Je pratique le C++

Partie 5/5

Nous vous proposons une série d'articles sur la pratique de C++ pour que vous puissiez tous vous y mettre. Ce mois-ci on aborde la librairie Boost du site www.boost.org. Boost est une communauté de programmeurs C++ dont le but est de pousser les librairies qu'ils font vers la standardisation C++. Boost est une librairie C++ portable. Vous pouvez l'utiliser avec Visual C++ sur Windows et GCC ou CLang sur Linux.

Christophe PICHAUD | .NET Rangers by Sogeti
Consultant sur les technologies Microsoft
christophepichaud@hotmail.com | www.windowscpp.net

Comment obtenir Boost ?

Très simplement en allant dans la rubrique download du site et vous téléchargez un zip de 120 Mb ou un targz de 80 Mb. Vous décompressez votre archive et vous êtes presque prêt. La décompression vous donne une arborescence de 430 Mb. Il y a énormément de documentation HTML c'est la raison pour laquelle la taille du dossier est conséquente.

Compilation de Boost

Malgré le fait que Boost soit en majorité un ensemble de templates et de fonctions inline - donc juste les fichiers d'en-têtes suffisent - il y a des librairies qu'il faut compiler.

La première chose à faire est de lancer la compilation de la librairie. Cela se fait en deux étapes. Premièrement, il faut compiler le custom Make de Boost qui se nomme Bjam. Il faut lancer bootstrap.bat. Maintenant que Bjam est compilé, on va builder la librairie avec notre chaîne de compilation particulière à une plateforme. Je travaille avec Visual Studio 2013 donc voici ma ligne de commande :

```
bjam toolset=msvc-12.0 variant=debug,release threading=multi link=shared
```

Je fais une compilation en debug et en release et indiquant que je veux une librairie multi-thread et en mode dynamique (DLL). La compilation prend un peu de temps, mais disons qu'en 15 minutes c'est terminé.

Introduction à Boost.Serialization

Maintenant que nous avons listé l'ensemble des librairies disponibles dans Boost, nous allons passer au code ! Examinons une librairie très utile qui sait sérialiser et dé-sérialiser des classes en format binaire ou XML sans forcer. Cette librairie permet de sérialiser des containers, ce qui évitera de parcourir nos différentes collections pour sérialiser des éléments simples. Par contre, nous devons faire attention à une collision de nom dans l'espace de noms disponible. Je m'explique... Boost sérialise les boost ::string, les boost ::vector & co... Ce qui veut dire qu'il va falloir transvaser nos objets écrits avec la STL standard dans des objets Boost compatibles à moins que notre application soit écrite complètement avec Boost. C'est un choix d'architecture ! Nous allons partir sur une application que j'ai écrite pour ma fille en 2012. Il s'agit d'une application de dessin où l'on dispose des éléments prédéfinis sur une page comme des images, des rectangles, des ronds, des triangles, etc. Cette application a permis à ma fille de 8 ans à l'époque de savoir manier la souris et de jouer avec le Ribbon pour changer les caractéristiques des objets (couleur, épaisseur de traits, etc.).

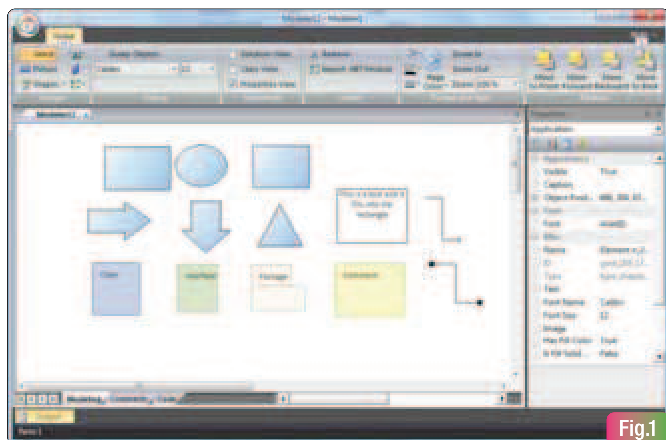


Fig.1

Le résultat de la sérialisation XML

Commençons par la fin... Le dessin est le suivant : Fig.1.

Le but du jeu est de sauvegarder ce dessin sous forme de document XML Fig.2.

