Nom :	
Prénom :	
No. groupe :	
No. carte :	

Programmation et structures de données en C– 2I001

TME solo du 28 novembre 2017
1h30
Aucun document n'est autorisé. Le memento qui a été distribué est reproduit à la fin de cet énoncé.

Les calculatrices, baladeurs et autres appareils électroniques sont interdits. Les téléphones mobiles doivent être éteints et rangés dans les sacs.

Le barème sur 22 points (11 questions) n'a qu'une valeur indicative.

Le mémento qui vous a été distribué est reproduit à la fin de l'énoncé.

TME solitaire numéro : `TS28-11_5-6`.

Vous devez récupérer l'archive suivante :

`/Infos/lmd/2017/licence/ue/2I001-2017oct/fournis/TS28-11_5-6/fournis.zip`

Pour récupérer les fichiers vous devrez décompresser le fichier .zip de la façon suivante :

```
unzip fournis.zip
```

Il vous sera demandé un mot de passe, tapez : `6m4TRE`

Vous respecterez strictement les prototypes et l'organisation des fichiers et prendrez garde à libérer la mémoire que vous aurez allouée.

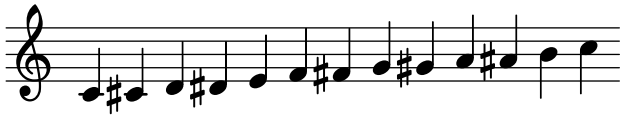
Attention, la soumission dun code qui ne compile pas donnera lieu a une forte pénalité.

Vous n'avez le droit qu'à un terminal, un éditeur de texte et, pour récupérer les fichiers fournis, un navigateur de fichiers. Le navigateur de fichiers sera fermé dès que les fichiers fournis auront été récupérés. Toute autre application est interdite. Vous soumettrez votre répertoire avec la commande remcptcave habituelle. Compte tenu du nom du TME, ce sera donc de la façon suivante :

```
remcptcave 0 TS28-11_5-6 <nom du repertoire>
```

Introduction

Le *dodécaphonisme* est une technique de composition musicale inventée en 1923 par Arnold Schönberg. La gamme musicale est composée de notes, qui sont séparées d'un intervalle d'un demi-ton. Il y a en tout 12 demi-tons et 12 notes différentes :



Dans cette technique de composition, une suite de notes, appelée *série*, peut être exploitée de différentes façons :

- forme droite : la série originelle
- forme rétrograde : la série est prise par la fin
- forme renversée : les intervalles sont renversés, c'est-à-dire qu'on effectue une symétrie par rapport à une hauteur donnée, vue comme un axe.
- forme rétrograde renversée : à la fois rétrograde et renversée

Le but du TME est de manipuler des morceaux, considérés comme des *séries*, en les convertissant dans ces différentes formes.

Le sujet ne nécessite aucune connaissance préalable en musique.

Exercice 1 : Note

Question 1 (2 points)

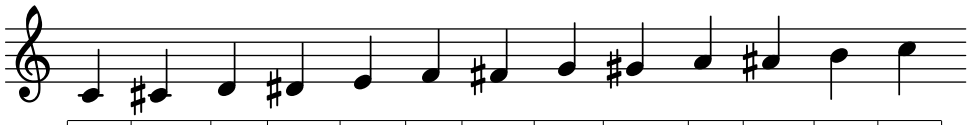
Une note de musique est définie par sa hauteur (une parmi les 12 valeurs possibles) et par sa durée. Le fichier `note.h` contient la définition de la structure représentant une note :

```
typedef struct _note {
int hauteur;/* En demi-tons*/
int duree; /* durée en fonction de la durée de base */
} Note;
```

Vous trouverez dans un fichier déjà fourni `note.c` la fonction déjà écrite

```
void afficherNote(const Note *n);
```

Cette fonction, à partir d'une hauteur de note comprise entre 0 et `NB_DEMI_TONS` −1, affiche le nom de la note en français accompagné d'un éventuel *dièse*, le symbole #, suivi de la durée.



do	do#	ré	ré#	mi	fa	fa#	sol	sol#	la	la#	si	do
0	1	2	3	4	5	6	7	8	9	10	11	0

TABLE 1 – Table de correspondance entre noms de note et numéros de note.

