2024 年ソフトウェア演習2B

第３ 回課題

**B243392　　ALCANDER IMAWAN　　2024年6月25日**

見やすくするために、Report2のプログラムをいかに添付した。

**プログラム**

Message.h

1: #include <string>

2: #include <vector>

3: #include <iostream>

4:

5: class Message {

6:

7: private:

8: std::vector<std::string> message;

9:

10: public:

11: Message(); // Constructor

12: Message(const std::string& message\_string);

13: Message(const std::vector<std::string>& message\_vector);

14:

15: ~Message(); //Destructor

16:

17: void addMessage(const std::string& message\_string);

18: std::string getMessage(int message\_id);

19: void showAllMessages(void);

20: int getNMessages(void);

21: };

Message.cpp

1: #include "Message.h"

2: #include <string.h>

3: #include <string>

4: #include <stdlib.h>

5:

6:

7: // Constructer initializing

8: Message::Message(){}

9:

10: //single message constructor

11: Message::Message(const std::string& message\_string){

12: message.push\_back(message\_string);

13: }

14:

15: //multiple vector messages constructor

16: Message::Message(const std::vector<std::string>& message\_vector){

17: message = message\_vector;

18: }

19:

20: Message::~Message(){}//Destructor

21:

22: //add ma message to the list

23: void Message::addMessage(const std::string& message\_string){

24: message.push\_back(message\_string);

25: }

26:

27: //get a message by message\_id

28: std::string Message::getMessage(int message\_id){

29: if(message\_id >= 0 && message\_id < message.size()){

30: return message[message\_id];

31: }

32: else {

33: return "Message ID not found";

34: }

35: }

36:

37: //showing all messages

38: void Message::showAllMessages(void){

39: for(const auto& msg : message){

40: std::cout << msg << std::endl;

41: }

42: }

43:

44: //get the num of messages

45: int Message::getNMessages(void){

46: return message.size();

47: }

main.cpp

1: #include "Message.h"

2:

3: int main (int argc, char \*argv[]){

4: //testing default constructor

5: Message obj1;

6: obj1.addMessage("Hello World.");

7: obj1.addMessage("Hello 2nd one\n");

8: std::cout << "Number of messages: " << obj1.getNMessages() << std::endl;

9: obj1.showAllMessages();

10:

11: //testing single message constructor

12: Message obj2("This is single message constructor\n");

13: obj2.showAllMessages();

14:

15: //testing vector of messages constructor

16: std::vector<std::string> vec = {"1st message", "2nd message", "3rd message"};

17: Message obj3(vec);

18: std::cout << "Number of vector messages: " << obj3.getNMessages() << std::endl;

19: obj3.showAllMessages();

20:

21: //testing getMessage

22: std::cout << "testing getMessage for index 1 vector : " << obj3.getMessage(1) << std::endl;

23:

24: return 0;

25: }

**Q1**

静的リンクライブラリの作成方法は以下の手順で示す。

1. ソースプログラムをオブジェクトファイルにコンパイルする。’.cpp’ファイルを’.o’ファイルにコンパイルする。
2. 静的リンクライブラリを作成する。
3. 静的リンクライブラリを’main.cpp’にリンクする。

コマンドラインは以下の通りになる。

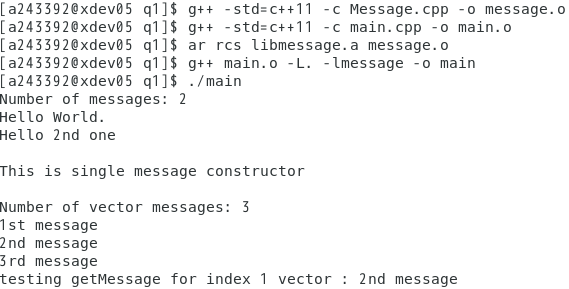
1. g++ -std=c++11 -c message.cpp -o message.o //message.oを作成する。
2. g++ -std=c++11 -c main.cpp -o main.o //main.oを作成する。
3. ar rcs libmessage.a message.o //libmessage.a静的ライブラリをmessage.oを用いて作る。
4. g++ -std=c++11 main.o -L -lmessage -o main //main.oをコンパイルするとき、libmessageをリンクしてコンパイルする。
5. ./main //プログラムを実行する。

