プログラムのライブラリ化

松本 和幸

2019 年 4 月 23 日

1　実験内容

1.1　目的

効率の良いプログラミング，特に複数人でのプログラミングにおいて重要となるライブラリ化について習得する．ライブラリ化について理解するために必要な，コンパイルの仕組みといった基礎的な知識も習得する．また，分割コンパイルを行う際に役立つなツールである make の使い方と，Makefile の書き方も習得する．

1.2　講義で説明する内容

以下について順に説明した後，今回の講義の課題（演習問題）について説明を行う．

- コンパイルの仕組み
- ライブラリ
- 分割コンパイル

1.3　課題

関数のライブラリ化を目的としたプログラムを作成する．

2　コンパイル

2.1　コンパイルの流れ

通常のコンパイル：

```
gcc -Wall hello.c -o outhello
```

実行ファイル “outhello” が生成される．このような実行ファイルはどのような流れで生成されるのかについて，図 1 に沿って順に説明する．

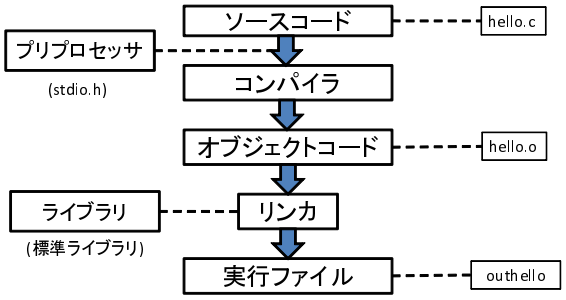


図 1: コンパイルの流れ

2.1.1 プリプロセッサ

ソースコードがコンパイラによりオブジェクトコードに変換される前に，プリプロセッサにより，ソースコード中の include 文やマクロ定義文が展開される．

たとえば，以下のようなソースコードの場合，

```
#include <stdio.h>
int main()
{
  printf("hello world\n");
  return 1;
}
```

プリプロセッサによって，include 文

```
#include <stdio.h>
```

が，ヘッダファイル stdio.h の内容に置き換えられる．置き換えられた後のファイルが，プリプロセス済みのソースコードである．

2.1.2 コンパイラ

コンパイラは，プリプロセッサの出力を解析し，アセンブラプログラムを生成する．アセンブラプログラムとは，アセンブリ言語（機械語と 1 対 1 に対応した，最も低水準な言語）によって表現されたプログラムのことを指す．

2.1.3 アセンブラ

アセンブラはアセンブラプログラムを読み込み，オブジェクトプログラムを生成する．オブジェクトプログラム（オブジェクトコード）とは，機械語（文字によって表現できない 1 と 0 からなるコード）によって表現されたプログラムのことである．

2.1.4 リンカ

リンカの役目はオブジェクトプログラムどうしを結合（リンク）し，実行ファイルを生成することである．

3 ライブラリ化

3.1 なぜライブラリが必要か

もしライブラリを使うことができないと，異なるプログラム間で同じソースコードを何度も書かなければならない．また，普段のプログラミングにおいて何気なく使用している，標準ライブラリ関数を使うことができなければ，画面への標準出力をする際でも，その都度，画面へのデバイスを開いたりする一連の処理のコードを書かなければならない．このように，ライブラリは効率良くプログラミングを行う上で非常に重要なものである．

3.2 ライブラリ関数

ライブラリ関数には，大きく分けると，以下の 2 つがある．

- 標準ライブラリ関数：
  ANSI-C 規格で定められているライブラリ関数群，マクロとして 24 のヘッダファイル内で定義されているもの．

  例) printf(), scanf(), fgets(), ….

