目次

0.1 unix コマンド

mv
cp
rm (ファイル名): remove の略でファイルを消去するコマンド． rm -r (ディレクトリ) でディレクトリの削除．ディレクトリ削除の場合はメッセージが出てしまい削除が出来ない場合があるため，問答無用の削除をする手段として -rf オプションを-r の代わりに用いることもある．

0.2 環境構築

sudo apt install flex
sudo apt install byacc
sudo apt install bison++
sudo apt install bison

0.3 make 入門

0.3.1 make コマンドの基本的な使い方

たとえば program というコマンドのソースが source-1.c source-2.c source-3.c program.h という 3 つのソースファイルと 1 つのヘッダファイルから構成され，各 source-*.c 中で program.h を include しているとする．一括してコンパイルしたい場合は
cc -o program source-1.c source-2.c source-3.c
とすればよいが，あるソースファイルだけを書き換えた場合でも，全てのソースをいちいちコンパイルすることになり面倒．そこで makefile という名前のファイルを作成し，その中に

makefile 1: makefile 中身

```
1 program: source-1.o source-2.o source-3.o
2   cc -o program source-1.o source-2.o source-3.o
3 source-1.o: source-1.c program.h
4   cc -c source-1.c
5 source-2.o: source-2.c program.h
6   cc -c source-2.c
7 source-3.o: source-3.c program.h
8   cc -c source-3.c
```

とかいておく．ソースコード (1) 中の 1 行目は program は source-1.o，source-2.o，source-3.o から構成される．2 行目は program を作成するには cc -o program source-1.o source-2.o source-3.o を実行すれば良い．しかし source-1.o， source-2.o， source-3.o を実行するには，4，6，8 行目を先に実行しなければならない．3 行目の source-1.o は source-1.c と program.h が必要であることを意味する．一般的には makefile は

makefile 2: makefile 一般

```
1 target: prerequisite
2 (TAB)command in order to make the target
```

という情報の羅列である．TAB の部分は必ず TAB でありスペースではダメ．ソースコード (2) 中の prerequisite は必須項目を表し，target はターゲットを表す．command はターゲットを更新するために実行されるコマンドを表す．複数の prerequisite や target やコマンドを使用する場合は下記のようになる．

makefile 3: makefile 一般

```
1 target1 target2 target3: prerequisite1 prerequisite2
2 (TAB)command1
3 (TAB)command2
4 (TAB)command3
```

ソースコード (2) 中のコロン (:) の左側には 1 つかそれ以上のターゲットを，右側には必須項目を 0 個以上記述する．

コンパイルを一回もしていない状態で makefile を実行 (make とコマンドする) とカレントディレクトリの makefile が読まれ，最初のターゲット名 (ここでは program) を作成しようとする．従って
> make
cc -c source-1.c
cc -c source-2.c
cc -c source-3.c
cc -o program source-1.o source-2.o source-3.o
と実行される．

ここで source-1.c の内容を書き換えるとファイル更新されたので source-1.c のタイムスタンプが新しくなった．再度 make を実行すると make は各ファイルのタイムスタンプを調べる．その結果 source-1.o より source-1.c のほうが新しいので
>make
cc -c source-1.c
cc -o program source-1.o source-2.o source-3.o
と source-2.c と source-3.c は再コンパイルされずに必要なコンパイルだけを実行してくれる．一方で program.h を書き換えた場合は全てのソースがこのヘッダファイルを参照しているため，
> make
cc -c source-1.c
cc -c source-2.c
cc -c source-3.c
cc -o program source-1.o source-2.o source-3.o
とすべてのファイルがコンパイルしなおされる．

0.3.2 suffix ルール

suffix = 英語では接尾辞という意味．ここでは拡張子という意味で使う．ソースコード (1) の makefile は同じファイルが何度も繰り返し現れて，非常に無駄が多い．もう少し短くまとめたものを作るために suffix(拡張子) を利用する．makefile が威力を発揮するのは suffix の関係指示をしたときである．ここで*.o は*.c から生成されることにする．とくに source-1.o は source-1.c から作られ，source-2.o は source-2.c から作られ ··· という風に．特に foo.c→foo.o のように拡張子のみが段階的に変化していくことに注目してほしい．そこで役に立つのがサフィックスルールである．

makefile 4: makefile 中身

```
1 .SUFFIXES: .c .o
2 .c.o:
3   cc -c $<
```

と書くと，*.c についてそれぞれ cc -c を実行し *.o を作成してくれる． $<は現在の必須項目を表す．

(厳密には $<は最初の必須項目のファイル名 prerequisite1 を表す．) つまり

makefile 5: makefile 中身

```
1 program: source-1.o source-2.o source-3.o
2   cc -o program source-1.o source-2.o source-3.o
3 .SUFFIXES: .c .o
4 .c.o:
5   cc -c $<
```

