

La sérialisation avec Boost

par **Pierre Schwartz** ([retour aux articles](#))

Date de publication : 15 juillet 2008

Dernière mise à jour :

Cet article va exposer les concepts de la sérialisation et les méthodes proposées par Boost pour la mettre en #uvre. Prérequis : programmation objet, C++

I - La sérialisation.....	3
II - Caractère sérialisable d'un objet.....	3
II-1 - Sérialisation de base.....	3
II-2 - Sérialisation d'objets hérités.....	4
II-3 - Versions de sérialisation.....	4
III - Les archives Boost.....	5
III-1 - Sérialisation texte.....	5
III-2 - Sérialisation binaire.....	6
III-3 - Sérialisation XML.....	6
IV - Sérialisation d'objets pointeurs.....	7
V - Conclusion.....	8
V-1 - Exemple récapitulatif.....	8
V-2 - Liens externes.....	9

I - La sérialisation

La sérialisation est un moyen de coder une donnée sous des formes diverses. La sérialisation d'un objet se fait en codant récursivement chacun de ses attributs. La sérialisation des objets permet de les stocker dans un fichier, en base de données ou même de les envoyer sur le réseau.

La définition de la méthode de sérialisation s'accompagne bien souvent de la méthode inverse permettant de reconstituer l'objet à partir d'un flux de données : c'est la désérialisation.

Boost fournit des classes de sérialisation permettant de sérialiser tous les types de base ainsi que les conteneurs de la STL. Par défaut, Boost propose des classes de sérialisation texte et binaire. Vous pouvez bien sûr implémenter vos propres classes de sérialisation si le besoin s'en fait sentir.

II - Caractère sérialisable d'un objet

II-1 - Sérialisation de base

Pour qu'un objet soit sérialisable, il suffit qu'il implémente la fonction `void serialize(Archive& , const unsigned int)`. Cette fonction template prend comme premier argument la sérialisation à utiliser (texte, binaire ...) et en second argument un numéro de version permettant de différencier les sérialisations.

L'implémentation de la fonction `serialize` doit uniquement définir quels attributs ajouter à la sérialisation et dans quel ordre les sérialiser. Cette unique fonction `serialize()` définit à la fois la sérialisation et la désérialisation. Les attributs seront ainsi sérialisés et désérialisés dans le même ordre (ce comportement est cependant modifiable).

Exemple sur une classe personnalisée contenant une poignée de champs :

```
class DVPExemple{
public:
    DVPExemple(){}
private:
    int a,b;
    string c;
    std::list<std::map<std::string, int> > d;
};
```

Il va suffire de définir la fonction `serialize` sur l'objet `DVPExemple`. Cela pourra se faire de manière intrusive en ajoutant une fonction membre dans `DVPExemple`, ou bien en définissant une fonction amie extérieure à la classe `DVPExemple`.

Méthode intrusive

```
class DVPExemple{
public:
    template<class Archive>
    void serialize(Archive& ar, const unsigned int version){
        ar & a & b & c & d;
    }

    int a,b;
    string c;
    std::list<std::map<std::string, int> > d;
};
```


Dans cet exemple, la fonction `serialize` lie chacun des attributs de notre classe à l'archive via l'opérateur `&`. Ce code fonctionnera parce que les attributs `a`, `b`, `c` et `d` sont des types sérialisables. On aurait aussi pu ajouter la sérialisation de n'importe quel objet sérialisable, aussi complexe soit-il.

Méthode non intrusive

```
class DVPExemple{
public:
    DVPExemple(){}
    int a,b;
    string c;
    std::list<std::map<std::string, int> > d;
};

template<class Archive>
void serialize(Archive& ar, DVPExemple& data, const unsigned int version){
    ar & data.a & data.b & data.c & data.d;
}
```

Ces 2 implémentations donnent exactement le même comportement et nous avons maintenant une classe `DVPExemple` tout à fait sérialisable. On utilisera l'une ou l'autre méthode selon qu'on souhaite rendre sérialisable une classe personnelle ou une classe externe.

