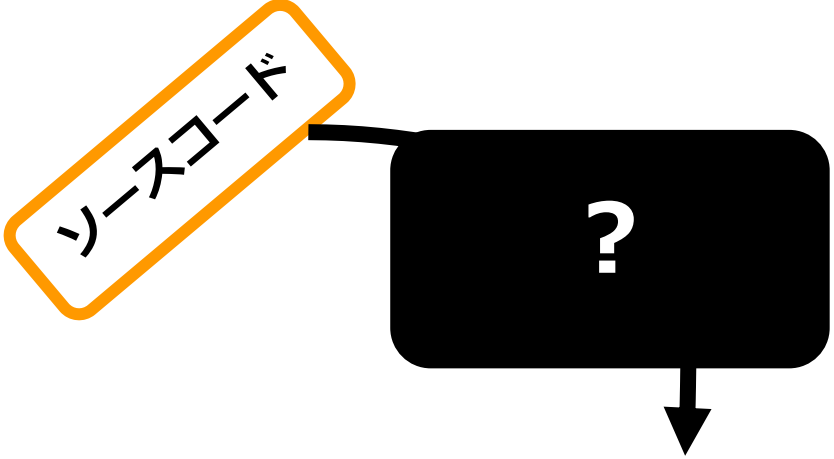


ばいなり！

T研 M2 島津

2015/11/05

具体的内容

こんぱいる！

ビルドツールってなにするの？
**　(make, ninja, rake, autotools, cmake, gyp…)**

静的ライブラリ(.a)ってなに？

- arコマンド

共有ライブラリ(.so)ってなに？

- LD_LIBRARY_PATH
- ldd

こんぱいる！　ってどういうこと？

2つのイメージ（予想）

```
$ gcc hello.c
$ ./a.out
hello.
```

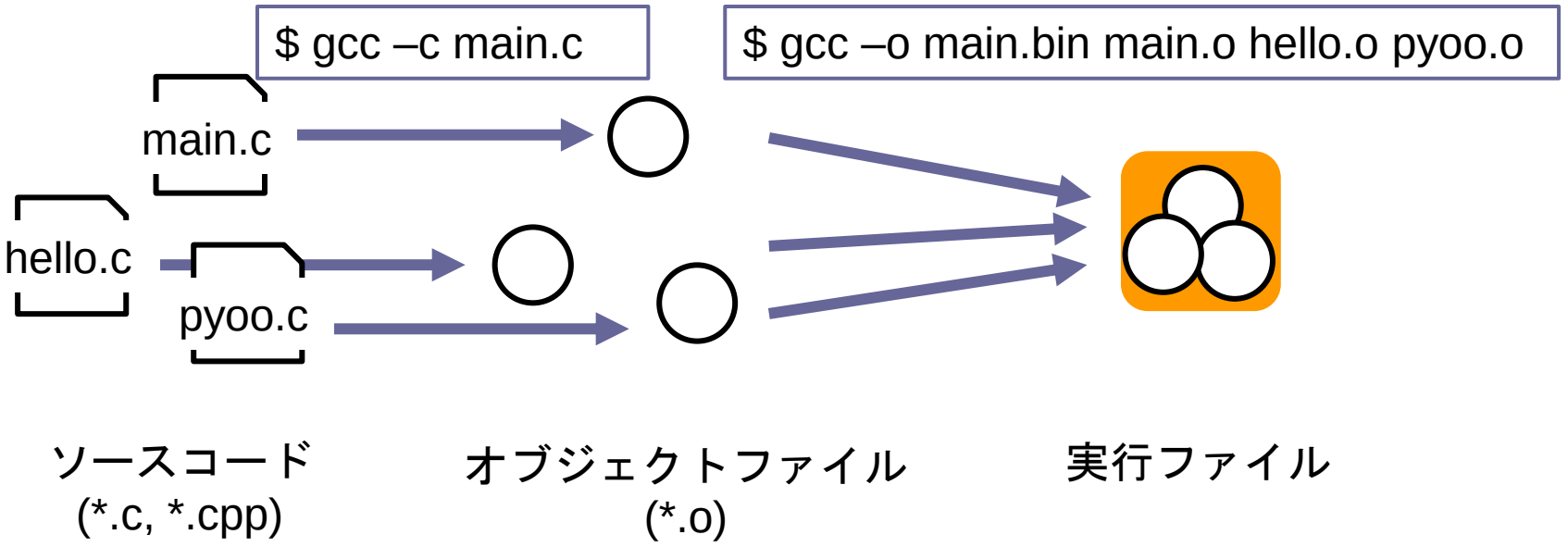
```
$ ./configure --prefix=../hogedir
$ make –j16
ぶわー
$ make install
```

