1　課題 1

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、動作確認をおこなった。

プログラム 1: program1.cpp

```cpp
#include <dirent.h>

#include <cstring>
#include <iostream>
#include <string>
#include <vector>

int main(int argc, char* argv[]) {
  // 引数として与えられたディレクトリ名を取得
  if (argc < 2) {
    std::cerr << "Usage: " << argv[0] << " <directory_path>" << std::endl;
    return 1;
  }

  std::string directory_path = argv[1];

  // ディレクトリが存在するかチェック
  DIR* dir = opendir(directory_path.c_str());
  if (!dir) {
    std::cerr << "Error: Directory does not exist or is not a directory."
              << std::endl;
    return 1;
  }

  // ファイル名を格納する vector クラスの変数を宣言
  std::vector<std::string> filenames;
  struct dirent* entry;

  // ディレクトリ内のファイル名を取得
  while ((entry = readdir(dir)) != nullptr) {
    if (entry->d_type == DT_REG) {
      filenames.push_back(entry->d_name);
    }
  }

  // ディレクトリをクローズ
  closedir(dir);

  // 格納されたファイル名を表示
  std::cout << "Files in directory '" << directory_path << "':" << std::endl;
  for (const std::string& filename : filenames) {
    std::cout << filename << std::endl;
  }

  return 0;
```

```
46 }
```

指定されたディレクトリ内のファイル名を取得し、それらを `filenames` に格納する。ディレクトリ内のファイルを走査するために、`std::filesystem` ライブラリを使用する。`std::filesystem::directory_iterator` を使ってディレクトリ内のファイルを取得し、その名前を `filenames` に追加し、出力した。

1.1 実行結果

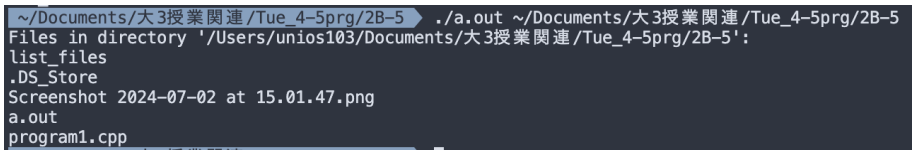

図 1: 課題 1 の実行結果

2 課題 2

`calc.cpp` 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、動作確認をおこなった。

プログラム 2: calc.cpp

```cpp
extern "C" {
double my_add(double a, double b) { return a + b; }
double my_sub(double a, double b) { return a - b; }
}
```

プログラム 3: program2.cpp

```cpp
#include <dlfcn.h>

#include <iostream>

typedef double (*func_t)(double, double);

int main() {
  // ライブラリのロード
  void* handle = dlopen("../library/libcalc.so", RTLD_LAZY);
  if (!handle) {
    std::cerr << "Error loading library: " << dlerror() << std::endl;
    return 1;
  }

  // 関数のシンボルを取得
  func_t my_add = (func_t)dlsym(handle, "my_add");
  func_t my_sub = (func_t)dlsym(handle, "my_sub");

  if (!my_add || !my_sub) {
```

```
    std::cerr << "Error finding symbols: " << dlerror() << std::endl;
    dlclose(handle);
    return 1;
  }

  double a = 5.0, b = 3.0;
  std::cout << "my_add(" << a << ", " << b << ") = " << my_add(a, b)
            << std::endl;
  std::cout << "my_sub(" << a << ", " << b << ") = " << my_sub(a, b)
            << std::endl;

  dlclose(handle);

  return 0;
}
```

与えられた関数 (`my_add`, `my_sub`) を含む `calc.cpp` をコンパイルしてライブラリを作成した。その後、プログラム中で `dlopen` を使ってライブラリをロードし、`dlsym` を使って関数を取得、実行した。

2.1 実行結果

図 2: 課題 2 の実行結果

3 課題 3

与えられた抽象クラスを継承した `Add` クラスを実装し、そのメンバ関数を実装する。`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、実行する。

プログラム 4: plugin.hpp

```
#ifndef __PLUGIN_H__
#define __PLUGIN_H__

#include <memory>
#include <string>

class PluginInterface {
 public:
  virtual ~PluginInterface() = default;
  virtual std::string getPluginName(void) = 0;
  virtual double exec(double a, double b) = 0;
};

