

Gestion des ressources avec SDL via les SDL_RWops, en C++

par Jean Christophe Beyler ([Autres articles](#))

Date de publication : 25/04/2007

Dernière mise à jour : 25/04/2007

Lorsqu'un programme nécessite des ressources externes, il est généralement important de bien définir comment celles-ci vont être stockées. Les possibilités sont nombreuses et dépendent vraiment de ce que cherchent les développeurs. Dans le cas où on souhaite cacher les données, limiter le risque de corruptions ou modifications des données ou simplement pour réduire le nombre de fichiers visibles à l'utilisateur, la mise en archive des fichiers externes est souvent la meilleure solution. Par contre, cela veut dire qu'il faut trouver un moyen de récupérer ces données. Cet article présente une solution utilisant les fonctions associées à la structure SDL_RWops.

- I - Introduction
- II - La structure et les fonctions
- III - Un premier exemple
 - III.1 - Le programme de base
 - III.2 - Utilisation de SDL_RWops
- IV - Un gestionnaire de ressources
 - IV.1 - La création d'un fichier ressource
 - IV.1.A - L'entête
 - IV.1.B - Le code
 - IV.1.C - Utilisation
 - IV.2 - La lecture d'un fichier ressource
 - IV.2.A - La classe GestionFich
 - IV.2.B - Les constructeurs
 - IV.2.C - Le destructeur
 - IV.2.D - La fonction chargeZone
 - IV.2.E - La fonction libereZone
 - IV.2.F - La fonction rechercheZone
 - IV.3 - La création d'une classe de gestion
 - IV.3.A - La classe Gestionnaire
 - IV.3.B - Les constructeurs et le destructeur
 - IV.3.C - Les fonctions chargeBMP et libere
 - IV.4 - L'utilisation du gestionnaire
- V - Pour aller plus loin
 - V.1 - Utilisation d'une archive zip
- VI - Téléchargements
- VII - Conclusion
- VIII - Remerciements

I - Introduction

Généralement, lorsque nous programmons quelque chose de conséquent, des fichiers externes sont utilisés. Par exemple, le programme peut nécessiter des fichiers de configuration, des fichiers sonores ou des images.

Au départ, il est plus simple de laisser ces ressources dans une arborescence dédiée et donc les zones du code qui en ont besoin, ouvriront directement les fichiers concernés. Mais lorsque la phase de programmation se termine, souvent les programmeurs se demandent s'il est bon de laisser ces ressources dans les mains des utilisateurs.

Cet article ne se veut pas être un encouragement de la tendance de vouloir tout cacher. Il y a généralement beaucoup plus à gagner en laissant tout à disposition que de tout vouloir masquer. Un artiste, qui aime le jeu, adaptera les images ou les sons pour le rendre plus beau ou agréable, chose qui ne sera rarement fait si tout est caché. Il va de même pour le code source, un programmeur peut ajouter une fonctionnalité intéressante que les programmeurs de base n'avaient pas imaginée.

Mais, il arrive que dans certains cas, le programme ne peut pas permettre une telle transparence. Pour de telles raisons, cet article est écrit afin de montrer une technique facile à mettre en place.

En utilisant la bibliothèque SDL, on découvre la présence d'une structure nommée **SDL_RWops**. Chaque fonction de chargement d'image, par exemple, propose une version qui prend en argument un pointeur vers une structure **SDL_RWops** à la place d'un nom de fichier. Cela permet de charger en mémoire les données et directement les utiliser. On pourra, par exemple, générer une image directement à l'exécution et ensuite charger cette image dans une structure de type **SDL_Surface** sans passer par un fichier intermédiaire.

Plusieurs fonctions sont associés à cette structure et cet article va les présenter dans un premier temps avant de montrer un exemple basique. Ensuite, nous allons compliquer les choses en ajoutant un gestionnaire de ressources intelligent qui permettra de rendre transparent la recherche des ressources.

Puis, dans la cinquième partie, nous allons utiliser la bibliothèque `zip` qui permet d'utiliser directement des archives `.zip` et donc simplifier la vie au gestionnaire.

Mais avant d'arriver à cet endroit de l'article, commençons par présenter la structure et les fonctions concernées.

II - La structure et les fonctions

Cette partie va présenter la structure **SDL_RWops** et les fonctions associées. Bien que nous n'aborderons et n'accéderons jamais directement le contenu de la structure, c'est toujours bon de savoir ce qui s'y trouve :

La structure SDL_RWops

```
typedef struct SDL_RWops {
    int (*seek)(struct SDL_RWops *context, int offset, int whence);
    int (*read)(struct SDL_RWops *context, void *ptr, int size, int maxnum);
    int (*write)(struct SDL_RWops *context, const void *ptr, int size, int num);
    int (*close)(struct SDL_RWops *context);

    Uint32 type;
    union {
        struct {
            int autoclose;
            FILE *fp;
        } stdio;
        struct {
            Uint8 *base;
            Uint8 *here;
            Uint8 *stop;
        } mem;
        struct {
            void *data1;
        } unknown;
    } hidden;
} SDL_RWops;
```

Comme vous le voyez, la structure contient des pointeurs de fonctions pour gérer la mise à jour de la position du curseur, la lecture, l'écriture et la fermeture du **SDL_RWops**.

