TP n° 13

Système de mixage sonore

L'objectif de ce TP est de réaliser un logiciel de mixage sonore par la modélisation d'un système de producteurs d'échantillons sonores, de filtres (application de traitements sur les échantillons sonores) et de consommateurs d'échantillons sonores (ces derniers peuvent servir à enregistrer les échantillons dans un fichier ou encore à les jouer directement sur la carte son).

Dans ce TP, vous utiliserez les classes fournies par la STL (*Standard Template Library*) lorsque vous aurez besoin de structures de données (tableaux, listes etc.).

Introduction : gestion de mémoire et « *smart pointer* ».

En C++, la gestion mémoire est l'un des problèmes majeurs. En effet, lorsqu'une allocation dynamique est effectuée, il faut libérer explicitement la variable allouée. Cependant, lorsqu'au sein d'une application plusieurs pointeurs désignent la même variable (par exemple par l'appel du constructeur de copie ou de l'opérateur d'affectation), il peut parfois être difficile de savoir quand la libérer : ceci peut se faire lorsque plus aucun pointeur ne la désigne.

Un « *smart pointer* » est un type de pointeur qui prend en charge la libération de la mémoire allouée ; il existe différentes stratégies pour réaliser ce but ; celle que nous utiliserons ici s'appelle « comptage de références » ("reference counting") et est mise en oeuvre dans la classe *counted_ptr* :

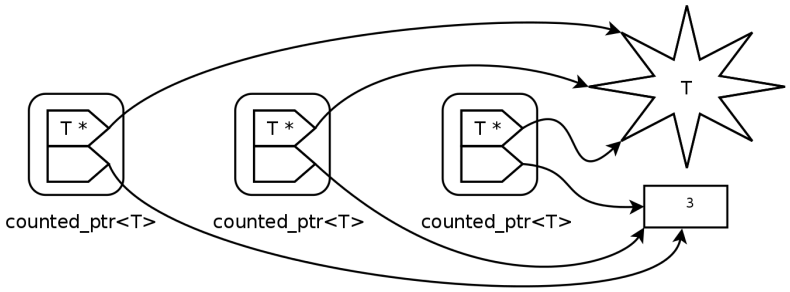


FIGURE 1 – Trois *counted_ptr* désignent une même variable allouée dynamiquement

Un *counted_ptr* compte le nombre de pointeurs qui désignent une variable donnée ; lorsque le nombre de pointeurs qui désignent cette variable atteint 0, le *counted_ptr* peut libérer automatiquement la mémoire allouée pour cette variable.

La classe *counted_ptr* dispose de tous les opérateurs associés aux pointeurs à l'exception de l'arithmétique pointeur qui ne sera pas utilisée dans le cadre de ce TP. En plus de désigner une variable allouée dynamiquement, un *counted_ptr* désigne un compteur associé à cette variable. Le rôle de ce compteur est de comptabiliser le nombre de *counted_ptr* qui désignent la variable en question.

Important :

- Dans ce TP, les variables créées dynamiquement seront obligatoirement désignées avec des *counted_ptr*.
- Attention : Une variable désignée avec un « smart pointer » ne doit pas être désignée par un pointeur « standard » : le système de gestion mémoire fourni par les « smart pointers » risquerait de libérer une variable alors que des pointeurs standards la désignent encore.
- On ne doit pas désigner avec un « smart pointer » une variable qui n'a été créée dynamiquement.

L'interface de la classe *counted_ptr* est fournie Figure 2 ; son implémentation vous est donnée.

```
/// Classe gérant un "smart pointer".
template <class T>
class counted_ptr {
public:
  ///  Constructeur
  /// \param pointer Si non nul, un pointeur vers une variable allouée dynamiquement.
  explicit counted_ptr(T * pointer = 0) ;

  ///  Destructeur, libère la donnée s'il s'agit du dernier pointeur qui la désigne.
  virtual ~counted_ptr() ;

  ///  Constructeur de copie.
  /// \param  smartPointer : le pointeur à copier.
  counted_ptr(const counted_ptr & smartPointer) ;

  ///  Opérateur d'affectation.
  /// \param  smartPointer : l'opérande de droite de l'affectation.
  counted_ptr & operator = (const counted_ptr & smartPointer) ;

  /// Constructeur de transtypage.
  /// Permet à un pointeur de T de pointer sur une instance de classe U, U héritant de T.
  /// \param smartPointer : pointe sur une instance de U
  template <class U>
  counted_ptr(const counted_ptr<U> & smartPointer) ;

  /// Donner à counted_ptr<U> accès à la partie privée de counted_ptr<T>
  template <class U> friend class counted_ptr;

  ///  Accès à la variable désignée par le pointeur.
  /// \return  La variable désignée par le pointeur.
  T & operator * () const  ;

  ///  Opérateur d'accès aux attributs / méthodes de la variable pointée.
  /// \return  adresse de la variable désignée par le pointeur.
  T * operator -> () const ;

}; // counted_ptr<T>
```

