Projet: Éditeur de musique

Manipulation de fichiers WAV

MP2I Lycée Pierre de Fermat

Baissez le volume de votre ordinateur lorsque vous faites ce DM.

Le but de ce projet est d'implémenter en C un logiciel permettant de manipuler des fichiers WAV, qui permettent de représenter des sons, et de créer de la musique à partir d'un fichier texte. Vous trouverez sur Cahier de Prépa une archive contenant des fichiers pour ce projet.

Modalités et notation

Bien que ce DM prenne la forme d'une suite de questions guidées, votre rendu ne sera pas noté question par question mais plutôt évalué dans sa globalité : qualité du code, des commentaires et des tests, etc... Une grille de correction est en ligne sur Cahier de Prépa, ça ne sera peut être pas exactement celle utilisée mais ça vous donnera une bonne idée de ce qui est attendu.
Certaines questions sont marquées d'un ✍ : elles demandent une réponse écrite, qui devra être notée dans un rapport au format Word, PDF, Markdown, ou autre.
Explicitement, vous rendrez :

- une archive contenant le code source, les tests, ainsi qu'un document README.txt expliquant **comment compiler et exécuter** vos programmes ;
- un court rapport au format PDF / Word / Markdown résumant les étapes principales de votre projet, et répondant aux différentes questions ✍ du DM. En particulier, la dernière partie du DM vous demandera d'implémenter une fonctionnalité additionnelle et de la décrire aussi précisément que possible dans ce rapport.

Le DM est individuel, mais vous êtes encouragé/e/s à travailler à plusieurs, du moment que vous ne rendez que du code que vous avez écrit vous-même.

Échéances

La date finale de rendu du projet est le **samedi 4 janvier 2025 à 21h00**. La progression minimale conseillée est la suivante :

- Avant le 30 novembre : jusqu'à la question 9 incluse.
- Avant le 10 décembre : jusqu'à la question 15 incluse.
- Avant le 20 décembre : projet de base fini (jusqu'à la question 20 incluse).

Ceci vous laisse les vacances pour la dernière partie.

Machines virtuelles

Il n'y a pas de son sur les machines virtuelles. Lorsque vous créerez des fichiers audio, il faudra donc les transférer sur votre machine physique pour les lire. On rappelle qu'il est possible de copier coller les fichiers de la machine virtuelle vers Windows.

*
* *

Ondes sonores

Le son est une vibration de l'air, qui se propage sous forme d'onde. Lors d'une telle vibration, l'air est compressé et détendu, localement. En particulier, lorsqu'un son atteint vos oreilles, la vibration de l'air fait également vibrer les tympans : c'est ce qui vous permet d'entendre.

On ne rentrera pas dans les détails physiques du son, on se contentera de le modéliser.

Définition 1. Un ***son*** de durée $T \in \mathbb{R}^+$ est modélisé par une fonction :

$$U : [0, T] -> \mathbb{R}$$

Cette fonction, à valeurs adimensionnées, décrit, en un point fixé, la variation de pression :

- Si $U(t) << 0$, alors à l'instant t, au point fixé, la pression de l'air est faible.
- Si $U(t) >> 0$, alors à l'instant t, au point fixé, la pression de l'air est forte.

Dans la suite, on identifiera un son et la fonction qui le modélise, on dira donc "le son U" pour dire "le son modélisé par U".

Définition 2. Le ***son sinusoïdal*** de fréquence $f \in \mathbb{R}^{+*}$, de durée $T \in \mathbb{R}^+$, et d'amplitude $U_0 \in \mathbb{R}^+$ est le son modélisé par :

$$\forall t \in [0; T], U(t) = U_0 \sin(2\pi f t)$$

Les ondes sinusoïdales sont des briques de base de construction du son. Pour des fréquences allant d'environ $20Hz$ à $20000Hz$, ces ondes sont audibles par les humains.

Pour $f = 440Hz$, on obtient le "La 440", une note de référence en musique. L'archive du projet contient un fichier `la440.wav` avec quelques secondes de cette note.