Maintenant serait un bon moment pour expliquer pourquoi la structure contient ces pointeurs. En fait, une variable de type **SDL_RWops** peut provenir d'un nom de fichier, d'un pointeur **FILE*** ou directement de la mémoire. Dépendant de la provenance, ces pointeurs de fonctions seront initialisés différemment. Par exemple, si on ouvre un **SDL_RWops** à partir de **FILE***, à la fermeture du **SDL_RWops** on peut demander qu'à la libération de la structure, le fichier soit fermé par exemple.

Cela nous mène à présenter les fonctions qui vont créer une structure **SDL_RWops** à partir de différentes sources de données.

Les fonctions associées

```
SDL_RWops * SDLCALL SDL_RWFFromFile(const char *file, const char *mode);
SDL_RWops * SDLCALL SDL_RWFFromFP(FILE *fp, int autoclose);
SDL_RWops * SDLCALL SDL_RWFFromMem(void *mem, int size);
SDL_RWops * SDLCALL SDL_RWFFromConstMem(const void *mem, int size);

SDL_RWops * SDLCALL SDL_AllocRW(void);
void SDLCALL SDL_FreeRW(SDL_RWops *area);
```

Comme vous le voyez, on peut créer une structure **SDL_RWops** à partir d'un nom de fichier. Le mode permettra d'ouvrir le fichier en lecture mais aussi en écriture si tel est le désir du programmeur !

On peut aussi ouvrir le fichier à partir d'un pointeur **FILE*** mais aussi directement d'une zone mémoire. La différence entre **SDL_RWFFromMem** et **SDL_RWFFromConstMem** consiste dans la permission d'écrire dans la zone fournie en paramètre. Bien sûr, dans le cas d'une zone mémoire, il faudra fournir la taille.

Enfin, pour la fonction utilisant un pointeur **FILE***, le dernier paramètre **autoclose** permet de fermer automatiquement le fichier lorsque vous appellerez **SDL_FreeRW**. Mais cela n'est pas vraiment important puisque, généralement, nous n'avons même pas besoin d'appeler cette fonction de libération de mémoire, elle sera appelée en interne. Par contre, pour les fonctions utilisant directement une zone mémoire, il faudra libérer la mémoire après la fin de l'utilisation des **SDL_RWops** concernés. Du coup, il faudra être sûr de le faire au bon moment sinon le comportement global du programme sera indéfini.

Les deux dernières fonctions servent pour l'allocation et la libération d'une structure **SDL_RWops**. Dans cet article, nous n'allons jamais appeler ces fonctions puisque ce sera entièrement géré en interne.

III - Un premier exemple

Dans cette partie, nous allons voir une première utilisation des fonctions **SDL_RWops**. Bien que cela sera très basique, à la fin de cette section, vous allez voir comment est programmée en interne la fonction **SDL_LoadBMP**. En effet, vous allez découvrir que vous appelez déjà sans le savoir ces fonctions.

III.1 - Le programme de base

Dans cette partie, nous allons montrer une utilisation basique de ces fonctions et comment elles s'intègrent dans un code normal. Avant d'utiliser les **SDL_RWops**, voici un code qui affiche simplement deux images dans une fenêtre.

Nous n'allons pas présenter tout le code puisque c'est le même code qu'on peut trouver [ici](#). Par contre, nous allons montrer ce qui a été ajouté. Tout d'abord, avant la boucle événementielle, le programme charge deux images :

Chargement des images

```
SDL_Surface *im1, *im2;
SDL_Rect rect_im1, rect_im2;

...

//Chargement des deux images et mise en place des positions
im1 = SDL_LoadBMP("image1.bmp");
im2 = SDL_LoadBMP("image2.bmp");

rect_im1.x = 10;
rect_im1.y = 10;
rect_im2.x = 330;
rect_im2.y = 10;

if((im1==NULL)|| (im2==NULL)) {
    done=true;
}
```

Ce code commence donc en appelant **SDL_LoadBMP** et ensuite initialise les deux structures **SDL_Rect** qui seront là pour positionner les deux images à l'écran. Ensuite, après la gestion des événements, il suffit de mettre une couleur de fond et dessiner les deux images :

Affichage des images

```
/* Gestion de l'affichage */
/* On remplit l'image de noir */
SDL_FillRect(screen, NULL, 0);

/* On dessine la premiere image */
SDL_BlitSurface(im1, NULL, screen, &rect_im1);
/* On dessine la deuxieme image */
SDL_BlitSurface(im2, NULL, screen, &rect_im2);
```

Vous trouverez ce code source basique [ici](#). Si jamais ce code vous semble déjà compliqué, vous feriez mieux de lire les tutoriels sur la SDL [ici](#).

