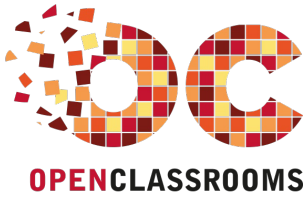
Sérialisation avec Boost

Par Grota
et Romain Porte (MicroJoe)



www.openclassrooms.com

Dernière mise à jour le 25/10/2010

Sommaire

Sérialisation avec Boost

Par  Grota et Romain Porte (MicroJoe)

Mise à jour : 25/10/2010
Difficulté : Intermédiaire Durée d'étude : 2 heures

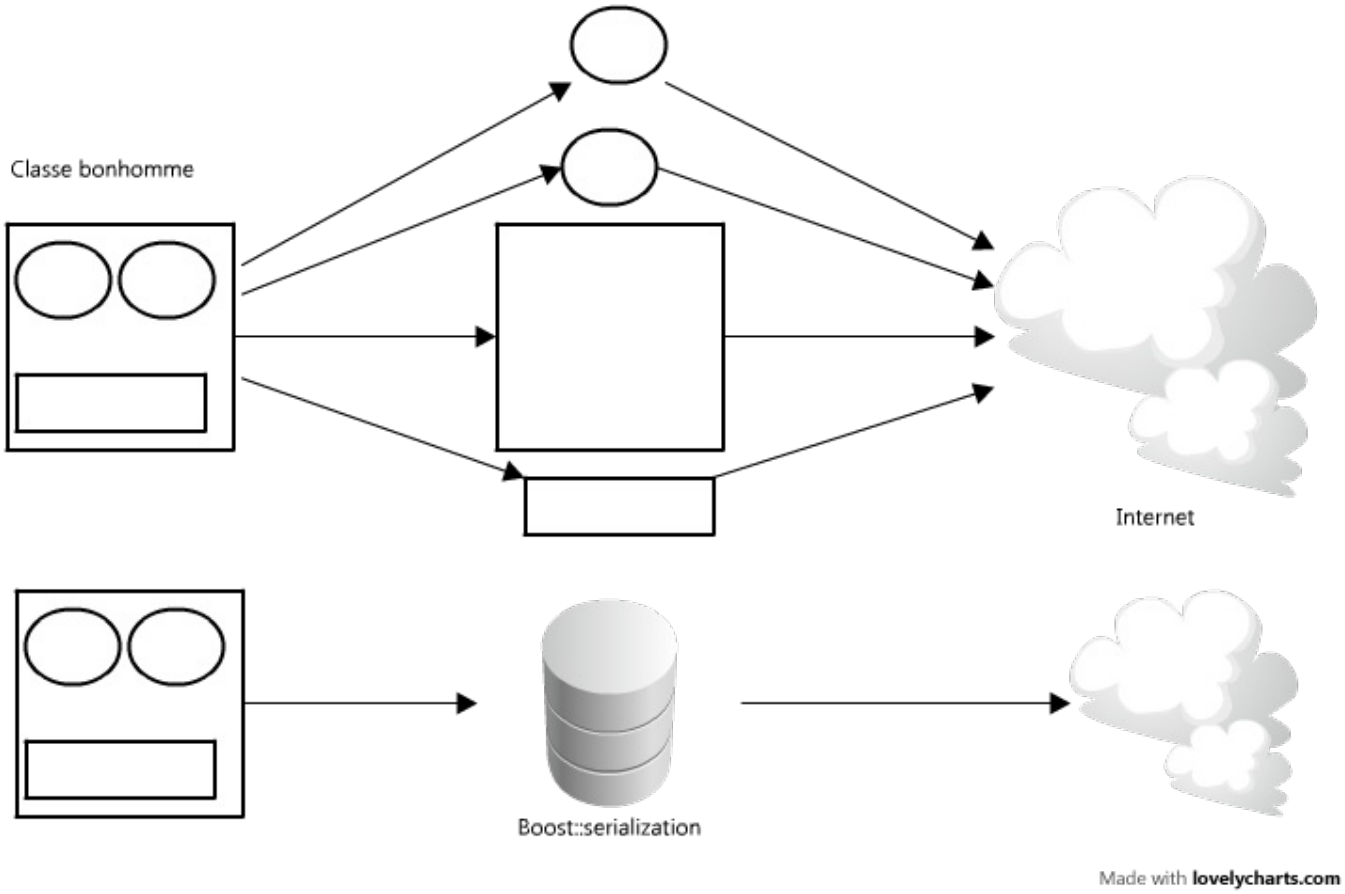
La **sérialisation**, nommée occasionnellement *marshalling*, est le procédé informatique de transformation d'un élément complexe vers des éléments plus simples, comme par exemple le rapport entre un fichier texte et sa représentation binaire. Cette simplification rend nombre d'opération **plus facile**. Nous nous y intéresserons plus particulièrement dans le contexte des classe **C++**.