- ライブラリ関数：
  自分，もしくは他の人が作成した，ANSI-C 規格で定められていないライブラリ関数．

  例) SDL(Simple DirectMedia Layer) や OpenGL に含まれる関数群

また，ライブラリの種類は，その読込方法の違いで，静的ライブラリと動的ライブラリに分けられる．

- 静的ライブラリ（スタティックライブラリ）：
  実行ファイルに全ファイルを入力するため，起動時の実行速度が遅くなる．静的ライブラリのファイルの拡張子は一般的に .a とする．

- 動的ライブラリ（ダイナミックリンクライブラリ）：
  実行時に動的にライブラリをリンクする．この場合，必要な時に読み込む，つまり，1 回のみのメモリへのロードのため，静的ライブラリを用いるよりも少しだけ速い．また，ライブラリ内に変更があった際に，そのライブラリを用いているすべてのプログラムをコンパイルし直す必要がないため，システムの管理が容易になる．動的ライブラリのファイルの拡張子は一般的に .so とする．共有ライブラリとも呼ばれる．

静的ライブラリと動的ライブラリの作成方法の違いについては，サンプルメイクファイル (makefile_sample.tar.gz，makefile_sample_so.tar.gz) を参考にせよ．

3.3 アーカイブファイル

複数のファイルを 1 つのファイルにまとめる操作をアーカイブと呼ぶ．アーカイブの方法は幾つもあるが，ライブラリ関数を作成する場合は，ライブラリの作成コマンドである，ar コマンドを用いる．アーカイブのインデックス作成には，ranlib コマンドを用いる．

4 分割コンパイル

巨大なプログラムを作成する際，1 つのソースコードで作成すると，同時に複数人でソースコードの編集作業ができない．また，1 人でソースコードを管理する場合でも，どこに何の関数が書いてあるかが分からなくなることがある．そこで，分割コンパイルという方法を用いて，大きなソースコードを複数の小さな単位（たとえば機能単位）に分割することでソースコードの管理とコンパイルを容易にする．分割コンパイルの特徴を以下にまとめる．

- 1 つの実行ファイルを複数のファイルで構成する．
- 1 つのプログラムを複数のファイルに分割する．
- 各々を個別にコンパイル，リンクする．(変更したファイルのみ再コンパイル)

4.1 分割方法

分割の方法は，以下に示すように，様々である．

- 関数単位
- 機能や役割が似ている関数群単位
- まとまった機能単位

分割する単位はプログラムを作成する人の自由であるが，複数人で開発を行う場合は，あらかじめ分割方法は決めておくべきである．そのほうが役割分担しやすく，後で他の人の書いたソースコードを読みやすくなる．

4.2 分割コンパイルの注意点

分割コンパイルにおいては，関数のプロトタイプ宣言と外部変数について注意する必要がある．プロトタイプ宣言が参照箇所より前か後かによって，結果が異なってくる．変数の宣言では，extern, static 宣言を，適切に使い分けなければならない．ライブラリを用いる際に，ライブラリで宣言されている外部変数と同じ名前で定義されている変数の型が異なると，意図した動作をしないことがある．場合によっては，エラーが出てコンパイルが通らないこともある．

- extern, static がそれぞれどのように使用されるかについて調べよ．

4.3 make

Makefile に記述されているコマンドを処理するプログラム．この授業では，GNU make を用いる．

4.4 Makefile

目的とする実行ファイルの生成の手順を記述したファイルを Makefile と呼ぶ．デフォルトでは，ファイル名も Makefile または makefile とする必要がある．このファイルの形式はテキストファイルであり，共通の書式に従い，記述する必要がある．

4.4.1 Makefile の作り方

Makefile は，以下のような共通の書式で記述する．コマンドを書く行の先頭は必ずタブにする．

```
ターゲット: 依存関係 依存関係 ...
tab　コマンド
tab　コマンド
tab ...
```

より簡単に記述するために，ユーザ定義マクロや，内部マクロなどを用いることができる．

Makefile のサンプル-1:

```
project1: main.o data.o io.o
	gcc data.o io.o main.o -o project1
data.o: data.c data.h
	gcc -c data.c
main.o: main.c data.h io.h
	gcc -c main.c
io.o: io.c io.h
	gcc -c io.c
clean:
	rm -rf *o project1
```