 **Attention, l'injection d'objets non initialisés dans une archive peut avoir un comportement indéfini, en particulier pour les booléens. Cet avertissement peut cependant être nuancé suivant les compilateurs. Il faut garder à l'esprit la bonne pratique de programmation de toujours manipuler des objets où l'initialisation est maîtrisée.**

II-2 - Sérialisation d'objets hérités

Boost recommande de ne pas appeler une éventuelle fonction `serialize()` d'une classe mère. Pour éviter ce genre de dérives, il est recommandé de déclarer la fonction `serialize` comme privée. Ainsi une classe fille ne pourra pas l'utiliser dans un appel par l'opérateur `&`. La solution préconisée est de déclarer la classe mère amie de la classe `boost::serialization::access`, et d'appeler la sérialisation de l'objet parent via

```
template<class Archive>
void serialize(Archive & ar, const unsigned int version)
{
    ar & boost::serialization::base_object<classe_mere>(*this); // sérialisation de la classe mère
    ar & [attributs propres à l'objet] // sérialisation des éléments propres à l'objet courant
}
```

II-3 - Versions de sérialisation

Boost propose de distinguer des versions de sérialisation via le dernier argument de la fonction `serialize()`. Cet argument sera automatiquement renseigné lors de l'appel à la sérialisation, il prendra la valeur définie par la macro `BOOST_CLASS_VERSION(nom_de_la_classe, numero_de_version)`. Exemple

```
class DVPExemple{
private:
    friend class boost::serialization::access;
    template<class Archive>
    void serialize(Archive& ar, const unsigned int version){
        if (version ....){
            // traitement propre à la version
        }else{
            // ...
        }
    }
public:
    // reste de la classe
};
```

```
BOOST_CLASS_VERSION(DVPExemple, 2)
```

Nous avons vu comment sérialiser un objet, mais la fonction `serialize` ne spécifie que l'ordre des éléments à sérialiser. Rien n'est encore dit sur le flux de données sérialisées. C'est le rôle des archives Boost. Ce sont elles qui vont définir le type de sérialisation (binaire, xml, texte ...) à mettre en #uvre.

III - Les archives Boost

Les archives vont encoder / decoder les données à sérialiser, dans l'ordre défini par les fonctions `serialize()` des objets à sérialiser. Pour chaque type de sérialisation, Boost propose 2 classes d'archive, permettant respectivement de coder et de decoder les données.

III-1 - Sérialisation texte

Les 2 classes d'archive texte sont `boost::archive::text_oarchive` (text output archive) et `boost::archive::text_iarchive` (text input archive). Ces archives définissent les opérateurs `<<` et `>>` permettant d'y injecter et d'en extraire des objets sérialisables. Les archives texte sont liées à des flux, comme par exemple des fichiers texte. Exemple

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

DVPExemple d;
std::ofstream ofile(filename);
boost::archive::text_oarchive oTextArchive(ofile);
oTextArchive << d;    // sérialisation de d

std::ifstream ifile(filename);
boost::archive::text_iarchive iTextArchive(ifile);
iTextArchive >> d;    // désérialisation dans d
```

Voici un exemple de code sérialisé dans une archive texte.

```
// classe à sérialiser
class Test{
public:
    Test(){}
    Test(int i, std::string c, float f1, float f2){
        a=i;
        s=c;
        v.push_back(f1);
        v.push_back(f2);
    }


    template<class Archive>
    void serialize(Archive& ar, const unsigned int version){
        ar & a & s & v;
    }

    int a;
    std::string s;
    std::vector<float> v;
};

Test t(2, std::string("essai"), 2.0, 3.5);
std::ofstream ofile(filename);
boost::archive::text_oarchive oTextArchive(ofile);
oTextArchive << t;    // sérialisation de t
```

Ce qui nous donne comme chaîne sérialisée :

