on voit bien que le stockage est contigu */
  for (i=0;i<15;i++)
      printf("%d ",pb[i]);

  /* Affichage : 0 1 2 0 1 2 0 1 2 0 1 2 0 1 2 */
```

Questions :

- d'après vous, pourquoi utilise-t-on des pointeurs intermédiaires pour l'allocation dynamique de tableaux multidimensionnels plutôt que la méthode précédente.