

Exportation de classes C++ dans une bibliothèque dynamique sous Linux

par hiko-seijuro (hiko-seijuro.developpez.com)

Date de publication : 09/01/2007

Dernière mise à jour : 09/01/2007

L'objectif de cet article est de présenter comment exporter des classes c++ dans une bibliothèque dynamique sous linux, puis comment les charger. (suppression des destructers, qui rendaient une partie du tutoriel flou, lors de la mise à jour)

- I - Introduction
- II - Les fonctions permettant la gestion des bibliothèques dynamiques sous Linux
 - II-A - Description des primitives
 - II-B - Exemple d'utilisation des primitives
 - II-C - Options de compilation
- III - Création, exportation et utilisation des classes C++
 - III-A - Créateurs
 - III-B - Exemple d'exportation d'une classe
 - III-C - Utilisation de la classe exportée
 - III-D - Compilation de la bibliothèque et du programme
- IV - Gestion des classes plus poussée
 - IV-A - Polymorphisme
 - IV-B - Mise en place d'une "usine" de création d'objets
 - IV-C - utilisation des bibliothèques
- V - Compléments
- VI - Conclusion
- VI - Remerciements

I - Introduction

Les bibliothèques dynamiques sous Windows (les DLL) permettent, d'utiliser des classes C++ de manière simple. Avec un peu d'astuce et grâce aux primitives C permettant la gestion des bibliothèques dynamiques, il est possible de manipuler les classes présentes dans les bibliothèques dynamiques.

L'avantage principal de cette exportation est que la simplicité et la robustesse du code est accrue. Le principal inconvénient est la difficulté de mise en oeuvre et le fait de devoir passer par des primitives C qui sortent un peu du cadre du C++ (ceci est un avis personnel).

Pour aborder cet article, vous devez connaître les bases du C++ et les bases de la bibliothèque standard (STL), plus particulièrement les conteneurs (list, map, vector).

L'article présent est décomposé en trois parties : la première décrivant les primitives C permettant de manipuler les bibliothèques dynamiques, la seconde expliquant les bases pour exporter les classes C++ dans les bibliothèques dynamiques et la dernière qui présente un aspect plus poussé pour gérer de manière plus robuste l'exportation dans les bibliothèques dynamiques.

II - Les fonctions permettant la gestion des bibliothèques dynamiques sous Linux

II-A - Description des primitives

Avant d'aborder le sujet en profondeur, nous allons commencer par expliquer comment créer des bibliothèques dynamiques sous Linux en C. Pour cela, nous avons à notre disposition des primitives :

- *dlopen* : qui permet d'*ouvrir* une bibliothèque dynamique
- *dlsym* : qui permet de charger une fonction
- *dLError* : qui permet d'obtenir le message d'erreur généré par la dernière primitive appelée
- *dlclose* : qui permet de *fermer* la bibliothèque ouverte

L'ordre d'appel des primitives est obligatoirement celui la :

- 1 *dlopen* : en lui passant comme paramètre le nom du fichier représentant la bibliothèque
- 2 *dlsym* (autant de fois que nécessaire) : en lui passant comme paramètre le nom de la fonction à charger
- 3 *dlclose*

La primitive *dLError* doit être utilisée à chaque appel d'une des autres primitives excepté *dlclose*.

II-B - Exemple d'utilisation des primitives

Voici un exemple illustrant l'utilisation d'une bibliothèque dynamique :

Exemple d'utilisation des primitives

```
#include <dlfcn.h>
#include <stdio.h>

int main(int argc, char **argv)
{
    void (*func)();

    // Ouverture de la bibliothèque
    void *hndl = dlopen("./libdyn.so", RTLD_LAZY);
    if(hndl == NULL)
    {
        printf("erreur dlopen : %s\n", dlerror());
        exit(EXIT_FAILURE);
    }

    // Chargement de la fonction "func"
    func = dlsym(hndl, "func");
    if (func == NULL)
    {
        printf("erreur dlsym : %s\n", dlerror());
        dlclose(hndl);
        exit(EXIT_FAILURE);
    }

    // Exécution de la fonction "func"
    func();

    // Fermeture de la bibliothèque
    dlclose(hndl);
}
```

Exemple d'utilisation des primitives

```
return EXIT_SUCCESS;  
}
```

Une entité primordiale dans ce programme est la variable *hndl* représentant un pointeur vers le code chargé de la bibliothèque. Cette variable est passée comme paramètre pour chaque appel de primitive suivante.



