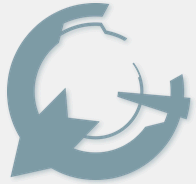
実用C++ハンズオン

分割ビルド編



東京工業大学
デジタル創作同好会

目次

- 翻訳単位について
- #includeとヘッダファイルについて
- 関数のファイル分割
- クラスのファイル分割
- インクルードガード
- ライブラリの使用

目次

- 翻訳単位について
- #includeとヘッダファイルについて
- 関数のファイル分割
- クラスのファイル分割
- インクルードガード
- ライブラリの使用

C++最大の初心者バイバイ

それは...

C++最大の初心者バイバイ

複数ファイルでの
コンパイル方法が良く分からない！

C++での分割ビルドを巡る状況

他のモダン言語なら言語レベルでサポートされがちな

- **「モジュール」概念**
- **「パッケージ」概念**

これらを言語レベルでは**サポートしていない！！**

(モジュールはC++20から加わったけど普及してるかは微妙)

代わりに何があるのか

それは...

代わりに何があるのか

それは...

翻訳単位

代わりに何があるのか

翻訳単位を理解すれば分割コンパイルが分かる

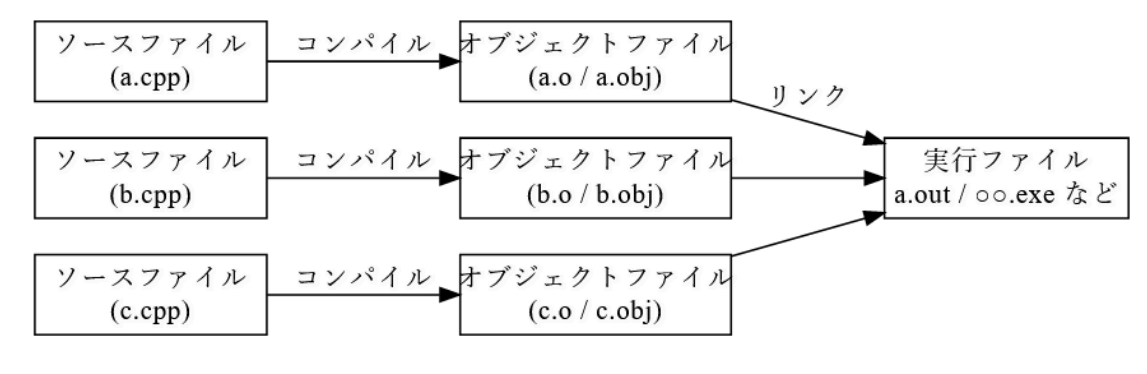
翻訳単位とは

翻訳単位とは

ソースファイル1つ = 1つの翻訳単位

ね、簡単でしょう？

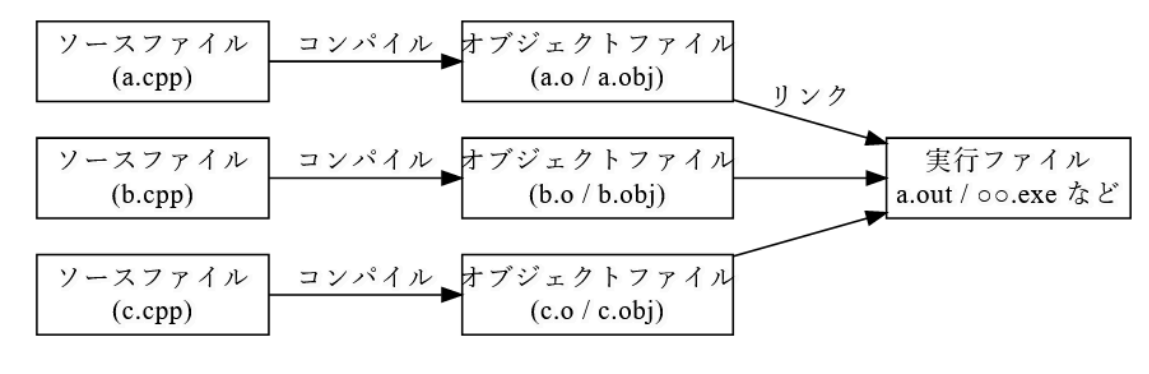
C++のビルドの流れ



1. ソースファイル(=翻訳単位)ごとに「コンパイル」される
2. オブジェクトファイルを「リンク」して最終的なバイナリになる

オブジェクトファイルとは: 機械語+リンクの為の補助情報の入ったファイル

C++のビルドの流れ



重要なこと:「コンパイル」の段階では他の翻訳単位のことは知ったこっちゃない

演習: 2つのファイルをコンパイル

それぞれ `func.cpp` , `main.cpp` とする

```cpp
#include <iostream>
using namespace std;

void func() {
    cout << "Hello" << endl;
}
```

```cpp
void func();

int main() {
    func();
    return 0;
}
```

演習: 2つのファイルをコンパイル

1. コンパイル

```
g++ -c func.cpp
g++ -c main.cpp
```

→ `func.o`, `main.o` ができる

これが**オブジェクトファイル！**

