ソフトウェア演習２B

プラグイン：ライブラリの動的ロード

動的ロード

- 動的リンク
  - ライブラリの動的リンクとは、プログラムのコンパイル時にライブラリのリンク情報を組み込んでおいて、プログラムの実行時に自動的にライブラリを動的にリンクすることである
- 動的ロード
  - ライブラリの動的ロードとは、プログラムの中でライブラリを読み込む（ロードする）ことである

プログラム中でライブラリを動的ロードする方法

1. ヘッダファイル dlfcn.h をインクルードする

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

2. 関数 dlopen でライブラリをロードする

```
void* handle = dlopen (“./libhogehoge.so”, RTLD_LAZY);
```

ファイル名をパス付きで指定した場合は、環境変数にパスが登録されていなくてもOK
ファイル名のみを指定した場合は、環境変数に設定されたパスから探索

3. 使用しなくなったら関数 dlclose を実行する

```
dlclose (handle);
```

ロードしたライブラリ中の関数を使う方法

- 関数を使用するためには、その関数に関する以下の情報を知っておかないといけない
  - 関数名
  - 関数の戻り値
  - 関数の引数

- 関数 dlsym を用いて関数へのポインタを取得する

double add (double a, double b) という関数を使用する場合：

```
double (*add_func) (double, double) =
  (double(*) (double, double)) dlsym (handle, “add”);


auto add_func = reinterpret_cast<double(*)(double, double)>(dlsym (handle, “add”));

double a = add_func (10, 20);
```

動的ロードでクラスを使用する方法

1. ライブラリ内でクラスを実装する

2. ライブラリ内でクラスのインスタンスを生成する関数を実装する

3. ライブラリをロードしたプログラム中でインスタンスを生成する関数を呼び出すことでクラス変数を使用することができる
   - ライブラリ中の関数を使用する方法は前のスライドの通り
   - ただし、使用するクラスは抽象クラスを継承したクラスである必要がある

動的ロードでクラスを使用する方法：抽象クラスの実装

- 次のような抽象クラスを実装する
  - このクラスは継承されることを前提として定義されていて、このクラスのインスタンスは生成できない。

```
class  Message {
public:
  virtual ~Message() = default;
  virtual Output (void) = 0;
};
```

動的ロードでクラスを使用する方法：抽象クラスを継承したクラスを実装

- 次のようなクラスを実装する

```
class  MessageEng: public Message {
public:
  MessageEng() {}
  ~MessageEng() {}
  void Output (void) override {
    std::cout << “hello” << std::endl;
  }
};
```

- クラスのインスタンスを作成する関数を実装する

```
#include <memory>
extern “C” {
  std::unique_ptr<Message> new_instance (void) {
    return std::unique_ptr<Message> (new MessageEng);
  }
}
```

← CとC++で実装したものが混在する場合に必要

動的ロードでクラスを使用する方法：プログラムでの使用

- 実装したMessageEngクラスを動的ライブラリとして作成する
  1. g++ -c –fPIC MessageEng.cpp
  2. g++ -shared MessageEng.o –o libmessageEng.so
- プログラム中で関数dlsymを使って関数new_instanceを使えるようにする

```
auto new_instance =
  reinterpret_cast<std::unique_ptr<Message>(*)(void)>(dlsym (handle, “new_instance”));
```

- 関数new_instanceを使ってクラスのインスタンスを生成して使用する

```
auto instance = new_instance();
instance->Output; // instance はポインタ変数なのでそのメンバにアクセスする際には->演算子を使用する！
```

プラグイン

- 動的ロードを用いるとプログラムに追加機能を与えるようなプラグインを実装することができる
- 抽象クラスを用いると、入出力の仕様は共通で動作の異なるプラグインを実装することができる
  - 例：画像処理のプラグイン・・・画像を入力として、色を変化させたり画像をぼかすような効果をもつような様々なプラグインを実装できる
- プラグイン用のディレクトリを用意しておき、プログラム実行時にそのディレクトリ内の動的リンクライブラリを読み込みプラグインとして使用する

ディレクトリの走査

- ディレクトリ内のファイルを調べるには DIR構造体を使用する
- 以下の関数を使ってディレクトリ内のファイルを調べる
  - opendir：指定したディレクトリをオープンする
  - readdir：opedirで取得したDIR構造体からファイル名を順番に取得する
  - closedir：オープンしたディレクトリをクローズする
- 指定したディレクトリ内のファイル名を列挙する例：

```
#include <dirent.h>
#include <string.h>
…
DIR* dir = opendir (argv[1]);
struct dirent* file;
while ((file = readdir (dir)) != nullptr) {
  if (strcmp (file->d_name, “.”) != 0 && strcmp (file->d_name, “..”) != 0) {
    std::cout << file->d_name << std::endl;
  }
}
```

おまけ：unique_ptrについて

- g++で実装されているスマートポインタという機能（クラス）の一つ
- 通常 new で動的にメモリ領域を確保した場合には、delete を実行してメモリ領域を解放する必要がある。
  - new でクラスのインスタンスを生成した場合、delete を実行しないとそのインスタンスは残ったままになる
    → デストラクタが呼び出されない
- ポインタ変数をスマートポインタで管理するとクラスのインスタンスが使われなくなったときに自動的にデストラクタを呼び出してくれる！
- スマートポインタの使い方の例：

```
// 通常のポインタ変数
Message* instance1 = new Message;

// スマートポインタの使用

std::unique_ptr<Message> instance2 = std::unique_ptr<Message>(new Message);
```