Makefile Document

GNU Make 강좌

임대영 RAXIS@hitel.net

v1.0, 1997년 8월 28일

1.1 make 유틸리티

make - GNU make utility to maintain groups of programs

The purpose of the make utility is to determine automatically which pieces of a large program need to be recompiled,

and issue the commands to recompile them.

우리말로 하면 make는 프로그램 그룹을 유지하는데 필요한 유틸리티이다. make유틸리티의 목적은 프로그램 그룹 중에서
어느 부분이 새롭게 컴파일되어야 하는지를 자동적으로 판단해서 필요한 커맨드(gcc따위)를 이용해서
그들을 재컴파일 시킨다고 되어 있다.

make는 일련의 프로그램 개발에만 쓰이지 않고, 컴파일러처럼 일종의 명령어 방식으로 처리되는 모든 곳에서 쓰일 수가 있다. 가령 LaTeX와 같은 경우도 .tex 파일에서 .dvi 파일을 만들고 다시 .ps 파일로 만드는 과정을 make를 사용해서 간단하게 만들 수가 있다.

Makefile은 make가 이해할 수 있도록 일종의 쉘 스크립트 언어같이 되어 있다(makefile database라 하기도 한다). 이 파일에는 결과 파일을 생성시키기 위한 파일들간의 관계, 명령어 등을 기술하고 있는데 이 강좌의 주된 목적이 바로 Makefile의 작성에 있다.

1.2 make의 필요성

우선은 make의 사용을 프로그램 개발과 유지 쪽으로 국한시키기로 한다.
보통 라인 수가 많아지면 여러 개의 파일로 나누어 (모듈로 나누어) 개발을 하게 된다.
이들은 알게 모르게 서로 관계를 가지고 있는데, 어느 하나를 필요에 의해 바꾸게 되었을 때
그 파일에 있는 함수를 이용하는 다른 파일도 새롭게 컴파일되어야 한다.

하지만 파일 수가 많은 경우 이를 일일이 컴파일을 하게 될 때,
그 불편함과 함께 컴파일하지 않아도 될 것도 컴파일을 하게 될 수도 있고,
컴파일해야 할 것도 미처 못하게 되는 경우가 있다(링크 에러의 원인이 되기도 하는데
에러의 원인을 제대로 찾기가 힘이 든다).

앞에서도 얘기했듯이 이런 상황에서 지능적으로 관계 있는 것만 새롭게 갱신을 할 필요가 있을 때
make파일은 빛을 발하게 된다.

2. 간단한 Makefile

2.1 Makefile 의 내부 구조

Makefile은 기본적으로 아래와 같이 목표(target), 의존 관계(dependency), 명령(command)의 세개로 이루어진 기분적인 규칙(rule)들이 계속적으로 나열되어 있다고 봐도 무방하다. make가 지능적으로 파일을 갱신하는 것도 모두 이 간단한 규칙에 의하기 때문이다.

```
-------------------------------------------------------------------------
target ... : dependency ...
                command
                ...
                ...
-------------------------------------------------------------------------
```

여기서 목표(target) 부분은 명령(command)이 수행이 되어서 나온 결과 파일을 지정한다. 당연히 목적 파일(object file)이나 실행 파일이 될 것이다.
명령(command)부분에 정의된 명령들은 의존 관계(depenency)부분에 정의된 파일의 내용이 바뀌었거나, 목표 부분에 해당하는 파일이 없을 때 이곳에 정의된 것들이 차례대로 실행이 된다. 일반적으로 쉘에서 쓸 수 있는 모든 명령어들을 사용할 수가 있으며 bash에 기반한 쉘 스크립트도 지원한다.

=> 참고: 참고로 목표 부분에는 결과 파일만 올 수 있는 것이 아니고, 보통 make clean 에서와 같이 간단한 레이블(label) 기능을 제공하기도 한다.

=> 명령 부분은 꼭 TAB 글자로 시작해야 한다. 그냥 빈칸 등을 사용하면 make 실행 중에 에러가 난다. make가 명령어인지 아닌지를 TAB 가지고 구별하기 때문!

2.2 Makefile 예제

간단한 Makefile을 만들어 본다. 우리가 만들려고 하는 프로그램은 main.c read.c write.c로 구성되어 있고 모두 io.h라는 헤더 파일을 사용한다고 가정한다. 이들을 각각 컴파일해서 test 라는 실행 파일을 생성시킨다.