```
22 serialization::archive 4 0 0 2 5 essai 2 0 2 3.5
```

 Certains compilateurs peuvent générer des erreurs à la compilation **use of undefined type 'boost::STATIC_ASSERTION_FAILURE<x>'**. Pour y remédier, il faut passer une référence constante sur l'objet à sérialiser lors de l'appel à l'opérateur <<.

III-2 - Sérialisation binaire

Le principe est très similaire à l'archive texte, il suffit d'utiliser les classes `boost::archive::binary_iarchive` et `boost::archive::binary_oarchive`. L'injection et la récupération d'objets se font aussi par les opérateurs << et >>. La sérialisation binaire a l'avantage de nécessiter moins de place, idéale pour être sauvegardée dans un fichier binaire ou pour être envoyée sur le réseau.

III-3 - Sérialisation XML

La sérialisation XML est une spécialisation de la sérialisation texte. Les classes à utiliser sont `boost::archive::xml_oarchive` et `boost::archive::xml_iarchive`. Prenons maintenant le même exemple que pour l'archive texte. Il faut cependant rajouter la macro `BOOST_SERIALIZATION_NVP` à chaque appel aux opérateurs << et >>. Cette macro permet d'indiquer le nom des noeuds XML en rapport avec le nom des attributs dans le code C++. L'injection dans l'archive devient `oTextArchive << BOOST_SERIALIZATION_NVP(t);` La fonction `serialize` devient


```
template<class Archive>
void serialize(Archive& ar, const unsigned int version){
    ar & BOOST_SERIALIZATION_NVP(a) &
    BOOST_SERIALIZATION_NVP(s) &
    BOOST_SERIALIZATION_NVP(v);
}
```


```
#include <boost/archive/xml_oarchive.hpp>

Test t(2, std::string("essai"), 2.0, 3.5);
std::ofstream ofile(filename);
boost::archive::xml_oarchive oTextArchive(ofile);
oTextArchive << BOOST_SERIALIZATION_NVP(t); // sérialisation de t
```

Ce qui nous donne comme chaîne sérialisée :

```
<?xml version="1.0" encoding="UTF-8" standalone="yes" ?>
<!DOCTYPE boost_serialization>
<boost_serialization signature="serialization::archive" version="4">
<t class_id="0" tracking_level="0" version="0">
  <a>2</a>
  <s>essai</s>
  <v>
    <count>2</count>
    <item_version>0</item_version>
    <item>2</item>
    <item>3.5</item>
  </v>
</t>
</boost_serialization>
```

 **Attention** lors de l'utilisation de la macro `BOOST_SERIALIZATION_NVP`, Boost se sert du paramètre pour construire le noeud XML correspondant. En injectant des caractères tels que des `->` ou des `[]` vous obtiendrez des erreurs à la génération XML.

 Boost propose également d'autres variantes des archives texte, permettant de prendre en compte les caractères larges, notamment via les objets `std::wstring`, ce sont les archives `warchive` et `wiarchive`

IV - Sérialisation d'objets pointeurs

Boost permet de sérialiser des objets pointeurs bien qu'ils ne correspondent qu'à des adresses mémoires.

```
DVPExemple* d = new DVPExemple();
oTextArchive << d;
```

Cependant comment faire lorsqu'on manipule des objets polymorphiques et qu'un pointeur peut en réalité masquer une instance d'une classe fille ? Prenons un simple exemple d'un conteneur contenant des pointeurs sur une classe mère mais où certains éléments sont des instances d'une classe fille. Peut-on quand même sérialiser ce conteneur ?

Boost permet de sérialiser des objets pointeurs polymorphiques en tenant à jour une table des classes utilisables ainsi qu'une table des casts possibles d'une classe sur l'autre. Ainsi, pour sérialiser des objets pointeurs polymorphiques, il va falloir renseigner ces tables d'informations Boost. La table des classes sérialisables doit être définie dans chaque objet archive. Typiquement une sauvegarde/chargement nécessite deux archives et nécessitera deux inscriptions dans les tables Boost.

Boost rajoute également une couche de traitement permettant de repérer les objets pointés plusieurs fois. Boost s'arrangera pour que les pointeurs qui étaient égaux avant une sérialisation le soient encore après une désérialisation. Les emplacements mémoires seront très probablement différents puisqu'il s'agira d'objets différents, mais leur contenu restera le même et les relations entre eux aussi.

La définition des classes filles sérialisables se fait via

```
archive.register_type(static_cast<Fille *>(NULL)); // inscription des pointeurs sur une classe fille
```

Et la définition des casts possibles se fait par