*Faites attention à bien charger la fonction (avec *dlsym*) avant de l'utiliser (c'est une erreur fréquente)*

II-C - Options de compilation

Pour qu'un fichier source C soit compilé dans une bibliothèque dynamique, il faut appliquer des options spécifiques lors de sa compilation. Le fichier source est tout ce qu'il y a de plus classique. Cela se déroule en deux étapes :

- 1 Compilation de la bibliothèque en utilisant l'option *-shared*
- 2 Compilation du programme utilisant la bibliothèque avec l'option *-ldl* qui permet de lier la bibliothèque contenant les primitives précités

Voici le code source de l'exemple de base, servez-vous en pour mieux comprendre comment cela fonctionne et n'hésitez pas à consulter les pages du manuel linux (man *dlopen*).

Source [source de l'exemple de base](#)

III - Création, exportation et utilisation des classes C++

Maintenant que les bases concernant les bibliothèques dynamiques sous Linux ont été présentées, nous allons voir comment exporter des classes entières en C++.

III-A - Créateurs

La première partie a montré comment exporter des fonctions dans des bibliothèques dynamiques. L'astuce, pour exporter des classes C++, consiste à créer une fonction spécifique : le créateur.

Le créateur est une fonction qui permet de créer une instance de la classe. La différence avec un constructeur est que le créateur est extérieur à la classe. Le créateur fait appel au constructeur et retourne l'instance créée.

Comme exemple, tout au long de l'article, nous allons utiliser les formes géométriques. Donc, pour un cercle, on aura ceci comme créateur :

le créateur de cercle

```
circle *Create()
{
    return new circle();
}
```

De plus, nous sommes dans un contexte C++, il faut donc rajouter le lien indiquant que les fonctions se situent dans un contexte C. Ainsi, nous pourrons les manipuler comme des fonctions classiques. En effet, les primitives utilisent les fonctions C, voilà pourquoi on doit se placer en contexte C. Voici donc ce qui est obtenu :

Créateur en C++

```
extern "C"
{
    circle *Create()
    {
        return new circle();
    }
}
```

III-B - Exemple d'exportation d'une classe

Maintenant que le créateur a été expliqué, nous allons pouvoir exporter notre première classe. Vous devez savoir que chaque méthode qui sera utilisée par le programme appelant la bibliothèque doit être virtuelle. Ceci est impératif car sinon la compilation provoquera une erreur. Voici la déclaration de la classe "circle" :

code source de circle.h

```
#ifndef CIRCLE_H_
#define CIRCLE_H_

#include <iostream>

class circle
{
public:
    virtual void draw();
};
```

code source de circle.h

```
typedef circle *(*maker_circle)();
#endif
```

La classe *circle* ne possède qu'une méthode nommée *draw* elle permet d'afficher un cercle sur la console, rien de bien compliqué. Les 2 *typedefs* permettent d'alléger la syntaxe du programme principal que l'on verra dans le point suivant. Comme vous pouvez le constater, la méthode est déclarée virtuelle.



J'insiste énormément sur la virtualité de la méthode car c'est une erreur qui peut être difficile à détecter et facile à faire.

Maintenant nous allons passer au code source de *circle.cpp* implémentant les méthodes de classe *circle*, le créateur :

code source de circle.cpp

```
#include "circle.h"

using namespace std;

void circle::draw()
{
    cout << "   ###   " << endl;
    cout << "  #   #  " << endl;
    cout << " #     # " << endl;
    cout << "#   #   " << endl;
    cout << "#   #   " << endl;
    cout << "  ###   " << endl;
}

extern "C"
{
    circle *make_circle()
    {
        return new circle();
    }
}
```

La méthode *draw* est implémentée comme une méthode classique. Le créateur est conforme à ce qui a été décrit précédemment.

Respectez bien ce qui a été expliqué et vous verrez que cela ne sera pas aussi compliqué que cela n'y paraît.

III-C - Utilisation de la classe exportée

Maintenant que le code de la bibliothèque a été expliqué, nous allons nous intéresser à l'utilisation de cette bibliothèque. Voici à quoi ressemble le code source du programme utilisant la bibliothèque :

Code source du programme

```
#include "circle.h"
#include <iostream>
#include <dlfcn.h>

using namespace std;

int main(int argc, char **argv)
{
    void *hndl;
```

Code source du programme

```
maker_circle pMaker;

// Ouverture de la librairie
hndl = dlopen("./libcircle.so", RTLD_LAZY);
if(hndl == NULL)
{
cerr << "dlopen : " << dlerror() << endl;
exit(EXIT_FAILURE);
}

// Chargement du créateur
void *mkr = dlsym(hndl, "make_circle");
if (mkr == NULL)
{
cerr << "dlsym : " << dlerror() << endl;
exit(EXIT_FAILURE);
}
pMaker = (maker_circle)mkr;

// Création, affichage du cercle
circle *my_circle = pMaker();
my_circle->draw();
dlclose(hndl);

return EXIT_SUCCESS;
}
```

Dans ce code, vous pouvez distinguer deux *includes* importants : celui concernant "circle.h" qui permet de connaître le code de la méthode et les 2 typedefs, ainsi que celui contenant les primitives (dlfcn.h).