Ouvrons la documentation de Boost.Serialization pour prendre la chose du bon côté et pour arriver au résultat ci-dessus. Si vous avez l'œil, vous verrez que l'application que je vous propose est réalisée avec les Microsoft Foundation Classes (MFC) car elle me permet de faire un joli Ribbon et de mettre en place le support du Document/Vue qui me permet de faire une application qui charge/enregistre mes données. Bref, MFC rocks. Je crois d'ailleurs qu'on va refaire une série Je Pratique le C++ sous Windows en 5 épisodes pour que vous puissiez développer vous aussi des applications qui rockent ! On verra... revenons à nos moutons. Le XML. Boost.Serialization fonctionne avec ce que l'on appelle une Archive dans laquelle on a accès à des données. Il faut nourrir l'archive avec des données. J'ai donc une classe d'objets à dessiner et des objets. Le modèle de données est constitué de 2 classes : CSimpleShape et CShapeCollection.

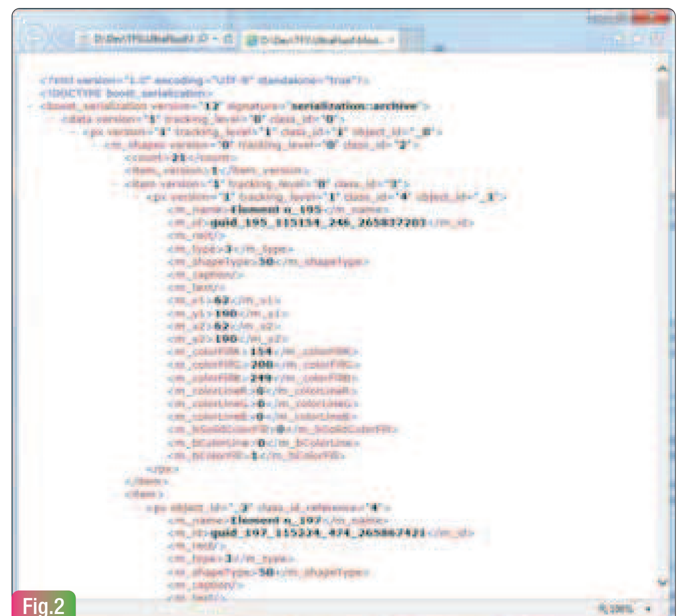


Fig.2

La classe CSimpleShape

Cette classe contient les propriétés suivantes :

```
public:
  wstring m_name;
  wstring m_id;
  string m_rect;
  long m_type;
  long m_shapeType;
  wstring m_caption;
  wstring m_text;
  long m_x1;
  long m_y1;
  long m_x2;
  long m_y2;
  int m_colorFillR;
  int m_colorFillG;
  int m_colorFillB;
  int m_colorLineR;
  int m_colorLineG;
  int m_colorLineB;
  bool m_bSolidColorFill;
  bool m_bColorLine;
  bool m_bColorFill;
};

BOOST_CLASS_VERSION(CSimpleShape, 1)
```

La classe CShapeCollection

Cette classe contient la liste des elements :

```
public:
  vector<boost::shared_ptr<CSimpleShape> > m_shapes;
};

BOOST_CLASS_VERSION(CShapeCollection, 1)
```

C'est un vecteur de CSimpleShape tout simplement... Pour que ces éléments soient stockés sous forme de fichier texte, binaire ou XML il faut préciser quelques petites choses dans les classes. Il faut spécifier cela en private :

```
private:
  friend class boost::serialization::access;
  template<class Archive>
  void save(Archive & ar, const unsigned int version) const
  {
     ar & BOOST_SERIALIZATION_NVP(m_shapes);
  }

  template<class Archive>
  void load(Archive & ar, const unsigned int version)
  {
     ar & BOOST_SERIALIZATION_NVP(m_shapes);
  }

  BOOST_SERIALIZATION_SPLIT_MEMBER()
```