FIGURE 2 – Interface de la classe *counted_ptr* utilisée pour gérer des « smart pointers »

1 Mixage sonore

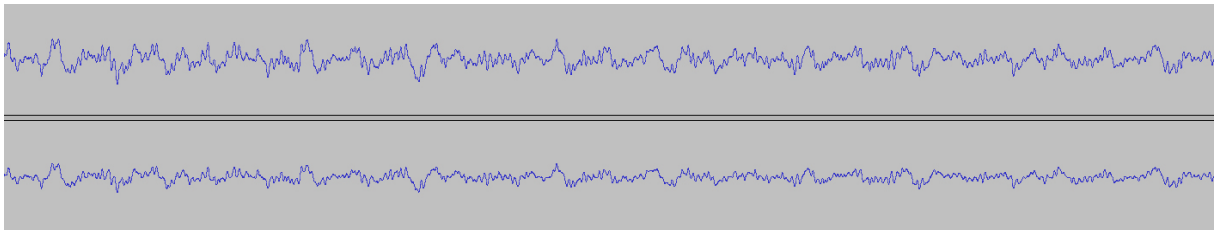


FIGURE 3 – Exemple de courbe sonore (en stéréo)

Le but de cette partie est de modéliser un système de mixage sonore. Dans notre système, un signal sonore sera considéré comme une suite de nombres réels (appelés échantillons) décrivant la forme de l'onde sonore. Chaque échantillon aura une valeur dans l'intervalle $[-1\,;1]$, 0 étant considérée comme une valeur neutre. Une seconde de son sera codée sous la forme d'une suite de 44100 échantillons décrivant la forme de la courbe sonore sur cette seconde (son qualité CD). Le premier échantillon correspond à l'amplitude du signal au temps 0s et le $x^{ième}$ échantillon correspond à l'amplitude du signal au temps $x/44100$ s. Un exemple d'une telle courbe vous est fourni sur la Figure 3

Le système que nous nous proposons de modéliser pour mixer plusieurs sons repose sur deux concepts : le **flot** et le **composant**.

1. Un **flot** correspond à une file d'échantillons. Il représentera une **sortie** d'un composant qui pourra ensuite être connectée à l'**entrée** d'un autre composant. Un flot doit implémenter l'interface *flot*.
2. Le rôle du **composant** est d'effectuer des traitements sur les échantillons (production d'échantillons, réglage de volume etc...). Dans la suite de ce TP, nous considérerons trois types de composants.

   le producteur : il s'agit d'un composant ne disposant que de sorties. Ce type de composant peut par exemple représenter un système de lecture du son à partir d'un fichier ou de l'entrée de la carte sonore. Ce type de composant doit implémenter l'interface *producteur*.
   Note : il est de la responsabilité d'un producteur de créer ses flots de sortie.

   le consommateur : il s'agit d'un composant ne disposant que d'entrées. Ce type de composant peut par exemple représenter un système d'écriture du son dans un fichier ou de restitution sonore. Ce type de composant doit implémenter l'interface *consommateur*. Un consommateur *ne crée pas de flot*, mais doit brancher ses entrées aux flots de sortie d'un ou plusieurs producteurs.

   le filtre : il s'agit d'un composant disposant d'entrées et de sorties (donc à la fois producteur et consommateur). Ce type de composant peut par exemple représenter un filtre de réglage de volume ou un effet écho. Ce type de composant doit implémenter l'interface *filtre*.

méthode *calculer* : Chaque composant posséde une méthode *calculer* (définie dans l'interface *composant*). C'est à l'intérieur de cette méthode que les différents calculs associés aux composants devront être placés.
À l'intérieur de cette méthode, le traitement ne pourra effectuer qu'une seule lecture par entrée et une seule écriture par sortie.
Le traitement d'un signal sonore ne pourra donc être effectué que par appels successifs à la méthode *calculer* de tous les composants d'une chaîne de filtrage.