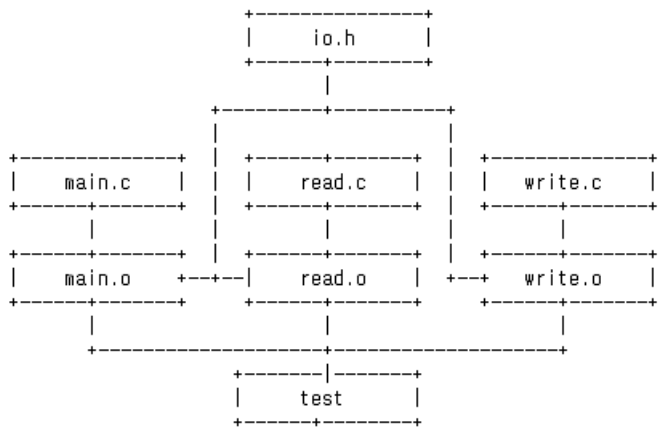
```
% gcc -c main.c
% gcc -c read.c
% gcc -c write.c
% gcc -o test main.o read.o write.o
```

위의 방식은 make를 쓰지 않고 그냥 명령어를 주는 방식이다. 파일의 수가 작아서 오히려 더 간단하게 보일 수 있으나, 파일이 100개정도 된다고 가정하면... 아찔...
그리고, 아래는 위와 똑같은 일을 수행하는 Makefile의 내용이다.

Makefile예제 1

```
-------------------------------------------------------------------------
test : main.o read.o write.o
                gcc -o test main.o read.o write.o
main.o : io.h main.c
                gcc -c main.c
read.o : io.h read.c
                gcc -c read.c
write.o: io.h write.c
                gcc -c write.c
-------------------------------------------------------------------------
```

make는 Makefile의 내용을 보고, 내부적으로 어떻게 파일들이 의존하고 있는지 조사한다. 위의 Makefile을 바탕으로 의존 관계를 그림으로 나타내 보면 아래와 같다.



위의 그림에서 보면 test 가 만들어지기 위해서는 main.o read.o write.o가 필요하게 각각의 목적 파일들은 모두 자신의 소스 파일과 io.h 에 의존함을 알 수가 있다.

가령 main.c를 고쳤다고 생각한다면 main.o가 컴파일되어 다시 생기고, test 도 다시 링크되어 갱신된다. 만약 io.h가 바뀌었다고 가정하면 모든 파일들이 컴파일되어서 목적 파일이 생기고, 그것들이 링크가 되어 test가 생긴다.

위와 같이 파일들을 구성한 다음 Makefile을 실행시켜 보자. Makefile의 실행은 그냥 make라고만 치면 된다.

```
% make
gcc -c main.c
gcc -c read.c
gcc -c write.c
gcc -o test main.o read.o write.o <- OK
```

2.3 매크로의 사용

간단한 매크로 기능을 사용해 보자. main.o read.o write.o라는 것을 OBJECTS 라는 매크로로 바꾸는 것이 아래의 예제 2에 나와 있다.

Makefile예제 2

```
---------------------------------------------------------------------------
OBJECTS = main.o read.o write.o

test : $(OBJECTS)
        gcc -o test $(OBJECTS)

main.o : io.h main.c
        gcc -c main.c
read.o : io.h read.c
        gcc -c read.c
write.o: io.h write.c
        gcc -c write.c
---------------------------------------------------------------------------
```

위에서 보다시피 매크로는 그냥 프로그램 짤 때와 같이 사용해서 값을 대입한다. 대신 사용할 때는 반드시 $(..) 안에 넣어서 사용한다. 매크로 치환을 위한 특수한 방법이 아닐까... 매크로의 사용법은 위와 같이 간단하므로 다양하게 정의해서 사용할 수 있다. 매크로에대한 자세한 설명은 다음 장에서 언급하기로 한다.

2.4 레이블의 사용

목표 부분에 해당하는 부분이 그냥 레이블과 같이 사용될 수도 있다고 이미 설명하였다. 예제 2 에다가 목적 파일들을 모두 삭제하는 명령어를 추가하기로 한다.

