TP C#6 : A l'assaut des Fichiers !

1 Avant de partir ...

1.1 Rappels et Consigne de rendu

Comme pour les autres TPs, nous vous rappelons que votre code doit compiler !
Nous vous considérons grand désormais et nous ne tolérerons plus de code fourni avec ses erreurs. Il est donc conseillé de vous servir des outils de Visual Studio comme la liste d'erreurs ou le débuggeur afin de vérifier que votre code compile et soit fonctionnel.

En ce qui concerne les consignes de rendu, il vous est demandé de fournir une archive de type zip nommée : rendu-tp-login_x.zip. (Si vous vous appelez Xavier Loginard, sinon mettez votre propre login).

```
rendu-tp-login_x.zip
        -login_x/
            - AUTHORS
            - MirrorWriting/
                - MirrorWriting.sln
                - MirrorWriting/         => Sans les dossiers "bin" et "obj"
            - WavReader/
                - WavReader.sln
                - WavReader/             => Sans les dossiers "bin" et "obj"
            - WavMixer/
                - WavMixer.sln
                - WavMixer/              => Sans les dossiers "bin" et "obj"
```

1.2 Notions abordées

Ce que vous allez découvrir dans ce TP gravite autour d'une notion phare dans le monde de la programmation, à savoir *la manipulation de fichier*.
Voici la liste des notions abordées :

- Ecriture et Lecture d'un fichier
- L'Hexadécimal et le binaire
- Un cas particulier de fichier : .wav

Ne prenez pas peur, ces notions ne sont pas très difficiles, elles sont juste nouvelles ! Vous allez voir cela va bien se passer.

2 ... penser à faire le plein de connaissance !

Avant d'aborder le cas des fichiers nous allons faire un petit rappel sur les classes, les structures et le mot clé static.

2.1 Les classes (rappel)

En C# comme en POO, les classes sont en quelque sorte la base de tout programme (lui même composé de hiérarchies de classes reliées les unes aux autres).

Une classe contient :
- des attributs (variables)
- des méthodes (fonctions ou procédures)
- un constructeur (méthode spéciale utilisée pour l'instanciation)
- un destructeur (le yin [1] du constructeur)

Lorsqu'on instancie un objet appartenant à une classe, C# alloue de la mémoire virtuelle pour stocker cet objet, et libère cette mémoire grâce au destructeur.

2.2 Les structures (rappel)

Les structures peuvent paraître très semblables aux classes mais il existe néanmoins quelques différences :
- pas d'héritage de structure
- un constructeur ne peut pas être défini sans paramètres
- les structures sont de types valeurs
- les champs initialisés doivent être `static` ou `const`

2.3 Le mot clé static

Depuis le début, nous vous avons fait utiliser ce mot clé, sans pour autant avoir pris le temps de vous expliquer correctement ce qu'il représente. Le mot clé `static` sert à définir une méthode, un attribut, ou même une classe qui ne peut pas être instancié. En d'autres termes les méthodes ou attributs qui ne sont pas liés à un objet et qui sont accessibles sans aucune instanciation (sans utiliser `new`).

En C# , certaines méthodes n'ont pas besoin d'être instanciées ou n'en ont pas l'utilité, par exemple Console.WriteLine() est une méthode statique (ou méthode de classe) dans le sens où quand vous l'utilisez, vous le faites sans avoir instancié d'objet de type Console.

C'est la même chose pour la plupart des méthodes que vous avez du coder jusqu'à maintenant, vous les appeliez dans la fonction main, sans instanciation. La méthode `Main` est également statique, en effet la classe `Program` n'est jamais vraiment instanciée, elle n'est appelée qu'une fois, tout comme la méthode `Main`, et ceci au lancement du programme.

Pour ce qui est des classes statiques, il s'agit de classes qui ne contiennent que des champs ou des méthodes statiques. La classe `Convert` doit vous être familière, et bien c'est une classe statique ce qui vous permet d'écrire ce genre d'instructions directement : `string quaranteDeux = Convert.ToString(42);`

3 Fichiers ? Vous avez dit Fichiers ?

3.1 Généralités

Nous passons enfin aux choses sérieuses ! Que dire à propos des fichiers ?
Les fichiers représentent avant tout le moyen le plus utilisé en informatique pour stocker des données. En effet si votre ordinateur dispose au mieux de 16 Go de Ram pour stocker des informations, il possède en revanche des disques durs de plusieurs To (1000 Go pour rappel). Les fichiers, vous l'aurez compris, sont donc stockés sur des périphériques spécifiques et nous avons besoin d'interagir avec ces fichiers pour récupérer et utiliser les données qu'ils contiennent.