Le nom des notes est stocké dans un tableau `NomsNotes` de taille `NB_DEMI_TONS`, accessible via les fichiers `notes.h` et `notes.c`, comme indiqué dans la Table 1.

Ecrivez la fonction

```
void renversement(const Note *note1, Note *note2);
```

qui, étant données deux notes, calcule l'intervalle de hauteurs entre les deux notes, et modifie la deuxième note pour que sa hauteur change selon l'intervalle opposé :

$$\text{intervalle} = \text{hauteur}_2 - \text{hauteur}_1$$
$$\text{hauteur}_2 = \text{hauteur}_1 - \text{intervalle}$$

Le renversement est aussi appelé *miroir* par rapport à la première note : cela revient à faire une symétrie axiale par rapport à l'axe représentée par la première note sur la portée.

Il faut veiller à ce que la note résultante reste comprise entre 0 et `NB_DEMI_TONS` -1 : on prendra le reste de la division euclidienne par `NB_DEMI_TONS`.

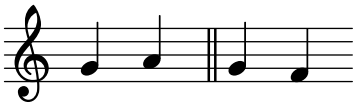


FIGURE 1 – Les notes 7 et 9 forment un intervalle de $9 - 7 = 2$. Le renversement est donc la suite de notes $7, 5$ car $5 = 7 - 2$.

Question 2 (2 points)

Vous écrirez une fonction `main` dans un fichier nommé `main_note.c` que vous créerez pour tester vos fonctions et écrirez la ou les commandes que vous avez utilisées pour compiler votre programme dans un fichier nommé `TMESOLO.txt`.

Exercice 2 : lecture d'un morceau

On considère désormais des morceaux : des suites de notes. Les notes dans la suite de notes sont rassemblées par paquets contigus de même taille appelés *mesure*.

On veut charger un morceau à partir d'un fichier.

Un morceau de musique est décrit de la façon suivante dans un fichier :

```
4
0
1
0
1
2
1
...
```

La première ligne correspond au nombre de notes par mesure, en l'occurrence, 4. Ensuite, on représente une note sur deux lignes : la première ligne correspond à la hauteur, et la seconde à la durée. Dans notre exemple, les trois premières notes sont donc `do1 do1 ré1`.

On ne connait pas à l'avance le nombre de notes dans le fichier. On va donc utiliser une liste pour stocker les notes dans un premier temps.

Le fichier `musique.h` contient la définition de la structure définissant une structure de liste classique :

```
typedef struct _listeNotes {
    Note note;
    struct _listeNotes *suivant;
} ListeNotes;
```

Question 3 (4 points)
Dans le fichier `musique.c`, écrivez la fonction

```
ListeNotes* lireListeMorceau(char* nomFichier, int* tempsParMesure,
   int *nbNotes);
```

qui lit dans le fichier de nom `nomFichier` la suite de notes, et retourne la liste dans laquelle les notes sont stockées. Elle stockera aussi dans `tempsParMesure` le nombre de temps par mesure, et dans `nbNotes` le nombre de notes de la liste.

Question 4 (2 points)
FACULTATIF Modifiez la fonction précédente, pour qu'elle considère les lignes sans nombre comme des lignes de commentaire (qui sont donc ignorées).

Question 5 (1 point)
Ajoutez la fonction

```
void detruireListeNotes(ListeNotes *listeNotes);
```

qui détruit la liste de notes.

Question 6 (1 point)
Comme on connaît maintenant leur taille, on désire maintenant convertir la liste de notes vers un tableau de notes pour manipuler plus facilement les morceaux. Ajoutez la fonction

```
Note* listeVersTableauMorceau(ListeNotes* morceau, int nbNotes);
```

qui convertit la liste en tableau de notes.