Encodage du son

Pour encoder un son, on échantillonne ses valeurs à divers instants discrets.
On fixe f_{ech} une ***fréquence d'échantillonnage***, et on note $\tau_{\text{ech}} = \frac{1}{f_{\text{ech}}}$ le temps d'échantillonnage.
On encode un son U de durée T par les $\lfloor \frac{T}{\tau_{\text{ech}}} \rfloor + 1$ valeurs suivantes :

$$[U(0), U(\tau_{\text{ech}}), U(2\tau_{\text{ech}}), \ldots, U(\lfloor \frac{T}{\tau_{\text{ech}}} \rfloor \tau_{\text{ech}})$$

Q1. ✍ Soient $A, f, f_{\text{ech}} \in \mathbb{R}^+$. Pour $i \in \mathbb{N}$, donner la valeur de l'échantillon numéro i du son sinusoïdal de fréquence f, d'amplitude A.

De plus, on encode chaque valeur comme un entier signé sur L bits. Ainsi, les sons que l'on encodera prendront des valeurs comprises entre -2^{L-1} et $2^{L-1} - 1$.

Plus f_{ech} et L sont grands, et plus l'encodage est fidèle au son réel. En pratique, L est toujours un multiple de 8 (pour pouvoir utiliser des octets), et on utilise souvent les valeurs suivantes :

$$\begin{aligned} f_{\text{ech}} &= 44100Hz \\ L &= 16 \end{aligned}$$

Le choix de f_{ech} vient du **théorème de Nyquist–Shannon**, qui dit que pour échantillonner un son de fréquence f, il faut une fréquence d'échantillonnage supérieure à $2f$: prendre $f_{\text{ech}} = 44100$ permet ainsi d'échantillonner des fréquences allant jusqu'à 20000 Hertz, la limite de l'oreille humaine.

Fichiers WAV

Les fichiers WAV sont les fichiers les plus simples d'encodage du son, car ils utilisent précisément la méthode décrite dans la section précédente. Un fichier WAV est constitué de 44 octets d'en-tête, suivi d'octets de données. Les données sont N valeurs échantillonées à une fréquence f_{ech}, chacune écrite sur $\frac{L}{8}$ octets. La spécification précise des fichiers WAV est trouvable en ligne, on en donne ici une version légèrement simplifiée qui suffira pour le DM. En particulier nos fichiers ne seront pas en stéréo.

En-tête On considère un fichier WAV de fréquence d'échantillonage f_{ech}, de L bits, avec N valeurs échantillonées. La constitution de l'en-tête de ce fichier WAV est la suivante :

octets	type	description
0-3	caractères	Les caractères 'R', 'I', 'F', 'F', chacun prenant 1 octet
4-7	entier	Le nombre d'octets dans le fichier après ce champ, i.e. $36 + \frac{L}{8}N$, écrit sur 4 octets
8-15	caractères	Les caractères 'W', 'A', 'V', 'E', 'f', 'm', 't', ' ' (le dernier est un espace).
16-19	entier	L'entier 16, écrit sur 4 octets
20-21	entier	L'entier 1 écrit sur 2 octets
22-23	entier	L'entier 1 écrit sur 2 octets
24-27	entier	La fréquence d'échantillonnage f_{ech}, écrite sur 4 octets
28-31	entier	La quantité d'octets nécessaire par seconde de son, i.e. $\frac{f_{\text{ech}}L}{8}$, écrit sur 4 octets
32-33	entier	Le nombre d'octets pris par une valeur échantillonnée, i.e. $\frac{L}{8}$, écrit sur 2 octets
34-35	entier	L écrit sur 2 octets
36-39	caractères	Les caractères 'd', 'a', 't', 'a'.
40-43	entier	Le nombre d'octets suivant ce champ, i.e. $\frac{L}{8}N$, écrit sur 4 octets.

Attention, les entiers sont stockés en ***petit-boutiste***, ce qui signifie qu'ils sont stockés par ordre décroissant d'importance des octets. Par exemple, si l'on souhaite écrire sur 4 octets le nombre 1981, qui s'écrit 0x000007BD en hexadécimal, on stockera les octets suivants dans l'ordre : 0xBD, 0x07, 0x00, 0x00.