Mais qu'est ce qu'un fichier ? une suite de données et c'est tout ? Une suite de données oui ! mais pas que. En effet si ce n'était qu'une suite de données, comment différencierions nous un type de fichier d'un autre ? Un fichier est globalement composé de deux parties : *metadata* et *data*.

1. ou le coté obscur c'est vous qui voyez

Les *metadatas* ou *header* regroupent toutes les informations nécessaires à l'identification du fichier, par exemple son type, sa taille, son type de compression ...
Tandis que les *datas* sont à proprement parler nos données, mais nous y reviendrons un peu plus tard dans ce TP.

N'oubliez pas de vous lavez les mains, car nous allons passer à la manipulation de fichiers !

3.2 Manipulation de fichiers en C# (et en douceur !)

En C# , il existe plusieurs approches pour s'amuser avec les fichiers, mais elles gravitent toutes autour de la notion de flux (*Stream* en anglais).

Les flux sont une représentation de données abstraites qui insiste sur le fait que les données peuvent être traitées à la file, avec une notion de consommation des données au fur et à mesure : quand je lis le premier octet d'un flux, si je recommence, j'obtiendrai l'octet suivant ...

Vous devez vous en douter mais il existe une classe en C# qui gére les flux, son nom ? C'est la classe `Stream` voyons ! Mais c'est plutôt sa classe fille qui nous intéresse ici : FileStream.

Voici donc sans plus attendre un exemple d'utilisation de ces classes :

```
Stream stream; // FileStream stream est également possible

try
{
        stream = new FileStream("toto.txt", FileMode.Create, FileAccess.Write);
}
catch (IOException e)
{
        Console.WriteLine("Cannot open the file : " + e.Message);
}
// Jusque ici on ne s'est occupé que de créer un fichier toto.txt
// On va maintenant écrire dedans :

for (int i = 0; i < 26; ++i)
        f.WriteByte((byte)('A' + i));
// toto.txt contient maintenant : ABCDE...
// Je vous vois septique, vérifions donc :

f.Seek(0, SeekOrigin.Begin); // Retourne au début du fichier
Console.WriteLine((char) f.ReadByte()); // Affiche le caractère 'A'
Console.WriteLine((char) f.ReadByte()); // Affiche le caractère 'B'
f.Seek(1, SeekOrigin.Current); // Avance de 1
Console.WriteLine((char)f.ReadByte()); // Affiche le caractère 'D'
f.Close(); // Ne pas oublier de fermer le stream une fois qu'on ne s'en sert plus
```

Deux étapes sont importantes pour la manipulation de fichier :
- L'ouverture : c'est là qu'on précise le mode de lecture (FileMode.Create, FileMode.Open) et les droits d'accès (écriture, lecture, ou les deux)
- La fermeture : quand on ouvre un fichier, on le ferme lorsqu'on n'en a plus besoin ! Vous n'aurez pas toujours la chance de coder avec des langages qui possèdent un garbage collector !

3.3 Les ASCII fichiers

Les "fichiers texte" sont des fichiers simples à lire [2] par opposition aux fichiers binaires qui contiennent uniquement du binaire, regroupé sous forme hexadécimale.
Parmi les plus communs on retrouve les fichiers de configuration ou encore les codes sources (.cs, .ml, .hxx, ...).
Pour le moment nous allons nous intéresser uniquement aux fichiers texte. Attention l'extension d'un fichier ne définit pas forcément son contenu, on peut renommer un .jpg en .txt, le contenu sera toujours le même, lisible par MSPaint, mais pas par Notepad [3].

Pour manipuler les fichiers texte, on utilise les classes StreamReader [4] et StreamWriter [5]. Ces classes ajoutent quelques méthodes bien pratiques, comme respectivement le ReadLine et WriteLine que vous connaissez bien. Les constructeurs de ces classes acceptent beaucoup de choses. Vous pouvez par exemple utiliser le flux d'un fichier déjà ouvert, ou alors ouvrir un nouveau fichier en donnant son nom au constructeur.

Impatient de commencer enfin ce TP ? Passons donc aux exercices !

4 Exercice 1 : MirrorWriting

Dans cet exercice nous allons travailler uniquement avec des "fichiers texte". Commençons tout de suite avec la première méthode. Les méthodes de cet exercice sont à implémenter dans le fichier program.cs de la solution MirrorWriting.