Attention : pour les composants de type *consommateur* ou *filtre*, les calculs ne pourront être effectués que si les entrées disposent d'un nombre suffisant d'échantillons : ceci devra être vérifié par l'appel de la méthode *yaDesEchantillons* qui permet de savoir si chaque entrée possède au moins un échantillon.

Le diagramme de classe des interfaces susmentionnées (qui vous sont fournies) est présenté Figure 4.

Lors de la conception d'un système de filtrage, l'utilisateur devra créer des instances de composants, connecter les entrées des composants aux sorties d'autres composants afin de créer la structure de son système de filtrage, puis appeler la méthode calculer de chaque composant dans l'ordre adéquat (voir le programme exemple fourni).

La Figure 5 vous présente un exemple de chaîne de filtrage consistant à changer le volume d'un son. Cette chaîne est composée d'un producteur (lecture disque) dont la sortie 0 est connectée à l'entrée 0 d'un filtre de volume dont la sortie 0 est connectée à un consommateur (écriture disque). D'autre part, la figure présente un zoom sur le composant *volume* qui peut être vu comme un *filtre composé* constitué de plusieurs composants (*signal_constant* consistant à fournir un signal constant et *multiplication* qui consiste à multiplier les échantillons fournis sur les deux entrées).

Important : Dans votre code, vous ne devrez plus utiliser de pointeurs pour désigner des variables allouées dynamiquement mais des *counted_ptr*<T> et T sera impérativement du type de l'une des interfaces fournies.

