UNIXゼミ最終回

川崎 赤塚 堤

1999年5月11日

目 次

第 1 章　gcc による C/C++開発 **2**
1.1　gcc 2
1.2　簡単な C のプログラム 2
1.3　リンク 4

第 2 章　make **6**
2.1　簡単な`makefile` 6
2.2　マクロ 7
2.3　make の詳細な使い方 8
2.3.1　マクロの参照 8
2.3.2　複数のターゲットファイル 8
2.3.3　長いマクロの定義 8
2.3.4　慣習的に定義されるマクロ 9
2.3.5　動的マクロ 9
2.3.6　環境変数とマクロ 9
2.3.7　暗黙のルール 9

第1章　gccによるC/C++開発

1.1　gcc

UNIX 上で C/C++による開発をする場合，たいていの場合，gcc にお世話になります．gcc とは GNU C Compiler の略で GNU によるフリーな C/C++汎用コンパイラです．作りが非常に洗練されており，複数の環境に移植されています．並列環境では異なったコンパイラを使うこともありますが，とりあえずは環境に慣れるという意味で，このコンパイラを使っていくことにします．

1.2　簡単なCのプログラム

最初に，C のプログラミングから離れて久しい方々のために，とりあえず一番簡単な形の C のプログラムでウォーミングアップしてみましょう．

```
#include <stdio.h>

main()
{
  printf("Hello, UNIX world !¥n");
}
```

このプログラムを emacs や vi などで入力し，`first.c` などの名前で保存してください．次にこのプログラムをコンパイルしてみます．コマンドラインから次のように入力してください．

```
$ gcc first.c Enter
```

ミスがなければコンパイルが完了したと思います．ここで，実行形式のファイルが完成しているはずなので，それを確認してみます．`a.out` というファイルがそれです．

```
$ ls -la Enter
total 5
drwxr-xr-x   2 544      everyone        0 May  3 12:38 .
drwxr-xr-x   4 544      everyone        0 May  3 12:35 ..
-rw-r--r--   1 755      everyone   299710 May  3 12:49 a.out
-rw-r--r--   1 544      everyone       70 May  3 12:49 first.c
$ _
```

さて，早速，実行してみましょう．コマンドラインから，`./a.out`[1]と入力してください．

[1] UNIX では，セキュリティ上の配慮から，PATH の通ってないディレクトリのコマンドは，たとえカレントディレクトリであったとしても場所を明示的に指定しなければならない．

```
$ ./a.out Enter
Hello, UNIX world !
$ ▬
```

Windows や，MS-DOS などの開発環境で C/C++による開発経験がある人はここで「あれっ?」と思うかもしれません．UNIX では，指定がない場合，コンパイル時に作成される実行ファイルは，a.out という名前になります．これが嫌な場合，もしくは実行ファイルに好きな名前を付けたい場合には，コンパイル時に

```
$ gcc -o first first.c Enter
```

のように-o <実行ファイル名>というオプションを付加します．
基本的にはこれでプログラムのコンパイル，実行の全てがうまくいきました．しかし，よく見てみると a.out のファイルサイズは 299710 バイト=300KB もあります．これはデバッグ[2]用の情報が含まれているためで，開発中はこのままでも良いのですが，最終的な完成版として世に出す場合には，少しダイエットしてもらう必要があります．次のように入力してください．

```
$ strip a.out Enter
```

驚くことに，あの大きいファイルのほとんどの部分がなくなり，たった 3072 バイトになりました．約 100 分の 1 になったことになります．

```
-rw-r--r--   1 755      everyone     3072 May  3 13:05 a.out
```

次に，数学関数を使った例をコンパイルしてみます．

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

main()
{
  printf("cos(3.14)=%f¥n", cos(3.14));
}
```

このプログラムを second.c という名前で保存し，コンパイルします．