演習: 2つのファイルをコンパイル

2. リンク

```
g++ main.o func.o
```

→ `a.out` ができる

実行ファイル完成！

演習: 2つのファイルをコンパイル

普通はこうでいい

```
g++ main.cpp func.cpp
```

プロトタイプ宣言

先ほどの `main.cpp` の中の「プロトタイプ宣言」

```
// ↓これ！
void func();

int main() {
    func();
    return 0;
}
```

プロトタイプ宣言

先ほどの `main.cpp` の中の「プロトタイプ宣言」

```
// ↓これ！
void func();

int main() {
    func();
    return 0;
}
```

もしこれが無かったら？

プロトタイプ宣言

コンパイルエラー発生！

```
error: ‘func’ was not declared in this scope
```

関数 `func` が定義されていないと言われている

プロトタイプ宣言

コンパイルエラー発生！

```
error: ‘func’ was not declared in this scope
```

関数 `func` が定義されていないと言われている

重要なこと:「コンパイル」の段階では他の翻訳単位のことは知ったこっちゃない

プロトタイプ宣言

「今は見つからないがとにかくどこかにある」と
コンパイラに教えるのがプロトタイプ宣言！

リンカエラー

プロトタイプ宣言だけちゃんとしておいて
`func.cpp` とリンクしなかったら？

```
g++ main.cpp
```

リンカエラー

リンカエラー発生！

```
main.cpp:(.text+0x9): undefined reference to `func()'
collect2: error: ld returned 1 exit status
```

コンパイルではなくリンクの段階で
「`func`なんて見つからないよ」と言われる

途中まとめ

- C++プログラムのビルドは「コンパイル」→「リンク」の順で行われる
- 「コンパイル」は「翻訳単位」＝「ソースファイル」ごとに行われる
- 「コンパイル」段階では他の翻訳単位のことは知ったこっちゃない
- プロトタイプ宣言によって「後で見つかる」ことを伝えられる

目次

- 翻訳単位について
- #includeとヘッダファイルについて
- 関数のファイル分割
- クラスのファイル分割
- インクルードガード
- ライブラリの使用