III.2 - Utilisation de **SDL_RWops**

Pour utiliser les fonctions qui vont créer une structure **SDL_RWops**, nous allons écrire une fonction nommée **chargerBMP** :

La fonction chargerBMP

```
SDL_Surface* chargerBMP(char *nom)
{
    SDL_RWops *tmp = SDL_RWFromFile(nom, "rb");
    if(tmp == NULL) {
        return NULL;
    }

    SDL_Surface *res = SDL_LoadBMP_RW(tmp, 1);
    return res;
}
```

Nous avons écrit cette fonction pour montrer que, dans la fonction **main** précédente, il ne doit pas y avoir de différences entre la première version et celle-ci. En effet, la première chose à faire est d'abstraire du code qui va gérer les ressources. Donc, pour le **main**, il ne sait pas qu'il vient de provoquer un passage par les **SDL_RWops**.

Avant de regarder le changement dans le **main**, regardons en détail les appels de **chargerBMP**. Tout d'abord nous appelons **SDL_RWFromFile**. Le premier paramètre est le nom de fichier à ouvrir et le deuxième est le mode. Nous passons *rb* comme mode puisqu'un fichier **BMP** sera un fichier en écriture seulement (nous ne voulons pas permettre à l'utilisateur de réécrire dans le fichier image), et c'est aussi un fichier binaire.

Une fois que nous avons testé que le fichier est correctement ouvert, nous allons charger l'image directement en utilisant la fonction **SDL_LoadBMP_RW**. Cette fonction est exactement comme **SDL_LoadBMP** sauf qu'à la place de prendre un nom de fichier, il prend une structure **SDL_RWops** qui représentera le **BMP**. Le deuxième paramètre définit si la structure **SDL_RWops** sera libérée à la fin de la fonction **SDL_LoadBMP_RW**. Si nous passons 0, la structure **SDL_RWops** ne sera pas libérée et le fichier ne sera pas fermé. Dépendant de ce que nous voulons, on passera 0 ou 1, mais ici, nous voulons qu'à la libération de la surface, la structure soit libérée et le fichier soit fermé.

Finalement, dans la fonction **main**, voici comment nous appelons la fonction **chargerBMP** :

Les changements dans la fonction main

```
//Chargement des deux images et mise en place des positions
im1 = chargerBMP("image1.bmp");
im2 = chargerBMP("image2.bmp");
```

Comme vous le voyez, rien ne change sauf l'appel de fonction. C'est donc en interne que nous allons gérer les ressources du programme. Maintenant que nous avons présenté brièvement ces fonctions, regardons comment le code source de la SDL définit la fonction **SDL_LoadBMP** :

La fonction SDL_LoadBMP

```
#define SDL_LoadBMP(file)    SDL_LoadBMP_RW(SDL_RWFromFile(file, "rb"), 1)
```

En fait, ils font exactement comme la fonction **chargerBMP** ! La seule différence est le test que nous avons ajouté entre les deux appels. Ce test n'est pas obligatoire car **SDL_LoadBMP_RW** vérifie que le pointeur soit non nul. Mais, par précaution, je préfère mettre un test de plus que de risquer un comportement indéfini.

Vous trouverez le code source de cette nouvelle version [ici](#).

IV - Un gestionnaire de ressources

Une fois que nous avons compris comment fonctionne les fonctions associées aux **SDL_RWops**, nous pouvons aller plus loin. Dans cette section, nous allons montrer comment faire une archive à nous, facile d'utilisation et qui permet de limiter le nombre de fichiers ressources visibles à l'utilisateur.

IV.1 - La création d'un fichier ressource

Cette sous-partie va montrer une façon naïve de créer un fichier ressource. Ce n'est pas vraiment viable comme solution, il faudrait ajouter une technique pour crypter une partie et compresser le fichier. Mais pour une question de simplicité, nous allons juste juxtaposer les fichiers et ajouter une entête facile à lire.

Finalement, nous allons aussi prendre des raccourcis pour beaucoup de choses pour simplifier l'écriture de ce fichier. La première chose à remarquer est la structure du fichier.

IV.1.A - L'entête

Avant de pouvoir écrire le code, il faut décider une structure de l'entête. Nous allons utiliser la structure la plus simple :

La structure de l'entête

- 1 Le nombre de fichiers
- 2 La liste des noms de fichiers, point de départ et leur taille
- 3 Le contenu de chaque fichier

Voici la déclaration des structures mises en jeu :

La structure de l'entête

```
#define TAILLE_NBR 7
typedef struct sEntete_interne_dep{
    unsigned char nbr_entrees[TAILLE_NBR];
}SEntete_interne;

#define TAILLE_NOM 32
#define TAILLE_POS 10
#define TAILLE_TAILLE 10
typedef struct sEntete_interne_elem{
    unsigned char nomFichier[TAILLE_NOM];
    unsigned char position[TAILLE_POS];
    unsigned char taille[TAILLE_TAILLE];
}SEntete_interne_elem;
```

Comme vous le voyez, l'entête est constituée de chaînes de caractères de longueurs fixes. Cela permet de savoir exactement sa taille. Le fichier commencera donc par un entier qui tiendra sur 7 caractères. Cet entier correspond au nombre de fichiers dans l'archive. Ensuite, nous aurons une entrée par fichier. Chacune contiendra le nom du fichier, sa position et sa taille. Chaque élément de cette entête est rempli à droite par des espaces.