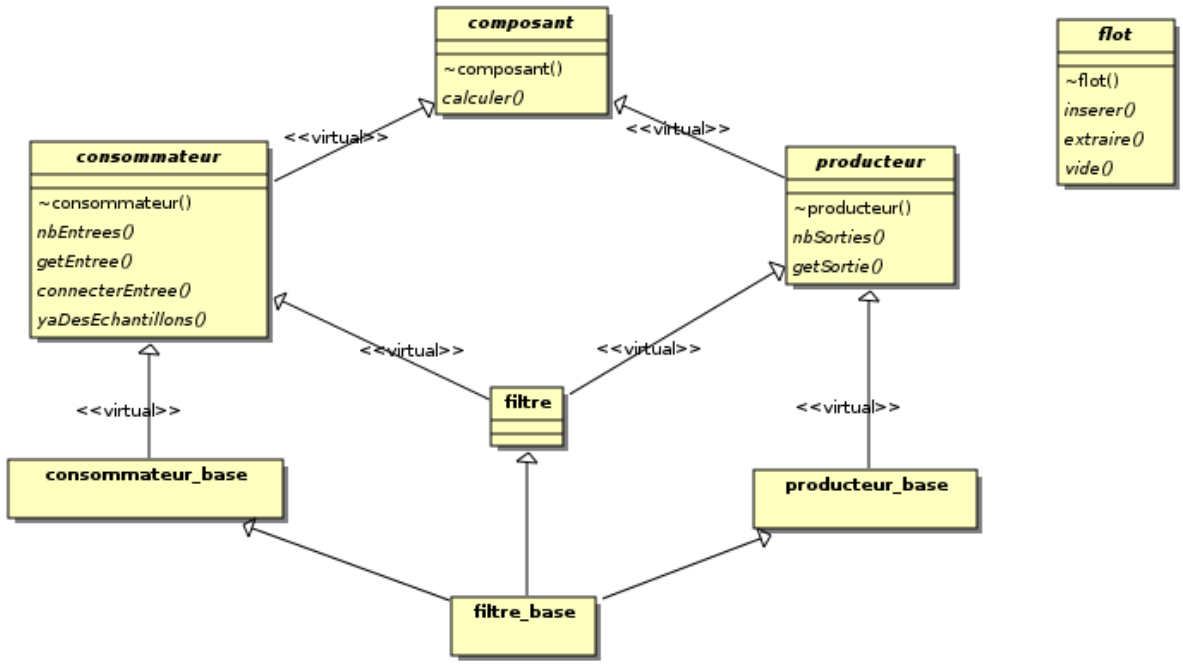


FIGURE 4 – Diagramme des interfaces et classes abstraites

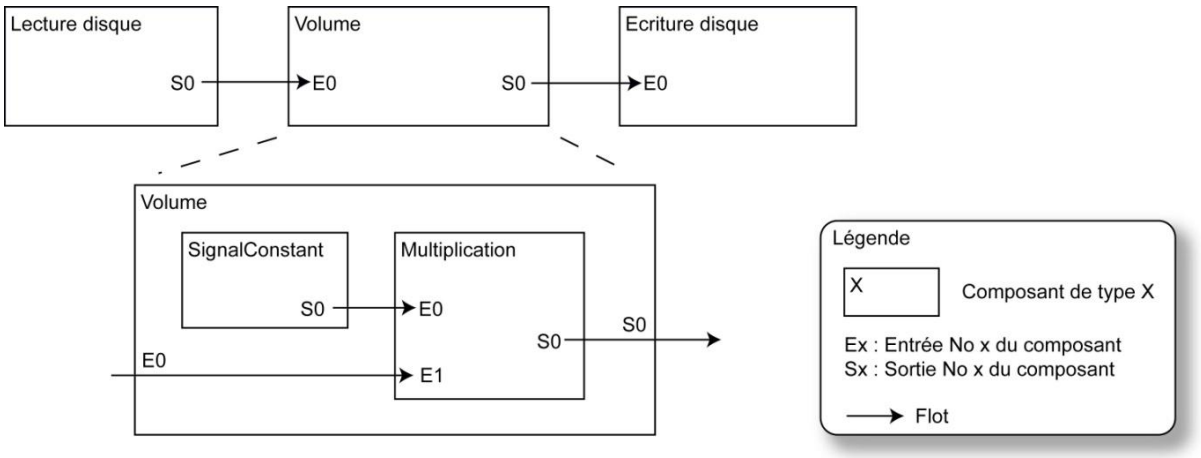


FIGURE 5 – Exemple de chaîne de filtrage sonore

1.1 Les fondations du système

Question 1 : Programmez la classe *imp_flot* qui implémente l'interface *flot*. Cette classe utilisera une instance de std :: deque (géré en file) pour stocker les échantillons.
NB : la pré-condition de la méthode *extraire* est : !*vide*().

Question 2 : Programmez la classe *signal_constant* qui aura pour rôle de fournir sur sa sortie une valeur constante fournie en paramètre lors de la création de l'instance. Cette classe devra implémenter l'interface *producteur*.
Testez votre implémentation à l'aide du programme fourni ; celui-ci crée un signal constant et enregistre ce signal dans un fichier texte en utilisant le composant *enregistreur_fichier_texte* fourni.

Question 3 : Programmez la classe abstraite *producteur_base* qui hérite [1] de l'interface *producteur* et implémente les fonctionnalités de gestion des flots communes à tous les producteurs ; n'hésitez pas à créer une méthode (protected) connecterSortie sur le modèle de connecterEntree.

1.2 Le composant harmonique

Nous allons maintenant programmer un second producteur d'onde sonore, qui aura pour rôle de produire une *harmonique* (une onde sinusoïdale pure). La formule permettant de produire cette onde est la suivante :

$$H(i, f, \phi) = sin(\frac{i}{44100} * 2 * \pi * f + \phi)$$

Dans cette équation, i correspond au numéro de l'échantillon à produire, f correspond à la fréquence de l'harmonique produite (par exemple la note *la* a une fréquence de 440Hz) et ϕ correspond au déphasage.

Question 4 : Programmez la classe *harmonique* qui implémente la classe abstraite *producteur_base*. Le constructeur de ce producteur d'harmoniques devra être paramétré par f et ϕ. Ce producteur posséde une unique sortie qui sera remplie par les échantillons de l'harmonique.
Remarque : le fichier constantes.h contient la définition de deux constantes (*pi* et *frequency* valant 44100), il vous est demandé d'utiliser ces constantes dans votre code.