本当にやっていること

- どっちかっていうと前者
- 後者は、いちいつ打つのが面倒だからコマンド一発で出来るように簡単にしている

  $(CC) $(CFLAGS) -c –o $(OBJDIR)/hoge.o hoge.c

こんぱいる！のイメージ図



こんぱいる！のやることの手順

オブジェクトファイルの生成　（狭義のコンパイル）

```
$ gcc –c hello.c
$ gcc –c main.c
$ ls
hello.c hello.o main.c main.o
```

- .c -> .o
- .oはすでにコンパイル済み（機械語）のプログラムが入っている

バイナリの生成 (リンク)

```
$ gcc –o wei.bin main.o hello.o
```

- .oにあるプログラムを**リンク**する
  - main.oとhello.oの関数を一つのファイルにまとめる
  - 関数名を見て、ジャンプする命令のアドレスを決める (cf. アーキテクチャのA型の命令)

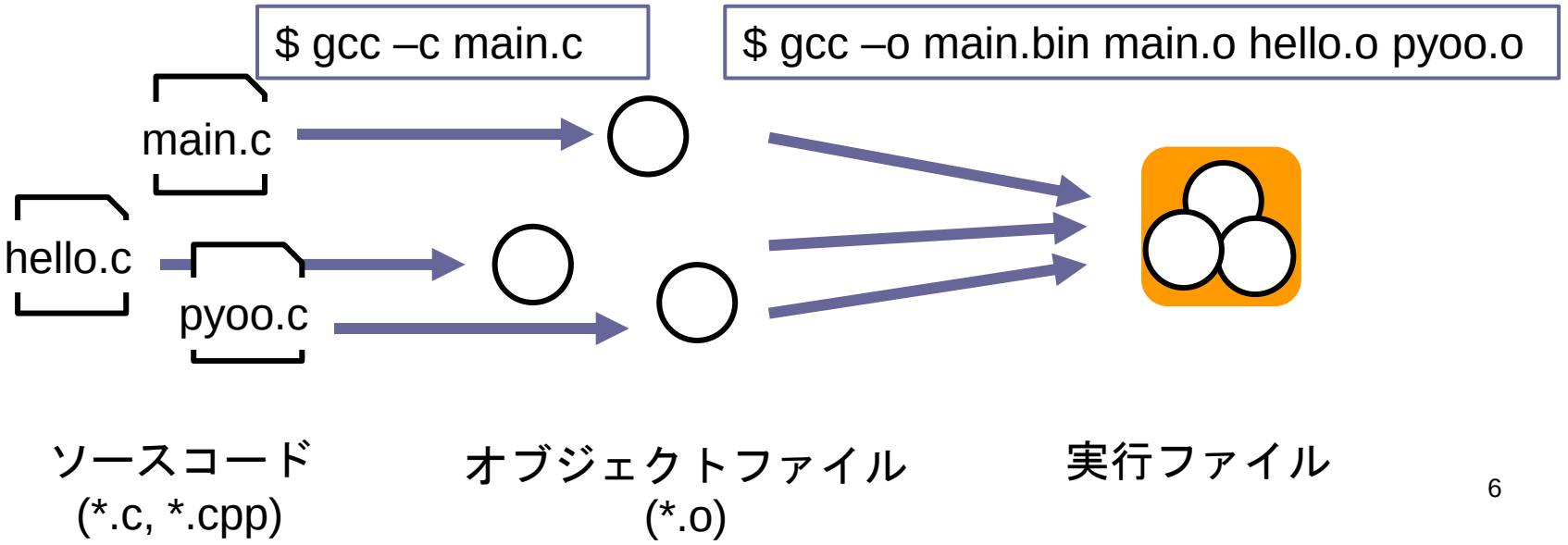