Pour trouver un fichier, il suffira donc de parcourir la liste des entrées et trouver le nom correspondant. Ensuite, nous aurons la position et la taille dans le fichier.

IV.1.B - Le code

Nous continuerons cette partie en présentant le code pour créer facilement un fichier d'archive. Le code commence bien sûr par les inclusions habituelles :

Les inclusions

```
#include <fstream>
#include <stdlib.h>
#include <string.h>
#include <iostream>
#include <sstream>

#include "gestionfich.h"

using namespace std;
```

Ensuite, nous avons écrit ce code simplement dans la fonction **main** vu sa complexité. Vous verrez qu'il n'y a pas grand chose de plus simple. Nous commençons par vérifier le nombre d'arguments :

La vérification des arguments

```
int main(int argc, char **argv)
{
    //On verifie le nombre d'arguments
    if(argc<=2) {
        cout << "Usage : " << argv[0] << " <Nom de l'archive> " << "<Fichier 1> [Fichier2] ...
[FichierN]" << endl;
        return EXIT_FAILURE;
    }
}
```

Ensuite, nous tentons d'ouvrir le fichier archive en écriture, si cela échoue, le programme sortira sans rien faire.

L'ouverture du fichier archive

```
//On tente d'ouvrir l'archive en ecriture
ofstream output(argv[1], ios_base::binary);

if(output != NULL) {
```

La première donnée dans le fichier sera le nombre de fichiers qu'il contient. Cela se fait simplement, il faut juste s'assurer d'avoir le bon nombre de caractères pour assurer une entête de taille fixe. On utilise donc la macro **TAILLE_NBR** et la fonction **resize** de la classe **string**.

L'écriture du nombre d'éléments

```
//Ecriture du nombre d'elements
ostringstream oss;
oss << (argc-2);
string nombre = oss.str();
nombre.resize(TAILLE_NBR, ' ');
output << nombre;
```

Ensuite, nous devons calculer la position de départ du premier fichier. Il faut donc calculer la taille totale de l'entête. En effet, bien que les structures soient de tailles fixes, nous avons un certain nombre d'éléments dans l'entête. Il faudra donc calculer la taille dépendant de ce nombre :

Calcul de la position du premier fichier

```
//Position de depart
long pos = sizeof(SEntete_interne_elem)*(argc-2) + sizeof(SEntete_interne);
```

Une fois que nous avons la position de départ du premier fichier, nous pouvons écrire les entrées des fichiers contenus dans l'archive. Ceci commence par l'écriture du nom du fichier et de sa position. Ensuite, il faut calculer la taille du

fichier et l'écrire à la suite. Il n'y a rien de spécial dans ce code mais il faut simplement faire attention de mettre tous les éléments à la bonne taille. Comme avant, ceci se fait en ajoutant des espaces sur la droite de la chaîne de caractères. Il faut donc transformer la position et la taille en string.

L'écriture des informations des fichiers

```
for(int i = 2 ; i < argc; i++) {
    //Ecriture du nom de fichier
    string nom(argv[i]);
    nom.resize(TAILLE_NOM, ' ');
    output << nom;

    //Ecriture de la position
    ostringstream osspos;
    osspos << pos;
    string strpos = sspos.str();
    strpos.resize(TAILLE_POS, ' ');
    output << strpos;

    //Ecriture de la taille du fichier a ajouter
    ifstream fich(argv[i]);
    if(fich == NULL) {
        cerr << "Erreur d'ouverture du fichier " << argv[i] << endl;
        output.close();
        return EXIT_FAILURE;
    }
    fich.seekg(0, ios_base::end);
    long taille = fich.tellg();

    //On augmente la position pour le prochain fichier
    pos += taille;
    fich.close();

    //On transforme la taille en chaine avant de l'ecrire
    ostringstream osstaille;
    osstaille << taille;
    string strtaille = osstaille.str();
    strtaille.resize(TAILLE_TAILLE, ' ');
    output << strtaille;
}
```

Enfin, nous allons pouvoir écrire le contenu des fichiers. Ceci se fait très simplement en utilisant la fonction **rdbuf** de la classe **ifstream**.

L'écriture des fichiers

```
//Maintenant on ecrit les fichiers
for(int i = 2 ; i < argc; i++) {
    ifstream input(argv[i]);
    if(input != NULL) {
        output << input.rdbuf();
    }
    input.close();
}
```

Cela termine donc ce fichier d'archive, nous pouvons le fermer et sortir du programme.

Fin du programme

```
output.close();
}

return EXIT_SUCCESS;
}
```

Certes, ce petit programme est un peu naïf et l'entête n'est pas très intelligente mais il suffit largement pour ce que nous voulons faire. Il va permettre à un programme de lire facilement les informations et récupérer les fichiers qu'il cherche. La prochaine section montre comment faire.

Vous trouverez le code source de création d'archive [ici](#).

IV.1.C - Utilisation

Enfin, pour créer notre archive, nous devons simplement faire cela :

La classe GestionFich

```
creation archive.arc image1.bmp image2.bmp
```

Ceci va créer une archive **archive.arc** contenant les deux images BMP.