D'une manière pratique, la sérialisation peut vous aider dans la réalisation de **fichier pour votre application**, dans les transfert sur le **réseau** ... La **simplicité** de mise en oeuvre apportée par Boost est considérable, c'est pourquoi c'est cette librairie que j'ai choisi.
Sommaire du tutoriel :

Installation de Boost

Quelques mots sur la sérialisation

Pour mieux comprendre ce qu'est la sérialisation, prenons un exemple courant : le transfert de données *via* le réseau. C'est **très simple** à mettre en place en ce qui concerne des **données peu complexes** (texte d'un chat, formulaires…), mais ça devient complexe quand il s'agit de transférer des **classes** : il vous faut transférer les différents membres, donc produire du code sans grand intérêt et très dupliqué pour gérer cela. Boost::serialization permet d'**automatiser** la chose ; et cela pourra également vous servir pour des **sauvegardes**, des **formats de fichiers**…



Installation sous Windows

La plupart des librairies Boost sont sous forme de fichiers d'en-tête, ce qui signifie que vous n'avez rien à compiler : il suffit d'inclure les fichiers concernés dans vos programmes. Malheureusement, j'ai dit « la plupart » : nous allons devoir en compiler certaines.

La compilation de Boost nécessite un compilateur (sans blague). Ma préférence ira à la méthode correspondant à MinGW. Il est nécessaire d'avoir le compilateur et ses outils dans votre variable d'environnement PATH.

Si vous avez déjà installé le SDK Qt 4, vous pouvez utiliser le Qt command prompt. Il est intéressant car il fournit un environnement contenant make, gcc et les binutil de manière préconfigurée. Dans les autres cas, ouvrez une invite de commande MS-DOS (tapez Win + R, et dans la boîte de dialogue, entrez `cmd` puis validez).

Vérification

Pour vérifier que votre environnement de compilation est correct, entrez `gcc` ; vous devriez obtenir un résultat similaire à celui-ci :

Code : Console

```
C:\Qt\4.5.3>gcc
gcc: no input files

C:\Qt\4.5.3>
```

Cependant, si vous obtenez la sortie suivante, c'est que votre configuration n'est pas saine. Assurez-vous d'avoir les outils MinGW dans votre PATH.

Code : Console

```
C:\Users\william>gcc
'gcc' n'est pas reconnu en tant que commande interne
ou externe, un programme exécutable ou un fichier de commandes.

C:\Users\william>
```

Téléchargement

Obtenez Boost sur cette page du site officiel. Décompressez l'archive dans un répertoire, mais assurez-vous que le chemin complet ne contienne pas d'espace.

C:\boost_1_43_0 est correct.	**C:\Program Files\boost_1_43_0** ne l'est pas.

Vous devez aussi obtenir la dernière version de bjam, l'utilitaire qui va permettre de compiler Boost. Vous le trouverez ici. Prenez la version binaire destinée à Windows. Vous trouverez un fichier bjam.exe : placez-le dans le même dossier que Boost.

Compilation

Déplacez-vous dans le dossier de Boost. Dans mon cas, il s'agit de C:\boost_1_43_0.

Exécutez la commande suivante :

Code : Console

```
bjam --with-
serialization toolset=gcc variant=release link=static threading=multi runtime-
link=static stage
```

Pour de plus amples précisions concernant l'installation sous Windows, ou en cas de problème, voyez la documentation de Boost à ce sujet.

Installation sous GNU/Linux

Il est très probable que votre distribution fournisse des paquets Boost : il s'agira vraisemblablement des paquets boost-dev ou libboost-dev (renseignez-vous sur la documentation officielle de votre distribution). L'avantage de cette méthode est que les librairies seront liées automatiquement si vous ajoutez -lboost à vos options de compilateur.

