Plugin sytem

I: Theory

Before using the system, a little bit of theory in order to understand how it is going to work.

I.1: Principle

The aim of this system is to allow the improvements of a software without modifying the main program. At the execution of the main program, it is going to load each plugin found in a certain file. And at the end, it will be able to use the object describe by the plugin without being compiled with the plugin.

This system requires that the plugin follows precise rules in order to be understand by the main program. This rules are going to be explained in the next part.

I.2: How it works

In order to be understand by the main program, the plugin needs to follow the structure describe in Interface.hpp and Interface.cpp. These files are integrasted in the compilation of the main project and they should be integrated in the compilation of the plugin.

The plugin describes an object which inherit from Interface. Interface describes which function needs to be implemented in the plugin in order to be called by the main project. In the folder template_plugin, there are two files which describe the basis of a plugin: template_plugin.hpp and template_plugin.cpp.

When your plugin is done, it needs to be compile with interface.hpp and interface.cpp, and converted in a shared library. When it's done, the plugin folder should be containing all header files and the shared library. This folder and the shared library need to have the same name. By paste this folder in the libraries folder of your main project is going to provoke the load of the plugin during the next execution.

The main program, when it is executed, look at the libraries' folder and list every folder in it. It tries to load every shared library which have for address ./libraries/folder/folder.so. Load a shared library, in this case, means create a pointer on the plugin object. But each plugin have is own name so the program uses a function called "load" present in each plugin. This function, even if it has the same name in each plugin, has not the same implementation. For each plugin, the "load" function construct a pointer on an object and return it as a child of Interface.

Like this, the main program just need a vector of pointer of Interface to store each plugin object. It can use them by using common function like get_name, get_description or another function you add in Interface. Moreover, you can give a rank for the plugin. With this rank, the list of plugins will be ordered as you wish.

I.3: Warning

At that moment, the system is not tested on MacOS or Windows. The compilation of shared library is different in each operating system and the address of a file can use different character (like an antislash). I'm not able to test those operating systems at the moment.

II: Practice

This practice part shows how to create and compile a plugin system. You can also try yourself by calling "./script" followed by "./prog" in a terminal. It is going to compile samples of plugin and the main program. After launching the program, it is going to display the name of each plugin ordered by the rank.

II.1: Plugin interface

interface.hpp shows differents methods which a plugin needs to implements. If you would add a method, just be sure that you add it in interface.hpp as a pure virtual method.

```
#ifndef INTERFACE_HPP
#define INTERFACE_HPP

#include <climits>
#include <iostream>

class Interface
{
    public :
        Interface() ;

        virtual ~Interface() = 0 ;

        virtual std::string get_name()
            { return name_ ; }

        virtual std::string get_description()
            { return description_ ; }

        virtual short get_rank()
            { return rank_ ; }

    protected :
        std::string name_ ;
        std::string description_ ;
        unsigned short rank_ ;
} ;
```

```
#endif // INTERFACE_HPP
```

II.2: Plugin compilation

You can use each compilation line without taking care of the order. The fact is that they are completely separated. In order to compile the plugin into a shared library, you need to use this command line (interface.cpp and interface.hpp are present in your folder and your plugin is in a folder 'plugin', written in plugin.cpp and plugin.hpp):

```
g++ -std=c++11 -fPIC -shared interface.cpp plugin.cpp -o plugin.so
```

In order to compile the main project, you just need to add the libdl in the compilation. For the project where we use a PluginList, the command line is written under. Maybe you just need to add -ldl in your makefile.

```
g++ -std=c++11 interface.cpp plugin.cpp plugin_list.cpp main.cpp -ldl -o prog
```

II.3: Plugin use

I will explain the example, it will be simpler to adapt it if you understand the whole thing. In order to be clear, we are going to load two plugins called 'A' (in './libraries/a') and 'B' (in './libraries/b'):

When you use a PluginList, you need to use the address of the library folder as parameter. In this case, we are going to call PluginList("./libraries"). It is going to look every folder in './libraries' and it will find 'a' and 'b'. It is going to create a Plugin object with the address of the subfolder for each subfolder, and add them into a its own vector of Plugin.

The Plugin is going to extract the name of the library thanks to the address by removing everything before the last ''. It is going to try to open the library and if it's possible, it will create a functor with the load function. If everything works, it calls the load function and receives a pointer on the object.

At this moment, a Plugin object is, for the main program, an interface on a real object that is defined in a plugin. If you would call a function, write the call in a Plugin method and use this method in the main program. For exemple, if you want to add a run() method:

1. Add a pure virtual method in Interface: `virtual void run() = 0 ;`
2. Add a method in Plugin in order to call the run function of a real plugin object. You can obviously use the same name: `void Plugin::run() { pointer_->run() ; }`
3. Optional, you can add a method in PluginList in order to call the run function on each plugin thanks to the `std::vector< Plugin * >`, just like `display_name()`.