```

```
#endif /* __PLUGIN_H__ */
```

プログラム 5: add.cpp

```
#include "plugin.hpp"

class Add : public PluginInterface {
 public:
  std::string getPluginName(void) override { return "Add"; }

  double exec(double a, double b) override { return a + b; }
};

extern "C" PluginInterface* create() { return new Add(); }

extern "C" void destroy(PluginInterface* p) { delete p; }
```

プログラム 6: main.cpp

```
#include <dlfcn.h>

#include <iostream>

#include "plugin.hpp"

typedef PluginInterface* (*create_t)();
typedef void (*destroy_t)(PluginInterface*);

int main() {
  // ライブラリのロード
  void* handle = dlopen("./libadd.so", RTLD_LAZY);
  if (!handle) {
    std::cerr << "Error loading library: " << dlerror() << std::endl;
    return 1;
  }

  // create 関数のシンボルを取得
  create_t create = (create_t)dlsym(handle, "create");
  if (!create) {
    std::cerr << "Error finding create symbol: " << dlerror() << std::endl;
    dlclose(handle);
    return 1;
  }

  // destroy 関数のシンボルを取得
  destroy_t destroy = (destroy_t)dlsym(handle, "destroy");
  if (!destroy) {
    std::cerr << "Error finding destroy symbol: " << dlerror() << std::endl;
    dlclose(handle);
    return 1;
  }

  // Add クラスのインスタンスを作成
  PluginInterface* plugin = create();
```

```
36   if (!plugin) {
37     std::cerr << "Error creating plugin instance" << std::endl;
38     dlclose(handle);
39     return 1;
40   }
41
42   // プラグインの名前と exec 関数のテスト
43   std::cout << "Plugin Name: " << plugin->getPluginName() << std::endl;
44   double a = 5.0, b = 3.0;
45   std::cout << "exec(" << a << ", " << b << ") = " << plugin->exec(a, b)
46             << std::endl;
47
48   // インスタンスを破棄
49   destroy(plugin);
50   dlclose(handle);
51
52   return 0;
53 }
```

`Add` クラスを実装し、`getPluginName` と `exec` メソッドを定義した。次に、このクラスを含む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た。最後に、プログラム中でライブラリを動的にロードし、`create` 関数を使って `Add` クラスのインスタンスを生成し、メソッドを呼び出した。

3.1 実行結果

```
~/Documents/大3授業関連/Tue_4-5prg/2B-5/program3  g++ -shared -fPIC -o libadd.so add.cpp
~/Documents/大3授業関連/Tue_4-5prg/2B-5/program3  g++ -o main main.cpp -ldl

~/Documents/大3授業関連/Tue_4-5prg/2B-5/program3  ./main
Plugin Name: Add
exec(5, 3) = 8
```

図 3: 課題 3 の実行結果

4 課題 4

課題 3 で作成した `Add` クラスの他に、他の四則演算を行うクラス (`Subtract`, `Multiply`, `Divide`) を実装し、それぞれ動的リンクライブラリを作成する。プログラム中でディレクトリを走査してライブラリを動的にロードして使用するプログラムを作成し、動作確認をおこなった。

プログラム 7: plugin.hpp

```
1 #ifndef __PLUGIN_H__
2 #define __PLUGIN_H__
3
4 #include <memory>
5 #include <string>
6
7 class PluginInterface {
```

```
 public:
  virtual ~PluginInterface() = default;
  virtual std::string getPluginName(void) = 0;
  virtual double exec(double a, double b) = 0;
};

#endif /* __PLUGIN_H__ */
```

プログラム 8: add.cpp

```
#include "plugin.hpp"

class Add : public PluginInterface {
 public:
  std::string getPluginName(void) override { return "Add"; }

  double exec(double a, double b) override { return a + b; }
};

extern "C" PluginInterface* create() { return new Add(); }

extern "C" void destroy(PluginInterface* p) { delete p; }
```