Si ce n'est pas le cas, référez-vous à la documentation pour compiler Boost sous les variantes d'UNIX. La démarche est aussi valable pour Mac OS X.

Sérialisation basique

C'est le procédé de sérialisation le plus simple. Prenons un exemple pour décrire ce procédé : la classe **Note**, qui représente la note d'un devoir. Il existe différents types de notes : note sur 20, note sur 40…

Préparatifs

Voici la classe Note :

Code : C++

```
class Note
{
private:
        friend class boost::serialization::access;

        template<class Archive>
        void serialize(Archive & ar, const unsigned int version) {
                ar & numerateur;
                ar & denominateur;
        }

        int numerateur;
        int denominateur;
public:
        Note() {};
        Note(int n, int d) :
                        numerateur(n), denominateur(d){}
};
```

C'est un code plutôt habituel, sauf les lignes 4 à 10 que je vais expliquer.

Code : C++

```
friend class boost::serialization::access;
```

La méthode que nous utilisons ici est dite intrusive : nous modifions la classe pour la préparer à la sérialisation. Nous verrons dans la prochaine partie une méthode non intrusive (pas de modification de la classe).
Nous déclarons donc comme classe amie `boost::serialization::access` : étant donné qu'il s'agit d'une classe **friend**, Boost pourra récupérer le contenu de la classe pour le sérialiser. Sans cette porte dérobée, l'encapsulation ne nous aurait pas laissé accéder aux variables dont la portée est **`private`** .

Code : C++

```
template<class Archive>
        void serialize(Archive & ar, const unsigned int version) {
                ar & numerateur;
                ar & denominateur;
        }
```

La première chose à remarquer est la présence d'un *template* pour la classe **Archive** : c'est tout à fait normal. En effet, la sérialisation de Boost permet de stocker les objets dans des fichiers texte, des fichiers XML… Nous avons donc besoin de généricité. La méthode **serialize** sera automatiquement appelée lors de l'archivage. Nous devons déclarer toutes les variables de la classe dans cette méthode, ou elles ne seront pas sérialisées.

Note concernant l'opérateur **&** : c'est une utilisation intéressante de la surcharge qui est utilisée ici. Lors de l'archivage, **&** sera équivalent à <<, et lors de la restauration, il sera équivalent à >>. Vous comprendrez tout lorsque vous verrez le code source complet.

Sérialisation

Voici le code complet de cet exemple :

Code : C++

```
#include <fstream>

#include <boost/archive/text_oarchive.hpp>
#include <boost/archive/text_iarchive.hpp>

class Note
{
private:
        friend class boost::serialization::access;

        template<class Archive>
        void serialize(Archive & ar, const unsigned int version) {
                ar & numerateur;
                ar & denominateur;
        }

        int numerateur;
        int denominateur;
public:
        Note() {};
        Note(int n, int d) :
                        numerateur(n), denominateur(d) {}
};


int main()
{
        std::ofstream ofs("fichierDeSerialisation");

        // Vous avez vu comme je travaille bien ? :)
        const Note maNoteDePhysisque(20,20);


        {
                boost::archive::text_oarchive oa(ofs);
                oa << maNoteDePhysisque;
        }

        /** Quelque temps plus tard… ***/

        Note monAncienneNote;

        {
                std::ifstream ifs("fichierDeSerialisation");
                boost::archive::text_iarchive ia(ifs);

                ia >> monAncienneNote;
        }

        return 0;
}
```

Qu'est-ce qui change ?

Nous avons inclus des *headers* spécifiques à la sérialisation aux lignes 3 et 4. Nous avons aussi inclus **fstream**, qui servira lors du stockage dans un fichier.

Nous avons ajouté un *main* qui sérialise et désérialise. Observons-le plus en détail :

Code : C++

```
std::ofstream ofs("fichierDeSerialisation");

        // Vous avez vu comme je travaille bien ? :)
        const Note maNoteDePhysisque(20,20);
```

```
        {
                boost::archive::text_oarchive oa(ofs);
                oa << maNoteDePhysisque;
        }
```

On ouvre d'abord un fichier pour stocker notre objet à la ligne 28. Puis nous créons à la ligne 31 un objet Note. La vraie sérialisation commence dans le bloc des lignes 34 à 37 : on utilise notre fichier comme une **text_archive** (ligne 35), puis on fait simplement appel à l'opérateur << pour stocker **maNoteDePhysisque** dans l'archive. À la fin du bloc, les destructeurs sont automatiquement appelés, et l'archive est refermée (attention cependant, vous ne devez pas détruire le flux de fichier **oa** car l'archive s'en charge !).