Makefile예제 3

```
-------------------------------------------------------------------------
OBJECTS = main.o read.o write.o

test : $(OBJECTS)
        gcc -o test $(OBJECTS)

main.o : io.h main.c
        gcc -c main.c
read.o : io.h read.c
        gcc -c read.c
write.o: io.h write.c
        gcc -c write.c

clean :
        rm $(OBECTS)

-------------------------------------------------------------------------
```

레이블로 사용될 때는 당연히 의존 관계 부분은 없어도 된다. 그리고 clean을 실행시키려면 아래와 같이 한다.

```
% make clean
rm main.o read.o write.o <- OK
```

3. 매크로(Macro) 와 확장자(Suffix) 규칙

3.1 매크로란 무엇인가? (What is Macro)

앞에서 매크로에 대해서 대충 언급을 했다. 프로그램을 짜본 사람이나 로터스, 한글, 엑셀 등의 모든 패키지에서 매크로라는 것을 사용하게 된다. 은연중에 매크로의 정의는 대충 짐작하고 있을 것이다. 이미 알고 있는바와 같이 매크로는 특정한 코드를 간단하게 표현한 것에 지나지 않는다. Makefile에서 사용되는 매크로는 비교적 그 사용법이 간단하기 때문에 금방 익혀서 사용할 정도가 된다.

매크로의 정의는 프로그램을 작성할 때 변수를 지정하는 것처럼 하면 된다. 그리고, 매크로를 사용하기 위해서는 $(..)을 이용하면 된다. 아래는 매크로의 간단한 예제이다.

=> 참고: 매크로의 사용에서 ${..}, $(..), $..를 모두 사용할 수 있습니다. 그러나 대부분의 책에서는 $(..) 을 사용하라고 권하는군요.

Makefile예제 4

```
-------------------------------------------------------------------------
OBJS = main.o read.o write.o

test : $(OBJS) <- (1)
gcc -o test $(OBJS)
-------------------------------------------------------------------------
```

첫 번째 장에서 다루었던 예제와 거의 비슷하다. 매크로는 사실상 복잡한 것을 간단하게 표시한 것에 지나지 않는다. (1) 번을 매크로를 안 쓰고 표현한다면 아마 아래와 같이 될 것이다.

Makefile예제 5

```
-------------------------------------------------------------------------
test : main.o read.o write.o
gcc -o test main.o read.o write.o
-------------------------------------------------------------------------
```

=> 참고: 예제 5가 더 쉽지 않느냐고 반문하는 사람은 매크로의 위력을 잘 모르는 사람입니다. 거의 모든 소프트웨어에서 매크로를 지원하는 이유를 한번 잘 생각해 봅시다. 예제 4 의 (1)부분이 이해하기 난해하다고 하실 지는 모르겠지만, 대충 형식이 정해져 있기 때문에 조금만 익숙해지면 오히려 더 편할 겁니다.

make에 관해 설명한 책에 다음과 같은 명언(?) 이 나온다.

Macro makes Makefile happy. (매크로는 Makefile 을 기쁘게 만든다.)

이 말은 Makefile을 작성함에 있어 매크로를 잘만 이용하면 복잡한 작업도 아주 간단하게 작성할 수 있음을 말해 주는 말이 아닐까 생각한다. 매크로에 대해서는 더 이상 말할 것이 없다. (너무 간단하죠 ?) 이제 남은 것은 여러분들이 자신의 매크로를 어떻게 구성하느냐이다. 어떤 것을 매크로로 정의해야 할지는 여러분들의 자유이며, 나중에 전반적인 지침을 설명할 것이다.

3.2 미리 정해져 있는 매크로 (Pre-defined macro)

여러분들보다 머리가 약간 더 좋은 사람들이 make 라는 것을 만들면서 미리 정해 놓은 매크로들이 있다. 'make -p' 라고 입력해 보면 make에서 미리 세팅되어 있던 모든 값들(매크로, 환경 변수(environment) 등등)이 엄청 스크롤 된다. 이 값들을 보고 미리 주눅 들 필요는 없다. 어차피 대부분의 내용들은 우리가 재정의 해주어야 하기 때문에 결론적으로 말하면 우리가 모두 작성한다고 생각하는 것이 마음이 편하다.,,

아래에는 대부분이 UNIX 계열의 make에서 미리 정해져 있는 매크로들 중에 몇 가지만 나열해 본 것이다.

