VE280 LAB 6

Topic: Class Inheritance

Only one exercise. It will take most of your time exploring and understanding what needs to be done. The code you need to write is quite simple.

In addition of working on class inheritance, you will also practice code reading and learn about the plugin architecture.

Click here to jump to what you should do

1. Plugin based software development

Most of you use Visual Studio Code or CLion for this course. Both IDEs have a lot of powerful or interesting plugins or gadgets, like Rainbow Bracket on CLion, or Bracket Pair Colorizer on Visual Studio Code. Another example is Google Chrome, with some useful extensions like SwitchyOmega or Tampermonkey. They are all examples of a plugin.

From a simplified informal point of view a plugin can be seen as a small piece of software that can be loaded to extend or bring in new features to a host application. For this to work, the host software must expose a plugin API. Then plugins can be developed independently from each others or the main application, i.e., the core software does not need them to compile and run properly. Plugins can hence be implemented following the plugin API, be compiled as shared libraries, and loaded at startup or even at run-time by the host software.

API, Application Programming Interface, of an application defines how other applications can access data from this application.

For instance when developing a music player one might want to introduce plugins to play various file types such as mp3, wav, and flac. In such a case one will want to have a generic play_file() function which would redirect the job to an appropriate function, for example play_mp3(), or play_wav() depending on the file type. Of course if a file type is not supported, e.g., play_flac() does not exist, the program should not crash but simply report that this file type is not supported. In particular this shows the necessity for each plugin to register itself and present some meta-information about itself to the main program.

In a slightly more formal way the plugin architecture is split into four sub-concepts:

- **Discovering**: mechanism allowing the host application to discover available plugins. Usually plugins are found in a specific folders. In this lab, the folder is `./plugins`. This concept is implemented in `PluginManager::discoverPlugin`.
- **Registering**: mechanism allowing the host application to potentially accept and register the discovered plugins; At this stage a plugin announces its features, version, and any other information necessary to the well functioning of the application. In this lab, it is `PluginManager::registerPlugin`.
- **Hooking**: sometimes called mount points or extension points, application hooks can be seen as the core of the plugin manager; They allow the plugin to “attach” itself to the application; Hence the core application can get control over the plugin. In this lab, all the plugins are stored in a vector `PluginManager::pluginList` so that the host application can access and control them.
- **Exposing**: the core application should also expose an API to plugins such that they can call some of its functions; Evidently, not all functions from the core application should be

accessible. The set of such functions can be seen as the part of the plugin manager API that is exposed to the plugins. **This is not covered in this lab.**

2. Core Applications

In this lab you will work on a single program that simulates the described plugin architecture. This program accepts one program argument, which provides the name of the folder that stores all the plugin files.

Plugin files are files with extension `.so`. Such files are dynamic libraries. Dynamic libraries, often mentioned together with static libraries, will be discussed more in EECS 370, EECS/VE 482.

When the program starts, it first scans through the folder, find all files with extension `.so` and load such files as plugins. If the program fails to load one of the plugin file, it will print an error message and continues on the next file.

The program is a Read-Eval-Print Loop program. It is like a simple shell. A shell is where you type and execute the Linux commands introduced in Lecture 1. You can also type some commands in this program. The program will execute your command and print a message.

Here are commands that you can use

```
load <.so filename>                load a plugin by its filename
remove <plugin name>               remove a plugin by its name
print <plugin name>                print the info of the plugin
exit                               exit the program
```

For any other message you type, the program will repeat it.

You can try loading the plugin or removing the plugin when the program is running. A plugin here detects certain patterns in your input. If your input matches such patterns or rules, the program will apply this plugin to handle your command. For details about the plugin `cat`, check the information in the next section 3.1 Cat.

```
dadsh > cat 3
# and then watch the cat for 3 seconds
dadsh > remove cat
dadsh > cat 3
dadsh > load plugins/cat.so
dadsh > cat 3
```

The source code and Makefile of the core applications are provided in the starter files.

Build with

```
make
# or
make l6
```

Execute with

```
./l6 <plugin directory>
```

In most cases you will run with

```
./l6 plugins
```

3. Plugins

Two plugins are provided and they will be loaded when the program starts.

3.1 Cat

This plugin watches for input in the format below

```
dadsh > cat <number>
```

Try it yourself. The number is the execution time (unit: second) of this command. Like for `cat 5`, the plugin will show a cat for 5 seconds.

Its source files are in `plugins/cat`. The compiled dynamic library file is at `plugins/cat.so`

You do not need to understand the source code in `plugins/cat/cat.cpp`. There are some core concepts of EECS/VE 482 in the source code.

3.2 Simple

This plugin watches for input in the format below

```
simple
```

It will print one message.

Its source files are in `plugins/simple`. The compiled dynamic library file is at `plugins/simple.so`

Please make sure you understand the source file here before you start writing your code.

I suggest you try the input list below

```
dadsh > simple
dadsh > remove simple
dadsh > simple
dadsh > load plugins/simple.so
dadsh > simple
```

3.3 Additional Information

In the source code shared with you we use some keywords that have not been introduced in the lectures.

- **override**: In a member function declaration or definition, `override` specifier ensures that the function is virtual and is overriding a virtual function from a base class. The program is ill-formed (a compile-time error will be generated) if this is not true. https://en.cppreference.com/w/cpp/language/override
- **extern "C"**: In C++, when used with a string, `extern` specifies that the linkage conventions of another language are being used for the declarator(s). In short, normally we cannot compile C++ source code and C source code together. But with `extern "C"`, we can first

compile the C++ source code into a library or link file. Then we can compile such file with other C source code.

4. TODO

- Try the program with the provided commands. Try removing one plugin and loading it back. See how such actions change the way that the program handles your input.
- Read through the code file. Understand how the program registers the plugins and how the program applies the plugins to handle the input.
- The source code contains some complex Linux functions and also usage of STL containers or algorithms. Take advantage of a search engine and website resources.
- Write a plugin yourself in `styled.cpp`
  - name: `styled`
  - author : `Second Lobster`
  - description: (Just Write anything you like)
  - help: (Just Write anything you like)

  It should reply to this specific message `WhosYourDaddy` with

  ```
  You are not playing Warcraft 3, guys...
  ```

  When using `print styled`, it should output as below

  ```
  Second Lobster writes styled because he thinks it is fun
  ```

- Write a plugin yourself in `add.cpp`
  - name: `add`
  - author : `Meual`
  - description: `add two integers`
  - help: `Usage: add <integer> <integer>`

  It should output the sum of two integers.

  Assume that all users follow the input format `add <integer> <integer>` and type valid integers.

  When using `print add`, it should output as below

  ```
  Add operations is great     --Meual
  ```

- For JOJ to successfully compile your plugins, make sure that your plugin source codes are in the same directory as `plugin.h`. Namely, you should use

```
#include "plugin.h"
```

5. How to compile

Follow the format below

```
g++ <cpp file> -fPIC -shared -o <output file name> -Wall -Werror -std=c++11
```

There are examples in `Makefile`, check them by yourself.

6. Submission

Zip your `styled.cpp`, `add.cpp` and submit on JOJ.

7. Hint

- Read through the code. Though there are some complex functions that you have never learnt, they have comments.
- All plugins should follow a certain kind of standard so that no matter what the plugin will be, the host application can recognize and connect to the plugin.
- The topic of this lab is class inheritance. Make use of what you have learnt about class inheritance.

Reference

[1] Manuel Charlemagne, VE482 Intro to Operating Systems