Utilisation de la bibliothèque TinyXML

Par khayyam90

Date de publication : 26 janvier 2006

Dernière mise à jour : 15 janvier 2007

Cet article montre l'utilisation de la bibliothèque TinyXML pour lire, écrire et modifier des fichiers XML en C++. Je ne vais pas faire le détail des classes mises en jeu, la doc officielle est là pour ça. Je vais présenter les méthodes d'accès et de modification de données. Je vais illustrer tout ça par un exemple de classe de gestion d'utilisateurs. Ce n'est pas un article sur le XML, je ne détaillerai donc pas les normes XML.

I - Installation

Tout au long de ce document, je vais parler (entre autres) de DOM et d'XML. En voici quelques précisions :

Définition du W3C

Le DOM (Document Object Model) définit une méthode standard pour accéder à des données et pour les manipuler. Le DOM représente un document XML sous la forme d'un arbre.

Extrait de l'introduction à XML

XML est l'abréviation de Extensible Markup Language. Contrairement aux « on dit », XML n'est pas un langage de programmation. On ne peut pas faire de tests, ni inclure un fichier dans un autre. En très gros, XML sert uniquement à stocker des données. XML est simplement une méthode pour représenter les données. Celles-ci sont écrites entre des balises ou sous forme d'attributs, et l'ensemble est écrit sous forme d'un arbre.

TinyXML est une bibliothèque Open Source de gestion de fichiers XML. Elle est distribuée sous la licence zlib :

- vous avez le droit d'utiliser, modifier et redistribuer le code source gratuitement ou non
- l'origine des sources ne doit pas etre masquée
- Les modifications que vous apportez doivent être clairement visibles pour ne pas entrainer de confusion avec la version originale

Plus d'infos sur cette licence **ici**.

TinyXML est utilisable sur les environnements Windows 32 bits, sur tous les systèmes POSIX (BSD, Unix) et sur Linux. Les sources sont disponibles sur le site de développement **http://sourceforge.net/projects/tinyxml**. TinyXML est également disponible en version précompilée pour MinGW sous forme de DevPack.

La bibliothèque peut être compilée de deux manières, en utilisant ou non la STL, ainsi le DevPack contient un fichier libtinyxml.a et un fichier libtinyxmlstl.a. Dans le cas où vous décidez d'utiliser la version STL, vous devez définir la variable de compilation TIXML_USE_STL, soit par la ligne de commande du compilateur soit en la rajoutant dans le code source de votre application. L'option pour l'édition de liens sera donc elle aussi différente : -ltinyxml ou bien -ltinyxmlstl. Ne vous trompez pas !

Donc pour résumer, pour utiliser TinyXML, il faut

- la compiler avec votre compilateur favori ou bien récupérer une version précompilée
- copier tinyxml.h et tinystr.h dans le répertoire include de votre compilateur (ou à tout autre endroit que le compilateur saura trouver)
- copier le fichier .a ou .lib (qui dépend si vous utilisez ou non la STL) dans le répertoire lib de votre compilateur (ou à tout autre endroit que le compilateur saura trouver)
- lier la bibliothèque dans tous les projets qui l'utiliseront

II - Récupération d'informations à partir d'un fichier XML

L'utilisation de TinyXML est très simple, il suffit de quelques lignes pour récupérer les informations d'un fichier XML. Je vais continuer dans la lancée de **mon précédent article sur les serveurs multi-threads** et orienter mes exemples dans ce sens : je vais présenter les fonctions de gestion des utilisateurs autorisés à acceder à un serveur. Mais soyons d'accord, dans le cas d'une grande liste d'utilisateurs, une solution plus efficace serait d'interroger une base de données.

Je choisis de charger la liste des utilisateurs lors du démarrage du serveur. Notez qu'on pourrait très bien ne rien charger et aller interroger le fichier XML à chaque opération. Chaque utilisateur possède un nom et un mot de passe ainsi qu'un indice définissant ses droits. Le fichier xml que nous avons à gérer a donc la forme suivante:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<lstUsers>
 <user name="toto" pass="13" indice="1"/>
 <user name="tata" pass="142" indice="2"/>
 <user name="titi" pass="azerty" indice="1"/>
</lstUsers>
```

Notre fichier XML possède un noeud "lstUsers" comportant 3 enfants, chacun étant un noeud "user". Et chaque user comporte 3 attributs. On pourrait aussi tout à fait imaginer d'autres attributs ou bien une imbrication des informations avec une hiérarchie plus développée, ca ne pose aucun probleme.

On commence le programme par inclure le header de la bibliothèque:

```
#include <tinyxml.h>
```

Pour simplifier l'écriture et ne pas avoir à écrire std:: devant tous les élements des bibliothèques standards (voir la **FAQ** pour plus de précisions) rajoutons la ligne :

```
using namespace std;
```

L'une des particularités intéressantes de TinyXML est qu'il reconnaît automatiquement l'encodage d'un fichier XML. Ainsi, il n'est pas nécessaire de renseigner l'attribut "encoding". Cela vous permettra par exemple d'extraire des données comportant des caractères spéciaux (accents, caractères chinois...). Et dans la série des petits "plus" qui sont agréables à utiliser, on retrouve l'indentation automatique des fichiers XML enregistrés.

La force de TinyXML réside dans sa légèreté et dans sa simplicité d'utilisation. Ainsi il pourra se révéler moins efficace, en terme de rapidité ou d'occupation mémoire, que d'autres outils d'analyse XML plus importants. C'est pourquoi TinyXML n'est pas forcément l'outil le mieux adapté pour la lecture de gros fichiers ; il reste toutefois idéal pour lire des petits fichiers XML (fichiers de configuration par exemple).

II-1 - Le chargement du fichier XML

Le principe est le suivant : on charge le fichier xml, on vérifie que le chargement s'est bien passé, on se positionne dans le noeud "lstUsers", puis on parcourt ce noeud en extrayant de chaque noeud enfant les attributs qui nous intéressent. Et d'un point de vue pratique, j'ai décidé de stocker toutes les informations recueillies dans une liste std::list. La manipulation des informations ultérieurement n'en sera que plus simple.

Le côté fortement orienté objet de la bibliothèque va nous aider : en effet, le document est considéré comme un noeud xml au même titre que le noeud "lstUsers" ou même le noeud "user", nous avons donc un grand nombre de fonctions membres héritées de la classe 'noeud' (TiXmlNode) qui se retrouvent virtuelles et présentes dans toutes les classes filles.

```
TiXmlDocument doc("users.xml");
if(!doc.LoadFile()){
    cerr << "erreur lors du chargement" << endl;
    cerr << "error #" << doc.ErrorId() << " : " << doc.ErrorDesc() << endl;
    return 1;
}
```

Tout d'abord on crée un objet du type TiXmlDocument qui va représenter notre document XML. Celui-ci est construit à partir du nom du fichier XML. L'étape suivante est tout simplement le chargement du fichier. C'est dans cette fonction qu'on va retrouver la lecture du fichier et la construction de l'arbre DOM représentatif du fichier XML. Et dans le cas où le chargement se passe mal (fichier introuvable ?) on récupère le numéro et l'intitulé du message d'erreur, respectivement dans doc.ErrorId() et dans doc.ErrorDesc().

Jusque là, c'est pas trop compliqué, passons maintenant à l'extraction des données :

II-2 - L'extraction des données

On cherche à parcourir la liste des utilisateurs, on va donc se positionner dans le noeud lstUsers, et on va boucler sur tous ses enfants. Pour celà, nous allons utiliser un pointeur vers un objet de l'arbre DOM, que nous allons déplacer d'un enfant à l'autre. Ce pointeur est du type TiXmlElement, il hérite de la classe TiXmlNode. En effet, un élément de l'arbre peut lui aussi être un noeud.

Et à chaque nouvel utilisateur rencontré, nous allons enrichir une liste. Il nous faut donc définir la notion d'utilisateur au niveau de C++. Ne cherchant pas la complication, créons une classe basique:

```
class user{
 public:
  string name, pass;
  int droits;
};
```

Et nous stockerons les informations extraites dans une list<user>.