と makefile に記述した場合
program は source-1 source-2 source-3
source-1.o は source-1.c から作成される
source-2.o は source-2.c から作成される
source-3.o は source-3.c から作成される
ということを表している．ここで source-3 だけを更新して make すると
> make
cc -c source-3.c
cc -o program source-1.o source-2.o source-3.o
となるがこのときに make は以下のような順番で依存関係のチェックを行う．

- program は source-1.o，source-2.o，source-3.o から作成されるので，*.o を新しく作成するべきかどうか調べる
- .c.o: により source-1.o は source-1.c から作成されることを知り，source-1.c のタイムスタンプを調べる → source-1.o のほうが新しい → コンパイルする必要無し
- 同様に source-2.o は source-2.c から作成されることを知り，source-2.c のタイムスタンプを調べる → source-2.o のほうが新しい → コンパイルする必要無し
- source-3.o は source-3.c より古い (source-3.c が更新されている) ので，cc -c source-3.c を実行 ($<には現在の必須項目である source-3.c が入っている)
- source-3.o が更新されたので， cc -o program source-1.o source-2.o source-3.o を作成

ただしこれでは *.c と program.h の依存関係が示されていない．そこで下記のようにすることで souce-1.o は source-1.c と program.h から作成されることを表すことができる．

makefile 6: makefile 中身

```
1 .c.o: program.h
2   cc -c $<
```

なお，make でよく使われるデフォルトルールとして，あらかじめ決まっているものがあり，ソースコード (5) 中の
.SUFFIXES: .c .o
はデフォルトで定義されており，省略することが可能．

0.3.3 マクロ

もうすこし読みやすくするためにマクロというものを使用する．マクロは変数のようなもので

makefile 7: makefile 中身

```
1 MACRO=value
```

で代入し$(MACRO) や${MACRO}で値を参照できる．目的のファイルは program で　オブジェクトファイルは 3 つの*.o なので以下のようにかける．

makefile 8: makefile 中身

```
1 TARGET=program
2 OBJS=source-1.o source-2.o source-3.o
```

また C コンパイラもマクロで定義することにする．

makefile 9: makefile 中身

```
1 CC=cc
```

このように抽象化しておくことで cc の代わりに gcc が使いたくなった場合に簡単に変更できる．(FreeBSD や Linux では cc=gcc だが Solaris などの cc は商用コンパイラである) program は OBJS に依存するので

makefile 10: makefile 中身

```
1 $(TARGET): $(OBJS)
2   $(CC) -o $@ -$(OBJS)
```

とかける．$@は現在のターゲットのファイル名が代入される．さらに clean というターゲットを作成し，いつでも作業途中の全ファイルを消せるように追加しておく．

makefile 11: makefile 中身

```
1  TARGET=program
2  OBJS=source-1.o source-2.o source-3.o
3  CC=cc
4  .SUFFIXES: .c .o
5  $(TARGET): $(OBJS)
6    $(CC) -o $@ -$(OBJS)
7  .c.o: program.h
8    $(CC) -c %<
9  clean:
10   rm -f $(TARGET) $(OBJS)
```

rm はファイルを消す remove の略であり，-f オプションは–force の略で存在しないファイルを無視する (確認を行わない) という意味である．ソースコード (11) の makefile は

- make とタイプすると実行ファイル program を作成する
- 一部のファイルを更新すると必要な部分だけを再コンパイルしてくれる
- make clean とすることで program と*o を削除する

という makefile が完成した.

なお　マクロは make の引数で指定することが可能で

>make CC=gcc

とすると$(CC) の部分が gcc に置き変わり gcc でコンパイルされる．make のデフォルトルールとして

CC=cc

は省略可能.