Predefined Macro 예제 6

```
-------------------------------------------------------------------------
ASFLAGS = <- as 명령어의 옵션 세팅
AS = as
CFLAGS = <- gcc 의 옵션 세팅
CC = cc (= gcc)
CPPFLAGS = <- g++ 의 옵션
CXX = g++
LDLFAGS = <- ld 의 옵션 세팅
LD = ld
LFLAGS = <- lex 의 옵션 세팅
LEX = lex
YFLAGS = <- yacc 의 옵션 세팅
YACC = yacc
MAKE_COMMAND = make
-------------------------------------------------------------------------
```

=> 참고: 직접 make -p를 해서 한번 확인해 보세요. 과연 make는 내부적으로 어떤 변수들을 사용하고 있는지 알아봅시다. 매크로는 관습적으로 대문자로 작성되니까 이점에 유의해서 보세요. make는 쉘상에서 정의한 환경 변수값들을 그대로 이용한다는 것을 알고 계시기 바랍니다.

위에 열거한 매크로는 make에서 정의된 매크로중 그야말로 일부에 지나지 않는다. 하지만 프로그램을 작성함에 있어 가장 많이 사용하게 될 매크로 들이다. 이들 매크로는 사용자에 의해 재정의 가능하다. 가령 gcc의 옵션 중에 디버그 정보를 표시하는 '-g' 옵션을 넣고 싶다면, 아래와 같이 재정의 한다.

CFLAGS = -g

예제 6 의 각종 FLAG 매크로들은 대부분 우리가 필요에 의해 세팅해 주어야 하는 값들이다. 왜 굳이 make에서 값도 정의되지 않은 매크로를 우리가 정의해서 써야 하는지 의문을 던질지도 모른다. 우리가 더 이쁜 이름으로 매크로를 정의할 수도 있다고 하면서...

여기서 한가지 사실을 생각해 봐야 할 것이다. make에서 위에 나온 것들을 왜 미리 정해 두었을까? (왜일까요?) make에서 이들 매크로를 제공하고 있는 이유는 내부적으로 이들 매크로를 사용하게 되기 때문이다. 어떻게 이용하는지는 확장자 규칙(Suffix rule)을 설명하면서 해답을 제공할 것이다. 이제 예제 4 의 Makefile을 매크로를 이용하여 깔끔하게(?) 작성해 보자.

Makefile예제 7

```
---------------------------------------------------------------------------
OBJECTS = main.o read.o write.o
SRCS = main.c read.c write.c <- 없어도 됨

CC = gcc <- gcc 로 세팅
CFLAGS = -g -c <- gcc 의 옵션에 -g 추가

TARGET = test <- 결과 파일을 test 라고 지정

$(TARGET) : $(OBJECTS)
$(CC) -o $(TARGET) $(OBJECTS)

clean :
        rm -rf $(OBJECTS) $(TARGET) core

main.o : io.h main.c <- (1)
read.o : io.h read.c
write.o: io.h write.c
---------------------------------------------------------------------------
```

위의 Makefile 을 동작시켜 보자.

```
% make
gcc -g -c main.c -o main.o
gcc -g -c read.c -o read.o
gcc -g -c write.c -o write.o
gcc -o test main.o read.o write.o <- OK

% make clean
rm -rf main.o read.o write.o test core <- OK
```

그런데 여기서 한가지 이상한 점을 발견하게 될 것이다. .c 파일을 .o 파일로 바꾸는 부분이 없는데 어떻게 컴파일이 되었을까? 빼먹고 타이핑 못한 것은 아닐까 하고... 절대 아님!

앞에서 CFLAGS 같은 매크로는 make 파일의 내부에서 이용된다고 하였다. 그렇다면 make는 과연 어디에서 이용을 할까? 바로 컴파일하는 곳에서 이용을 하는 것이다. 따라서 우리는 CFLAGS를 셋팅해 주기만 하면 make가 알아서 컴파일을 수행하는 것이다. (얼마나 편리합니까!)

=> 참고: 확장자 규칙에서 다시 한번 자세히 설명을 하겠습니다.
(1) 에 해당하는 부분은 어떤 파일이 어디에 의존하고 있는지를 표시해 주기 위해서 꼭 필요하다. .c 파일을 컴파일하는 부분은 일괄적인 루틴으로 작성할 수 있기 때문에 이들 파일간의 의존 관계(dependency)를 따로 표시해 주어야 한다.

