Sérialisation avec UtilXML (février 2009) par Michel Michaud

UTILXML est un fichier d'en-tête et un fichier d'implémentation fournissant des fonctions pour faire une sérialisation en XML facilement. Pour en faire profiter nos programmes existants, on doit modifier les fonctions `Sérialiser`/`Désérialiser` des types `struct` ainsi que les fonctions générales de sérialisation (`Récupérer` et `Enregistrer`). En boni, on pourra enlever nos fonctions de sérialisation des types `enum`. Il n'est pas nécessaire de lire et comprendre le code de UTILXML pour l'utiliser, mais il devrait être compréhensible (à 95 %) si on a vu les 14 premiers chapitres du livre[1]. Pour utiliser UTILXML, on doit ajouter *UtilXML.cpp* à notre projet et inclure *UtilXML.h* au besoin. Les éléments étant placés dans le *namespace UtilXML*, on mettra aussi un `using namespace UtilXML;` dans nos fichiers sources (ce document suppose que c'est fait).

Fonction `Enregistrer`

- Il faut commencer par ouvrir le fichier normalement. Le nom du fichier devrait porter le suffixe *xml*.
- On utilise ensuite `ÉcrireEntêteXML` si on veut un vrai fichier XML standardisé, qu'on pourra ouvrir aisément dans Internet Explorer ou Visual Studio. Cette fonction reçoit simplement le fichier (le `ofstream`) en paramètre. Le fichier sera aussi le premier paramètre de la plupart des autres fonctions de UTILXML;
- On poursuit en mettant une balise globale (racine) avec `ÉcrireBaliseDébutFichier` qui permet aussi de mettre un numéro de version. Chaque appel d'une fonction `ÉcrireBaliseDébut`*`Xyz`* (on verra aussi `ÉcrireBaliseDébutStruct` plus loin) devra être complémenté par un appel de la fonction `ÉcrireBaliseFin`. On a donc, jusqu'à maintenant :

```
void EnregistrerInfoDémo(const string& p_nomFichier, ... /*données à sérialiser*/ )
    {
    ofstream ficDémo(p_nomFichier.c_str());            // Nom de fichier ayant le suffixe .xml

    ÉcrireEntêteXML(ficDémo);                         // Écrira <?xml version="1.0" ... ?>
    ÉcrireBaliseDébutFichier(ficDémo, "démo", "1");  // Écrira <démo version="1">

    // Sérialisation des données spécifiques...

    ÉcrireBaliseFin(ficDémo, "démo");                // Écrira </démo>
    }
```

- Pour la sérialisation des données spécifiques, il faut utiliser les fonctions de UTILXML prévues à cette fin et les fonctions de sérialisation de nos types `struct` que nous aurons faites ou refaites pour ça (on verra comment plus loin). Par exemple, pour sérialiser un `int`, on utilisera `SérialiserInt`, pour un `double`, `SérialiserDouble`, pour un `TypeDate` personnel, on aurait fait un fonction `SérialiserDate`. Les fonctions de base ont trois paramètres : le fichier, le nom de la balise et la valeur à sérialiser. Le nom de la balise devrait être le nom normal de la variable. Par exemple :

```
void EnregistrerInfoDémo(..., int p_nombre, double p_tauxTaxe, const TypeDate& p_dateDébut, ...)
    {
    ...
    SérialiserInt(ficDémo, "nombre", p_nombre);
    SérialiserDouble(ficDémo, "tauxTaxe", p_tauxTaxe);
    SérialiserDate(ficDémo, "dateDébut", p_dateDébut);
    ...
    }
```

  Les données de base seront sérialisées sous la forme `<`*`nomChamp`* `type="`*`son type`*`">`*`valeur`*`</`*`nomChamp`*`>`, par exemple, `<nombre type="int">123</nombre>`.