```
$ gcc second.c Enter
/tmp/cca134051.o: In function ‘main’:
/tmp/cca134051.o(.text+0xe): undefined reference to ‘cos’
$ ▬
```

エラーが出ました．このエラーは，cos という関数がどこにもないということをいっています．C のコンパイラは printf や scanf のような関数は暗黙の内に探し出し，プログラムにリンク[3]してくれますが，数学関連の関数や，通常使わないような関数，または，ユーザー定義の関数に関しては，勝手にリンクしてくれるというわけではありません．ここでは，明示的にリンクしなければなりません．従って，正しくは，

[2]プログラムからバグを取り除く作業．実はプログラムを作成する時間の 80%がこの作業に費やされるといっても過言ではない．

[3]プログラム同士をまとめること．数学関連の関数は，libm.a に実装されており，このプログラム (ライブラリ) とリンクしなければならない．

```
$ gcc second.c -lm Enter
```

とすることになります．-lmというのは，-l<ライブラリ名>という書式で，ここでは数学ライブラリである，'m' をリンクしたいという意味になります．
次にもっと高度な例を挙げます．この例では，2 つのソース (ファイル) から成るプログラムを提示しています．

main.c

```
#include <stdio.h>
extern int func(int);

main()
{
  int n;
  n = func(5);
  printf("func(5) = %d¥n", n);
}
```

func.c

```
#include <math.h>

int func(int n)
{
  return n * n + 2 * n + cos((double)n / 10.23) * 10;
}
```

このプログラムをコンパイルするには，次のようにします．

```
$ gcc main.c func.c -lm Enter
```

何も問題がなければコンパイルが完了し，a.outが作成されたはずです．
ここまでで，基本的なコンパイルの方法を説明してきました．これだけのことを知っていれば，面倒と思わない限り，どんなプログラムでもコンパイルできます．

1.3 リンク

今までの所では，実行ファイルを作成することをコンパイルといいましたが，実際には，この表現は正しくありません．実行ファイルが作成されるまでには，**コンパイル→リンク**[4]という過程を経ています．**コンパイル**は，単に指定されたファイルをオブジェクトファイルというファイルに変換することであり，**リンク**とは，それらを結合し一つにすることを意味しています．また，リンク時には，プログラム中で使われていてプログラム中で定義されていない関数を標準ライブラリや指定されたライブラリの中から探し出し，それらもリンクするということを行います．
直前の例ならば，図 1.1 のようなプロセスをたどっています．最初に，main.c，func.cはコンパイル

[4] もっと正確にいうと，**プリコンパイル→コンパイル→リンク**である．

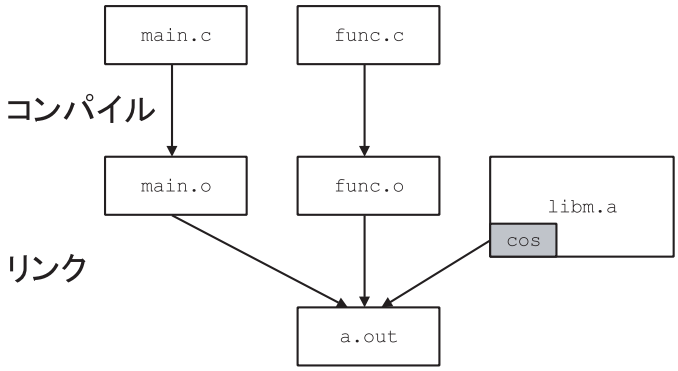

図 1.1: コンパイル，リンクのプロセス

され，それぞれ main.o，func.o を生成します．次に，gcc はこの中で使われている関数を全て探し出します．ここでは，-lm で明示的に示されている数学ライブラリ libm.a から cos のオブジェクトを探し出し，a.out にリンクしています．また，この図では省略されていますが，実際には，printf などもこの過程で探され，リンクされています．

この過程を全て自分で行うとすれば，

```
$ gcc -c main.c Enter
$ gcc -c func.c Enter
$ gcc -o a.out main.o func.o Enter
```

とすることになります．また，例えば，この後に func.c だけを書き換え，再コンパイルする場合，main.c には変更がありませんから，main.o は既に作成されているものが使えます．従って，