`#include` とは何か

`#include` とは何か

ファイルの内容をその場に広げるだけの機能

#include とは何か

極端な話

```
#include <iostream>     // a.txt

int main() {
    std::cout <<
```

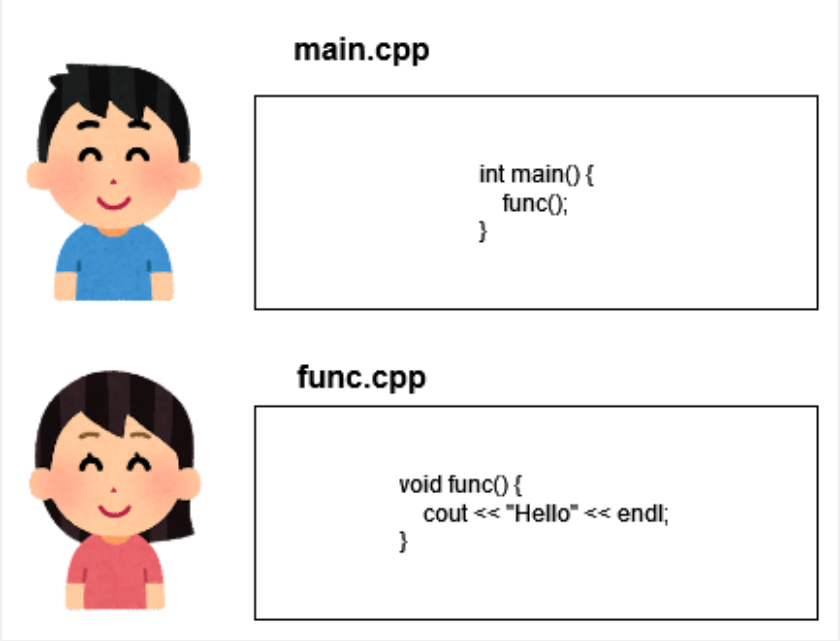
```
    "Hello World" << std::endl;    // b.txt
}
```

```
#include "a.txt"
#include "b.txt"
```

こんなんでも動く

ヘッダファイル

複数人で手分けして開発する場合を考える



ヘッダファイル

`func.cpp` に定義された関数を利用するには
全部プロトタイプ宣言を書かなければいけない →**めんどい**

main.cpp

```
void funcA();
int funcB(int a, int b);
double funcC(float p

int main() {
   funcA();
}
```

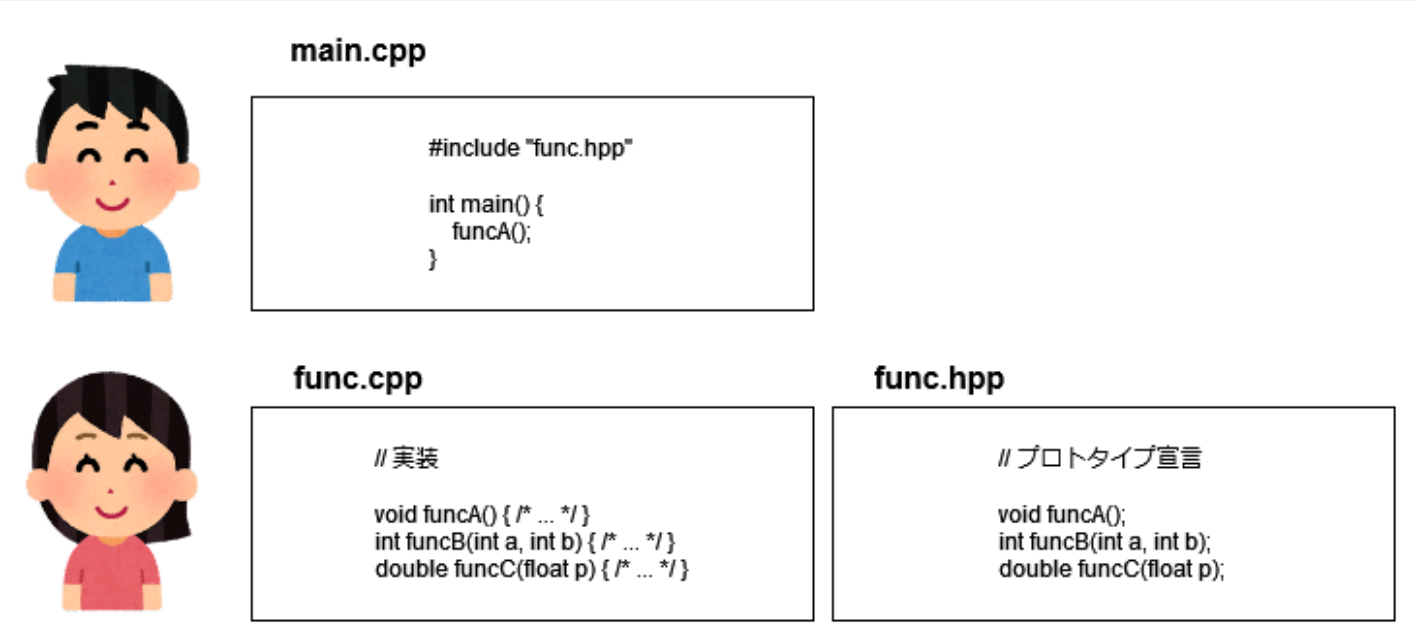
func.cpp

```
void funcA() { /* ... */ }
int funcB(int a, int b) { /* ... */ }
double funcC(float p) { /* ... */ }
```