=> 참고: 파일간의 의존 관계를 자동으로 작성해 주는 유틸리티가 있습니다. 이것은 다음 장에서 다루기로 합니다.

3.3 확장자 규칙 (Suffix rule)

확장자 규칙이란 간단히 말해서 파일의 확장자를 보고, 그에 따라 적절한 연산을 수행시키는 규칙이라고 말할 수 있다. 가령 .c 파일은 일반적으로 C 소스 코드를 가리키며, .o 파일은 목적 파일(Object file)을 말하고 있다. 그리고 당연히 .c 파일은 컴파일되어서 .o 파일이 되어야 하는 것이다.

여기서 한가지 매크로가 등장하게 된다. .SUFFIXES 라고 하는 매크로인데 우리가 make 파일에게 주의 깊게 처리할 파일들의 확장자를 등록해 준다고 이해하면 될 것이다.

```
.SUFFIXES = .c .o
```

위의 표현은 '.c' 와 '.o' 확장자를 가진 파일들을 확장자 규칙에 의거해서 처리될 수 있도록 해준다. .SUFFIXES 매크로를 이용한 예제를 살펴보자.

Makefile예제 8

```
-------------------------------------------------------------------------
.SUFFIXES = .c .o

OBJECTS = main.o read.o write.o
SRCS = main.c read.c write.c

CC = gcc
CFLAGS = -g -c

TARGET = test

$(TARGET) : $(OBJECTS)
        $(CC) -o $(TARGET) $(OBJECTS)

clean :
        rm -rf $(OBJECTS) $(TARGET) core
main.o : io.h main.c
read.o : io.h read.c
write.o: io.h write.c
```

위의 Makefile 을 동작시켜 보자.

```
% make
gcc -g -c main.c -o main.o
gcc -g -c read.c -o read.o
gcc -g -c write.c -o write.o
gcc -o test main.o read.o write.o <- OK
```

확장자 규칙에 의해서 make는 파일들간의 확장자를 자동으로 인식해서 필요한 작업을 수행한다. 즉 아래의 루틴이 자동적으로 동작하게 된다.

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

=> 참고: gmake에서는 약간 다르게 정의되어 있지만, 우선은 같다고 이해합시다. $< , $@ 에 대해서는 곧 설명합니다.
우리가 .SUFFIXES = .c .o 라고 했기 때문에 make 내부에서는 미리 정의된 .c (C 소스 파일)를 컴파일해서 .o (목적 파일)를 만들어 내는 루틴이 자동적으로 동작하게 되어 있다. CC와 CFLAGS 도 우리가 정의한 대로 치환될 것임은 의심할 여지가 없다.

make 내부에서 기본적으로 서비스를 제공해 주는 확장자들의 리스트를 열거해 보면 아래와 같다. 각 확장자에 따른 자세한 설명은 생략한다.

.out .a .ln .o .c .cc .C .p .f .F .r .y .l .s .S .mod .sym .def .h .info .dvi .tex .texinfo .texi .txinfo .w .ch .web .sh .elc .el

Makefile내부에서 .SUFFIXES 매크로의 값을 세팅해 주면 내부적으로 정의된 확장자의 연산이 동작을 하게 된다. 따라서 확장자 규칙은 make가 어느 확장자를 가진 파일들을 처리할 것인가를 정해 주는 것이라고 생각할 수 있다.

그러나 이것은 필자만의 생각일지 몰라도 make에서 자동적으로 확장자를 알아서 해주는 것이 좋긴 하지만, 필자는 일부러 위의 .c.o 에 해당되는 부분을 그냥 정의해서 쓰길 더 좋아한다. 이것은 지금까지의 습관상 그렇지만 왠지 우리가 정의하는 것이 더 자유롭게(flexible) 사용할 수 있을 것 같기 때문이다. 그리고 이런 기능은 우리가 작성을 해봐야 make의 메카니즘을 더 잘 이해할 수 있다고 생각한다.