4.4.2 内部マクロ

以下のような内部マクロを用いることができる．

$* ：ターゲット名から拡張子を除いた部分
$@ ：ターゲット名
$< ：最初の依存関係のファイル名

4.4.3 ユーザ定義マクロ

内部マクロ以外に，ユーザがマクロを自由に定義できる．

OBJECTS = 必要なオブジェクトファイル
CC = 使用するコンパイラ
CFLAGS = コンパイルオプション

以下，.c から .o を作成する場合の記述方法について示す．

OBJS = data.o io.o main.o と定義している場合，$(OBJS:.o=.c) とすると，data.c io.c main.c を表すことができる．逆に，SRCS = data.c io.c main.c と定義している場合，$(SRCS:.c=.o) とすると，data.o io.o main.o を表せる．また，内部マクロを用いて，以下のような書きかえもできる．

```
data.o: data.c data.h
       $(CC) $(CFLAGS) data.c
              ↓
.c.o:
       $(CC) $(CFLAGS) $<
```

```
project1: $(OBJS)
          $(CC) $(OBJS) -o project1
                ↓
TARGET: project1
        $TARGET : $(OBJS)
        $(CC) $(OBJS) -o $@
```

内部マクロおよびユーザ定義マクロを使って，Makefile のサンプル-1 を書き直すと，以下のようになる．

```
# コンパイラは gcc とする
CC = gcc
# ソースファイルは data.c main.c io.c である
SRCS = data.c main.c io.c
OBJS = $(SRCS:.c=.o)
# コンパイルオプション
CFLAGS = -c
# 実行ファイル
TARGET = project1
$(TARGET) : $(OBJS)
            $(CC) $(OBJS) -o $@
.c.o:
            $(CC) $(CFLAGS) $<
clean :
            rm -rf $(OBJS) $(TARGET)
```

4.5 演習-1

1. math.h に含まれている数学関数を 5 つ以上調べ，それを使用したサンプルプログラムを作り，そのソースコードと実行結果を示せ.
2. ar, ranlib について調べてまとめよ.
3. ar のオプション d, r, p, t について調べ，1 で作成したプログラムをアーカイブファイルにして，それに対して適用し，その実行結果を示せ.
4. Makefile で使うことができる内部マクロについて 3 つ以上調べ，それらを使って Makefile を書いてみよ.
5. LaTeX で作成した文書をコンパイルし，“make clean” で dvi ファイル等を削除，“make pdf” で pdf ファイルを作成する Makefile を作れ.

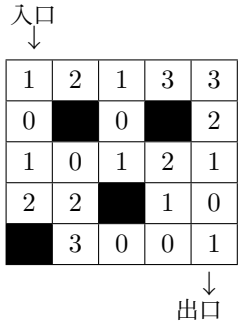
4.6 演習-2

二つの正の整数の最大公約数を求めるプログラムを作成せよ．ただし，**以下の条件を必ず満たすこと**.

- ユークリッドの互除法を用いて求める.
- メイン関数と最大公約数を求める関数を別ファイルで作成.
- Makefile を作成する.

4.7 演習-3

図 2 に示す迷路の経路を見つけるプログラムを作成せよ.



1	2	1	3	3
0	■	0	■	2
1	0	1	2	1
2	2	■	1	0
■	3	0	0	1

図 2: 迷路

ただし，**以下の条件を必ず満たすこと**.

- 1 回の移動で，前後左右いずれかの方向に 1 マスのみ移動できる
- 現在地から確認できるのは前後左右のマスの情報のみ

- 黒いマスは通れない
- 数字は移動コストを表す
- 移動のための関数 visit, 移動コストを管理する関数 calc_cost をライブラリ化する
- Makefile を作成する

また，プログラムを工夫して，できるだけ移動コストの総和が最小になる経路を探してみよう.

参考文献

[1] Robert Mecklenburg 著、矢吹 道郎 監訳、菊池 彰 訳, GNU Make 第 3 版, オライリー・ジャパン, 2015.

[2] GNU make `https://www.mlab.im.dendai.ac.jp/~tobe/xp-2/gmake.html`

[3] 自動化のための GNU Make 入門講座 - Makefile の基本：ルール. `http://objectclub.jp/community/memorial/homepage3.nifty.com/masarl/article/gnu-make/rule.html`