ヘッダファイル

プロトタイプ宣言などをまとめたファイル(**ヘッダファイル**)を作っておく

→最初に#includeするだけでOK！便利！



ヘッダファイルによる分割の例

```hpp
// func.hpp

void func();
```

```cpp
// func.cpp
#include "func.hpp"

void func() { /* ... */ }
```

```cpp
// main.cpp
#include "func.hpp"

int main() {
    func();
}
```

ヘッダファイルに書くこと

ヘッダファイルに書くのはプロトタイプ宣言だけではない

- 定数定義
- クラス定義(後述)
- 変数extern宣言
- etc...

詳細は後述

補足: 拡張子について

Cのヘッダファイルは `.h` 、C++のヘッダファイルは `.hpp` にすることが多い

(ただし仕様としては何でもよく、これらは文化的なもの)

途中まとめ

- `#include` はファイルの中身をその場に展開するだけの機能
- 関数を使う側がプロトタイプ宣言などをいちいち書くのはだるいので、ヘッダファイルに書いてincludeしてもらう
- 拡張子は `.h` / `.hpp`

目次

- 翻訳単位について
- #includeとヘッダファイルについて
- 関数のファイル分割
- クラスのファイル分割
- インクルードガード
- ライブラリの使用

関数のファイル分割

ここまでの復習

- 関数定義はソースファイルに書く
- プロトタイプ宣言はヘッダファイルに書いてincludeしてもらう

早速実践してみよう

演習: 関数のファイル分割

以下のソースコードを `main.cpp`, `hoge.cpp`, `hoge.hpp` に分けてみよう

```cpp
#include <iostream>
using namespace std;

int hoge(int a, int b) {
    return a + b;
}

int main() {
    cout << hoge(2, 3) << endl;
}
```

注意ポイント: 関数の二重定義

ヘッダファイルに「プロトタイプ宣言」でなく
「関数定義」を書いてしまった場合どうなるか

```
// hoge.hpp

int hoge(int a, int b) {
    return a + b;
}
```

注意ポイント: 関数の二重定義

2ファイル以上からincludeすると...

```
// a.cpp
#include "hoge.hpp"

// ...
```

```
// b.cpp
#include "hoge.hpp"

// ...
```

注意ポイント: 関数の二重定義

リンカエラー発生！！

```
b.cpp:(.text+0x0): multiple definition of `hoge(int, int)'
collect2: error: ld returned 1 exit status
```

複数の翻訳単位に同じ関数の定義がある
→リンク段階で「重複してるよ」と言われてしまう

目次

- 翻訳単位について
- #includeとヘッダファイルについて
- 関数のファイル分割
- クラスのファイル分割
- インクルードガード
- ライブラリの使用

クラス・構造体

これをどう分割するか

```
class A {
    int val;
    void method() { /* ... */ }
};

int main() {
    A hoge;
}
```