Cette partie de code nous montre le template à définir. Il s'agit d'une fonction template load et save qui travaille sur une Archive. La syntaxe pour enrôler une donnée dans l'archive se fait au travers de l'opérateur &. Cette partie de code enregistre le vecteur d'éléments. Mais pour que cela fonctionne, il faut aussi que le classe CSimpleShape enregistre ses éléments dans l'archive.

```
  template<class Archive>
  void save(Archive & ar, const unsigned int version) const
  {
     // ar & name;
     // ar & id;
     ar & BOOST_SERIALIZATION_NVP(m_name);
     ar & BOOST_SERIALIZATION_NVP(m_id);
     ar & BOOST_SERIALIZATION_NVP(m_rect);
     ar & BOOST_SERIALIZATION_NVP(m_type);
     ar & BOOST_SERIALIZATION_NVP(m_shapeType);
     ar & BOOST_SERIALIZATION_NVP(m_caption);
     ar & BOOST_SERIALIZATION_NVP(m_text);
     ar & BOOST_SERIALIZATION_NVP(m_x1);
     ar & BOOST_SERIALIZATION_NVP(m_y1);
     ar & BOOST_SERIALIZATION_NVP(m_x2);
     ar & BOOST_SERIALIZATION_NVP(m_y2);
     ar & BOOST_SERIALIZATION_NVP(m_colorFillR);
     ar & BOOST_SERIALIZATION_NVP(m_colorFillG);
     ar & BOOST_SERIALIZATION_NVP(m_colorFillB);
     ar & BOOST_SERIALIZATION_NVP(m_colorLineR);
     ar & BOOST_SERIALIZATION_NVP(m_colorLineG);
     ar & BOOST_SERIALIZATION_NVP(m_colorLineB);
     ar & BOOST_SERIALIZATION_NVP(m_bSolidColorFill);
     ar & BOOST_SERIALIZATION_NVP(m_bColorLine);
     ar & BOOST_SERIALIZATION_NVP(m_bColorFill);
  }
```

Le code du load et du save est le même. L'archive sait dans quel sens on travaille, soit en lecture, soit en écriture. Il faut noter que les MFC fournissent une infrastructure similaire avec les opérateurs << et >> ; je trouve que c'est plus lisible à mon goût, mais bon. Pour pouvoir compiler ce code, il faut disposer des en-têtes suivantes :

```
#define BOOST_SERIALIZATION_DYN_LINK TRUE
#define BOOST_ALL_DYN_LINK TRUE

#include <boost/smart_ptr/shared_ptr.hpp>
#include <boost/archive/tmpdir.hpp>
#include <boost/serialization/nvp.hpp>

#include <boost/archive/xml_iarchive.hpp>
#include <boost/archive/xml_oarchive.hpp>
#include <boost/archive/text_iarchive.hpp>
#include <boost/archive/text_oarchive.hpp>
#include <boost/serialization/shared_ptr.hpp>
#include <boost/serialization/base_object.hpp>
#include <boost/serialization/string.hpp>
#include <boost/serialization/list.hpp>
#include <boost/serialization/vector.hpp>
#include <boost/serialization/map.hpp>
#include <boost/serialization/utility.hpp>
#include <boost/serialization/assume_abstract.hpp>
using namespace boost;
```

L'écriture des données

Maintenant que nous avons nos classes qui savent utiliser une Archive, voyons le code qui donne l'ordre de faire Load ou Save. Vous allez voir, c'est très simple :

```
boost::shared_ptr<CShapeCollection> data(new CShapeCollection());
for( vector<std::shared_ptr<CElement>>::iterator i = m_objects.m_objects.begin() ; i!
=m_objects.m_objects.end() ; i++ )
{
    std::shared_ptr<CElement> pElement = *i;
    boost::shared_ptr<CSimpleShape> pNewElement(new CSimpleShape());
    pNewElement->m_name = pElement->m_name;
    pNewElement->m_id = pElement->m_objectId;
    pNewElement->m_type = pElement->m_type;
    pNewElement->m_shapeType = pElement->m_shapeType;
    pNewElement->m_caption = pElement->m_caption;
    pNewElement->m_text = pElement->m_text;

    CPoint p1 = pElement->m_rect.TopLeft();
    CPoint p2 = pElement->m_rect.BottomRight();
    pNewElement->m_x1 = p1.x;
    pNewElement->m_y1 = p1.y;
    pNewElement->m_x2 = p2.x;
    pNewElement->m_y2 = p2.y;

    pNewElement->m_colorFillR = GetRValue(pElement->m_colorFill);
    pNewElement->m_colorFillG = GetGValue(pElement->m_colorFill);
    pNewElement->m_colorFillB = GetBValue(pElement->m_colorFill);
    pNewElement->m_colorLineR = GetRValue(pElement->m_colorLine);
    pNewElement->m_colorLineG = GetGValue(pElement->m_colorLine);
    pNewElement->m_colorLineB = GetBValue(pElement->m_colorLine);

    pNewElement->m_bSolidColorFill = pElement->m_bSolidColorFill;
    pNewElement->m_bColorLine = pElement->m_bColorLine;
    pNewElement->m_bColorFill = pElement->m_bColorFill;

    data->m_shapes.push_back(pNewElement);
}

std::ofstream xofs(filename.c_str());
boost::archive::xml_oarchive xoa(xofs);
xoa << BOOST_SERIALIZATION_NVP(data);
```

La variable filename est remplie par l'ouverture d'une Common Dialog Windows SaveAs et toute la magie consiste à déclarer une xml_archive et à utiliser l'opérateur << pour sauver toutes nos données.