Question 7 (2 points)
Vous ajouterez la fonction

```
void afficherMorceau(Note* notes, int nbNotes, int tempsParMesure);
```

qui affiche le morceau sur la sortie standard. Tous les `tempsParMesure` temps, affichez une barre de mesure `|`. Par exemple, la suite de notes (0,1), (0,1), (0,1), (2,1) (4, 2) avec 4 temps par mesure est représentée par :

```
do1 do1 do1 ré1 | mi2
```

Question 8 (2 points)
Vous écrirez une fonction `main` dans un fichier nommé `main.c` que vous créerez pour tester vos fonctions et écrirez la ou les commandes que vous avez utilisez pour compiler votre programme dans le fichier nommé `TMESOLO.txt` (en les distinguant bien des commandes utilisées pour le fichier `main_note.c`).
Vous pourrez tester votre lecture et affichage sur le fichier de morceau fourni `auclairdelalune.morc`. Le début de la sortie devrait ressembler à l'exemple de la question précédente.

Exercice 3 : transformations dodécaphoniques

Les transformations ne changent que la hauteur des notes, pas leur durée.

Pour illustrer, on considère l'exemple de la Figure 2.

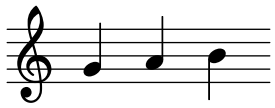


FIGURE 2 – Forme droite : `(7,1) (9, 1) (11, 1)`

Question 9 (2 points)

La première transformation envisagée d'une suite de notes est la transformation vers la forme *rétrograde*. La première note devient la dernière, la seconde, l'avant-dernière et ainsi de suite.

Écrivez dans `musique.c` la fonction

```
void retrogradeMorceau(Note* notes, int nbNotes);
```

L'exemple devient ce qui est montré en Figure 3.

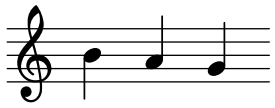


FIGURE 3 – Forme rétrograde à partir de la Figure 2 : `(11,1) (9, 1) (7, 1)`

Question 10 (2 points)

La seconde transformation à écrire est la transformation vers la forme *renversée*. On utilise pour cela la fonction *renversement* écrite pour deux notes, et on l'applique successivement aux couples consécutifs de deux notes.

Écrivez dans `musique.c` la fonction

```
void renversementMorceau(Note* notes, int nbNotes);
```

Cela donne pour note exemple les notes de la Figure 4.

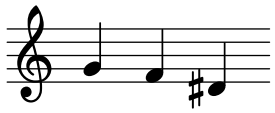


FIGURE 4 – Forme renversée à partir de la Figure 2 : `(7,1) (5, 1) (3, 1)`

Question 11 (2 points)

Complétez votre fonction `main` du fichier `main.c` pour tester les transformations. On testera aussi la transformation *rétrograde inverse* en considérant que cela revient à faire deux transformations à la suite, d'abord *rétrograde*, ensuite *renversée*.

Remarquez que les transformations ci-dessus modifient le tableau de notes. Si on veut modifier plusieurs fois le même morceau, il faudra donc le copier, avec la fonction suivante présente dans `musique.c`.

```
Note* copierMorceau(Note* notes, int nbNotes);
```

qui copie un morceau dans un nouveau morceau, alloué dynamiquement.

Vous écrirez aussi un `Makefile` permettant de compiler séparément vos fichiers et de créer le programme nommé `dodecaphonisme`.

Mémento de l'UE 2I001

Ce document a pour vocation de présenter un bref descriptif des fonctions offertes par les bibliothèques standards et qui sont susceptibles d'être utilisées dans l'UE.

Entrées - sorties

Prototypes disponibles dans `stdio.h`.

