2024 年ソフトウェア演習2B

第 5 回課題

B243392　　ALCANDER IMAWAN　　2024 年 7 月 22 日

Q1

プログラムの実行時に引数としてディレクトリ名を与え，そのディレクトリ内に存在するファイル名を vector クラスの変数に格納するプログラムを作成し，動作確認を行いなさい.

プログラム

main.cpp

```
1: #include <iostream>
2: #include <filesystem>
3: #include <vector>
4: #include <string>
5:
6: namespace fs = std::filesystem;
7:
8: int main(int argc, char* argv[]) {
9:         if (argc < 2) {
10:                 std::cerr << "How to use: " << argv[0] << " <directory_path>" << std::endl;
11:                 return 1;
12:         }
13:
14:         std::string directoryPath = argv[1];
15:         std::vector<std::string> fileNames;
16:
17:         try {
18:                 for (const auto& entry : fs::directory_iterator(directoryPath)) {
19:                         if (entry.is_regular_file()) {
```

```
20:                         fileNames.push_back(entry.path().filename().string());
21:                     }
22:                 }
23:         } catch (const fs::filesystem_error& e) {
24:                 std::cerr << "Error accessing directory: " << e.what() << std::endl;
25:                 return 1;
26:         }
27:
28:         std::cout << "Files in directory " << directoryPath << ":" << std::endl;
29:         for (const auto& fileName : fileNames) {
30:                 std::cout << fileName << std::endl;
31:         }
32:
33:         return 0;
34: }
35:
```

解説

main 関数では、コマンドライン引数の数を確認し、ディレクトリパスが指定されていない場合は使用方法を表示して終了する。コマンドライン引数からディレクトリパスを取得し、ファイル名を格納するためのベクターを定義する。try ブロック内で、ディレクトリ内のエントリを反復処理し、通常のファイルである場合はファイル名をベクターに追加する。例外が発生した場合（例えばディレクトリにアクセスできない場合）には、エラーメッセージを表示して終了する。最後に、ベクター内のファイル名を順に出力する。

動作確認

```
D:\Toyohashi_Gikadai\TUT_Software_Enshuu2\report5\Q1>q1 ../q3
Files in directory ../q3:
Add.cpp
libadd.so
plugin.hpp
q3.cpp
q3.exe
```

Q2

calc.cpp のソースコードをコンパイルして動的リンクライブラリを作成し，作成したライブラリをプログラム中で動的ロードして使うプログラムを作成しなさい．動作確認も行うこと．

プログラム

calc.cpp

```
1: extern "C" {
2:         double my_add (double a, double b) {return a + b;}
3:         double my_sub (double a, double b) {return a - b;}
4: }
```

q2.cpp

```
1: #include <iostream>
2: #include <windows.h>
3:
4: typedef double (*AddFunc)(double, double);
5: typedef double (*SubFunc)(double, double);
6:
7: int main() {
8:         // Load the DLL
9:         HMODULE hLib = LoadLibrary("libcalc.so");
10:        if (!hLib) {
11:                std::cerr << "Failed to load DLL: " << GetLastError() << std::endl;
12:                return 1;
13:        }
14:
15:        // Get the function pointers
16:        AddFunc my_add = (AddFunc)GetProcAddress(hLib, "my_add");
17:        SubFunc my_sub = (SubFunc)GetProcAddress(hLib, "my_sub");
18:
19:        if (!my_add || !my_sub) {
20:                std::cerr << "Failed to get function address: " << GetLastError() << std::endl;
```