IV.2 - La lecture d'un fichier ressource

Nous avons vu comment le fichier d'archive est créé. Il est temps de comprendre comment s'en servir et quelles sont les possibilités d'une telle solution.

IV.2.A - La classe GestionFich

Pour simplifier encore plus le système nous allons écrire une classe qui se charge de donner le point de départ de chaque fichier, de charger en mémoire la zone si nécessaire, de libérer si elle n'est plus utilisée. Voici sa déclaration :

La classe GestionFich

```
class GestionFich
{
    private:
        //Le fichier
        string nomarchive;

        //Entete
        SEntete *entete;
        int nbr_entrees;

        //Les elements
        map<string, SZone*> elements;

        //Fonctions privees
        //Fonction qui charge une zone, cree une structure SZone associee
        SZone *chercheZone(string);

    public:
        GestionFich();
        GestionFich(string nom);
        ~GestionFich();

        SDL_RWops* chargeZone(string);
        void libereZone(string);
};
```

Cette classe contient le nom de l'archive, un pointeur sur un tableau contenant toutes les informations des fichiers contenus dans l'archive (comme on a vu précédemment) et une **map** qui contiendra la correspondance entre le nom de fichier et la zone en mémoire où se trouve ce fichier.

Comme constructeur, cette classe accepte le constructeur vide et un constructeur qui donnera le nom de l'archive. Finalement, nous aurons deux fonctions : une pour chercher un nom de fichier et le mettre en mémoire et une fonction pour libérer la zone.

En principe, vous avez dû remarquer que le tableau des entêtes est de type **SEntete**, or la structure que j'ai présentée avant s'appelait **SEntete_interne_elem**. La raison est assez simple : même si notre entête est écrit en chaînes de caractères, cela ne nous oblige pas à nous en servir par la suite comme tel, nous utiliserons plutôt :

La structure SEntete

```
typedef struct sEntete {
    string nomFichier;
    long position;
    long taille;
}SEntete;
```

Cela rendra l'utilisation de cette archive plus facile. Enfin, la dernière structure à présenter s'appelle **SZone**, elle contient simplement la position en mémoire du fichier extrait de l'archive et sa taille.

La structure SZone

```
typedef struct sZone {
    void *depart;
    long taille;
}SZone;
```

IV.2.B - Les constructeurs

Les constructeurs de cette classe sont très simples : on initialise le nom de l'archive avec le nom passé en argument et on met le pointeur **entete** à NULL.

Les constructeurs

```
//Constructeurs
GestionFich::GestionFich()
{
    nomarchive = "Donnees.arc";
    entete = NULL;
}

GestionFich::GestionFich(string nom)
{
    nomarchive = nom;
    entete = NULL;
}
```

IV.2.C - Le destructeur

Le destructeur est un peu plus compliqué. En effet, on efface le tableau **entete** mais on doit aussi effacer la map **elements**.

Pour effacer la map, on utilise un itérateur et on récupère chaque pointeur de type **SZone**. Ensuite, on transtype en pointeur char* et on libère la zone et le pointeur vers la structure.

Finalement, on libère la mémoire en appelant la fonction **clear**.

Le destructeur

Le destructeur

```
//Destructeur
GestionFich::~GestionFich()
{
    delete[] entete;

    //On efface la map
    map<string, SZone*>::iterator iter;
    for(iter = elements.begin(); iter != elements.end(); iter++) {
        //On efface la zone
        SZone *tmp = iter->second;

        if(tmp != NULL) {
            char *donnees = reinterpret_cast<char*> (tmp->depart);
            delete[] donnees;
            delete tmp;
        }
    }
    //On libere la memoire
    elements.clear();
}
```

IV.2.D - La fonction chargeZone

La fonction **chargeZone** est le point d'entrée de la classe **GestionFich**. C'est grâce à elle que nous allons récupérer un fichier dans une archive et la mettre en mémoire. Nous allons supposer que les fichiers récupérés sont supposés en lecture seule. Ceci nous permettra de stocker les informations du chargement des fichiers. Cela implique que si nous chargeons deux fois le même fichier, le système va le charger une seule fois et retournera deux fois les mêmes informations.

La fonction **chargeZone** est en fait une fonction qui retourne un pointeur vers une structure de type **SDL_RWops**. Elle appelle la fonction **chercheZone** qui va chercher dans la **map** la zone correspondante. Si la zone ne se trouve pas dans la map, elle la chargera en mémoire. Voici le code de cette fonction :

La fonction chargeZone

```
SDL_RWops* GestionFich::chargeZone(string nom)
{
    //On regarde dans la map d'abord
    SZone *tmp = chercheZone(nom);

    //Gestion de l'erreur
    if(tmp == NULL) {
        return NULL;
    }

    //Creer un SDL_RWops
    SDL_RWops *res = SDL_RWFromConstMem(tmp->depart, tmp->taille);
    return res;
}
```

En principe, il n'y a rien de très particulier. La fonction **chercheZone** retourne un pointeur vers une structure **SZone**. Ensuite, nous testons sa valeur avant de passer les informations à la fonction **SDL_RWFromConstMem**.