ビルドツールの必要性

ファイルが多くなると・・・？

- たとえば、chromiumだったら20000ファイルを超える！
- 全部のファイルに対してコマンドラインを書いてシェルスクリプトを書く？
- つらみしか感じない

いい感じにしてくれるツールが？

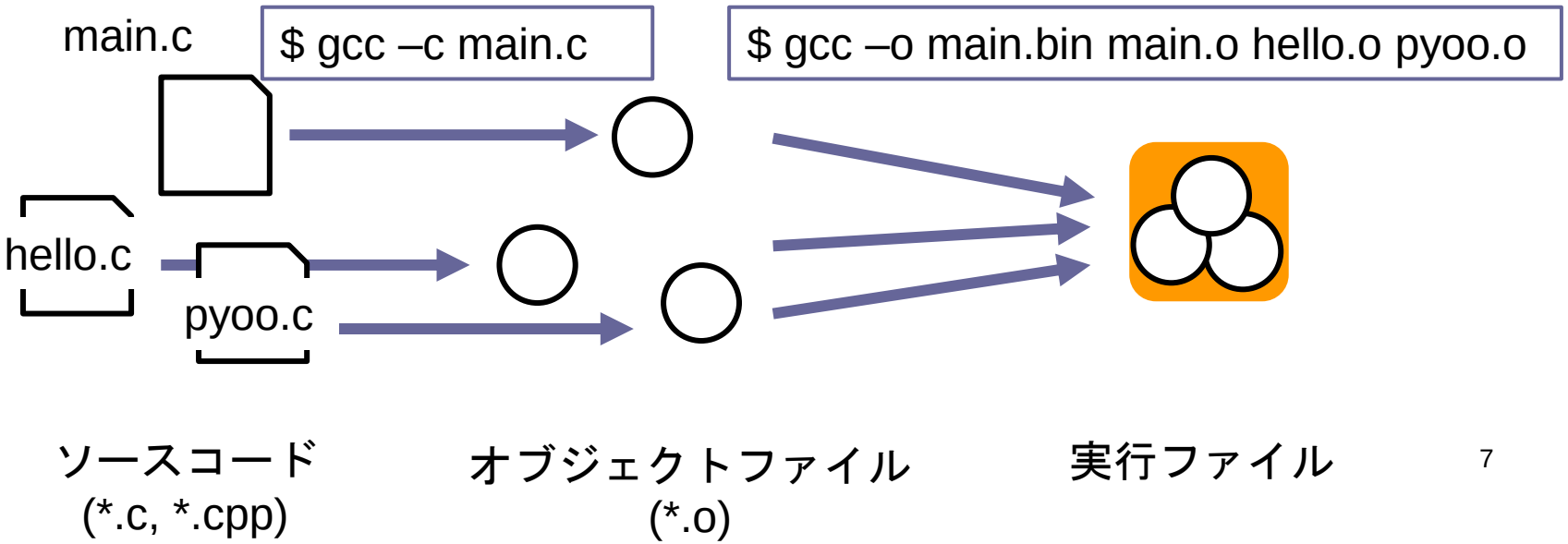
- ビルドツール



ビルドツール – make, ninja, rake…

make, ninja, rake

- ”依存関係”を解決しながらコマンドを実行してくれるツール
- ”依存”するファイルが更新されると、そのコマンドが実行される
- 変えたファイルに**関係するコマンドだけ**しか実行されない！



makeの例

makeコマンドの設定ファイル：makefile

- こんな感じで書けば、main.o/hello.o/pyoo.oを使ってwei.binが生成できる

```
wei.bin: main.o hello.o pyoo.o
        gcc -o $@ $^

%.o: %.c
        gcc -c $<
```