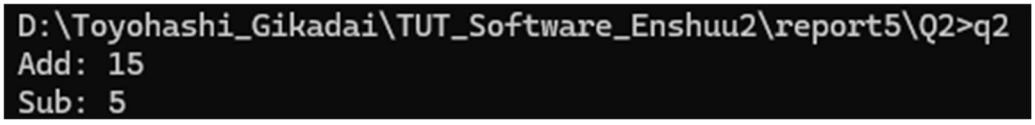
```
21:             FreeLibrary(hLib);
22:             return 1;
23:         }
24:
25:         // Use the functions
26:         double result_add = my_add(10.0, 5.0);
27:         double result_sub = my_sub(10.0, 5.0);
28:
29:         std::cout << "Add: " << result_add << std::endl;
30:         std::cout << "Sub: " << result_sub << std::endl;
31:
32:         // Unload the DLL
33:         FreeLibrary(hLib);
34:
35:         return 0;
36: }
```

解説

動的ライブラリの作成手順をいかにある。

1. calc.cpp を calc.so にコンパイルする。
   g++ -shared -o libcalc.so -fPIC calc.cpp
2. q2.cpp をコンパイルし、リンクする。-L を使わない理由は q2.cpp の中に、LoadLibrary() 関数を使っているからである。この関数の機能は-L の機能と同じである。
   g++ -o q2 q2.cpp
3. コンパイルしたファイルを実行する。
   ./q2.exe

動作確認



Q3

plugin.hpp のソースコードに示す抽象クラスを継承したクラス Add を実装しなさい．メンバ関数はそれぞれ以下のような動作をするものとする．次に，実装した Add クラスを動的リンクライブラリとしてコンパイルし，作成したライブラリをプログラム中で動的ロードして使うプログラムを作成しなさい．動作確認も行うこと．

プログラム

plugin.hpp

```
1: #ifndef __PLUGIN_H__
2: #define __PLUGIN_H__
3:
4: #include <string>
5: #include <memory>
6:
7: class PluginInterface {
8: public:
9:    virtual ~PluginInterface() = default;
10:    virtual std::string getPluginName (void) = 0;
11:    virtual double exec (double a, double b) = 0;
12:    };
13:
14: #endif /* __PLUGIN_H__ */
```

Add.cpp

```
1: #include "plugin.hpp"
2: #include <string>
3:
4: class Add : public PluginInterface {
5: public:
6:         virtual std::string getPluginName() override {
7:                 return "AddPlugin";
8:         }
```

```
9:
10:         virtual double exec(double a, double b) override {
11:                 return a + b;
12:         }
13: };
14:
15: // Export function to create an instance of Add
16: extern "C" PluginInterface* createPlugin() {
17:         return new Add();
18: }
19:
20: // Export function to destroy an instance of Add
21: extern "C" void destroyPlugin(PluginInterface* plugin) {
22:         delete plugin;
23: }
24:
```

q3.cpp

```
1: #include <iostream>
2: #include <memory>
3: #include <windows.h>    // For dynamic linking on Windows
4: #include "plugin.hpp"
5:
6: typedef PluginInterface* (*CreatePluginFunc)();
7: typedef void (*DestroyPluginFunc)(PluginInterface*);
8:
9: int main() {
10:         // Load the dynamic library
11:         HMODULE hLib = LoadLibrary("libadd.so");
12:                 if (!hLib) {
13:                         std::cerr << "Failed to load library: " << GetLastError() << std::endl;
14:                         return 1;
15:                 }
```

```
16:     CreatePluginFunc createPlugin = (CreatePluginFunc)GetProcAddress(hLib,
"createPlugin");
17:     DestroyPluginFunc destroyPlugin = (DestroyPluginFunc)GetProcAddress(hLib,
"destroyPlugin");
18:
19:     // Create and use the plugin
20:     std::unique_ptr<PluginInterface> plugin(createPlugin());
21:     std::cout << "Plugin Name: " << plugin->getPluginName() << std::endl;
22:     double result = plugin->exec(10.0, 5.0);
23:     std::cout << "Result: " << result << std::endl;
24:
25:     // Clean up
26:     destroyPlugin(plugin.release());
27:     FreeLibrary(hLib);
28:
29:     return 0;
30: }
```