Entrées, sorties formatées

```
int printf(const char *format, ...);
```

La fonction `printf` écrit sur la sortie standard (par défaut le terminal), un message correspondant au texte spécifié dans la chaîne `format`. Le texte affiché peut contenir des portions *variables* spécifiées par des codes de conversion précédés par le caractère `%`. Les arguments suivant `format` doivent correspondre (ordre et type) aux codes de conversion présents. Voici quelques codes de conversion courants :

- `%d` : entier
- `%c` : caractère
- `%s` : chaîne de caractères
- `%f` : nombre réel

`printf` renvoie le nombre de caractères imprimés et la chaîne de format peut contenir des codes de contrôle permettant le formatage comme `\n` pour forcer le passage à la ligne et `\t` pour insérer une tabulation.

```
int scanf (const char *format, ...);
int sscanf(const char *entree, const char *format, ...);
```

Les fonctions `scanf` et `sscanf` permettent de saisir et analyser un texte saisi sur l'entrée standard, par défaut le clavier (`scanf`) ou depuis une chaîne de caractères passée en argument (`sscanf`). Le texte devra respecter le `format` spécifié et les arguments suivants doivent correspondre à des pointeurs sur des variables de type appropriés. Les codes de conversion son identiques à ceux de `printf`.

Entrées, sorties caractères

```
int getchar(void);
```

Lit un caractère dans le flux d'entrée standard (par défaut le clavier). La fonction retourne le code **EOF** en cas d'erreur, le caractère lu dans le cas contraire.

```
int putchar(int c);
```

Affiche un caractère dans le flux de sortie standard (par défaut le terminal). La fonction retourne le code **EOF** en cas d'erreur, le caractère lu dans le cas contraire.

```
int puts(const char *s);
```

Affiche une chaîne de caractères dans le terminal et passe à la ligne, renvoie **EOF** en cas d'erreur.

Fichiers

Prototypes disponibles dans `stdio.h`.

```
FILE *fopen(const char *path, const char *mode);
```

Ouvre un fichier dont le chemin est spécifié par la chaîne `path` et retourne un pointeur de type `FILE *` (`NULL` en cas d'échec). L'argument `mode` permet de spécifier le type d'accès à réaliser sur le fichier :

- `[r]` pour un accès en lecture,
- `[w]` pour un accès en écriture et le contenu précédent du fichier est écrasé,
- `[a]` pour un accès en écriture, le contenu du fichier est préservé et les écritures sont effectuées à la suite des contenus déjà présents.

En cas d'erreur la fonction retourne le code **NULL** sinon un pointeur vers le fichier ouvert.

```
int fclose(FILE *fp);
```

Cette fonction provoque la fermeture du fichier pointé par `fp`. En cas d'erreur la fonction retourne le code **EOF** sinon `0`.

```
int fprintf(FILE *stream, const char *format, ...);
```

Identique à `printf` mais l'argument `stream` permet de spécifier le flux de sortie.

```
int fscanf(FILE *stream, const char *format, ...);
```

Identique à `scanf` mais l'argument `stream` permet de spécifier le flux d'entrée.

```
int fputc(int c, FILE *stream);
```

Identique à `putchar` mais l'argument `stream` permet de spécifier le flux de sortie.

```
int fputs(const char *s, FILE *stream);
```

Identique à `puts` mais l'argument `stream` permet de spécifier le flux de sortie et il n'y a pas d'ajout de passage à la ligne.

```
int getc(FILE *stream);
```

Identique à `getchar` mais l'argument `stream` permet de spécifier le flux d'entrée.

```
char *fgets(char *s, int size, FILE *stream);
```

Lit au plus `size` octets dans le flux `stream`. La lecture s'arrête dès qu'un passage à la ligne est rencontré. Les octets lus sont stockés dans `s`. La fonction retourne `s` en cas de succès et `NULL` en cas d'erreur.

```
size_t fread(void *ptr, size_t size, size_t nmemb, FILE *stream);
```

Lecture binaire de `nmemb` éléments de `size` octets dans le fichier `stream`. Les données lues sont stockées en mémoire à partir de l'adresse `ptr`. La fonction retourne le nombre d'éléments effectivement lus.

```
size_t fwrite(const void *ptr, size_t size, size_t nmemb,
              FILE *stream);
```

Écriture de `nmemb` éléments de `size` octets dans le fichier `stream`. Les données à écrire sont lues en mémoire à partir de l'adresse `ptr`. La fonction retourne le nombre d'éléments effectivement écrits.