クラス・構造体

クラス定義はヘッダファイルに書く

```
class A {                   // a.hpp
    int val;
    void method() { /* ... */ }
};
```

```
#include "a.hpp"            // main.cpp

int main() {
    A hoge;
}
```

- クラス・構造体の定義情報はコンパイル段階で必要
- クラス・構造体は**複数の翻訳単位に重複して存在しても問題ない**

クラス・構造体

ただしメソッドは(基本的に)宣言と定義を分ける　これで完成！

```
class A {                   // a.hpp
    int val;
    void method();          // 宣言
};
```

```
#include "a.hpp"            // a.cpp

void A::method() { /* ... */ }  // 定義
```

```
#include "a.hpp"            // main.cpp

int main() {
    A hoge;
}
```

補足: クラスのメソッド定義

クラスのメンバ関数(メソッド)定義をクラスの外に書く場合
スコープ解決演算子 `::` を使う

```
class Hoge {
    void method() { /* 定義 */ }
};
```

↓

```
class Hoge {
    void method();      /* 宣言 */
};

void Hoge::method() { /* 定義 */ }
```

演習: クラスのファイル分割

以下のソースコードを `main.cpp`, `myclass.hpp`, `myclass.cpp` に分けてみよう

```cpp
#include <iostream>

class MyClass {
    int val;
public:
    MyClass(int _val) { val = _val; }
    void show() { std::cout << val << std::endl; }
};

int main() {
    MyClass obj{ 123 };
    obj.show();
}
```

途中まとめ

- 複数の翻訳単位で使うクラスは、定義をヘッダファイルに書いてinclude
- クラスのメソッド定義はソースファイルに書く

補足: 外部リンケージと内部リンケージ

Q. なぜクラスは複数の翻訳単位に重複して存在しても
リンカエラーにならないのか？

補足: 外部リンケージと内部リンケージ

Q. なぜクラスは複数の翻訳単位に重複して存在しても
リンカエラーにならないのか？

A. クラスは**内部リンケージ**を持つから

補足: 外部リンケージと内部リンケージ

プログラミングにおけるあまりにも当たり前な話:

「同じ名前のものは同じものを指す」

```
int a;       // このaと...

a = 10;      // このaは同じ！
```

補足: 外部リンケージと内部リンケージ

プログラミングにおけるあまりにも当たり前な話:

「同じ名前のものは同じものを指す」

```
int a;       // このaと...

a = 10;      // このaは同じ！
```

だからこそ同じ名前のものが二重定義されるとエラーになる

補足: 外部リンケージと内部リンケージ

外部リンケージ:
翻訳単位をまたいで「同じ名前のものは同じもの」と扱われる存在

内部リンケージ:
1つの翻訳単位の中でのみ「同じ名前のものは同じもの」と扱われる存在

補足: 外部リンケージと内部リンケージ

外部リンケージ:
翻訳単位をまたいで「同じ名前のものは同じもの」と扱われる存在

内部リンケージ:
1つの翻訳単位の中でのみ「同じ名前のものは同じもの」と扱われる存在

(普通の)関数は**外部リンケージ**
クラス・構造体は**内部リンケージ**

補足: 外部リンケージと内部リンケージ

- 外部リンケージを持つ存在
  - 普通の関数
  - 普通のグローバル変数
  - 完全特殊化されたテンプレート関数
- 内部リンケージを持つ存在
  - クラス・構造体・共用体・列挙体
  - const変数, constexpr変数
  - テンプレート関数
  - インライン関数
  - static変数・関数 および 無名名前空間の変数・関数

補足: 外部リンケージと内部リンケージ

複数の翻訳単位で何かを共有したい場合

外部リンケージを持つ存在は

- ソースコードに定義を書く
- ヘッダファイルに宣言を書いてinclude

内部リンケージを持つ存在は

- ヘッダファイルに定義を書いてinclude

とする(補足終わり)