- Il existe des fonctions semblables pour les `int`, `long`, `double`, `char`, `bool` et `string`. Il existe aussi des fonctions pour les `enum` et les vecteurs, mais elles ont des paramètres supplémentaires.
- Pour les `enum`, il faut donner le nom du type spécifique, par exemple, pour `p_couleur` du `TypeCouleur`, on pourrait sérialiser ainsi :

```
SérialiserEnum(ficDémo, "couleur", "TypeCouleur", p_couleur);
```

  Dans ce cas, seule la valeur numérique est inscrite, par exemple `<couleur type="enum TypeCouleur">2</couleur>`. On peut aussi passer un paramètre supplémentaire pour avoir la valeur en texte aussi :

```
SérialiserEnum(ficDémo, "couleur", "TypeCouleur", p_couleur, CouleurEnTexte(p_couleur));
```

  Qui donnerait, par exemple, `<couleur type="enum TypeCouleur">2(Jaune)</couleur>`. C'est utile seulement si on désire lire le fichier nous-mêmes, car les fonctions de UTILXML ne feront rien (aucun validation) avec cette valeur.

[1] C'est par choix que le code a été écrit en utilisant le moins de possibilités de C++, afin qu'il soit utilisable rapidement tout en restant assez compréhensible. On aurait pu faire mieux en utilisant des classes par exemple.

- Pour les vecteurs, il faut spécifier le type des données ainsi que la fonction à utiliser pour sérialiser ces données. Cette fonction peut être une fonction de UTILXML ou une fonction personnelle (pour les `struct`). Par exemple, si `p_amis` est un `vector<string>` et `p_congés` est un `vector<TypeDate>`, on pourra les sérialiser ainsi :

```
SérialiserVector(ficDémo, "amis", "std::string", p_amis, SérialiserString);
SérialiserVector(ficDémo, "congés", "TypeDate", p_congés, SérialiserDate);
```

- Les vecteurs sont sérialisés sous la forme suivante :

```
<nomChamp type="std::vector"
    <taille type="long">taille du vecteur</taille>
    <éléments type="nomType">
       <élément type="nomType">valeur du vecteur[0]</élément>
       <élément type="nomType">valeur du vecteur[1]</élément>
       ...
    </éléments>
 </nomChamp>
```

Fonction `Récupérer`

- Il faut commencer par ouvrir le fichier normalement. Le nom du fichier devrait évidemment porter le suffixe *xml*.
- Il faut évidemment désérialiser/vérifier dans le même ordre que la sérialisation. Les fonctions générales comme `ÉcrireEntêteXML` ont des fonctions `Vérifier` associées, par exemple `VérifierEntêteXML`, alors que les fonctions `Sérialiser` ont des fonctions `Désérialiser` (pour les types `struct` personnels, on fera aussi des fonctions de ce genre). Par exemple, pour l'ensemble des exemples donnés jusqu'à maintenant, on pourrait avoir :

```
ifstream ficDémo(p_nomFichier.c_str());

VérifierEntêteXML(ficDémo);
VérifierBaliseDébutFichier(ficDémo, "démo", "1");
DésérialiserInt(ficDémo, "nombre", p_s_nombre);
DésérialiserDouble(ficDémo, "tauxTaxe", p_s_tauxTaxe);
DésérialiserDate(ficDémo, "dateDébut", p_s_dateDébut);
DésérialiserEnum(ficDémo, "couleur", "TypeCouleur", p_s_couleur);
DésérialiserVector(ficDémo, "amis", "std::string", p_s_amis, DésérialiserString);
DésérialiserVector(ficDémo, "congés", "TypeDate", p_s_congés, DésérialiserDate);
VérifierBaliseFin(ficDémo, "démo");
```

- En fait, on voudra probablement faire une gestion classique des erreurs. Lors de la détection d'une erreur, les fonctions de UTILXML lèvent une exception du `TypeErreurXML`. La fonction pourra donc avoir la structure suivante :

```
bool RécupérerInfoDémo(const string& p_nomFichier, int& p_s_nombre, double& p_s_tauxTaxe, ...)
    {
    ifstream ficDémo(p_nomFichier.c_str());

    if ( ! ficDémo.is_open())
        return false;

    try
        {
        VérifierEntêteXML(ficDémo);
        VérifierBaliseDébutFichier(ficDémo, "démo", "1");
        DésérialiserInt(ficDémo, "nombre", p_s_nombre);
        DésérialiserDouble(ficDémo, "tauxTaxe", p_s_tauxTaxe);
        ...
        VérifierBaliseFin(ficDémo, "démo");
        }
    catch (TypeErreurXML)
        {
        return false;
        }

    return true;
    }
```