J'ai délimité une portée (j'ai utilisé des accolades ouvrantes et fermantes sans en avoir vraiment besoin) pour appeler les destructeurs le plus rapidement possible, ce qui permet d'être sûr que les données sont bien sauvegardées immédiatement. Vous pouvez tout à fait choisir de ne pas le faire.

Code : C++

```
Note monAncienneNote;

        {
                std::ifstream ifs("fichierDeSerialisation");
                boost::archive::text_iarchive ia(ifs);

                ia >> monAncienneNote;
        }
```

Nous allons désérialiser. Notez que cette opération n'a pas forcément lieu dans le même programme, ni sur le même ordinateur : vous pouvez très bien envoyer le fichier par le réseau puis l'ouvrir sur une autre machine. Le résultat sera exactement le même.

Nous créons un objet vide, puis, dans le bloc, nous ouvrons notre fichier de sérialisation comme une **text_archive** et nous extrayons son contenu dans l'objet précédemment créé. À la fin du bloc, les destructeurs sont automatiquement appelés, et les fichiers refermés. 

Comme vous le voyez, la syntaxe est très simple : << pour stocker, >> pour récupérer.

Vous pouvez utiliser cette méthode sur des classes plus complexes. Imaginons que vous vouliez créer une classe **Bulletin** :

Code : C++

```
class Bulletin
{
        friend class boost::serialization::access;
        template<class Archive>
        void serialize(Archive & ar, const unsigned int version) {
                ar & note1;
                ar & note2;
        }
        Note note1;
        Note note2;
protected:
        Bulletin(const Note & n1_, const Note & n2_) :
                        note1(n1_), note2(n2_) {}
public:
        Bulletin() {}
        virtual ~Bulletin() {}
};
```

Ce n'est pas un problème d'avoir des classes imbriquées dans la classe à sérialiser (dans le cas présent, Bulletin contient Note) ; cependant, il faut que chaque classe soit elle-même sérialisable.

Sérialisation non intrusive

Il est possible de sérialiser du code venant de boost.serialization. Cela implique cependant de violer le principe d'encapsulation, et ne doit donc être entrepris que lorsque les contraintes techniques l'imposent : en effet, il faut passer toute la classe en visibilité publique.

Revoyons donc la classe Note :

Code : C++

```
class Note
{
public:
        int numerateur;
        int denominateur;

        Note() {};
        Note(int n, int d) :
                        numerateur(n), denominateur(d) {}
};
```

Nous avons tout passé en visibilité publique. C'est mal. Vos poils se dressent. Vous avez froid dans le dos.  Mais vous continuez quand même… Pour sérialiser, il faut une fonction **serialize** sous forme de surcharge recevant en argument une instance de la classe Note :

Code : C++

```
namespace boost
{
        namespace serialization {

                template<class Archive>
                void serialize(Archive & ar, Note & g, const
unsigned int version)
                {
                        ar & g.numerateur;
                        ar & g.denominateur;
                }

        } // namespace serialization
} // namespace boost
```

L'utilisation sera exactement la même que celle d'une sérialisation intrusive !

Petite précision

Il n'est pas forcément nécessaire de passer complètement le contenu de la classe en public : si les accesseurs et les mutateurs donnent un accès suffisant (c'est-à-dire complet) au contenu de la classe, vous pouvez les utiliser. Il vous faudra cependant séparer la fonction **serialize** en **load** et **save**, et déclarer la séparation *via* la macro `BOOST_SERIALIZATION_SPLIT_MEMBER(NomDeLaClasse)` , comme dans l'exemple suivant.