4.1 WriteFile

```
static void WriteFile(string filename, string msg);
```

Comme son nom l'indique cette méthode écrit msg dans le fichier filename.
Implémenter cette méthode et utiliser la pour écrire "Hello world !" dans le fichier toto.txt. N'oubliez pas de tester si aucune erreur n'a été rencontré lors de la création de votre fichier.

4.2 ReadFile

```
static void ReadFile(string filename);
```

Après l'écriture, un peu de lecture nous détendra. Implémenter cette méthode qui va lire le contenu de filename afin de l'afficher dans la console. Si vous testez votre méthode sur votre fichier toto.txt, voici ce que vous pourriez lire sur la console :

```
Reading "toto.txt" :
Hello world !
```

Un fichier étant composé d'une ou plusieurs lignes, votre méthode doit être capable d'afficher toutes les lignes d'un fichier. Testez votre ReadFile avec un fichier test.txt contenant par exemple :

```
Ceci est un test.
Ma methode ReadLine peut
lire
plusieurs lignes !
```

2. La preuve étant que même un humain arrive à le lire ...
3. De même que renommer votre dossier de rendu en rendu-tp-login x.zip ne l'aura pas compressé par magie !
4. http://msdn.microsoft.com/fr-fr/library/system.io.streamreader.aspx
5. http://msdn.microsoft.com/fr-fr/library/system.io.streamwriter.aspx

4.3 MirrorWriting

```
static void MirrorWriting(string input, string output);
```

Vous avez vu précédemment comment faire pour lire et écrire dans un fichier, nous allons maintenant mélanger les deux. La méthode MirrorWriting va lire chaque ligne de input, et écrire cette ligne dans le fichier output. Facile ? Vous avez raison, elle va écrire la ligne dans output mais pas n'importe comment, MirrorWriting le fera en inversant les caractères de la ligne lue. Voici ce que donne cette méthode sur le fichier test.txt (sans la partie droite bien entendu) :

```
.tset nu tse iceC                    |          Ceci est un test.
tuep eniLdaeR edohtem aM             |          Ma methode ReadLine peut
eril                                 |          lire
! sengil srueisulp                   |          plusieurs lignes !
```

4.4 Bonus : ReverseFile

```
static void ReverseFile(string input, string output)
```

Maintenant que vous avez renversé les caractères de chaque ligne, pourquoi ne pas renverser le fichier en entier ? La méthode ReverseFile fonctionne sur le même principe que MirrorWriting sauf qu'elle renverse les lignes en commençant par écrire l'inverse de la dernière dans output et ainsi de suite en remontant jusqu'à la première ligne.

```
! sengil srueisulp
eril
tuep eniLdaeR edohtem aM
.tset nu tse iceC
```

5 Exercice 2 : WavReader

Les méthodes de cet exercice sont à implémenter dans le fichier program.cs de la solution WavReader.

On dit habituellement que la musique adoucit les mœurs ... Nous allons vérifier cela maintenant !

Le but de cet exercice est de lire les *metadatas* d'un fichier .wav. Si vous ne vous souvenez déjà plus ce qu'est une *metadata*, relisez la partie sur les fichiers, ou demandez à vos ACDC. Pour ceux dont la mémoire dépasse celle d'un poisson rouge je vous invite à lire attentivement la partie qui suit. Mais d'abord qu'est ce qu'un fichier .wav ?

5.1 Remous mélodieux

Wav ou Wave (Waveform Audio File Format) est un standard audio pour stocker de l'audio numérique développé par Microsoft et IBM. C'est un dérivé du format RIFF (Resource Interchange File Format) (RIFF) reconnaissable à sa façon de regrouper ses données en blocs de taille fixe.

Chaque .wav est composé d'un header de taille fixe de 44 octets suivi des données audio [6]. Ce header est donc découpé en plusieurs blocs de différentes tailles, dont voici un aperçu :

ChunkID	4 octets	"RIFF"
ChunkSize	4 octets	Taille du fichier moins un octet
Format	4 octets	"WAVE" pour les fichiers .wav
Subchunk1ID	4 octets	"fmt" pour les fichiers .wav
Subchunk1Size	4 octets	Non utilisé dans l'exercice
AudioFormat	2 octets	1 pour les .wav non compressés
NumChannels	2 octets	Nombre de canaux (Mono = 1, Stereo = 2)
ByteRate	4 octets	Non utilisé dans l'exercice
SampleRate	4 octets	Fréquence d'échantillonnage
BlockAlign	2 octets	Non utilisé dans l'exercice
BitsPerSample	2 octets	Non utilisé dans l'exercice
Subchunk2ID	4 octets	"data" pour les fichiers .wav
Subchunk2Size	4 octets	Non utilisé dans l'exercice

Pour la taille du fichier, il suffit juste de d'ajouter 8 à l'entier lu. Le reste du fichier (de l'octet 44 à la fin) correspond aux données audio qui sont regroupées par paquet de deux octets, et ces deux paquets représentent alternativement les données du canal droit et du canal gauche.