- もっと詳しい使い方はいろいろ調べてみてください！
- (個人的には次に説明するgypをつかってmakefileやninjaのファイルを生成するのがマイブームです)

configureは
autotoolsの一部

メタビルドシステム - autotools, cmake, gyp…

ビルドに必要なファイルを生成するシステム

- makefileやsln(Visual Studio向けプロジェクトファイル)、xcodeなど
- cmakeやgypなどではクロスプラットフォーム向けに書いたりもできる

やってくれること

- 環境に応じていい感じのmakefileなどを作ってくれる
- autotoolsだと、configureしたときに環境に応じたヘッダファイルも生成してくれる
- これらにより、同じプログラムをいろんな環境で**簡単に**動かせる

よくある使い方

```
$ ./configure –prefix=/home/denjo/gnuplot_install
makefileなどを生成！
$ make
gccつかったコマンドをいい感じに実行！
$ make install
いい感じの場所にファイルをコピー！
```

ビルドシステムのまとめ

結局のところなにをやっているのか？

- gccやclangを動かしている
- そのための便利ツールがmakeやninja
- それを準備するための便利ツールがautotoolsやcmakeやgyp

よくわからない挙動をした場合には・・・

- 立ち戻ってgccなどのオプションがいい感じになっているかをチェックする
- どうせコマンド実行しているだけなので、表示されたコマンドを手で実行してみるのも手

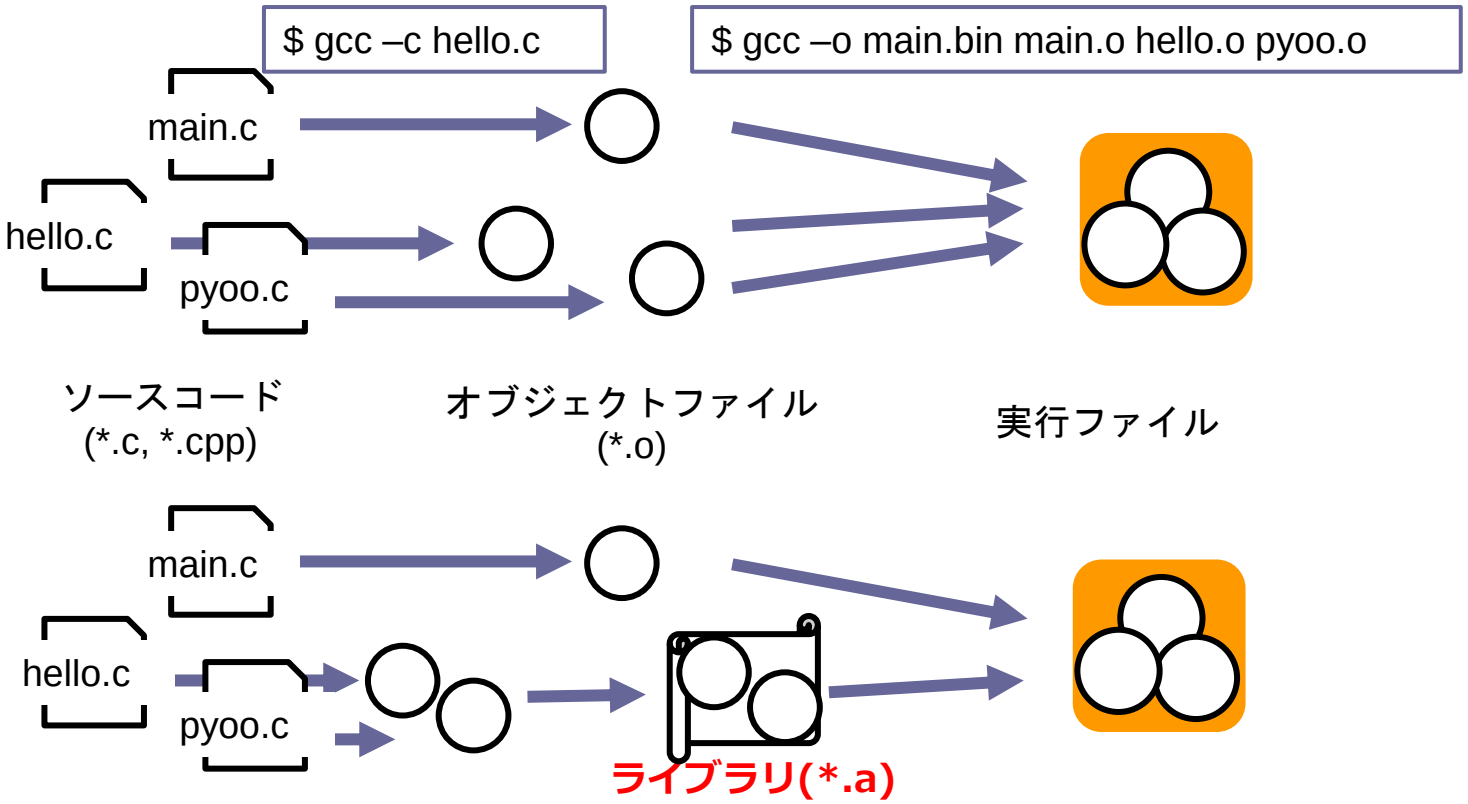
覚えておくと便利な環境変数

- C_INCLUDE_PATH, CPP_INCLUDE_PATH, LIBRARY_PATH
- デフォルト以外のincludeパスやライブラリの検索元(後述)を追加できる

ライブラリ

ライブラリとは？

- 一言で言うと、オブジェクトファイルの塊
- 利用しやすいように、オブジェクトファイルを一つのファイルにまとめたもの(ex. libc)



ライブラリの操作

使い方

- arコマンドで作成
- コンパイル時には.oと同じようにファイルを指定して使うか、-lhogeのようにする

```
$ gcc –c hello.c
$ gcc –c pyoo.c
$ ar rsv libhello.a hello.o pyoo.o
$ ls
libhello.a hello.c hello.o main.c pyoo.c pyoo.o
```