```
$ gcc -o a.out main.o func.c Enter
```

といった具合にコンパイルできます．ここでは main.c を再度コンパイルしていませんので，その分だけコンパイルにかかる時間を削減できます．この程度のプログラムではコンパイルの所要時間は大した物と感じないかもしれませんが，大きなプログラムを書く場合，この差は重要になります．

第2章　make

これまでの部分では，とりあえずのコンパイルの方法を説明しましたが，ファイルが多くなってくると面倒になってきます．せめて1回入力するだけで済んだらなどと考えるのも当然です．これらの処理を自動化する方法として，makeというプログラムが提供されています．

2.1　簡単なmakefile

makeでは，makefileというファイルに記述された通りに処理を行います．ここでは，次のようなファイルを作ってみましょう．

makefile

```
# a.outを作成するためのmakefile
a.out : main.o func.o
    gcc -o a.out main.o func.o -lm

main.o : main.c
    gcc -c main.c

func.o : func.c
    gcc -c func.c
```

このファイルは，

```
<ターゲットファイル1> [ターゲットファイル2] ... : [依存ファイル1] [依存ファイル2] ...[1]
    <コマンド>
    [コマンド]
```

のような形式で書かれています．また，#から行末まではコメントとして無視されます．ちなみに<実行するコマンド>の前にある空白は，タブ1個です．スペースを入れたりするとエラーになります．
このファイルでは，**ターゲットファイル**が**依存ファイル**よりも新しい場合，つまり，上書きされた場合や新しく作成された場合に指定された**コマンド**を実行します．つまり，このファイルは，

- main.oよりもmain.cが新しければ，gcc -c main.cを行う
- func.oよりもfunc.cが新しければ，gcc -c func.cを行う

[1] <param>や，[param]はBNF記法と呼ばれる物で，< >は省略不可，[ ]は省略可能なパラメータを示します．

- a.out よりも main.o あるいは，func.o が新しければ，
  gcc -o a.out main.o func.o -lm を行う

ということを記述しています．このファイルを実行するには，

```
$ make Enter
```

とするだけです．後は，makefile に記述した通りに作業を自動化してくれます．

2.2 マクロ

makefile では，マクロという機能を使えます．例えば，今書いた makefile は次のように書き直すことが出来ます．

```
# OBJS にオブジェクトファイルを列挙する
OBJS = main.o func.o

a.out : $(OBJS)
    gcc -o a.out $(OBJS) -lm

main.o : main.c
    gcc -c main.c

func.o : func.c
    gcc -c func.c
```

これは，main.o func.o と書くのが冗長であるため，それらを各部分を 1 箇所だけにしてみた例です．こうすると，main.o func.o と書く部分が少なくなるので，記述ミスを防ぐ効果があります．
また，コンパイラも常に gcc であるとは限りません．環境によっては，cc になるかもしれませんし，cl になるかもしれません．このときに，わざわざ全ての gcc という部分を書き換えるのは大変です．このような場合には，上と同じようにマクロを使って，

```
# CC にコンパイラの名前を入力
OBJS = main.o func.o
CC = gcc

a.out : $(OBJS)
    $(CC) -o a.out $(OBJS) -lm

main.o : main.c
    $(CC) -c main.c

func.o : func.c
    $(CC) -c func.c
```

と書き換えることが出来ます．こうすると，コンパイラが cc である環境に持っていったときに 1 箇所を書き換えればよいので負担が少なくなります．また，コマンドラインで，

```
$ make CC=cc Enter
```

としても，同様のことが実現できます．
以上のようにマクロをうまく使うことで makefile の書き換えを最小限に押さえることができます．

2.3 make の詳細な使い方

2.3.1 マクロの参照

マクロ名が 2 文字以上の場合，例えば，SRC を参照する場合，$(SRC) のようにして参照しますが，マクロ名が 1 文字の場合，単に$A のように書くことが出来ます．この後に出てくる$*も 1 文字のマクロの一種といえるでしょう．
また，マクロの参照ではなく，$そのものを表したい場合には，$$のように，$を 2 回重ねて書きます．

2.3.2 複数のターゲットファイル

次のような書き方もできます．