プログラム 9: divide.cpp

```
#include <stdexcept>

#include "plugin.hpp"
class Divide : public PluginInterface {
 public:
  std::string getPluginName(void) override { return "Divide"; }

  double exec(double a, double b) override {
    if (b == 0.0) {
      throw std::runtime_error("Division by zero");
    }
    return a / b;
  }
};

extern "C" PluginInterface* create() { return new Divide(); }

extern "C" void destroy(PluginInterface* p) { delete p; }
```

プログラム 10: multiply.cpp

```
#include "plugin.hpp"

class Multiply : public PluginInterface {
 public:
  std::string getPluginName(void) override { return "Multiply"; }

  double exec(double a, double b) override { return a * b; }
};
```

```
9
10 extern "C" PluginInterface* create() { return new Multiply(); }
11
12 extern "C" void destroy(PluginInterface* p) { delete p; }
```

プログラム 11: subtract.cpp

```
1  #include "plugin.hpp"
2
3  class Subtract : public PluginInterface {
4   public:
5    std::string getPluginName(void) override { return "Subtract"; }
6
7    double exec(double a, double b) override { return a - b; }
8  };
9
10 extern "C" PluginInterface* create() { return new Subtract(); }
11
12 extern "C" void destroy(PluginInterface* p) { delete p; }
```

プログラム 12: main.cpp

```
1  #include <dlfcn.h>
2
3  #include <filesystem>
4  #include <iostream>
5  #include <vector>
6
7  #include "plugin.hpp"
8
9  namespace fs = std::filesystem;
10
11 typedef PluginInterface* (*create_t)();
12 typedef void (*destroy_t)(PluginInterface*);
13
14 int main() {
15   std::string plugin_dir = "./plugin";
16
17   if (!fs::exists(plugin_dir) || !fs::is_directory(plugin_dir)) {
18     std::cerr << "Error: Directory does not exist or is not a directory."
19               << std::endl;
20     return 1;
21   }
22
23   std::vector<void*> handles;
24   std::vector<PluginInterface*> plugins;
25
26   for (const fs::directory_entry& entry : fs::directory_iterator(plugin_dir))
       {
27     if (fs::is_regular_file(entry.path())) {
28       void* handle = dlopen(entry.path().c_str(), RTLD_LAZY);
29       if (!handle) {
30         std::cerr << "Error loading library: " << dlerror() << std::endl;
31         continue;
```

```
      }

      create_t create = (create_t)dlsym(handle, "create");
      if (!create) {
        std::cerr << "Error finding create symbol: " << dlerror() << std::endl
    ;
        dlclose(handle);
        continue;
      }

      PluginInterface* plugin = create();
      if (plugin) {
        plugins.push_back(plugin);
        handles.push_back(handle);
      } else {
        dlclose(handle);
      }
    }
  }

  double a = 6.0, b = 2.0;
  for (PluginInterface* plugin : plugins) {
    std::cout << "Plugin Name: " << plugin->getPluginName() << std::endl;
    try {
      std::cout << "exec(" << a << ", " << b << ") = " << plugin->exec(a, b)
                << std::endl;
    } catch (const std::exception& e) {
      std::cerr << "Error: " << e.what() << std::endl;
    }
  }

  for (size_t i = 0; i < plugins.size(); ++i) {
    destroy_t destroy = (destroy_t)dlsym(handles[i], "destroy");
    if (destroy) {
      destroy(plugins[i]);
    }
    dlclose(handles[i]);
  }

  return 0;
}
```

`Add`, `Subtract`, `Multiply`, `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た。`main.cpp` 中では、`std::filesystem` を使ってディレクトリを走査し、各ライブラリを動的にロードして、プラグインのインスタンスを生成し、メソッドを呼び出して動作を確認した。

4.1 実行結果

```
~/Documents/大3授業関連/Tue_4-5prg/2B-5/program4 ./a.out
Plugin Name: Multiply
exec(6, 2) = 12
Plugin Name: Subtract
exec(6, 2) = 4
Plugin Name: Divide
exec(6, 2) = 3
Plugin Name: Add
exec(6, 2) = 8
```

図 4: 課題 4 の実行結果

5 自己チェック

以下の項目について、1 から 4 までの 4 段階で自己評価した。

- 【3】動的ロードの仕組みを理解した.
- 【2】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.
- 【2】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.