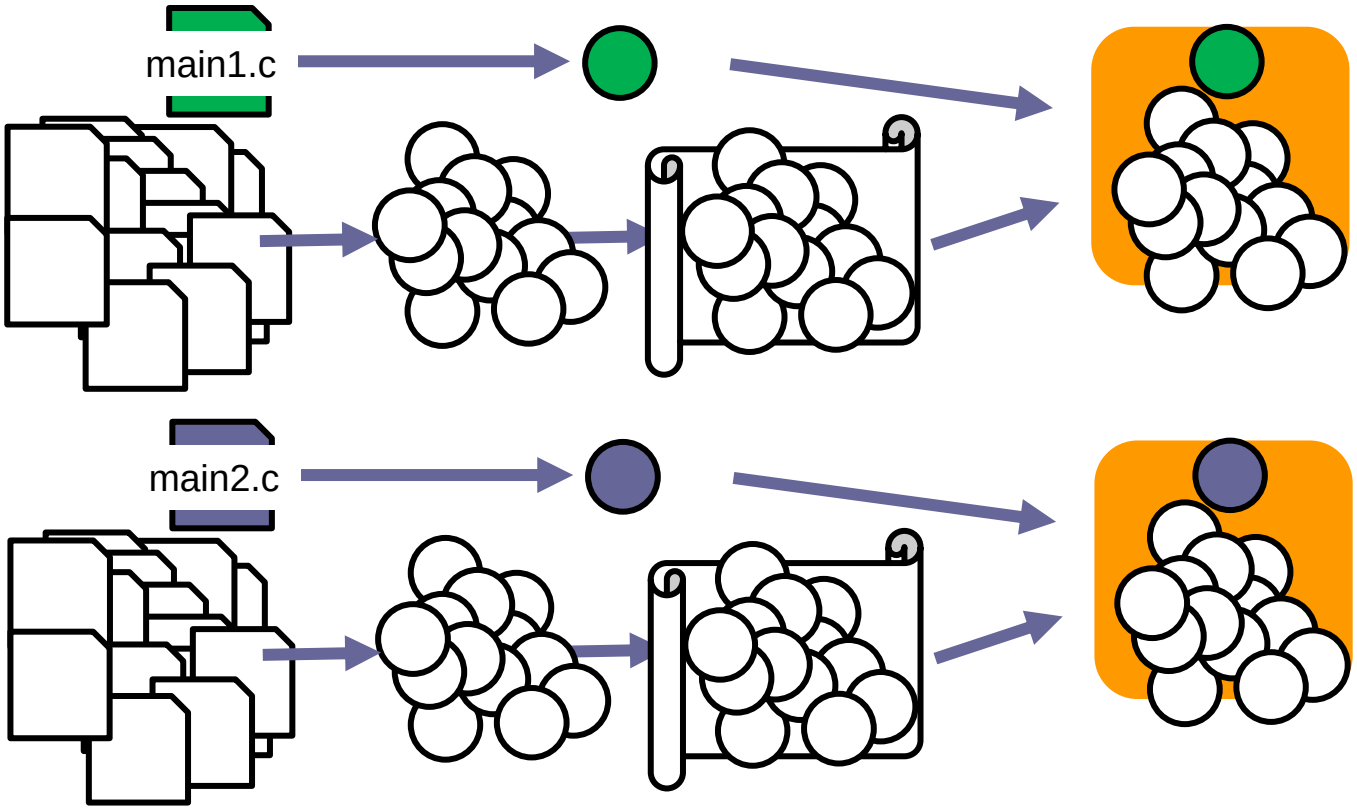
```
$ gcc –c main.c
$ gcc –o main.bin main.o –lhello –L.
libhello.a hello.c hello.o main.bin main.c main.o pyoo.c pyoo.o
```

-lと**-L**という２つのオプションは覚えておくとよい。