**解説**

学校のlinux環境は-std=c++11を設定必要がある。なぜかというと、学校の環境上ではデフォルトのC++コンパイラがバージョン11ではないからだ。

まず、すべてのソースファイルをコンパイルする。コンパイルしたソースファイルを用いて、静的ライブラリlibmessage.aをmessage.oに基づいて作る。Arはarchiverの省略である。次に、静的ライブラリをmainプログラムコンパイルするときにリンクする。-lmessageはlibmessage.aを指す。-lmessageを使わない時、’static\_library\_name.a’に変えても構わない。最後に、mainアプリを実行する。

**動作確認**



**Q2**

動的リンクライブラリの作成方法は以下の手順で示す。

1. ソースプログラムをオブジェクトファイルにコンパイルする。’.cpp’ファイルを’.o’ファイルにコンパイルする。でも、位置に依存しない形にしないといけない。そのため、’-fPIC’を使わないといけない。
2. 動的リンクライブラリ(シェアライブラリ)を作成する。
3. 動的リンクライブラリを’main.cpp’にリンクする。

コマンドラインは以下の通りになる。

1. g++ -std=c++11 -fPIC -c message.cpp -o message.o //message.oを作成する。
2. g++ -std=c++11 -shared -o libmessage.so message.o //libmessage.so動的ライブラリをmessage.oを用いて作る。
3. g++ -std=c++11 main.cpp -L libmessage.so -o main //main.oをコンパイルするとき、libmessage.soをリンクしてコンパイルする。
4. export LD\_LIBRARY\_PATH=.:$LD\_LIBRARY\_PATH //動的リンカーが動的ライブラリを探すため、’LD\_LIBRARY\_PATH’を環境関数に設定しないといけない。
5. ./main //プログラムを実行する。

**解説**

まず、message.oのオブジェクトを作る。-fPIC(Position-Independent Code)はファイルの場所を独立するファイルのフラグである。次に、動的ライブラリを作る。-sharedフラグは動的ライブラリを作るコマンド。mainアプリをコンパイルするとき、動的ライブラリをリンクする。動的リンカーが動的ライブラリを探すため、’LD\_LIBRARY\_PATH’を環境関数に設定しないといけない。最後に、mainアプリを実行する。

**動作確認**

**A screenshot of a computer

Description automatically generated**

**Q3**

静的ライブラリと動的ライブラリの削除時の動作の違いは、これらのライブラリがどのようにリンクされ、実行ファイルによって使用されるかに起因する。

静的ライブラリの場合、静的ライブラリとリンクすると、リンク時に静的ライブラリの内容が実行ファイルにコピーされる。つまり、実行可能ファイルが作成されると、もはや静的ライブラリファイルには依存しなくなる。ライブラリのコードは実行ファイルに直接埋め込まれる。したがって、実行ファイルを作成した後、静的ライブラリを削除しても、実行ファイルは実行される。

一方、動的（共有）ライブラリとリンクする場合、実行ファイルにはライブラリのコードは含まれない。その代わりに、実行時に解決されなければならない共有ライブラリへの参照が含まれる。動的ライブラリは、プログラムの実行時に利用可能である必要がある。動的ライブラリーが削除されると、動的リンカーは必要なライブラリーを見つけロードすることができなくなり、プログラムの実行に失敗する。

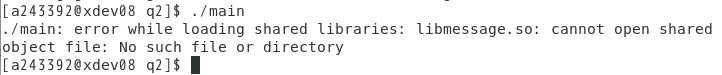
**動作確認**

**Q1のライブラリを消す結果**

**A screenshot of a computer

Description automatically generated**

**Q2のライブラリを消す結果結果**



**自己チェック項目**

4 静的リンクライブラリを作成できる.

4 静的リンクライブラリのリンク方法を理解した.

4 動的リンクライブラリを作成できる.

3 動的リンクライブラリのリンク方法を理解した.

4 それぞれのライブラリをリンクしたプログラムの実行時の振る舞いを理解した.