Dans l'archive du projet, vous trouverez un fichier "court.wav". Ce fichier contient $N = 3$ valeurs échantillonées, à une fréquence $f_{\text{ech}} = 44100Hz$, sur $L = 16$ bits. Les trois valeurs sont :

$$15387, 815, -6337$$

Dans un terminal, lancez la commande ***od -t x1 -A x court.wav***, pour afficher le contenu du fichier en hexadécimal.

Q2. Vérifiez que les octets affichés par la commande correspondent à la spécification. Vous trouverez dans l'archive du projet la table de correspondance ASCII entre les caractères et leurs codes hexadécimaux.

Q3. ✍ Donnez tous les octets d'un fichier WAV contenant 5 valeurs, avec $f_{\text{ech}} = 22050Hz$, $L = 16$, les 5 valeurs étant $978, 1630, -1, -1630, 32767$.
Vous donnerez votre réponse en écrivant 16 octets par ligne, comme la sortie de la commande `od`.

Manipulation de fichiers WAV en C

On suppose dans toute la suite du sujet que $L = 16$ et $f_{\text{ech}} = 44100$. En particulier, les valeurs échantillonnées sont entre -32768 et $+32767$. On rappelle l'existence en C du type `int16_t` dans la librairie `<stdint.h>`, qui est un type entier signé sur 16 bits.

Vous allez maintenant écrire un programme permettant de créer des fichiers WAV, en C. Vous allez devoir utiliser la librairie `<math.h>`, et on rappelle que pour compiler vos programmes, vous devrez rajouter l'option ***-lm*** dans la commande de compilation, pour indiquer au compilateur qu'il faut utiliser la librairie de maths. L'option ***-lm*** doit toujours être à la fin de la commande :

```
gcc bla.c -o nom_exec -lm
```

De plus, il est conseillé de compiler avec tous les warnings et d'activer l'option `-g` pour pouvoir utiliser valgrind. Le programme final utilisant 6 fichiers C différents, la commande de compilation sera très longue. Il est conseillé de la noter dans un fichier texte pour pouvoir la modifier et la copier-coller dans le terminal facilement [1].

Pour écrire des données non-textuelles dans un fichier, on peut utiliser des fonctions spécialisées de la librairie `<stdio.h>`, ou bien le faire à la main avec `fprintf(f, "%c", c)`. En effet, cette commande va écrire dans le fichier f le caractère c, on peut donc l'utiliser pour écrire **octet par octet**. Par exemple, le code suivant montre trois manières d'écrire des octets dans un fichier, et écrit les octets `0x43, 0x00, 0x6b` :

```c
FILE* f = fopen("mon_fichier", "w");

fprintf(f, "%c", 77); // 77 = 0x43
fprintf(f, "%c", 0x00);
fprintf(f, "%c", 'k'); // le code ASCII de k est 107 = 0x6b

fclose(f);
```

Ainsi, pour écrire l'entier 2,705,637,499 (`0xa144c07b` en hexadécimal) dans un fichier en *little-endian*, on peut écrire :

```c
fprintf(f, "%c%c%c%c", 0x7b, 0xc0, 0x44, 0xa1);
```

ou encore :

```c
fprintf(f, "%c", 123);
fprintf(f, "%c", 192);
fprintf(f, "%c", 68);
fprintf(f, "%c", 161);
```

Notons que cela revient essentiellement à faire une écriture en base 256 !

Q4. Écrivez une fonction `void write_int(FILE* f, int a, int size)` permettant d'écrire un entier dans un fichier, octet par octet, en *little-endian*. La fonction prendra en argument un `int` à écrire, et le nombre d'octets à écrire (a priori entre 1 et 4).

1. Vous pouvez aussi vous renseigner sur le concept de **makefile**, mais c'est un peu hors programme...

Nous allons créer un premier couple de fichiers .h/.c pour définir des structures permettant de stocker des sons échantillonnés. On propose la structure suivante pour représenter un son :

```
typedef struct sound{
  int n_samples; // nombre d'échantillons
  int16_t* samples; // tableau des échantillons
} sound_t;
```

Un son est donc une liste de valeurs échantillonnées.