Ensuite, le créateur est extrait de la bibliothèque comme s'il s'agissait d'une fonction C classique. Après avoir créé l'objet, vous pouvez utiliser ses méthodes. On peut apparenter cette manière d'agir à du C orienté objet (enfin de loin).

III-D - Compilation de la bibliothèque et du programme

Le dernier point concernant cette partie est la compilation. Pour cela nous allons utiliser ce makefile :

source du makefile

```
example: main.cpp libcircle.so
g++ -o example main.cpp -ldl

libcircle.so: circle.cpp circle.h
g++ -shared -o libcircle.so circle.cpp

clean:
rm -f example libcircle.so
```

Comme vous pouvez le constater ce makefile ressemble à celui de la première partie. La seule différence concerne le compilateur qui est ici spécifique à g++.

Maintenant vous savez comment exporter des classes C++ mais cela reste, d'un point de vue conceptuel, pauvre. Dans la prochaine partie nous allons rendre la conception plus robuste.

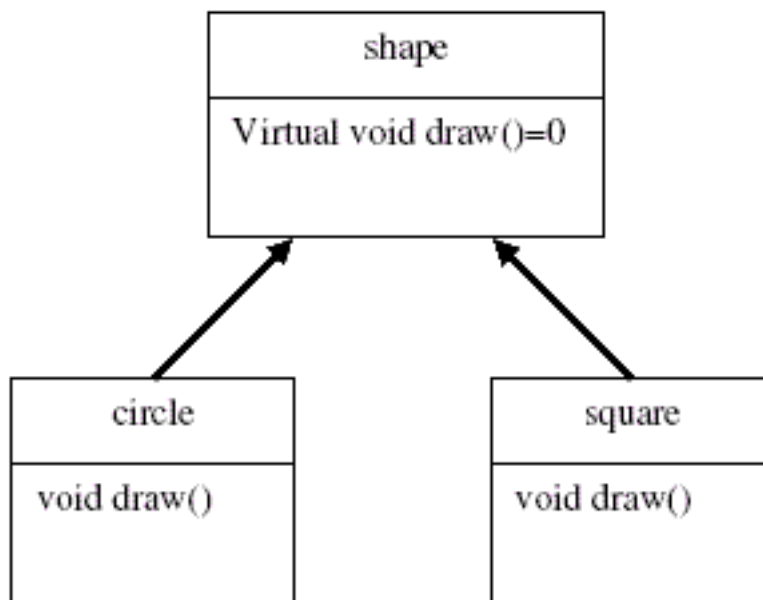
Source [code source de cette partie.](#)

IV - Gestion des classes plus poussée

Pour rendre l'exportation plus robuste nous allons utiliser le concept de polymorphisme pour mettre en place une usine de création d'objets(pattern factory).

IV-A - Polymorphisme

Commençons par un bref rappel concernant le polymorphisme. Prenons la hiérarchie de classe qui sera utilisée comme exemple :



arbre d'héritage de la bibliothèque

Vous pouvez constater que chaque classe possède la méthode *draw*. Le principe du polymorphisme est qu'à l'exécution de ces lignes de codes :

```

shape *c = new circle();
shape *s = new square();
c->draw();
s->draw();
    
```

la méthode *draw* de l'objet "c" correspond à son implémentation dans la classe *circle*, et respectivement celle de l'objet "s" à celle de la classe *square*.

Ainsi, pour résumer, le polymorphisme (d'héritage) est un processus spécifique à la POO qui permet d'assurer une certaine distance vis à vis de l'objet manipulé qui connaît son implémentation propre.

IV-B - Mise en place d'une "usine" de création d'objets

Tout d'abord le prototype du créateur va être adapté. En effet, nous n'allons pas renvoyer un "circle *", par exemple, mais un "shape *" dans **TOUS** les cas. (ne vous inquiétez pas vous allez comprendre un peu plus tard :))

Pour créer l'usine de création d'objets, nous allons utiliser la bibliothèque standard (STL) et plus particulièrement la classe *map* qui permet d'associer un nom à une fonction comme suit :

```
factory["circle"] = make_circle;
```

La variable *factory* sera déclarée comme variable globale externe dans le fichier "shape.h". L'explication sera donnée dans le point suivant. Voici comment est déclarée la variable *factory* :

```
extern map<string, maker_shape, less<string> > factory;
```

En plus de cette usine nous allons ajouter la déclaration suivante dans le bloc *extern "C"* d'un fichier source :

Classe enregistreuse

```
// Classe d'enregistrement
class registrar
{
public:
    registrar()
    {
        factory["circle"] = make_circle;
    }
};

registrar r;
```

Dans notre exemple, nous avons ajouté cette déclaration dans le bloc du fichier source de la classe *circle*. Le principe est simple. La classe "register" permet d'ajouter le créateur correspondant au source dans la variable *factory*. Cette classe est instanciée pour que l'ajout soit effectif. "Register" ne sera jamais utilisé, elle ne sert qu'à effectuer l'opération précédemment décrite.