Question 5 : Utilisez le programme fourni pour produire une harmonique correspondant à un *la* 440Hz et durant 2s, qui sera enregistrée dans un fichier ayant un seul canal de sortie (son mono) avec le composant *enregistreur_fichier* fourni. Référez-vous à l'Annexe 3 : Conversions de fichiers et écoute pour écouter le son produit.

1.3 Les fondations (suite)

Question 6 : Programmez la classe abstraite *consommateur_base* qui hérite[1] de l'interface *consommateur* et implémente les fonctionnalités communes à tous les consommateurs.

Question 7 : Programmez la classe *filtre_base* qui hérite des classes *filtre, producteur_base* et *consommateur_base* (il s'agit donc d'un héritage multiple).

1.4 Le composant Multiplicateur

La classe *multiplicateur* est un filtre possédant deux entrées et une sortie dont le rôle est de produire sur sa sortie un échantillon résultat de la multiplication des échantillons présents sur ses deux entrées.

Question 8 : De quelle classe doit hériter la classe *multiplicateur* afin de maximiser l'utilisation du code déjà produit ?

Question 9 : Programmez la classe *multiplicateur* et testez-la en reprenant votre précédent programme de test et en multipliant un *la* 440Hz avec un *la* 880Hz.

1. il s'agit ici obligatoirement d'un héritage virtuel

1.5 La puissance de la programmation générique : utilisation des foncteurs

Si nous souhaitions programmer un filtre additionnant deux signaux, il suffirait de reprendre l'implémentation du filtre multiplicateur et de changer l'implémentation de la méthode *calculer* afin que celle-ci effectue une addition au lieu d'une multiplication. En réalité, il en serait de même pour toute opération binaire utilisée pour combiner deux signaux sonores. Pour généraliser la notion de filtre appliquant une opération binaire sur deux canaux nous allons utiliser des *foncteurs*.

Un *foncteur* n'est rien d'autre qu'un objet dont la classe définit l'opérateur fonctionnel **operator()**. Les foncteurs ont la particularité de pouvoir être utilisés exactement comme des fonctions puisqu'il est possible d'appliquer leur opérateur fonctionnel avec la syntaxe d'un appel de fonction (Cf. le TP ferry et la programmation du comparateur pour les tris).

La STL (*Standard Template Library*), via l'inclusion du fichier d'en-tête functional, définit quatre classes de foncteurs binaires qui peuvent nous être utiles : std::minus<T>, std::plus<T>, std::multiplies<T>, std::divides<T> qui correspondent respectivement à la soustraction, l'addition, la multiplication et la division.

Voici un exemple d'utilisation d'un foncteur de multiplication :

```
// Déclaration d'un foncteur de multiplication
std::multiplies<double> multiplication;
// Utilisation du foncteur de multiplication
double resultat = multiplication(10.0, 20.0);
```

A la suite de l'exécution de ces deux lignes de code, resultat vaut $10 \times 20 = 200$.

Question 10 : Programmez la classe *operation_binaire* qui est une classe générique, paramétrée par le type de foncteur binaire à utiliser pour combiner deux signaux d'entrée en un seul signal de sortie.

Question 11 : Testez votre classe en remplaçant, dans votre programme de test, le filtre multiplicateur par un filtre *operation_binaire* correctement paramétré et jouant le rôle d'un filtre multiplicateur. Vous devriez obtenir le même résultat qu'avec le mutiplicateur programmé ci-dessus.
NB : pour comparer deux fichiers binaires, exécutez la commande : cmp fichier1 fichier2 ; si aucun message ne s'affiche, les deux fichiers ont même contenu.

1.6 Des filtres composés

Nous souhaitons concevoir un filtre consistant à régler le volume d'un son. Ce filtre possédera une entrée (le son) et une sortie (le son dont on a changé le volume). Le principe de ce filtre consiste à multiplier le son fourni en entrée par une constante (le volume).

Ce filtre devra être construit en utilisant les composants *signal_constant* et *multiplicateur* comme présenté sur la Figure 5.

Question préliminaire : quel est le bon « moment » pour connecter l'entrée « externe » du filtre composé *volume* sur l'entrée du composant interne *multiplicateur* ?