-lが**ライブラリ名**を指定していて、**-L**はライブラリの**検索元**を指定している。

静的ライブラリの問題点

コードサイズの肥大化

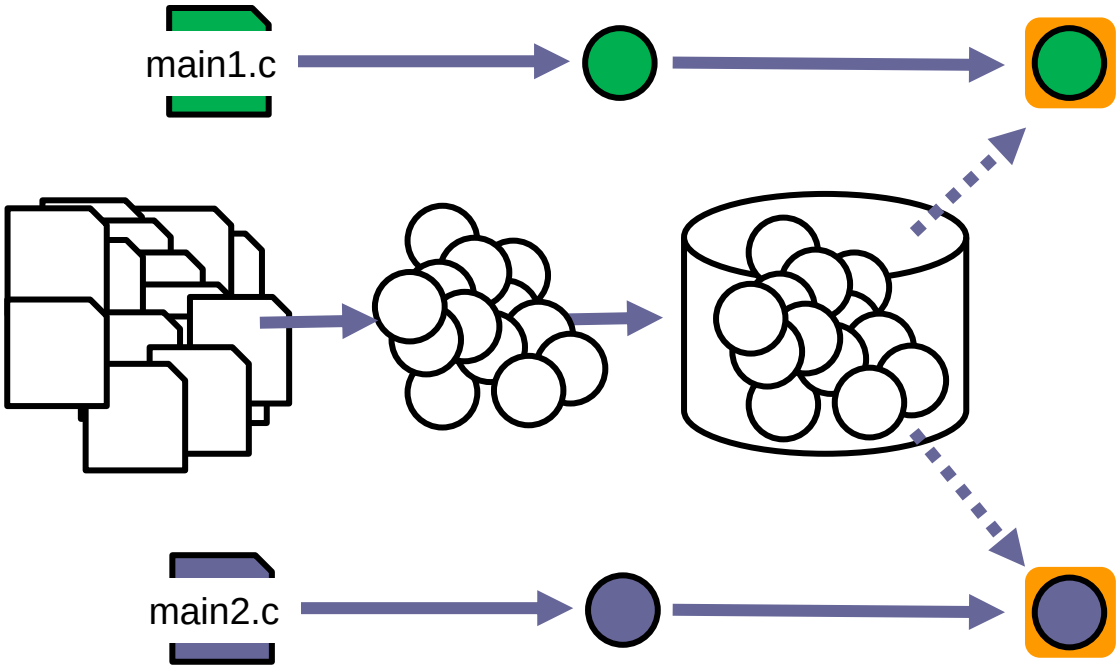
- 静的ライブラリは、バイナリがすべてのコードを持つ
- たくさんのプログラムが同じライブラリを使っていた場合…