Q5. Créez deux fichiers `sound.h` et `sound.c`, et implémentez dans ces fichiers une fonction permettant de libérer la mémoire allouée pour un élément de type `sound_t`. N'oubliez pas que la documentation va dans le fichier .h, alors que le code va dans le fichier .c. N'oubliez pas non plus d'ajouter un **include guard**.

On souhaite pouvoir écrire de tels sons dans des fichiers WAV, et à l'inverse pouvoir lire un son depuis un fichier WAV et le charger en mémoire dans un `sound_t`. Nous allons donc créer un deuxième couple de fichiers : `wav.h` et `wav.c`.

Q6. Écrivez une fonction qui écrit l'en-tête d'un fichier WAV qui aura n échantillons :

```
void write_header(FILE* f, int n);
```

Q7. Écrivez une fonction de signature

```
void save_sound(char* filename, sound_t* s);
```

qui permet de sauvegarder un son dans un fichier WAV.

Q8. Testez la fonction précédente en créant un `sound_t` à la main, de 3 ou 4 échantillons, et vérifiez avec **od -t x1 -A x nom_fichier** que le fichier généré correspond bien.

Nous allons maintenant écrire quelques fonctions pour générer des sons basiques. Ces fonctions iront dans des nouveaux fichiers : `waveform.h` et `waveform.c`

Q9. **BAISSEZ LE VOLUME DE VOTRE ORDINATEUR** avant de commencer cette question. Écrivez une fonction qui étant donné une durée T et une fréquence d'échantillonnage f_{ech} génère un son de durée T dont tous les échantillons ont une valeur aléatoire :

```
sound_t* white(float duree, int f_ech);
```

Testez cette fonction pour créer un fichier WAV de quelques secondes : vous devriez obtenir ce qu'on appelle un "bruit blanc", qui rappelle la pluie, la mer, le souffle du vent.

Q10. Écrivez une fonction qui, étant donné une amplitude, une durée, une fréquence, et une fréquence d'échantillonnage, crée et renvoie un son sinusoïdal correspondant :

```
sound_t* sine(float freq, int amplitude, float duree, int f_ech);
```

Testez cette fonction pour créer un fichier WAV contenant un son sinusoïdal de fréquence 440Hz de quelques secondes. Le fichier "la440.wav" dans l'archive du TP donne le résultat que vous devriez obtenir.

Q11. Implémentez des fonctions de même signature que la fonction précédente et permettant de générer des signaux en créneau, en triangle, en dent de scie :

```
sound_t* square(float freq, int amplitude, float duree, int f_ech);
sound_t* triangle(float freq, int amplitude, float duree, int f_ech);
sound_t* sawtooth(float freq, int amplitude, float duree, int f_ech);
```

Vérifiez que les fichiers WAV créés sont cohérents, i.e. que pour une fréquence fixée, les différentes formes d'ondes produisent bien la même note, avec un timbre différent. Vous pouvez aussi utiliser Audacity (https ://www.audacityteam.org/), un logiciel open source permettant entre autre de visualiser les signaux sonores.

Nous allons maintenant rajouter la possibilité d'enchaîner plusieurs sons. Pour cela, on propose la structure suivante, qui est simplement un tableau de sons que l'on va juxtaposer.

```
1 typedef struct track {
2   int n_sounds; // nombre de sons dans la piste
3   sound_t** sounds; // liste des sons
4 } track_t;
```

Le principe va être d'utiliser une piste (*track* en anglais) pour stocker de manière temporaire une suite de sons, que l'on viendra concaténer à l'aide d'une fonction. La structure et le code associé iront dans les fichiers `sound.h` / `sound.c`.

Q12. Implémentez une fonction qui libère la mémoire allouée pour une piste.

Q13. Implémentez une fonction `sound_t* reduce_track(track_t* t)` qui transforme une piste en un unique son.