```
list<user> user_list;
```

Il nous faut maintenant positionner notre pointeur d'élément sur le début de la liste des utilisateurs. Le premier noeud de notre document est le noeud "lstUsers". Plaçons nous sur le premier noeud "user" qu'il contient. La solution la plus évidente est

```
TiXmlElement *elem = doc.FirstChildElement()->FirstChildElement();
```

En effet, nous nous plaçons sur le premier noeud (lstUsers) puis sur son premier enfant (le noeud user name="toto" pass="13" indice="1"). Mais dans le cas où aucun noeud user n'existe, on se retrouvera avec un pointeur pointant sur un objet qui n'existe pas, ce qui va compromettre la suite du programme. C'est pourquoi, et la doc officielle l'explique bien, il est préférable d'utiliser des handles pour se déplacer dans notre arbre DOM. L'effet sera exactement le même, la sécurité en plus:

```
TiXmlHandle hdl(&doc);
TiXmlElement *elem = hdl.FirstChildElement().FirstChildElement().Element();
```

Il nous suffit ensuite de tester la valeur de elem (égale ou non à NULL) pour savoir si l'objet pointé existe. Ensuite, c'est parti, on peut parcourir tous les noeuds :

```
user cl;
if(!elem){
 cerr << "le noeud à atteindre n'existe pas" << endl;
 return 2;
}
```

```
while (elem){
 cl.name = elem->Attribute("name");
 cl.pass = elem->Attribute("pass");
 elem->QueryIntAttribute("indice", &cl.droits);
 user_list.push_back(cl);

 elem = elem->NextSiblingElement(); // iteration
}
```

On notera au passage l'utilisation de la fonction membre QueryIntAttribute pour récupérer la valeur de l'attribut "indice" sous forme d'un entier et non d'une chaîne de caractères.

Et pour vous convaincre, on passe en revue la liste après l'extraction:

```
list<user>::iterator i;
for(i=user_list.begin(); i!=user_list.end(); i++)
    cout << i->name << " " << i->pass << " " << i->droits << endl;
```

```
toto 13 1
tata 142 2
titi azerty 1
```

Bon, maintenant un utilisateur veut changer son mot de passe, comment fait-on ? On pourrait arrêter le serveur, éditer le fichier XML à la main et relancer le serveur, mais nous avons mieux : la modification (Waaaaaaa).

III - Modifications d'un fichier XML

Pour modifier un utilisateur il va falloir se positionner correctement dans l'arbre DOM puis changer un ou plusieurs attributs. On va donc boucler sur les utilisateurs jusqu'à trouver celui qui nous intéresse puis on va le changer. Et vous n'oublierez pas de mettre a jour la list<user>.

Je me sers donc d'un booléen pour quitter la boucle de parcours, mon itérateur sera ainsi positionné sur le noeud à modifier. La fonction membre SetAttribute nous permet ensuite de changer la valeur d'un attribut donné. Dans cet exemple, j'ai choisi de changer le mot de passe de l'utilisateur "tata" par "nouv".

```
bool trouve = false;
TiXmlHandle hdl(&doc);
TiXmlElement *elem = hdl.FirstChildElement().FirstChildElement().Element();

while(elem && !trouve){
    if( string(elem->Attribute("name")) == "tata"){
        trouve = true;
        break;
    }
    elem = elem->NextSiblingElement(); // iteration, passage a l'element suivant
}

if (!trouve)
    cerr << "user inexistant" << endl;
else{
    elem->SetAttribute("pass", "nouv");
    doc.SaveFile("users.xml");  // enregistrement de la modification
}
```

On notera aussi la conversion en string de elem->Attribute("name") pour pouvoir effectuer une comparaison entre deux chaînes de caractères.

Et maintenant un peu plus dur, supprimer des membres dans l'arbre DOM. Mais non, c'est pas si difficile, TinyXML faut tout pour vous.

IV - Suppression de noeuds

La suppression de noeuds s'effectue de la manière suivante, on boucle sur sur tous les noeuds frères jusqu'à trouver celui à supprimer. Et ensuite on appelle la fonction de suppression en spécifiant le noeud qui va perdre un fils. Il va donc nous falloir un pointeur sur l'élément à supprimer et un pointeur sur l'élément parent. Tous les deux seront du type TiXmlElement.

Dans cet exemple, nous supprimons l'utilisateur nommé "tata".