IV.2.E - La fonction libereZone

La fonction **libereZone** fait l'inverse. Elle prend le nom du fichier à libérer et cherche dans la map si une zone associée est en mémoire. Si c'est le cas, on la libère sinon on ne fait rien.

La fonction chargeZone

```
void GestionFich::libereZone(string nom)
{
    //On regarde dans la map
    map<string,SZone*>::iterator iter = elements.find(nom);

    if(iter != elements.end()) {
        SZone *tmp = iter->second;
        //Gestion de l'erreur
        if(tmp != NULL) {
            //Liberation de la memoire
            unsigned char *ptr = (unsigned char*) tmp->depart;
            delete[] ptr;
            delete tmp;

            //On enleve de la map
            elements.erase(nom);
        }
    }
}
```

IV.2.F - La fonction chercheZone

Nous arrivons enfin à la plus grande fonction de cette classe : **chercheZone**. C'est elle qui va devoir lire le fichier d'archive si la ressource recherchée n'est pas présente.

On commence donc par vérifier si la ressource est présente :

La fonction chercheZone

```
SZone *GestionFich::chercheZone(string nom)
{
    //On regarde dans la map d'abord
    map<string,SZone*>::iterator iter = elements.find(nom);
```

Si jamais l'élément n'est pas présent, on va devoir parcourir l'entête. Pour rendre plus efficace la recherche, nous chargeons en mémoire l'entête la première fois que nous cherchons une ressource. Ensuite, nous aurons le tableau directement en mémoire.

Chargement de l'entête

```
if(iter == elements.end()) {
    //Si l'entete n'est pas encore chargee
    if(entete==NULL) {
        //Ouverture du fichier
        ifstream fich(nomarchive.c_str(), ios::in|ios::binary);

        //Gestion de l'erreur
        if(fich == NULL) {
            return NULL;
        }
    }
}
```

On commence par lire le nombre d'entrées et on va allouer le tableau **entete**. Vous remarquerez la facilité pour lire la chaîne de caractères que nous avons écrit pour ce nombre d'entrées. La classe **ifstream** la transforme directement en entier en ignorant les espaces que nous avons ajoutés.

Allocation du tableau

```
//Chargement de l'entete
fich >> nbr_entrees;

//Gestion de l'erreur
```

Allocation du tableau

```
if(nbr_entrees == 0) {
    return NULL;
}

//Allocation du tableau
entete = new SEntete[nbr_entrees];
```

Une fois le tableau alloué, le reste est assez facile, nous parcourons l'entête et nous remplissons le tableau.

Lecture de l'entête

```
//Lecture du fichier de l'entete
for(int i=0;i<nbr_entrees;i++) {
    fich >> entete[i].nomFichier;
    fich >> entete[i].position;
    fich >> entete[i].taille;
}
//Fermeture du fichier
fich.close();
}
```

Lorsque nous arrivons à ce point du code, nous savons que la ressource n'est pas encore en mémoire et que le tableau de l'entête l'est. Donc nous allons juste parcourir le tableau :

Lecture de l'entête

```
//Parcourir le tableau
for(int i=0; i < nbr_entrees; i++) {
    if(entete[i].nomFichier== nom) {
```

Une fois la bonne entrée trouvée, nous allons allouer une zone mémoire pour la structure **SZone** et nous allons ouvrir le fichier. Ensuite, après avoir bien positionné le curseur et alloué un tableau assez grand, nous allons extraire le fichier.

Allocation et lecture

```
SZone *zone = new SZone;

//Ouverture du fichier
ifstream fich(nomarchive.c_str(), ios::in|ios::binary);

//Cherche la position dans le fichier
fich.seekg(entete[i].position, ios::beg);

//Allocation d'une zone pour lire le fichier
char* buf = new char[entete[i].taille];

//Remplir la structure
zone->depart = buf;
zone->taille = entete[i].taille;

//Copier la zone du fichier
fich.read(buf, zone->taille);

//Fermer le fichier
fich.close();
```

La dernière étape consiste à ajouter la paire (nom du fichier, pointeur sur la zone) à la map **elements** et retourner la zone.

Ajout à la map et retour

```
//Ajoute au map
```

Ajout à la map et retour

```

        elements.insert(make_pair(nom, zone));

        //Retourner le pointeur vers la structure
        return zone;
    }
}

```

Il nous reste le cas où la ressource était déjà chargée et le cas d'erreur. Ces deux cas sont simples :

Fin de la fonction

```

    else {
        // On retourne l'element
        return iter->second;
    }

    //Sinon on n'a rien trouve
    return NULL;
}

```

Comme vous le voyez, cette fonction est assez grande mais très simple. Nous chargeons l'entête la première fois et ensuite nous allons l'utiliser pour trouver la ressource si elle n'est pas déjà en mémoire.