Chaînes de caractères

Prototypes disponibles dans `string.h`.

Une chaîne de caractères correspond à un tableau de caractère et doit contenir un marqueur de fin `\0`.

```
size_t strlen (const char *s);
```

Renvoie la longueur d'une chaîne de caractères (marqueur de fin `\0` non compris).

```
int strcmp (const char *s1, const char *s2);
int strncmp(const char *s1, const char *s2, size_t n);
```

Comparaison entre chaînes de caractères éventuellement limité aux `n` premiers caractères. La valeur retournée est :

- 0 si les deux chaînes sont identiques,
- négative si `s1` précède `s2` dans l'ordre lexicographique (généralisation de l'ordre alphabétique),
- positive sinon.

```
char *strcpy (char *dest, const char *src);
char *strncpy(char *dest, const char *src, size_t n);
```

Copie le contenu de la chaîne `src` dans la chaîne `dest` (marqueur de fin `\0` compris). La chaîne `dest` doit avoir précédemment été allouée. La copie peut être limitée à `n` caractères et la valeur retournée correspond au pointeur de destination `dest`.

```
void *memcpy(void *dest, const void *src, size_t n);
```

Copie `n` octets à partir de l'adresse contenue dans le pointeur `src` vers l'adresse stockée dans `dest`. `dest` doit pointer vers une zone mémoire préalablement allouée et de taille suffisante. `memcpy` renvoie la valeur de `dest`.

```
size_t strlen(const char *s);
```

Retourne le nombre de caractères de la chaîne `s` (marqueur de fin `\0` non compris).

```
char *strdup(const char *s);
```

Cette fonction permet de dupliquer une chaîne de caractères, elle retourne un pointeur vers la chaîne nouvellement allouée. La nouvelle chaîne pourra être libérée avec la fonction `free`.

```
char *strcat(char *dest, const char *src);
char *strncat(char *dest, const char *src, size_t n);
```

Ajoute la chaîne `src` à la suite de la chaîne `dst`. La chaîne `dest` devra avoir été allouée et être de taille suffisante. La fonction retourne `dest`.

```
char *strstr(const char *haystack, const char *needle);
```

La fonction renvoie un pointeur sur la première occurrence de la sous-chaîne `needle` rencontrée dans la chaîne `haystack`. Si la chaîne recherchée n'est pas présente, la fonction retourne `NULL`.

Conversion de chaînes de caractères

Prototypes disponibles dans `stdlib.h`.

```
int atoi(const char *nptr);
```

La fonction convertit le début de la chaîne pointée par `nptr` en un entier de type `int`.

```
double atof(const char *nptr);
```

Cette fonction convertit le début de la chaîne pointée par `nptr` en un `double`.

```
long int strtol(const char *nptr, char **endptr, int base);
```

Convertit le début de la chaîne `nptr` en un entier long. l'interprétation tient compte de la `base` et la variable pointée par `endptr` est affectée avec l'adresse du premier caractère invalide (au sens de la conversion).

Allocation dynamique de mémoire

Prototypes disponibles dans `stdlib.h`.

```
void *malloc(size_t size);
```

Alloue `size` octets de mémoire et retourne un pointeur générique correspondant à l'adresse du premier octet de la zone, renvoie `NULL` en cas d'échec.

```
void *realloc(void *ptr, size_t size);
```

Permet de modifier la taille d'une zone de mémoire allouée dynamiquement. `ptr` doit correspondre à l'adresse du premier octet de la zone précédemment allouée par `malloc` ou `realloc`. `size` correspond à la taille en octet de la nouvelle zone allouée. `realloc` garantie que la nouvelle zone contiendra les données présentes dans la zone initiale.

```
void free(void *ptr);
```

Libère une zone mémoire allouée dynamiquement. `ptr` doit correspondre à l'adresse du premier octet de la zone précédemment allouée par `malloc` ou `realloc`.