Vous pourrez tester votre fonction avec la liste de sons sinusoïdaux suivants, qui forment le début de "Au clair de la lune" (les sons d'amplitude 0 simulent des silences) :

Fréquence	Durée	Amplitude
440Hz	0.5s	0
440Hz	0.4s	16000
440Hz	0.1s	0
440Hz	0.4s	16000
440Hz	0.1s	0
440Hz	0.5s	16000
493.88Hz	0.5s	16000
554.36Hz	1s	16000
493.88Hz	1s	16000

Taper des fréquences est assez fastidieux et peu intuitif. Nous allons donc passer par un système plus simple, habituel en musique : les notes de la gamme. Pour simplifier le code, nous n'allons pas utiliser les notations musicales comme *Do, Ré, Mi* ou *C, D, E*, mais plutôt des entiers.
Une **hauteur de note** (ou **pitch** en anglais) est un entier $n \in \mathbb{Z}$, et la fréquence correspondant à n est :

$$f(n) = 440 \times 2^{\frac{n}{12}}$$

Par exemple, $f(0)$ vaut 440Hz : la note de hauteur 0 est le "La 440". $f(12)$ vaut 880Hz : c'est aussi un La (en musique, doubler la fréquence donne la même note, une octave plus aigüe).
De plus, plutôt que de parler d'amplitude, entre 0 et $2^{L-1} - 1$, on parlera de volume, entre 0 et 1 (le volume sera donc un réel). Une note est donc caractérisée par son pitch, sa durée, et son volume.

La prochain étape est donc de pouvoir lire dans un fichier une suite de notes et d'en faire une piste. On propose le format suivant pour ce qu'on appellera une **mélodie** :

— Sur une première ligne : le nombre de notes n puis le type de son à utiliser ;

— Sur les n lignes suivantes : la hauteur, la durée et le volume de chaque note.

Par exemple, le fichier suivant encode “Au clair de la lune” avec une onde en dent de scie (les commentaires ne sont pas présents dans le fichier) :

```
9 sawtooth // 9 notes, en dent de scie
0 0.5 0   // son de volume 0 = silence
0 0.4 0.5 // au
0 0.1 0
0 0.4 0.5 // clair
0 0.1 0
0 0.5 0.5 // de
2 0.5 0.5 // la
4 1   0.5 // lu-
2 1   0.5 // -neuh
```

Les fonctions relatives à la lecture de mélodie iront dans deux fichiers `melody.h/.c`

Q14. Écrivez une fonction `pitch_to_freq` qui calcule la fréquence associée à une hauteur de note. Vous pourrez utiliser la fonction `pow` de la librairie `<math.h>`.

Q15. Écrivez une fonction `track_t* read_track(FILE* file)` qui lit dans un fichier une mélodie selon le format décrit précédemment, et crée une piste. L'archive du projet contient un exemple pour vous aider à vérifier que tout marche bien, dans le dossier `question_15/`

Maintenant, nous allons ajouter la possibilité de superposer des sons, et donc de pouvoir jouer plusieurs notes à la fois. Malheureusement, on ne peut pas simplement ajouter les échantillons un à un, car on risque un dépassement d'entier. Une solution envisageable est de faire une moyenne pondérée. Nous allons implémenter cette solution via une structure qui stockera une liste de pistes, chacune pondérée par un volume [2] :

```c
typedef struct mix {
  int n_tracks; // Nombre de pistes
  track_t** tracks; // Liste des pistes
  float* vols; // / Liste des volumes des pistes
} mix_t;
```

Cette structure et les fonctions liées iront également dans les fichiers `sound.h`/`sound.c`.

Q16. Écrire une fonction qui libère la mémoire allouée pour un `mix_t*`.

Q17. Écrire une fonction `sound_t* reduce_mix(mix_t* m)` qui permet de transformer un mix en un son en prenant la moyenne pondérée des valeurs pour chaque échantillon. **Attention**, lors du calcul de la moyenne, si vous faites une somme, vous risquez de dépasser la limite des entiers 16 bits. Il faudra repasser par le type int pour éviter les erreurs.

Q18. ✍ En notant n le nombre de pistes et $l_1, \dots l_n$ les longueurs des pistes, quelle est la complexité de la fonction précédente ?