Question 12 : Programmez la classe *volume*. Testez votre filtre.

Comme vous avez pu le remarquer dans la question précédente :

1. la création d'un filtre composé demande de gérer le couplage entre les entrées / sorties « externes » du filtre composé et les entrées / sorties correspondantes des composants internes ;
2. d'autre part, le corps de la méthode *calculer* d'un filtre composé consiste en l'appel de la méthode *calculer* de chacun des composants internes.

Vous allez maintenant devoir généraliser le principe du filtre *volume* :

Question 13 : Programmez la classe *filtre_ compose*, héritant de la classe *filtre_ base*, et implémentant l'ensemble des fonctionnalités facilitant la conception d'un filtre constitué de plusieurs composants.

Il est à noter que si votre classe est correctement conçue, l'implémentation d'un filtre composé ne devrait se faire qu'en héritant de la classe *filtre_ compose* et en implémentant le constructeur décrivant la structure du filtre.

Pour guider votre réflexion, voici quelques questions auxquelles vous devrez répondre au préalable :

1. quelles sont les opérations à la charge de l'utilisateur d'un filtre composé concret (celui qui écrit le programme client) ?
2. quelles sont les opérations à la charge du concepteur d'un filtre composé concret, comme par exemple le *mixeur* décrit ci-dessous ?
3. quelles sont les opérations à la charge du programmeur de la classe abstraite *filtre_ compose* ?

Question 14 : Pour valider votre classe *filtre_ compose*, programmez une nouvelle version du filtre *volume*, appelée *volume_ compose* à l'aide de la classe *filtre_ compose* puis testez ce filtre.

Question 15 : À l'aide de la classe *filtre_ compose*, créez la classe *mixeur*. Le constructeur de cette classe sera paramétré par le nombre d'entrées du mixeur et une valeur de volume pour chaque entrée. Ce composant possédera une sortie qui correspondra au mixage des signaux fournis en entrée. La formule de mixage est la suivante :

$$m = \sum_{i=0}^{n-1} e_i * v_i$$

Dans cette formule n correspond au nombre d'entrées, m correspond à l'échantillon résultant du mixage, e_i correspond à l'échantillon présent sur l'entrée i et v_i correspond au volume associé à l'entrée i.

Testez votre filtre.

Faites un diagramme sur lequel figure l'ensemble des composants internes et externes, les entrées et sorties de tous les composants (internes et externes), ainsi que les connexions entre les entrées et sorties (internes et externes).

1.7 Jouer avec le son...

Dans le répertoire raw, vous avez à votre disposition plusieurs fichiers de sons. Ces fichiers correspondent à plusieurs pistes sonores contenant de la batterie, basse et autres sons qui composés forment un morceau de musique.

Les fichiers de sons qui vous sont fournis sont des fichiers stéréo enregistrés à 44100Hz (la fréquence que vous utilisez depuis le début du TP) et contenant des suites d'échantillons sans aucun en-tête (format *brut* ou « *raw* »). Dans ces fichiers, les échantillons sont codés sous la forme d'entiers 16bits signés (type short int prenant ses valeurs dans l'intervalle $[-32768\,;32767]$). Si les fichiers sont enregistrés en mono, la suite d'échantillons correspond à une piste. Si les fichiers sont enregistrés en stéréo, ils contiennent deux pistes entrelacées i.e. un échantillon pour la voie gauche, un échantillon pour la voie droite etc... Ces fichiers sont des fichiers binaires et non des fichiers texte.

1.7.1 lecteur de fichier

Dans un premier temps, vous allez programmer un lecteur de fichiers sonores ; il s'agit d'un producteur lisant un fichier et produisant sur sa / ses sorties (mode mono ou stéréo) les échantillons lus. Pour lire dans un fichier binaire, vous utiliserez la classe `std::ifstream` (définie dans le fichier `fstream`) dont voici un extrait de la documentation.