```
bool trouve = false;
TiXmlHandle hdl(&doc);
TiXmlElement *elem = hdl.FirstChildElement().FirstChildElement().Element();

while(elem && !trouve){
    if( string(elem->Attribute("name")) == "tata"){
        trouve = true;
        break;
    }
    elem = elem->NextSiblingElement();
}

if (!trouve)
    cerr << "user à supprimer inexistant" << endl;
else{
    TiXmlElement *f = doc.FirstChildElement();
    f->RemoveChild(elem);
    doc.SaveFile("users.xml");
}
```

On notera la présence d'un break dans la boucle de parcours pour éviter d'incrémenter le pointeur sur l'objet à supprimer une fois qu'il a été trouvé.

L'objet *f correspond au noeud <lstUsers>, qui contient le noeud <user>. C'est l'objet parent qui va invoquer la fonction de suppression. Et de même que pour la modification d'un noeud, on enregistre le fichier après la suppression.

V - Ajout de noeuds

Autre opération nécessaire, l'ajout d'un nouvel utilisateur. Deux possibilités, on peut insérer le nouvel élément à la fin de l'arborescence ou bien à un endroit particulier. Je vais traiter les deux cas.

V-1 - Ajout à un emplacement précis

Ajoutons un utilisateur juste après l'utilisateur nommé "tata". On va donc parcourir notre arbre jusqu'à trouver l'utilisateur nommé "tata". Et on y ajoutera le nouvel element. De même que pour la modification d'un noeud, c'est le noeud parent qui doit invoquer la fonction d'ajout. Il nous faudra donc un pointeur vers ce noeud parent.

La création d'un nouvel utilisateur demande la création d'une nouvelle instance de la classe TiXmlElement. Le constructeur de la classe prend en argument le type du noeud ("user"). Et les différents attributs seront renseignés par la fonction membre SetAttribute.

```
bool trouve = false;
TiXmlHandle hdl(&doc);
TiXmlElement *elem = hdl.FirstChildElement().FirstChildElement().Element();

while(elem && !trouve){
    if( string(elem->Attribute("name")) == "tata"){
        trouve = true;
        break;
    }
    elem = elem->NextSiblingElement();
}

if (!trouve)
    cerr << "user inexistant" << endl;
else {
    TiXmlElement *f = doc.FirstChildElement(); // on récupère le noeud parent
    TiXmlElement le_nouveau ("user");
    le_nouveau.SetAttribute("name", "tutu");
    le_nouveau.SetAttribute("pass", "le_pass");
    le_nouveau.SetAttribute("rank", "4");

    f->InsertAfterChild(elem, le_nouveau);
    doc.SaveFile("users.xml");  // enregistrement des modifications
}
```

Dans cet exemple, nous avons inséré notre nouvel élément après un élément precis. Il en existe la fonction duale permettant d'insérer un élément avant un autre, elle fonctionne de la même manière : InsertBeforeChild.

V-2 - Ajout en fin d'arborescence

L'ajout en fin d'arborescence est plus simple car nous n'avons pas à rechercher un noeud précis après lequel insérer notre nouvel élément.

De même que pour l'ajout à un endroit précis et la suppression, c'est le noeud parent qui doit invoquer la fonction d'insertion. Ce noeud est pointé par f.

```
TiXmlElement *f = doc.FirstChildElement();
TiXmlElement le_nouveau ("user");
le_nouveau.SetAttribute("name", "tutu");
le_nouveau.SetAttribute("pass", "le_pass");
le_nouveau.SetAttribute("indice", "4");

f->InsertEndChild(le_nouveau);
doc.SaveFile("users.xml");  // enregistrement des modifications
```

VI - Implémentation d'une classe de gestion d'utilisateurs

Voici mon implémentation d'une classe de gestion d'utilisateurs. Cette classe implémente les opérations de chargement, de modification, de suppression et d'ajout d'utilisateurs.

- **user_mgr.cpp**
- **user.h**
- **user_mgr.h**

VII - Conclusion

Vous l'aurez compris, TinyXML se veut très facile à prendre en main. Il permet de gérer des données XML très simplement. Par contre, il ne permet pas de faire de la validation de fichiers par rapport à une grammaire. A chaque utilisation son outil, celui-là étant très léger, il trouvera facilement sa place dans des applications aussi diverses que variées.

VIII - Liens et remerciements

Et voici quelques liens connexes :

- **De la documentation sur le XML**
- **La doc officielle de TinyXML**
- **Le site officiel de TinyXML**

Je tiens à remercier toute l'équipe de la section C/C++ en particulier **Loulou24**, **Aurelien.Regat-Barrel** et **Ti-R** pour la relecture de cet article.