IV.3 - La création d'une classe de gestion

Une fois que nous avons un gestionnaire de fichier d'archive, nous allons ajouter un gestionnaire de ressources qui fera le lien entre les archives et le code du programme. La première grande raison de faire cela est pour partitionner le travail. Mais une autre grande raison est pour masquer quel type d'archive est utilisé. Le gestionnaire a simplement besoin d'un **SDL_RWops** pour faire son travail, il n'a pas forcément besoin de savoir comment cette ressource a été obtenue.

IV.3.A - La classe Gestionnaire

Voici la déclaration de la classe :

La classe Gestionnaire

```

class Gestionnaire
{
    private:
        GestionFich *gestion_entete;

    public:
        Gestionnaire();
        Gestionnaire(string nom);
        ~Gestionnaire();

        SDL_Surface *chargeBMP(string nom);
        void libere(string nom);
};

```

Cette classe est relativement petite pour le moment puisque nous avons simplement besoin d'un type de ressource : des fichiers BMP. Mais lorsque vous voudrez des fichiers textes, des fichiers sons et des fichiers vidéos, cela risque de se remplir rapidement.

Finalement, nous avons une fonction **chargeBMP** qui retourne directement une **SDL_Surface** vers l'image voulue et une fonction **libere** qui va appeler la fonction **libereZone** de la classe **GestionFich**.

IV.3.B - Les constructeurs et le destructeur

Le travail des constructeurs et du destructeur est simplement de créer et libérer une instance de type **GestionFich**.

La classe Gestionnaire

```
Gestionnaire::Gestionnaire()
{
    gestion_entete = new GestionFich();
}

Gestionnaire::Gestionnaire(string nom)
{
    gestion_entete = new GestionFich(nom);
}

Gestionnaire::~Gestionnaire()
{
    delete gestion_entete;
}
```

IV.3.C - Les fonctions chargeBMP et libere

Les fonctions **chargeBMP** et **libere** vont simplement faire le lien avec l'instance **gestion_entete**.

Certes, **chargeBMP** fait un peu plus, vu qu'il va créer une surface **SDL_Surface** en plus. Mais le code est relativement facile à comprendre :

Les fonctions chargeBMP et libere

```
SDL_Surface* Gestionnaire::chargeBMP(string nom)
{
    SDL_RWops *tmp = gestion_entete->chargeZone(nom);

    if(tmp != NULL) {
        return SDL_LoadBMP_RW(tmp, 1);
    }
    return NULL;
}

void Gestionnaire::libere(string nom)
{
    gestion_entete->libereZone(nom);
}
```

IV.4 - L'utilisation du gestionnaire

A partir de l'exemple de la section **III.2**, nous pouvons voir qu'il n'y a pas de grandes différences dans le main de ce programme. Nous avons ajouté une instance **gestion** de la classe **Gestionnaire** et lors du chargement des deux surfaces, nous appelons à présent la fonction de cette classe :

Le chargement

```
//Chargement des deux images et mise en place des positions
im1 = gestion.chargeBMP("image1.bmp");
im2 = gestion.chargeBMP("image2.bmp");
```

A la fin du programme, nous appelons la fonction **libere** bien que cela se ferait automatiquement dans le destructeur :

La destruction

```
//Parce qu'on veut faire proprement  
gestion.libere( "image1.bmp" );  
gestion.libere( "image2.bmp" );
```

Bien que cela se fait aussi simplement, vous voyez bien ce qui se passe en arrière plan, cela est loin d'être basique mais peut vraiment servir pour rendre le programme plus intéressant.

Vous trouverez le code source de cette version [ici](#).

V - Pour aller plus loin

La première chose importante à noter est que si nous demandons de charger plusieurs fois la même image, nous aurons deux structures **SDL_Surface** différentes. Ce sera le travail de l'utilisateur de les libérer. Un autre choix serait que le gestionnaire retourne le même pointeur vers la surface et conserve aussi cette information. Les deux sont possibles. Le choix dépendra de ce que vous cherchez.

Un autre point serait de dire : le problème avec un tel système est la quantité de mémoire qui va se retrouver utilisée en même temps. Cela est entièrement vrai. Si nous décidons que le gestionnaire va sauvegarder les pointeurs vers les ressources en interne et que l'utilisateur a simplement besoin de passer le nom avant chaque utilisation pour récupérer le pointeur, alors le gestionnaire pourrait faire du nettoyage de temps en temps pour alléger l'utilisation mémoire. Cela sort du cadre de ce tutoriel mais serait une solution intéressante.

V.1 - Utilisation d'une archive zip

La deuxième étape est d'utiliser directement des archives zip à la place de créer sa propre archive. Les avantages sont nombreux : l'archive sera compressée et nous avons la possibilité d'utiliser des dossiers. De plus, nous allons illustrer cela avec la bibliothèque `zzip` qui permet de faire des choses très intéressantes. Le fait d'avoir implémenté deux classes **GestionFich** et **Gestionnaire** fait que nous avons simplement à modifier la classe **GestionFich**, le reste du code restera identique.

Par rapport au code précédent, la seule fonction qui changera est **chercheZone**. Le reste de la classe reste intacte (à part la disparition des structures gérant les entêtes).