解説

Q2 と同じやり方で libadd.so を作成し、それを q3 にリンクする。Add.cpp で getPluginName()関数と exec()関数を作成した。Override は継承クラスのメソッドが基底クラスの仮想メソッドを正しくオーバーライドすることを保証し、コンパイル時にシグネチャの不一致やその他のエラーを検出するのに役立ち、開発者の意図を明示することで、コードの明快さと保守性を高める

CreatePluginFunc は PluginInterface*を返す関数のポインタであり、DestroyPluginFunc は PluginInterface*を入力する関数のポインタであり、void を返す。LoadLibrary はライブラリをリンクする関数である。

std::unique_ptr<PluginInterface> plugin(createPlugin())；
createPlugin 関数を呼び出してプラグインのインスタンスを作成し、std::unique_ptr でラップしてその寿命を自動的に管理する。

std::cout << "Plugin Name：” 「 << plugin->getPluginName() << std::endl;：

プラグインのインスタンスの getPluginName メソッドを使用して、プラグインの名前を表示する。double result = plugin->exec(10.0, 5.0);はプラグインインスタンスの exec メソッドを引数 10.0 と 5.0 で呼び出し、結果を保存する。std::cout << "Result：” 「 << result << std::endl;は exec メソッドの結果を表示します。

動作確認

```
D:\Toyohashi_Gikadai\TUT_Software_Enshuu2\report5\Q3>q3
Plugin Name: AddPlugin
Result: 15
```

Q4

課題 3 で作成した Add クラスの他の四則演算を行うクラスを実装して，動的それぞれの動的リンクライブラリを作成しなさい．次に作成したライブラリをディレクトリ名 plugin 内に置き，プログラム中でディレとクリを走査して，ディレクトリ内にあるライブラリを動的ロードをして使用するプログラムを作成しなさい．

プログラム

Add.cpp

```
1: #include "plugin.hpp"
2: #include <string>
3:
4: class Add : public PluginInterface {
5: public:
6:         virtual std::string getPluginName() override {
7:                 return "AddPlugin";
8:         }
9:
10:          virtual double exec(double a, double b) override {
11:                  return a + b;
12:          }
13: };
14:
```

```
15: // Export function to create an instance of Add
16: extern "C" PluginInterface* createPlugin() {
17:         return new Add();
18: }
19:
20: // Export function to destroy an instance of Add
21: extern "C" void destroyPlugin(PluginInterface* plugin) {
22:         delete plugin;
23: }
24:
```

Subtract.cpp

```
1: #include "plugin.hpp"
2: #include <string>
3:
4: class Subtract : public PluginInterface {
5: public:
6:         virtual std::string getPluginName() override {
7:                 return "SubtractPlugin";
8:         }
9:
10:         virtual double exec(double a, double b) override {
11:                 return a - b;
12:         }
13: };
14:
15: // Export functions
16: extern "C" PluginInterface* createPlugin() {
17:         return new Subtract();
18: }
19:
20: extern "C" void destroyPlugin(PluginInterface* plugin) {
21:         delete plugin;
```

```
22: }
```

Multiply.cpp

```
1: #include "plugin.hpp"
2: #include <string>
3:
4: class Multiply : public PluginInterface {
5: public:
6:         virtual std::string getPluginName() override {
7:                 return "MultiplyPlugin";
8:         }
9:
10:         virtual double exec(double a, double b) override {
11:                 return a * b;
12:         }
13: };
14:
15: // Export functions
16: extern "C" PluginInterface* createPlugin() {
17:         return new Multiply();
18: }
19:
20: extern "C" void destroyPlugin(PluginInterface* plugin) {
21:         delete plugin;
22: }
```