共有ライブラリ(.so, shared objects)

ライブラリの共通化

- 実行時に同じライブラリを使うようにする
- これが**共有ライブラリ**



実際の使い方

動き

- サーチパス(/etc/ld.so.confやLD_LIBRARY_PATH)から勝手にsoファイルを探してくる
  - lddコマンド：依存している共有ライブラリを見れる
- 実行時に探したいパスを追加するときには、ビルド時にオプションを追加
  - -Wl,-rpath=$PWDや-Wl,-R $PWDとか

コマンド

-fPICはとりあえず呪文ということで・・・
(興味あったら聞いてください)

```
$ cd lib
$ gcc -fPIC –c hello.c
$ gcc -fPIC –c pyoo.c
$ gcc –shared –fPIC –o libhello.so hello.o pyoo.o
$ ls
libhello.so hello.c hello.o pyoo.c pyoo.o
$ cd ..
$ gcc –Wl,-R $PWD/lib –o main.bin main.c –lhello –Llib
$ ls
lib main.c main.bin
```

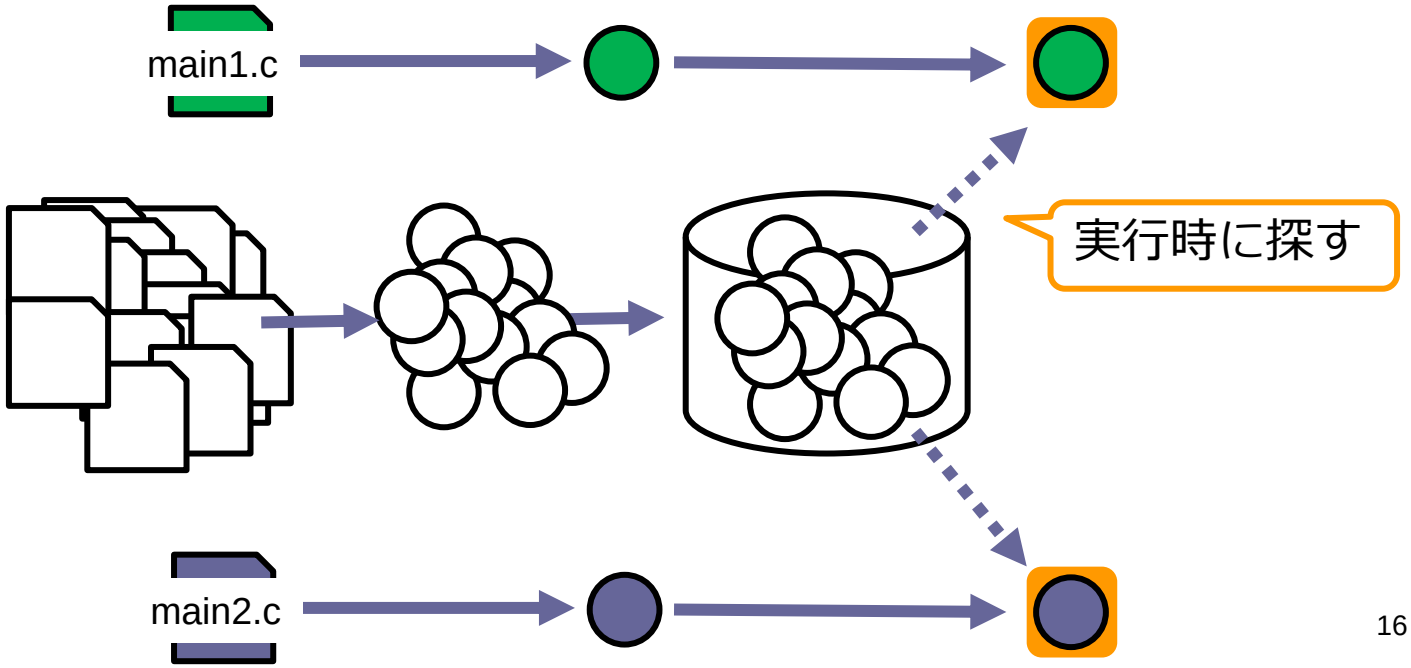
ディレクトリ構成

```
+- main.c
+- lib
   +- hello.c
   +- pyoo.c
```