目次

- 翻訳単位について
- #includeとヘッダファイルについて
- 関数のファイル分割
- クラスのファイル分割
- インクルードガード
- ライブラリの使用

インクルードガード

同じヘッダファイルを二重にincludeするとどうなるか？

```
// a.hpp
class A { /* ... */ };
```

```
// main.cpp
#include "a.hpp"
#include "a.hpp"

int main() {}
```

インクルードガード

コンパイルエラー発生！

```
a.hpp:1:7: error: redefinition of ‘class A’
In file included from main.cpp:1:
```

同じクラスを二重に定義していると言われた
それはそう

インクルードガード

そこでこのようにする

```
// a.hpp
#ifndef A_HPP
#define A_HPP

class A { /* ... */ };

#endif
```

インクルードガード

- 1回目にincludeしたとき
  ヘッダファイルの中身を取り込んでマクロ `A_HPP` を定義
- 2回目以降にincludeしたとき
  マクロ `A_HPP` が定義されているので `#ifndef` 〜 `#endif` の間は読まれない

全てのヘッダファイルには基本的にこれを施そう

演習: インクルードガードの実装

以下のヘッダファイルにインクルードガードを施そう

```
// hoge.hpp

struct Hoge {
    int a;
};
```

以下のソースコードが問題なくコンパイルできたらOK

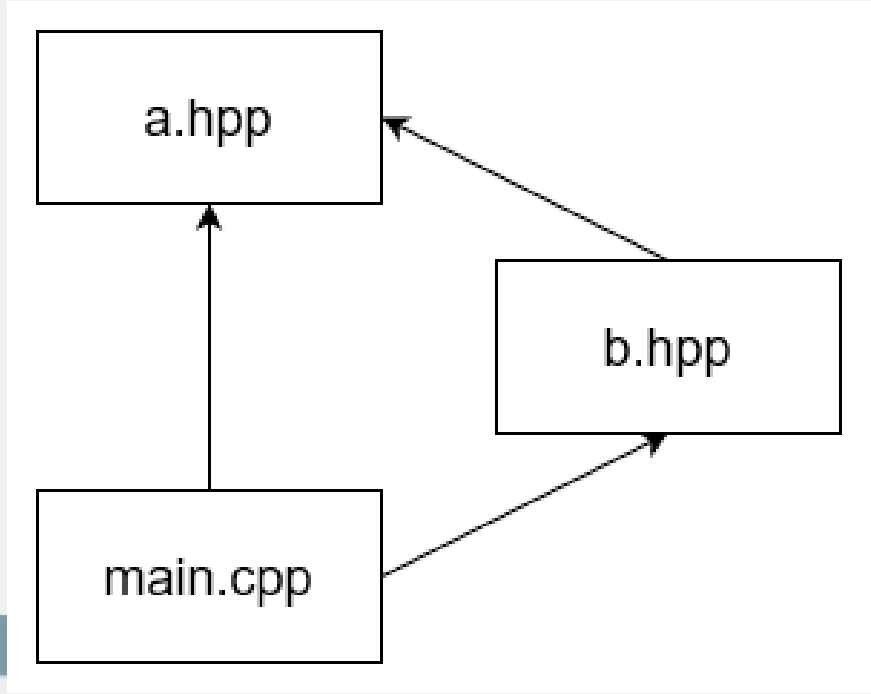
```
// main.cpp
#include "hoge.hpp"
#include "hoge.hpp"

int main() {}
```

補足: インクルードガードはただのミス防止か？

インクルードガードは普通に**プログラムを簡潔な構造にする上で必須**

例: このようなインクルード関係の場合を考えてみよう



補足: #pragma once

ここまでに解説したような古典的インクルードガードは面倒なので
一部のコンパイラはソースコードの先頭に `#pragma once` と書くだけで
同じことができるようになっている

```
#pragma once

class Hoge {
    void foo();
};

int bar(int p, int q);
```