```
boost::serialization::void_cast_register(static_cast<Fille *>(NULL), static_cast<Mere *>(NULL));
```

Et voilà, maintenant on peut sérialiser un conteneur comme celui-ci


```
// création du conteneur
std::vector<Mere*> v;
v.push_back(new Mere());
v.push_back(new Fille());

// création du flux de sortie et de l'archive
std::ofstream ofile("out.txt");
boost::archive::text_oarchive oTextArchive(ofile);

// inscription dans les tables Boost
oTextArchive.register_type(static_cast<Fille *>(NULL));
boost::serialization::void_cast_register(static_cast<Fille *>(NULL), static_cast<Mere *>(NULL));


// sérialisation
oTextArchive << v;
```

L'absence de la définition Boost d'un cast ou d'une classe provoquera la levée d'une exception `boost::archive::archive_exception` à l'exécution.

 *Chacune des classes sérialisables par pointeur devra posséder un constructeur par défaut sous peine de devoir surcharger les méthodes de chargement et de sauvegarde de données plus précises que la fonction `serialize()`.*

V - Conclusion

V-1 - Exemple récapitulatif

 Vous prendrez soin de déclarer correctement Boost dans votre compilateur, en indiquant les chemins des headers et en ajoutant éventuellement les instructions d'édition de liens `-lboost_serialization`.

Voici un exemple résumant les concepts de la sérialisation Boost :

```
#include <fstream>
#include <vector>
#include <iostream>
#include <sstream>

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

// pour la sérialisation de std::vector
#include <boost/serialization/vector.hpp>

using namespace boost::archive;

/**
 * classe basique à sérialiser
 */
class User{
public:
    User(){}
    std::string nom, prenom;
    int num, age;

    void display(){
        std::cout << nom << " " << prenom << " " << num << " " << age << std::endl;
    }

private:
    friend class boost::serialization::access;
    template<class Archive>
    void serialize(Archive& ar, const unsigned int version){
        ar & nom & prenom & num & age;
    }
};

// enregistrer une collection dans un fichier
template <class T>
void saveIntoFile(std::vector<User>& d, const char* file){
    std::ofstream ofile(file);
    T ar(ofile);

    ar << d;
    ofile.close();
}

// charger une collection depuis un fichier
template <class T>
void getFromFile(std::vector<User>& d, const char* file){
    std::ifstream ifile(file);
    T ar(ifile);

    ar >> d;
    ifile.close();
}

int main (){
    // créer un tableau d'objets
```



```
std::vector<User> d;
for (int i = 0; i<10; i++){
    User u;
    u.age = rand()%50;
    u.num = i;

    std::ostringstream ss1, ss2;
    ss1 << "nom" << i;
    u.nom = ss1.str();

    ss2 << "prenom" << i;
    u.prenom = ss2.str();
    d.push_back(u);

    u.display() ;
}

// sauver le tableau d'objets dans un fichier
saveIntoFile<text_oarchive>(d, "out.txt");

std::cout << std::endl << std::endl;
d.clear();

// relire les données depuis le fichier
getFromFile<text_iarchive>(d, "out.txt");
for (int i = 0; i<d.size(); i++)
    d[i].display();

return 0;
}
```

Ce code très simple permet d'enregistrer et de récupérer dans le même état des données enregistrées.

V-2 - Liens externes

- [La page officielle de Boost.serialization](#)
- [Plein de tutoriels sur Boost sur developpez.com](#)

La sérialisation s'avère un moyen puissant de coder des données, de les sauvegarder et de les échanger de manière persistante. Boost se charge des aspects les plus complexes et il vous suffit de définir la petite fonction `serialize()` pour chacune de vos classes à sérialiser. C'est bien peu de choses quand on voit la puissance de cet outil. Bien que souvent dépréciée des développeurs parce que trop lourde ou trop complexe à mettre en oeuvre, la sérialisation Boost se démarque par sa légèreté et sa simplicité d'utilisation. A consommer sans modération.

Je remercie toute l'équipe de la rubrique C++ pour leur relecture acharnée et **diogene** pour la correction orthographique.