共有ライブラリのサーチパス

実行時にいちいちファイルを探す…どこから？

- 確認1: lddコマンドを使ってみる
- 確認2: /etc/ld.so.confを見てみる
  (見るとinclude /etc/ld.so.conf.dをしているので、実際には/etc/ld.conf.so.dを見る)
- 確認3: gccのオプションに-vをつけてみる



よくあるシチュエーション

ライブラリをビルドしてインストールしたいけど、
自前でビルドしたものと元からはいってるものをちゃんとわけたい！

- ./configure –-prefix=… をつかってインストール先を設定
- LIBRARY_PATHとLD_LIBRARY_PATHの２つの環境変数を設定
  - LIBRARY_PATH: プログラムのビルド時につかうライブラリの位置
  - LD_LIBRARY_PATH: プログラムの実行時に含めるサーチパス

いろいろありそうなタイトルを付けたけど、
個人的に思い当たるのはこれだけだった・・・

まとめ

ビルドツールとは

- gcc, clangなどをいい感じに実行してくれる便利ツール
- C_INCLUDE_PATH/CPP_INCLUDE_PATH/LIBRARY_PATHを使って読み込むディレクトリを指定できる

ライブラリとは

- オブジェクトファイルを１つにまとめたもの
- 静的ライブラリを使うと最終的にできるバイナリにすべてが含まれる
- 共有ライブラリを使うと、実行時に検索・リンクが行われる
- LD_LIBRARY_PATHを使って、サーチパスを追加できる

おまけ

僕（島津）の環境の例

- 自前でビルドしたものは/home/shimazu/local以下におく
  - つまりprefixは $HOME/local にしてる
- インクルードパスやライブラリのパスを環境変数で追加

ディレクトリ構成

```
/home/shimazu
+- local
   +- bin
   +- lib
   +- include
   ...
```

```
export CPLUS_INCLUDE_PATH=$CPLUS_INCLUDE_PATH:$HOME/local/include
export C_INCLUDE_PATH=$C_INCLUDE_PATH:$HOME/local/include
export LIBRARY_PATH=$LIBRARY_PATH:$HOME/local/lib:$HOME/local/lib/x86_64-linux-gnu
export LD_LIBRARY_PATH=$HOME/local/lib:$LD_LIBRARY_PATH
export LD_LIBRARY_PATH=$HOME/local/lib/x86_64-linux-gnu:$LD_LIBRARY_PATH
```

- 参考になれば幸いです。