24 17	1e f3	3c 13	3c 14	16 f9	18 f9
gauche	droit	gauche	droit	gauche	droit

Faites attention, sur le site web les blocs ByteRate et SampleRate sont inversés, mais ils sont dans le bon ordre sur le Tp.
Autre petite indication, sachez qu'un char est codé sur un octet, c'est pour cela que les blocs ChunkID, Format, Subchunk2ID ont une taille de quatre octets pour contenir quatre chars. Mais qu'en est il de Subchunk1ID qui a une taille de quatre octets mais pour coder a priori trois chars (fmt) ? Je dis a priori car si on regarde la séquence d'octets que contient le bloc on obtient :

66	6d	74	20
f	m	t	

Vous ne voyez toujours pas ?
Cherchez sur internet à quel caractère ASCII correspond le nombre 20 et vous trouverez !

Petit rappel : un octet est codé sur un nombre de 2 chiffres en base hexadécimale, un octet sera donc compris entre 00 et FF en hexadécimal.

A partir de maintenant, votre mission, si toutefois vous l'acceptez, consiste à lire certains blocs d'un header d'un fichier .wav et à en extraire les informations. Vous ne devez lire que certains blocs, les autres seront ignorés. Pour cette mission vous aurez besoin de connaitre de nouvelles classes : `BinaryReader` [7] et `BitConverter` [8].

`BinaryReader` vous permettra de lire facilement des blocs d'octet, `BitConverter` fonctionne à peu près de la même manière que la classe Convert.

Des questions ? Soucis ? Choses pas claires ? Demandez à vos ACDC !

6. https://ccrma.stanford.edu/courses/422/projects/WaveFormat/
7. http://msdn.microsoft.com/fr-fr/library/system.io.binaryreader.aspx
8. http://msdn.microsoft.com/fr-fr/library/system.bitconverter(v=vs.95).aspx

5.2 CheckWav

```
static bool CheckWav(string filename);
```

Avant même de vouloir extraire de l'information, il faut s'assurer que le fichier .wav soit bien un fichier .wav et pas un imposteur [9] !
Vous allez donc implémenter une méthode qui va vérifier les blocs suivants :

- ChunkID doit être égal à RIFF
- Format doit être égal à WAVE
- Subchunk1ID doit être égal à fmt
- Subchunk2ID doit être égal à data

et renvoie un booléen pour dire si le fichier est valide ou non.

5.3 PrintInfos

```
static void PrintInfos(string filename);
```

Maintenant que vous connaissez la validité de votre fichier, vous pouvez commencer à en extraire les informations.
Rappelez vous votre mission, seuls certains blocs nous intéressent :

- ChunkSize
- AudioFormat
- NumChannels
- SampleRate

Le reste devra être ignoré [10].

Vous voilà maintenant prêt à faire avouer tous ses secrets à n'importe quel fichier .wav, et pour ce faire vous allez transformer votre `Main` en petite interface de torture.
On pourra s'attendre à ce genre de sortie [11] :

```
Wav torture !
A qui le tour ? : test.wav
Le fichier test.wav est valide
Que la torture commence !
...
Arggg.. Stop j'avoue tout :
Mon nom est : test.wav
Je pèse : 156984 octets
Je ne suis pas compressé !
Je possède 2 canaux audio
Ma fréquence d'échantillonnage est : 44100
```

Vous trouverez des fichiers de test sur les sites des ACDC.

5.4 Bonus : ReadHeader

```
static void ReadHeader(string filename);
```

Vous avez envie d'en savoir plus sur votre fichier, codez la méthode ReadHeader qui affichera le contenu de tous les blocs sur la console.

9. "l'extension ne fait pas le fichier" - *Unknown*
10. ignoré ne veut pas forcément dire non lu ...
11. Impressionnez nous en proposant des sorties originales !

6 Exercice 3 : WavMixer

Pour cet exercice, la méthode `Main` lira dans le fichier "sample.wav". La méthode MuteRightSpeaker sera appelée avec un BinaryWriter ouvert sur le fichier "first_output.wav". La méthode MuteAlternate sera appelée avec un BinaryWriter ouvert sur le fichier "second_output.wav".

Donc en exécutant votre programme, les deux fichiers seront générés.