On étend notre format de fichier pour prendre en compte le fait que l'on peut avoir plusieurs pistes. On écrira donc dans un fichier :

— Sur la première ligne, le nombre n de pistes ;

— Sur la deuxième ligne, n nombres donnant le volume de chaque piste ;

— Enfin, n blocs décrivant les n pistes selon le format précédent.

L'archive du DM contient des exemples de fichiers à ce format, dans le dossier `question_19/`.

2. Attention, la somme des volumes ne fait pas forcément 1 !

Q19. Créer dans les fichiers `melody.h` / `.c` une fonction `mix_t* load_mix(char* filename)` qui va charger depuis un fichier le mix correspondant.

La dernière étape est de rendre le programme utilisable, en lui faisant prendre des arguments à l'exécution.

Q20. Modifiez le main pour que le programme puisse prendre en argument deux noms de fichiers : un pour lire les notes, un pour le fichier WAV créé. Le programme affichera des messages informatifs, par exemple la taille du fichier créé, le temps total pris, la longueur en secondes du fichier WAV généré, etc... Votre programme devrait ressembler à :

```
fredfrigo@ubuntu:~$ ./wav_writer clair_de_lune.txt clair_de_lune.wav
Fichier clair_de_lune.wav généré (temps écoulé: 0.2s)
Durée du fichier: 3m27s (taille 10Mo)
```

Fonctionnalités additionnelles

Pour finir ce DM, vous devez ajouter une fonctionnalité de votre choix au programme. Cette partie ne sera pas notée sur l'ampleur de la fonctionnalité, mais plutôt sur la propreté du code, sur la clarté de vos explications, sur la pertinence des structures / fonctions utilisées, etc...

Avant d'ajouter cette fonctionnalité, **archivez votre programme actuel**, afin d'en avoir toujours une copie propre. **Vous devez rendre les deux versions**.

Vous devez implémenter la fonctionnalité choisie, et l'expliquer dans le rapport : à quoi elle sert, quelles structures et fonctions vous avez introduit ou modifié, etc... Si votre fonctionnalité modifie le format des fichiers textes lus, vous devez le décrire et fournir des fichiers exemples. Enfin, vous devez écrire dans un fichier `README.txt` un guide pour **compiler et exécuter** votre programme correctement.

Vous pouvez faire cette partie en binôme (et cette partie uniquement), auquel cas vous devez indiquer avec qui vous avez travaillé. Vous devez rédiger un rapport par groupe, et les deux membres du groupe rendront le code et le rapport. Répartissez vous les tâches équitablement, n'assignez pas un membre au code et l'autre au dessin de la couverture du rapport !

Quelques propositions d'idées : si vous choisissez autre chose, envoyez moi un mail avec l'idée générale de ce que vous voulez faire pour que je le valide.

- Se renseigner sur le format WAV et implémenter de quoi générer des fichiers en stéréo [3]
- Pouvoir lire dans un fichier WAV pour générer un `sound_t*`, et implémenter une fonctionnalité en lien (mélanger deux sons, renverser un son, utiliser des *samples* de batterie pour faire un *beat*, etc...).
- Pour les physicien/ne/s : une fois que vous aurez vu les filtres en éléctrocinétique, utiliser la méthode d'Euler pour simuler l'effet d'un filtre passe-bas ou passe-haut sur un son pour éliminer les hautes ou basses fréquences.
- Pour les musicien/ne/s : se renseigner sur la notion de timbre, sur la synthèse sonore additive, ou bien la notion d'enveloppe (attack/decay/sustain/release), et implémenter une fonctionnalité en lien pour générer des notes de base plus riches que ceux du projet.
- (**Absurdement difficile**) Se renseigner sur le format MIDI, et permettre à votre programme de lire directement depuis un fichier MIDI.

3. Une page assez bien faite : soundfile.sapp.org/doc/WaveFormat. Une note car ce n'est pas super clair : dans les fichiers stéréo, les échantillons de gauche et de droite sont entrelacés : un fichier WAV alterne échantillon gauche, droite, gauche, droite, etc...