Code : C++

```
//on rajoute
#include <boost/serialization/split_free.hpp>

class Note
{
public:
        Note() {};
        Note(int n, int d) :
                        numerateur(n), denominateur(d) {}
       int getNumerateur()
       {return numerateur;}

       int getDenominateur()
       {return denominateur;}

       void setNumerateur(int n)
       {numerateur = n;}

       void setDenominateur(int n)
       {denominateur = n;}

private:
        int numerateur;
        int denominateur;


};

namespace boost
{
        namespace serialization {

    void save(Archive & ar, Note & n, const unsigned int version)
const
    {
        ar  & n.getNumerateur();
        ar  & n.getDenominateur();
    }
    template<class Archive>
    void load(Archive & ar, Note & n, const unsigned int version)
    {
        int a,b;
        ar  & a;
        n.setNumerateur(a);
        ar  & b;
        n.setDenominateur(b);
    }


        } // namespace serialization
} // namespace boost
    BOOST_SERIALIZATION_SPLIT_MEMBER(Note)
```

Cas spécifiques

Sérialisation de classes dérivées

Pour sérialiser une hiérarchie d'objets, il faut inclure un nouvel en-tête dédié.

Code : C++

```
#include <boost/serialization/base_object.hpp>
```

De plus, le code de sérialisation sera modifié :

Code : C++

```
friend class boost::serialization::access;
    template<class Archive>
    void serialize(Archive & ar, const unsigned int version)
    {
        // serialize base class information
ar & boost::serialization::base_object<CLASSEdeBASE>(*this);
        ar & street1;
        ar & street2;
    }
```

Avec **CLASSEdeBASE** le nom de la classe de base. Aucune modification à l'exception de celle-ci.

La classe de base doit absolument être sérialisable !

Pour illustrer le propos, créons un objet qui représentera un devoir composé d'une note (quelle originalité ) et d'un texte de sujet.

Code : C++

```
#include <boost/serialization/base_object.hpp>

class DevoirSurTable : public Note
{
    friend class boost::serialization::access;
    template<class Archive>
    void serialize(Archive & ar, const unsigned int version)
    {
ar & boost::serialization::base_object<Note>(*this);
        ar & sujetDevoir;
    }
    std::string sujetDevoir;
public:
/* Note : le constructeur fournit ses données à la classe mère Note
*/
    DevoirSurTable(int note_numerateur, int note_denominateur,
string sujet) : Note(note_numerateur, note_denominateur),
sujetDevoir(sujet)  {};
    ~DevoirSurTable(){};
};
```

En conséquence, la classe de base `Note` sera archivée avec la classe dérivée `DevoirSurTable` . On peut tout à fait appliquer ce concept à une hiérarchie de plusieurs niveaux de dérivation : chaque classe doit alors sérialiser sa parente.

Pointeurs

Prenons comme exemple un relevé de notes :

Code : C++

```
class Releve
{
        friend class boost::serialization::access;
        Note * contenu[10];
```

```
        template<class Archive>
        void serialize(Archive & ar, const unsigned int version) {
                for (int i = 0; i < 10; ++i)
                        ar & contenu[i];
        }
public:
        Releve() {}
};
```

Les dix pointeurs seront archivés.

Eh, attends, ce sont des **pointeurs** : lorsqu'on les restaurera, ils ne pointeront plus sur la bonne zone de mémoire !

Mais si ! Boost a tout prévu : lors de la sérialisation, la cible du pointeur est sérialisée, et lors de la restauration, ce contenu sera toujours pointé par le pointeur. Magique ? 

Nous pouvons même simplifier encore le code car Boost détecte automatiquement la présence d'un tableau :

Code : C++

```
class Releve
{
        friend class boost::serialization::access;
        Note * contenu[10];
        template<class Archive>
        void serialize(Archive & ar, const unsigned int version) {
                        ar & contenu;
        }
public:
        Releve() {}
};
```

Le même fonctionnement simple (`ar & contenu;`) est correct pour tous les conteneurs de la STL.

Cependant, ce mode d'autodétection des pointeurs peut poser un problème lorsque l'on se frotte au polymorphisme. Voici une situation problématique :

Code : C++

```
class Base {
 /* … */
};
class Derived_one : public Base {
 /* … */
};
class Derived_two : public Base {
 /* … */
};
int main(){
 /* … */
    Base *b;
 /* Fonction serialize */
    ar & b;
 /* … */
}
```

Lors de la sérialisation ou restauration (`ar & b`), comment savoir si l'objet est de type **Derived_one**, **Derived_two**, ou même

Base ? Lors de la restauration, une exception « *unregistered class* » sera levée par Boost.