6.1 Un peu de théorie ...

Dans la partie précédente, nous avons travaillé sur les métadatas du fichier .wav. Comme vous l'avez peut être compris, dans cette partie nous allons manipuler sur les datas. C'est ici que commence la partie délicate ! En effet, nous allons modifier directement les données contenus du fichier, donc si vous vous trompez dans la manipulation, faites attention à vos oreilles. Commençons par un peu d'explication sur le contenus du bloc data.

Le bloc de data commence à partir de l'octet 36. À partir de là, les quatre prochains octets contiennent les valeurs ASCII du texte "data". Les quatre octets suivants représentent le Subchunk2Size, qui indique la taille de la donnée data, mais on ne prendra pas ce bloc en compte.

100	97	116	97
'd'	'a'	't'	'a'

Et c'est enfin ici que commence le son ! Nous allons pour l'instant nous intéresser au format 16 bits, plus répandu que le format 8 bits. Les données sont séparées par unité de temps, déterminées par la fréquence, et représentées sur deux octets (16 bits) en mono (pas de différence gauche/droite) ou quatre octets (2 * 16 bits) en stéréo (un canal pour le haut parleur gauche, un autre pour le haut parleur droite). Ainsi, pour une fréquence de 10 000 Hz, une unité de temps fera donc 0.1 ms (pour rappel, une unité de temps est égale à l'inverse de la fréquence).

Nous allons travailler sur des sons stéréo pour manipuler les différents canaux. Intéressons nous ensuite aux quatre octets. Les deux premiers octets font référence au haut parleur gauche, les deux suivants à celui de droite. Les valeurs étant sur 16 bits signés, elles sont comprises entre -32,768 et 32,767[12], 0 représentant le silence.

Maintenant que nous avons vu pas mal de théorie, passons enfin à la pratique !

6.2 InitializeHeader

```
static void InitializeHeader(BinaryReader br, BinaryWriter bw);
```

Vous devez implémenter une méthode qui prend un BinaryReader (correspondant au stream d'entré), ainsi qu'un BinaryWriter (correspondant au stream de sortie) qui devra :
- Réinitialiser le "curseur" du flux au début du fichier (regarder du côté de la méthode Seek)
- Écrire le header jusqu'au bloc "data" (Subchunk2ID) dans le BinaryWriter
- Vérifier que le (Subchunk2ID) est bien égal a "data", sinon sortir de la méthode et écrire une erreur sur la console et sortir de la méthode
- L'écrire dans le BinaryWriter
- Écrire le bloc Subchunk2Size dans le BinaryWriter

12. http://msdn.microsoft.com/en-us/library/exx3b86w(v=vs.100).aspx

Si tout se passe bien, ce main de test devrait marcher et ne pas écrire d'erreur :

```
static void Main(string[] args)
{
    BinaryReader br = new BinaryReader(File.Open("sample.wav", FileMode.Open));
    BinaryWriter bw = new BinaryWriter(File.Create("output.wav"));
    InitializeHeader(br, bw);
    br.Close();
    bw.Close();

    br = new BinaryReader(File.Open("output.wav", FileMode.Open));
    bw = new BinaryWriter(File.Create("test.wav"));
    InitializeHeader(br, bw);
    br.Close();
    bw.Close();
}
```

6.3 MuteRightSpeaker

```
static void MuteRightSpeaker(BinaryReader br, BinaryWriter bw);
```

Pour cette méthode, nous allons manipuler un son stéréo pour effacer le canal de droite. Ainsi le son ne proviendra que du canal gauche. Un conseil, pour récupérer les valeurs sur 16 bits, il existe la méthode ReadInt16() de BinaryReader qui renvoie un short. Exemple :

```
BinaryReader br = new BinaryReader(File.Open("sample.wav", FileMode.Open));
short input = br.ReadInt16();
```

Si vous n'avez pas d'écouteur, vous pouvez toujours visualiser votre résultat sur des logiciels de traitement de son tel qu' Audacity.

Attention à la taille du type que vous écrivez dans le fichier !

```
/* 0 est ici considéré comme un int, soit 4 octets */
bw.Write(0);

/* Pour une valeur sur 16 bits, il faut caster la valeur en short */
bw.Write((short)0);
```

6.4 Bonus : MuteAlternate

```
static void MuteAlternate(BinaryReader br, BinaryWriter bw);
```

En bonus, la méthode MuteAlternate effacera 5000 intervalles sur le canal droite, puis 5000 intervalles sur le canal gauche,... Votre méthode ne devra pas planter si la donnée n'est pas un multiple de 5000 !

7 Conclusion

In ACDC You Trust.