La lecture des données

La lecture des données est calquée sur l'écriture, on récupère les données et on les transvase dans la collection qui va s'afficher à l'écran :

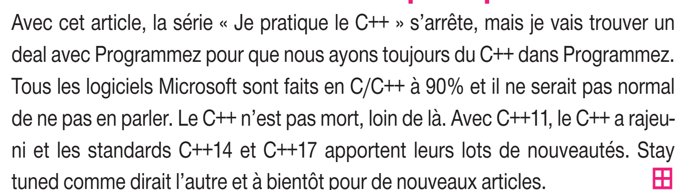
```
boost::shared_ptr<CShapeCollection> data(new CShapeCollection());
// load an archive
std::ifstream xifs(filename.c_str());
assert(xifs.good());
boost::archive::xml_iarchive xia(xifs);
xia >> BOOST_SERIALIZATION_NVP(data);
// Clear existing shapes
m_objects.RemoveAll();
```

```
for( vector<boost::shared_ptr<CSimpleShape> >::iterator i = data->m_shapes.begin() ; i!
=data->m_shapes.end() ; i++ )
{
    boost::shared_ptr<CSimpleShape> pElement = *i;
    //AfxMessageBox(pElement->m_name + " " + pElement->m_id);

    std::shared_ptr<CElement> pNewElement = CFactory::CreateElementOfType((Element
Type)pElement->m_type,
                                        (ShapeType)pElement->m_shapeType);
    pNewElement->m_name = pElement->m_name.c_str();
    pNewElement->m_objectId = pElement->m_id.c_str();
    pNewElement->m_caption = pElement->m_caption.c_str();
    pNewElement->m_text = pElement->m_text.c_str();
    pNewElement->m_pManager = this;
    pNewElement->m_pView = pView;

    CPoint p1;
    CPoint p2;
    p1.x = pElement->m_x1;
    p1.y = pElement->m_y1;
    p2.x = pElement->m_x2;
    p2.y = pElement->m_y2;
    pNewElement->m_rect = CRect(p1, p2);

    int colorFillR = pElement->m_colorFillR;
    int colorFillG = pElement->m_colorFillG;
    int colorFillB = pElement->m_colorFillB;
    pNewElement->m_colorFill = RGB(colorFillR, colorFillG, colorFillB);
    int colorLineR = pElement->m_colorLineR;
    int colorLineG = pElement->m_colorLineG;
    int colorLineB = pElement->m_colorLineB;
    pNewElement->m_colorLine = RGB(colorLineR, colorLineG, colorLineB);

    pNewElement->m_bSolidColorFill = pElement->m_bSolidColorFill;
    pNewElement->m_bColorLine = pElement->m_bColorLine;
    pNewElement->m_bColorFill = pElement->m_bColorFill;

    m_objects.AddTail(pNewElement);
    pView->LogDebug(_T("object created ->") + pNewElement->ToString());
}

// Redraw the view
Invalidate(pView);
```

Conclusion

L'utilisation de Boost et de Boost.Serialization est plutôt simple à appréhender. La documentation fournie est de bonne facture et les exemples sont légion dans la documentation. L'avantage de Boost.Serialization est que les archives sont portables... Donc pour faire du multiplateforme, c'est easy ! Le code complet de l'application de dessin est disponible ici : http://ultrafluid.codeplex.com .

Conclusion sur la série « Je pratique C++ »



Avec cet article, la série « Je pratique le C++ » s'arrête, mais je vais trouver un deal avec Programmez pour que nous ayons toujours du C++ dans Programmez. Tous les logiciels Microsoft sont faits en C/C++ à 90% et il ne serait pas normal de ne pas en parler. Le C++ n'est pas mort, loin de là. Avec C++11, le C++ a rajeuni et les standards C++14 et C++17 apportent leurs lots de nouveautés. Stay tuned comme dirait l'autre et à bientôt pour de nouveaux articles.