Pour éviter cela, deux méthodes sont possibles.

- Vous pouvez sérialiser toutes les classes dérivées. Lorsqu'un objet est sérialisé, sa classe est « apprise » par Boost. Les pointeurs sur cette classe seront archivés sans problème dès lors qu'un objet de cette classe aura été sérialisé :

Code : C++

```
int main(){
 /* … */
    Derived_one d1;
    Derived_two d2;
    Base *b;
 /* Fonction serialize */
    ar & d1;
    ar & d2;
    ar & b;
 /* … */
}
```

- Vous pouvez enregistrer manuellement les classes, comme ceci (notez l'ajout d'un *header*) :

Code : C++

```
#include <boost/serialization/export.hpp>
class Base {
 /* … */
};
class Derived_one : public base {
 /* … */
};
class Derived_two : public base {
 /* … */
};
BOOST_CLASS_EXPORT_GUID(derived_one, "derived_one")
BOOST_CLASS_EXPORT_GUID(derived_two, "derived_two")
```

La sérialisation ne posera alors pas de problème.

Le cas du polymorphisme comporte d'autres subtilités que je n'ai pas présentées ici : les détails à propos de ce type de sérialisation se situent dans la documentation.

Versions

Imaginons qu'un jour les notes changent du tout au tout : par exemple, chaque note recevrait un coefficient. Comment s'assurer que les fichiers créés avec une ancienne version de notre programme sont toujours valables ? En ajoutant un simple *header*, et en vérifiant les versions : chaque classe reçoit un numéro de version — 0 au début — qui permet de l'identifier. Regardons plutôt l'exemple suivant.

Ancienne version de la classe

Code : C++

```
#include <boost/serialization/version.hpp>

class Note
{
private:
        friend class boost::serialization::access;
```

```
        template<class Archive>
        void serialize(Archive & ar, const unsigned int version) {
                ar & numerateur;
                ar & denominateur;
        }

        int numerateur;
        int denominateur;
public:
        Note() {};
        Note(int n, int d) :
                        numerateur(n), denominateur(d){}
};
BOOST_CLASS_VERSION(Note, 0)
```

Nouvelle version de la classe

Code : C++

```
#include <boost/serialization/version.hpp>

class Note
{
private:
        friend class boost::serialization::access;

        template<class Archive>
        void serialize(Archive & ar, const unsigned int version) {
                ar & numerateur;
                ar & denominateur;
                if(version > 0)
                     ar & coefficient;
        }

        int numerateur;
        int denominateur;
        int coefficient;
public:
        Note() {};
        Note(int n, int d) :
                        numerateur(n), denominateur(d){}
};
BOOST_CLASS_VERSION(Note, 1)
```

Nous ne sérialisons et ne chargeons le coefficient que si la version est suffisante. Cette version est déclarée par le biais la macro **BOOST_CLASS_VERSION(classe, version)**. Cependant, ce code n'est pas encore optimal : on enregistre toujours dans la dernière version disponible. Nous pourrions donc séparer la fonction **serialize** en **load** et **save** :

Code : C++

```
class Note
{
private:
        friend class boost::serialization::access;
        template<class Archive>
        void save(Archive & ar, const unsigned int version) const {
                ar & numerateur;
                ar & denominateur;
                ar & coefficient;
        }
        template<class Archive>
        void load(Archive & ar, const unsigned int version) {
```

```
                ar & numerateur;
                ar & denominateur;
                if (version > 0)
                        ar & coefficient;

        }
        BOOST_SERIALIZATION_SPLIT_MEMBER()

        int numerateur;
        int denominateur;
        int coefficient;
public:
        Note() {};
        Note(int n, int d) :
                        numerateur(n), denominateur(d) {}
};
BOOST_CLASS_VERSION(Note, 1)
```

Pour poursuivre votre quête de sérialisation, rendez-vous sur la documentation officielle :

www.boost.org

Vous serez certainement intéressés par les autres types d'archivage (nous avons vu uniquement les archives texte), notamment les archivages XML et binaire. Il est même possible, si aucun de ceux-là ne vous convient, de créer votre propre format d'archive.