```
main.o sub.o : sample.h
    gcc -c main.c
```

この例は，次のものと全く同じ意味です．

```
main.o : sample.h
    gcc -c main.c

sub.o : sample.h
    gcc -c main.c
```

ただし，考えてみれば分かることですが，main.o，sub.o どちらの時にも，gcc -c main.c を行うという意味になってしまうので，この文には意味がありません．このような場合には，$*を使います．

```
main.o sub.o : sample.h
    gcc -c $*.c
```

$*は，ターゲットファイルから**拡張子を除いた**部分を意味するマクロです．従って，この例は次の物と同じ意味になります．

```
main.o : sample.h
    gcc -c main.c

sub.o : sample.h
    gcc -c sub.c
```

2.3.3 長いマクロの定義

マクロの定義を複数行に分けたい場合には，

```
SRC = main.c sub1.c sub2.c sub3.c sub4.c
SRC += sub5.c sub6.c sub7.c sub8.c
```

のようにします．この場合，最終的に，SRC の値は，

```
main.c sub1.c sub2.c sub3.c sub4.c sub5.c sub6.c sub7.c sub8.c
```

となります．また，sub4.c と sub5.c の間には自動的に空白が 1 文字入ります．
また，既に定義されているマクロの値の前に挿入するように値を追加したい場合には，

```
SRC = main.c sub1.c sub2.c sub3.c sub4.c
SRC := sub5.c sub6.c sub7.c sub8.c $(SRC)
```

のようにします．決して，

```
SRC = main.c sub1.c sub2.c sub3.c sub4.c
SRC = sub5.c sub6.c sub7.c sub8.c $(SRC)
```

のようにしてはいけません．このようにした場合，エラーが発生します．

2.3.4 慣習的に定義されるマクロ

make では，表 2.1 マクロを慣習的に用いています．これらは決して強制される物ではありませんが，普通，よく使われる物です．自分の書いた makefile の可読性を上げる意味でもこれらのマクロを積極的に使うようにすることが望ましいといえます．

2.3.5 動的マクロ

動的マクロとはその状況によって内容の変わるマクロのことをいいます．既に出てきた$*も動的マクロです．動的マクロのいくつかを図 2.2 に示します．

2.3.6 環境変数とマクロ

makeが実行される際に環境変数として定義されているものは全てマクロとして定義されます．従って，

```
$ setenv CFLAGS -O Enter
$ make Enter
```

とすると，CFLAGS というマクロが定義されることになります．ただし，makefile のなかで既に CFLAGS の定義があれば，そちらが優先されます．

2.3.7 暗黙のルール

次の例は暗黙のルールを適用した例です．

```
CC = gcc
CFLAGS = -O
```

表 2.1: 慣習的に用いられるマクロ

CC	C コンパイラのコマンド名
CFLAGS	C コンパイラのコンパイルオプション
CPPFLAGS	C コンパイラのプリプロセッサオプション

表 2.2: 動的マクロ

$@	ターゲットファイル名.
$*	ターゲットファイル名から拡張子を除いたもの.
$<	依存ファイルのファイル名．依存ファイルが複数の場合は，最初のファイル名.
$^	全ての依存ファイル名．依存ファイルが複数の場合は，全てのファイル名.
$?	全ての依存ファイルの内，ターゲットファイルよりも新しい物.

```
.c.o :
    $(CC) $(CFLAGS) -c $< -o $@

main.o : main.c sample.h
sub.o : sub.c sample.h
```

この例では，最初の部分の，

```
.c.o :
    $(CC) $(CFLAGS) -c $< -o $@
```

が非常に重要です．この部分は，ターゲットファイルが，????.c や，????.o である場合に，

```
gcc -O -c <依存ファイル> -o <ターゲットファイル>
```

としなさいといっています．そして，その下の部分では，このルールを暗黙の内に適用するため，コマンドが書かれていません．このような暗黙のルールを使うことによって makefile を簡素に書くことが出来ます．また，暗黙のルールでは，ターゲットファイルを直接指定したり，依存ファイルを指定したりすることはできません．そうした場合，暗黙のルールとは見なされなくなってしまいます．