Constructeur : `ifstream(const char * filename, std::ios_base::openmode mode = std::ios_base::in);`

filename est un pointeur vers un tableau de caractères correspondant au nom du fichier et doit se terminer par le caractère de code ascii 0. Ce type de chaîne de caractères peut être extrait d'une instance de `std::string` avec la méthode `c_str()`.

mode est une valeur résultant de la combinaison avec l'opérateur | (ou bit à bit) d'une ou plusieurs valeurs de la liste ci-dessous :

- **app** (append) Set the stream's position indicator to the end of the stream before each output operation.
- **ate** (at end) Set the stream's position indicator to the end of the stream on opening.
- **binary** (binary) Consider stream as binary rather than text.
- **in** (input) Allow input operations on the stream.
- **out** (output) Allow output operations on the stream.
- **trunc** (truncate) Any current content is discarded, assuming a length of zero on opening.

Lecture de données binaires : `std::istream & read ( char * mem, std::streamsize n );`

- **mem** : pointeur vers la zone mémoire dans laquelle les données lues dans le fichier seront stockées.
- **n** : nombre d'octets à lire dans le fichier et à stocker dans la zone mémoire désignée par `mem` qui doit avoir une capacité sufisante pour stocker les octets lus.

Autres méthodes utiles :

- `bool good() const` : rend vrai si aucune erreur n'a été détectée à l'issue de la dernière opération sur le fichier ;
- `bool eof() const` : rend vrai si la fin du fichier est atteinte ;
- `void close()` : ferme le fichier et libère les structures de données associées.

Remarque : chaque appel de calculer doit lire un échantillon (cas d'un fichier mono) ou deux échantillons (cas d'un fichier stéréo). Comme un échantillon est codé sur 16bits, il s'agit de lire 2 octets et les placer dans un `short int` pour chaque canal.

Question 16 : Programmez la classe *lecteur_fichier* qui sera paramétrée par le nom du fichier contenant les données d'un son et le nombre de canaux du fichier à lire (fichier mono ou stéréo). Ce producteur posséde une sortie par canal.
Pour gérer les différentes situations (fichier inexistant, fin de lecture du fichier...) vous utiliserez les exceptions et plus précisément des classes d'exception dérivant de la classe *composant_exception* qui vous est fournie.
Attention : Les valeurs lues dans le fichier et les échantillons du système sonore n'ont pas la même plage de valeurs.

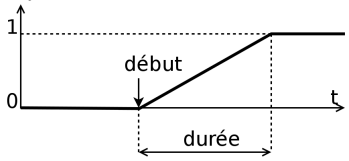
Question 17 : Faites un programme qui effectue le mixage de plusieurs pistes audio parmi celles qui vous sont fournies.

1.8 Des effets supplémentaires

Cette section est dédiée à la description de quelques effets à programmer pour enrichir votre système ; n'hésitez pas à créer des filtre auxiliaires supplémentaires pour faciliter leur programmation.

Pour tous les effets programmés, vous ferez un schéma représentant les composants et expliquant les connexions externes et internes.

Fade in. Le « *fade in* » (fondu à l'ouverture) fait progressivement passer le volume de 0 à 1. Cela permet de faire arriver un son progressivement. Les paramètres du constructeur sont la date de début de « *fade in* » et sa durée.



Fade out. Le « *fade out* » (fondu à la fermeture) est l'inverse du fade in, il permet de faire progressivement passer le son d'un volume de 1 (volume maximum) à 0. Cela permet de faire progressivement disparaître un son. Essayez d'imaginer une solution qui utilise le composant précédent.

Panning. Le « *panning* » (panoramique) joue sur la stéréo. Il s'agit d'un effet qui permet de faire circuler un son entre la voie de gauche et la voie de droite. Il est généralement géré par un filtre possédant la voie gauche et la voie droite en entrée ainsi qu'une autre entrée décrivant la courbe de « *panning* » (valeur -1 : son sur la voie gauche, valeur 1 : son sur la voie droite, valeur 0 : son à 50% du volume sur la voie gauche et la voie droite).

Compression. Cet effet permet de diminuer les sons trop forts, et au contraire rehausser les sons trop faibles. Cet effet est très utilisé par exemple à la télévision, pour « égaliser » les voix entre autre... Une méthode simple de programmation d'une sorte de compression (la vraie n'est pas vraiment celle-ci) consiste à appliquer la formule suivante sur les échantillons sonores :

$$C(x) = signe(x) * |x|^c \, ; \, c \in]0; 1]$$

c correspond à la « force » de l'effet. Si $c = 1$ la compression est inexistante, si $c < 1$ elle est active. La fonction $signe(x)$ vaut -1 si x est négatif et 1 si x est positif.