Fonctions personnelles de sérialisation/désérialisation des `struct`

- Les fonctions doivent recevoir les trois paramètres classiques : fichier, nom du champ et valeur.
- Pour la sérialisation, on commence par inscrire la balise globale avec `ÉcrireBaliseDébutStruct`, on sérialise ensuite les champs avec les fonctions `Sérialiser` appropriées, puis on termine avec un `ÉcrireBaliseFin`. Par exemple, avec le `TypeDate` suivant

```
struct TypeDate
    {
    int jour;
    TypeMois mois;
    int année;
    };
```

on ferait la fonction de sérialisation suivante :

```
void SérialiserDate(ofstream& p_es_fic, const string& p_nomChamp, const TypeDate& p_date)
    {
    ÉcrireBaliseDébutStruct(p_es_fic, p_nomChamp, "TypeDate");
    SérialiserInt(p_es_fic, "jour", p_date.jour);
    SérialiserEnum(p_es_fic, "mois", "TypeMois", p_date.mois);
    SérialiserInt(p_es_fic, "année", p_date.année);
    ÉcrireBaliseFin(p_es_fic, p_nomChamp);
    }
```

- Pour la désérialisation, on commence par `VérifierBaliseDébutStruct`, on désérialise ensuite les champs avec les fonctions `Désérialiser` appropriées, puis on termine avec un `VérifierBaliseFin`. Par exemple :

```
void DésérialiserDate(ifstream& p_es_fic, const string& p_nomChamp, TypeDate& p_s_date)
    {
    VérifierBaliseDébutStruct(p_es_fic, p_nomChamp, "TypeDate");
    DésérialiserInt(p_es_fic, "jour", p_s_date.jour);
    DésérialiserEnum(p_es_fic, "mois", "TypeMois", p_s_date.mois);
    DésérialiserInt(p_es_fic, "année", p_s_date.année);
    VérifierBaliseFin(p_es_fic, p_nomChamp);
    }
```

Autres fonctions de UTILXML

- Par défaut, le fichier XML est formaté de façon classique avec une indentation de 3. On peut modifier l'indentation en appelant la fonction `ChangerIndentation` avant la sérialisation. Par exemple, `ChangerIndentation(5)` pourrait donner le fichier suivant (où on aurait sérialisé une seule date) :

```
<?xml version="1.0" encoding="ISO-8859-1" standalone="yes"?>
<démo version="1.0">
     <prochainNoël type="struct TypeDate">
          <jour type="int">25</jour>
          <mois type="enum TypeMois">12</mois>
          <année type="int">2007</jour>
     </prochainNoël>
</démo>
```

- On peut aussi désactiver complètement le formatage en appelant `DésactiverFormatage()` avant la sérialisation. Le résultat sera alors sur une seule ligne sans espace ajouté. Par exemple :

```
<?xml version="1.0" encoding="ISO-8859-1" standalone="yes"?><démo version="1.0"><prochainNoël ...
```

  Cette fonction pourrait être utile si le programme sérialise dans deux fichiers XML en même temps, car la gestion du formatage ne serait pas vraiment fonctionnelle dans ce cas, puisque les calculs de niveaux sont faits globalement.

- La fonction `ChangerIndentation` réactive le formatage au besoin.

Encodage et décodage

- Les fonctions `SérialiserChar`/`SérialiserString`/`DésérialiserChar`/`DésérialiserString` s'occupe d'encoder et décoder les caractères problématiques en XML (principalement `<` et `&`, mais les `>` `"` et `'` sont aussi codés). Par exemple, `SérialiserString(ficDémo, "menu", "A&W")` donnerait `<menu type="std::string">A&amp;W</menu>`.

- Au besoin, les fonctions `EncoderXML`/`DécoderXML` sont disponibles (voir le fichier d'en-tête pour les détails), mais elles ne devraient pas être utiles pour la sérialisation/désérialisation puisque l'encodage/décodage est fait automatiquement par les fonctions de base.

Les fichier d'en-tête et d'implémentation donnent de nombreuses explications supplémentaires dans des commentaires. Voir aussi le programme *DemoXML* pour des exemples et d'autres explications.