La fonction `chercheZone`

```
SZone *GestionFich::chercheZone(string nom)
{
    //On regarde dans la map d'abord
    map<string,SZone*>::iterator iter = elements.find(nom);

    if(iter == elements.end()) {
        //Ouverture du fichier
        nom = nomarchive + "/" + nom;
    }
}
```

La fonction commence de la même façon : recherche de la zone et, si elle n'existe pas, nous allons devoir la charger en mémoire. Par contre, nous allons devoir calculer le nom du fichier en ajoutant devant le nom de l'archive sans l'extension, suivi d'un '/' et du nom du fichier. Nous faisons cela pour préparer l'appel à la fonction **zzip_open**.

L'appel `zzip_open`

```
ZZIP_FILE *fich = zzip_open(nom.c_str(), 0);
```

Le premier paramètre à la fonction est le nom du fichier à ouvrir et le deuxième est les drapeaux (que nous mettrons toujours à 0). Finalement, pourquoi avoir fait cela ?

Imaginons que notre archive s'appelle **Donnees.zip** et qu'il contienne **image1.bmp**. Pour récupérer le fichier, nous devons passer la chaîne *Donnees/image1.bmp*.

Cela peut sembler bizarre mais c'est extrêmement pratique car si jamais il existe un dossier **Donnees** qui contient le fichier **image1.bmp**, la fonction `zzip_open` va simplement ouvrir ce fichier sans chercher une archive. Cela reviendra à ouvrir un fichier normalement comme nous l'avons fait dans le premier exemple !

Par contre, si le dossier ou le fichier n'existent pas, alors il va chercher dans l'archive **Donnees.zip** pour un fichier **image1.bmp**. Quelle est la conséquence de cette idée ? Le programmeur peut développer tout le programme avec un dossier **Donnees** et modifier les images, les sons sans devoir recréer l'archive à chaque modification. Une fois le programme terminé, il suffira de mettre toutes les données dans une archive **Donnees.zip** et le code **zzip** sera identique.

Il reste à remarquer que si le fichier **Donnees/image1.bmp** existe et une archive **Donnees.zip** contient **image1.bmp** en même temps, la fonction **zzip_open** donnera la priorité à la version normale et non archivée.

Le code continue par vérifier que le fichier s'est bien ouvert et ensuite nous calculons la taille du fichier lorsqu'il sera extrait :

Vérification d'erreur et calcul de taille

```
if(fich == NULL) {
    cerr << "Erreur de l'ouverture du fichier " << nom << endl;
    return NULL;
}

//Allocation de la zone
SZone *zone = new SZone;

//Cherche la position dans le fichier
zzip_seek(fich, 0, SEEK_END);
zzip_off_t taille = zzip_tell(fich);
zzip_seek(fich, 0, SEEK_SET);
```

Nous continuons par l'allocation, la lecture, et, ensuite, le code est identique :

Allocation, lecture et fin

```
//Allocation d'une zone pour lire le fichier
char* buf = new char[taille];
zzip_ssize_t lu = zzip_read(fich, buf, taille);
if((lu<0) || (lu<taille)) {
    cerr << "Erreur de la lecture " << nom << " , " << lu << " , " << taille << endl;
    delete zone;
    return NULL;
}

//Fermeture du fichier
zzip_close(fich);

//Remplir la structure
zone->depart = buf;
zone->taille = taille;

//Ajoute au map
elements.insert(make_pair(nom, zone));

//Retourner le pointeur vers la structure
return zone;
}
else {
    // On retourne l'element
    return iter->second;
}

//Sinon on n'a rien trouve
return NULL;
}
```

Comme vous le voyez, cela ne change pas grand chose par rapport à notre code précédent sauf que le tout est internalisé. L'avantage est bien sûr que dans notre version d'archive nous n'avions pas une version compressée ou une version acceptant des dossiers dans l'archive.

Par contre, cela oblige d'avoir une dépendance de plus, ce qui peut être gênant dans certains cas.

Vous trouverez le code source de cette version [ici](#).

VI - Téléchargements

Voici les différentes versions du programme présentées dans cet article :

La structure de l'entête

- 1 [Programme basique](#)
- 2 [Première utilisation de SDL_RWops](#)
- 3 [Création d'archive](#)
- 4 [Utilisation de notre archive](#)
- 5 [Utilisation de la bibliothèque zip](#)

VII - Conclusion

Dans cet article, j'ai tenté de montrer comment utiliser l'API **SDL_RWops** pour mettre toutes les ressources dans un seul fichier. Afin de garder les ressources privées et loin des mains des utilisateurs, il faudrait bien sûr crypter une partie de l'archive mais il faut noter qu'aucune mesure de sécurité n'est vraiment efficace lorsque quelqu'un veut vraiment savoir ce que contient un fichier...

Le but de cet article a donc été de montrer comment archiver les ressources et les utiliser avec la SDL. En passant par une archive créée en interne, vous avez vu comment gérer les ressources, une table d'entête. Ensuite, avec la bibliothèque zzip, nous avons montré comment il est facile d'avoir une version compressée des ressources.

VIII - Remerciements

Merci à **loka** pour la relecture et à **FabaCoeur**.