Divide.cpp

```
1: #include "plugin.hpp"
2: #include <string>
3:
4: class Divide : public PluginInterface {
5: public:
6:         virtual std::string getPluginName() override {
```

```
7:                 return "DividePlugin";
8:         }
9:
10:         virtual double exec(double a, double b) override {
11:                 return a / b;
12:         }
13: };
14:
15: // Export functions
16: extern "C" PluginInterface* createPlugin() {
17:         return new Divide();
18: }
19:
20: extern "C" void destroyPlugin(PluginInterface* plugin) {
21:         delete plugin;
22: }
```

q4.cpp

```
1: #include <iostream>
2: #include <memory>
3: #include <vector>
4: #include <string>
5: #include <filesystem>
6: #include <windows.h>    // For dynamic linking on Windows
7: #include "plugin.hpp"
8:
9: namespace fs = std::filesystem;
10:
11: typedef PluginInterface* (*CreatePluginFunc)();
12: typedef void (*DestroyPluginFunc)(PluginInterface*);
13:
14: int main() {
15:         std::vector<std::unique_ptr<PluginInterface>> plugins;
```

```
16:     std::vector<void*> handles;
17:
18:     // Scan the plugin directory
19:     for (const auto& entry : fs::directory_iterator("/plugin")) {
20:         std::string path = entry.path().string();
21:
22:         if (path.find(".so") == std::string::npos) continue;
23:
24:         HMODULE hLib = LoadLibrary(path.c_str());
25:             if (!hLib) {
26:             std::cerr << "Failed to load DLL: " << GetLastError() << std::endl;
27:             continue;
28:         }
29:         CreatePluginFunc createPlugin = (CreatePluginFunc)GetProcAddress(hLib,
"createPlugin");
30:         DestroyPluginFunc destroyPlugin = (DestroyPluginFunc)GetProcAddress(hLib,
"destroyPlugin");
31:
32:
33:         // Create and store the plugin
34:         plugins.emplace_back(createPlugin());
35:     }
36:
37:     // Use the plugins
38:     for (const auto& plugin : plugins) {
39:         std::cout << "Plugin Name: " << plugin->getPluginName() << std::endl;
40:         double result = plugin->exec(10.0, 5.0);
41:         std::cout << "Result: " << result << std::endl;
42:     }
43:
44:     // Clean up
45:     for (size_t i = 0; i < plugins.size(); ++i) {
```

```
46:             DestroyPluginFunc destroyPlugin =
(DestroyPluginFunc)GetProcAddress((HMODULE)handles[i], "destroyPlugin");
47:             destroyPlugin(plugins[i].release());
48:             FreeLibrary((HMODULE)handles[i]);
49:         }
50:
51:         return 0;
52: }
```

解説

ライブラリの作り方

```
g++ -shared -o plugin/libadd.so -fPIC add.cpp
g++ -shared -o plugin/libsubtract.so -fPIC subtract.cpp
g++ -shared -o plugin/libmultiply.so -fPIC multiply.cpp
g++ -shared -o plugin/libdivide.so -fPIC divide.cpp
```

Add、Subtract、Multiply、および Divide の各クラスを実装し、それぞれを動的リンクライブラリとしてコンパイルし、れらのクラスはそれぞれ別々の動的リンクライブラリ（DLL）としてコンパイルされた。

次に、main.cpp ファイルで std::filesystem を使って特定のディレクトリを走査し、動的リンクライブラリ（.so ファイル）を探し、見つかった各ライブラリを動的にロードした。これにより、プラグインのように機能するクラスのインスタンスを作成することが可能になった。また、各ライブラリ内のクラスのインスタンスを生成した。最後に、各クラスのメソッドを呼び出して、その動作を確認した。

動作確認

```
D:\Toyohashi_Gikadai\TUT_Software_Enshuu2\report5\Q4>q4
Plugin Name: AddPlugin
Result: 15
Plugin Name: DividePlugin
Result: 2
Plugin Name: MultiplyPlugin
Result: 50
Plugin Name: SubtractPlugin
Result: 5
```

自己チェック項目

3 動的ロードの仕組みを理解した.
3 動的リンクライブラリに実装された関数をプログラム中で動的ロードして使うことができる.
3 動的リンクライブラリに実装されたクラスをプログラム中で動的ロードして使うことができる.