Echo. L'effet d'écho est obtenu en mixant au son original ce même son mais joué avec un décalage temporel. Pour réaliser cet effet, votre filtre doit donc avoir une mémoire permettant de stocker les échantillons traités.

Annexe 1 : Configuration du projet eclipse

1. créez un projet puis créez un répertoire src (pour vos fichiers sources) et un répertoire include (pour vos fichiers « .h »).
2. Ouvrez ensuite les propriétés de votre projet et sélectionnez l'onglet « C/C++ Build / Settings » :
   - C++ Compiler / Includes / Include Paths :
     - ajoutez le répertoire include du projet ;
     - ajoutez le répertoire /share/esir1/prog/13_mixage/include ;
   - C++ Linker / Libraries :
     - rubrique « Libraries » : indiquez le nom abrégé de la bibliothèque (outils) ;
     - rubrique « Library Search Path » : indiquez le répertoire contenant la bibliothèque outils : /share/esir1/prog/13_mixage/lib

Annexe 2 : Bibliothèques et fichiers pour le mixage

Les fichiers fournis pour ce TP se trouvent dans le répertoire /share/esir1/prog/13_mixage. Dans ce répertoire, vous trouverez les sous-répertoires :

- src : contient une ébauche de programme client, *à copier dans votre répertoire* src (voir Annexe 1 : Configuration du projet eclipse).

— include : contient l'ensemble des fichiers .h qui vous sont fournis ; *ces fichiers ne devront pas être modifiés ni copiés dans votre projet.*
— lib : contient la bibliothèque liboutils.a. Cette bibliothèque contient deux versions d'enregistreurs (enregistrement dans un fichier binaire « *raw* » et enregistrement dans un fichier texte) ; *même remarque.*
— raw : répertoire contenant des fichiers de sons que vous pourrez utiliser durant ce TP ; *même remarque.*
— doc : contient la documentation des classes fournies.

Annexe 3 : Conversions de fichiers et écoute

Les fichiers de sons qui sont mis à votre disposition pour vos expérimentations sont dans le répertoire /share/esir1/prog/13_mixage/raw.

Différents formats de fichiers de son

raw : données brutes (non compressées) sans en-tête ;

wav : données brutes (non compressées) avec un en-tête qui décrit le format des données ;

mp3 : données compressées (avec perte) avec en-tête ; format non-libre ;

ogg/vorbis : données compressées (avec perte) avec en-tête ; format ouvert et libre ;

flac : Free Lossless Audio Codec = format de codage libre sans perte.

Dans les salles de TP, on dispose de l'outil sox/play/rec pour convertir/écouter/enregistrer des fichiers audio. Voici quelques options relatives au format des données dans un fichier audio sans en-tête extraites de la documentation (man sox) :

-b BITS : The number of bits in each encoded sample.

-c CHANNELS : The number of audio channels in the audio file (1 = mono, 2 = stereo).

-e ENCODING : The audio encoding type. Sometimes needed with file-types that support more than one encoding type.

signed : PCM data stored as signed integers.

-r RATE : Gives the sample rate in Hz (or kHz if appended with 'k') of the file.

-L : little endian

Conversion de format

— Conversion du format raw vers wav : sox -b 16 -c 1 -e signed -r 44100 -L *f.raw f.wav*
— Conversion du format wav vers raw : sox *f.wav f.raw*

Ces opérations se font sans perte d'information ;

Écoute d'un fichier de son

— format brut, 16bits par échantillon, 1 canal, 44100Hz : play -b 16 -c 1 -e signed -r 44100 -L *f.raw*
— format brut, 16bits par échantillon, 2 canaux, 44100Hz : play -b 16 -c 2 -e signed -r 44100 -L *f.raw*
— format wav : play *f.wav*

Stockage des fichiers de sons

Étant donné la place occupée par les fichiers audio, n'hésitez pas à vous créer un répertoire dans /temporaire/reseau pour y stocker vos fichiers audio (voir www.istic.univ-rennes1.fr/fr/Intranet/ZoneEtudiants/Assistance/FAQ, rubrique « Plateforme Linux »).