注意: C++標準の機能ではない！！

補足: #pragma once

メリット

- 書くのは楽になる
- マクロ名の被りを心配しなくて良くなる

デメリット

- 利用するビルド環境で使えることを確かめる必要がある
- 「任意の環境でコンパイルできる」という保証は消える

これらと天秤にかけて使うかどうかを決めよう(自分は使ってません)

途中まとめ

- ヘッダファイルには問答無用でインクルードガードを付けよう
- `#pragma once` は考えて使おう

目次

- 翻訳単位について
- #includeとヘッダファイルについて
- 関数のファイル分割
- クラスのファイル分割
- インクルードガード
- ライブラリの使用

ライブラリの種類

- ヘッダオンリーライブラリ
- 静的ライブラリ
- 動的ライブラリ

ライブラリの使い方

- ヘッダオンリーライブラリ
- 静的ライブラリ
- 動的ライブラリ

とりあえず「静的ライブラリ」の使い方が分かればいい

静的ライブラリを構成する要素

- ヘッダファイル(`.h` / `.hpp`)
- ライブラリファイル(`.a` / `.lib`) ← 中身はただのオブジェクトファイル

この2つが取りこめればいい！

静的ライブラリを使う際に必要なこと

以下の4点

- インクルードパスをコンパイラに指定する
- ライブラリパスをコンパイラに指定する
- ライブラリ名をコンパイラに指定する
- ヘッダファイルをインクルードする

インクルードパス

`#include` で取り込むファイルをどこから探すかという情報

`gcc/g++` だと `-I` オプションで指定

例:

```
g++ hoge.cpp -I /abc/def/hoge
```

こうすると `/abc/def/hoge` からインクルードファイルが探索される

ライブラリパス

ライブラリファイルをどこから探すかという情報

`gcc/g++` だと `-L` オプションで指定

例:

```
g++ hoge.cpp -L /abc/def/hoge
```

こうすると `/abc/def/hoge` からライブラリファイルが探索される

ライブラリ名

何と言う名前のライブラリファイルをリンクするかという情報

`gcc/g++` だと `-l` オプションで指定

例:

```
g++ hoge.cpp -l:libhoge.a
```

こうすると `libhoge.a` がライブラリパスから探され読み込まれる

ライブラリ名

gcc/g++では `libxxx.a` の形式の名前の場合以下のようにも書ける

```
g++ hoge.cpp -lhoge
```

このように書いた場合も `libhoge.a` が読み込まれる

コンパイラへの指定

- インクルードパス
- ライブラリパス
- ライブラリ名

この3点をコンパイラに指定すれば必要なファイルが取ってこれる！

ヘッダファイルをincludeする

カレントディレクトリのヘッダファイルは `"hoge.hpp"` のようにしていた

```
#include "hoge.hpp"
```

そうでないヘッダファイルは `<hoge.hpp>` のようにする

```
#include <hoge.hpp>
```

演習: 簡単なライブラリのリンク

演習用のシンプルなライブラリ `liboisu` を配布します

main関数から `show_message()` を呼び出して `oisu-` と出たら成功です

補足: 他のコンパイラの場合

この講習会ではgcc/g++での方法を解説した

他のコンパイラでも前述の3つをコンパイラに指定する点は同じ

ただし指定方法が違ったりするので気を付けよう

補足: 他のコンパイラの場合

「いやだ！コンパイラごとの設定方法なんて考えたくない！」

→CMake編へ...

全体まとめ

- 関数は定義をソースに、宣言をヘッダファイルに
- クラスは定義をヘッダファイルに
- ヘッダファイルにはインクルードガード
- ライブラリを使う際は以下の3点をコンパイラに指定
  - インクルードディレクトリ
  - ライブラリディレクトリ
  - ライブラリ名
- カレントディレクトリ以外のヘッダファイルは <> でインクルード

おわり

お疲れさまでした！