IV-C - utilisation des bibliothèques

Le point le plus compliqué de l'article est celui là. Tout d'abord, il faut déclarer la variable *factory* comme variable normale. Les variables *factory* des bibliothèques ne sont en fait que des substituts. Ainsi, à chaque ajout de créateur dans la variable *factory*, c'est cette variable qui est manipulée et qui contiendra donc chaque créateur. Cette astuce permet d'éviter le chargement de symbole.

Le polymorphisme prend ici tout son sens, car chaque figure hérite de *shape* ce qui permet de garantir la validité de la procédure pour construire la variable "factory". En effet, chaque créateur renvoyant un "shape *", seul l'objet (et nous aussi quand même) sait de quelle classe il est une instance.

Voici le programme, simple, qui correspond à notre procédé :

main_simple.cpp

```
#include "circle.h"
#include <iostream>
#include <dlfcn.h>
#include <map>
```

main_simple.cpp

```
#include <string>

using namespace std;

map<string, maker_shape, less<string> > factory;

int main(int argc, char **argv)
{
    // Ouverture de la bibliothèque
    void *circle_hndl = dlopen("./libcircle.so", RTLD_LAZY);

    // Vérification de la bibliothèque
    if(circle_hndl == NULL)
    {
        cerr << "dlopen (circle) : " << dlerror() << endl;
        exit(EXIT_FAILURE);
    }

    // Création du cercle
    shape *my_circle = factory["circle"]();

    // Affichage du cercle
    my_circle->draw();

    // Fermeture de la bibliothèque
    dlclose(circle_hndl);

    return EXIT_SUCCESS;
}
```

Ce programme permet juste de vous faire constater la différence avec l'usine : aucun symbole n'est chargé de la bibliothèque ce qui évite un test supplémentaire. De plus, regardez le code suivant qui permet d'afficher une liste de figures :

```
list<shape *>::iterator shape_it;
for (shape_it = shape_list->begin(); shape_it != shape_list->end(); shape_it++)
{
    (*shape_it)->draw();
}
```

Il s'agit de l'affichage de chaque figure présente dans la liste, qu'il s'agisse d'un cercle ou d'un carré. Cela est rendu possible grâce au polymorphisme. Dans le code source, vous trouverez les deux exemples : le simple qui reproduit le fonctionnement du programme du point précédent et le complet qui permet de montrer les possibilités de l'usine. Je vous laisse le soin de vous plonger dans ce dernier exemple pour bien cerner cette partie de l'article. Cela n'est pas compliqué, il suffit de bien exécuter le programme et de comprendre ce qu'il fait, en regardant le code évidemment.

Source [voici le code source de cette partie](#)

V - Compléments

Tout d'abord voici un code source représentant un exemple basique avec quelques possibilités d'utilisation.

 [source exemple](#)

Il faut préciser que plusieurs opérations ne sont pas possibles comme par exemple l'héritage des classes présentes dans la librairie par des classes "personnelles". En effet, la construction de la table permettant de représenter l'arbre d'héritage se passe lors de la compilation et non à l'exécution. Pour plus de précisions, il faut aller voir cette page

 <http://ldp.org/HOWTO/Program-Library-HOWTO/shared-libraries.html>

VI - Conclusion

Nous avons vu dans une première partie qui présente les fonctions dédiées aux manipulations des bibliothèques dynamiques sous linux. Ces fonctions sont utilisées à la base en langage C.

Nous avons vu ensuite comment adapter un code C++ et utiliser les fonctions précédentes pour exporter des classes dans des bibliothèques dynamiques.

Enfin nous avons abordé un petit exemple illustrant plus concrètement les deux parties précédentes.

En conclusion, construire une bibliothèque dynamique demande un effort de réflexion. En effet, il faut penser à ce que la bibliothèque doit contenir mais aussi, et surtout, l'organisation arborescente des classes pour avoir une exportation, et une construction, plus performante.

Enfin, il faut garder à l'esprit que certaines opérations sont impossibles. On peut donc dire qu'exporter des classes C++ sous linux est plus ardu que de le faire sous Windows avec les MFC.

VI - Remerciements

Merci à **Laurent Gomila**, **Farscape**, **Nico-pyright(c)** et **Fearyourself** pour aide et leurs encouragements.