예제 8 의 내용을 약간 바꾸어 보자.
Makefile예제 9

```
---------------------------------------------------------------------------
.SUFFIXES = .c .o

OBJECTS = main.o read.o write.o
SRCS = main.c read.c write.c

CC = gcc
CFLAGS = -g -c
INC = -I/home/raxis/include <- include 패스 추가

TARGET = test

$(TARGET) : $(OBJECTS)
        $(CC) -o $(TARGET) $(OBJECTS)

.c.o : <- 우리가 확장자 규칙을 구현
        $(CC) $(INC) $(CFLAGS) $<-

clean :
        rm -rf $(OBJECTS) $(TARGET) core

main.o : io.h main.c
read.o : io.h read.c
write.o : io.h write.c
---------------------------------------------------------------------------
% make
gcc -I/home/raxis/include -g -c main.c
gcc -I/home/raxis/include -g -c read.c
gcc -I/home/raxis/include -g -c write.c
gcc -o test main.o read.o write.o <- OK
```

예제 8 과 예제 9 의 차이는 그저 .c .o 부분을 누가 처리하느냐이다. 그리고 예제 9에서는 INC 라는 매크로를 추가시켜서 컴파일할때 이용하도록 하였다.

3.4 내부 매크로 (Internal macro)

make에서는 내부 매크로라는 것이 있다. 이것은 우리가 맘대로 정할 수 있는 매크로는 절대 아니다. 대신 매크로를 연산, 처리하는데 쓰이는 매크로라고 하는 것이 더 적당할 것이다.

Internal Macro 예제 10

$* <- 확장자가 없는 현재의 목표 파일(Target)

$@ <- 현재의 목표 파일(Target)

$< <- 현재의 목표 파일(Target)보다 더 최근에 갱신된 파일 이름

$? <- 현재의 목표 파일(Target)보다 더 최근에 갱신된 파일이름

=> 참고: 책에서는 $< 와 $?를 약간 구분하고 있지만 거의 같다고 봐도 무방할 것입니다.

각 내부 매크로에 대한 예를 보기로 한다.

```
main.o : main.c io.h
gcc -c $*.c
```

$* 는 확장자가 없는 현재의 목표 파일이므로 $* 는 결국 main 에 해당한다.

```
test : $(OBJS)
gcc -o $@ $*.c
```

$@는 현재의 목표 파일이다. 즉 test에 해당된다.

```
.c.o :
gcc -c $< (또는 gcc -c $*.c)
```

$< 는 현재의 목표 파일보다 더 최근에 갱신된 파일 이름이라고 하였다. .o 파일보다 더 최근에 갱신된 .c 파일은 자동적으로 컴파일이 된다. 가령 main.o를 만들고 난 다음에 main.c를 갱신하게 되면 main.c는 $<의 작용에 의해 새롭게 컴파일이 된다.

=> 참고: 이제 예제 9 을 이해할 수 있겠습니까?

=> 참고: Makefile 파일을 작성해 놓고, 그냥 make만 치시면 make는 Makefile의 내용을 살펴보다가 첫 번째 목표 파일에 해당되는 것을 실행시키게 됩니다. 따라서 위의 예제에서는 make test 라고 해도 같은 결과를 내게 됩니다. 반면 clean에 해당하는 부분을 윗부분에 두게 되면 make는 항상 make clean을 수행하게 됩니다.

```
% make <- make clean 이 실행됨
rm -rf main.o read.o write.o test core

% make test <- 강제적으로 test 가 생성되게 한다.
gcc -I/home/raxis/include -g -c main.c
gcc -I/home/raxis/include -g -c read.c
gcc -I/home/raxis/include -g -c write.c
gcc -o test write.c main.o read.o write.o <- OK
```

Makefile의 이해를 돕기 위해서 Makefile을 하나 더 작성해 보기로 한다. make.tex 파일을 make.dvi로 만든 다음 이것을 다시 make.ps로 만드는 것이다. 보통의 순서라면 아래와 같다.

% latex make.tex <- make.dvi 가 만들어진다.

% dvips make.dvi -o <- make.ps 가 만들어진다.

보통의 가장 간단한 Makefile을 작성해 보면 아래와 같다.

Makefile예제 11

```
-------------------------------------------------------------------------
make.ps : make.dvi
        dvips make.dvi -o

make.dvi : make.tex
        latex make.tex
-------------------------------------------------------------------------
```

위와 같은 일을 하는 Makefile을 다르게 한번 작성해 보자. 매크로를 어느정도 사용해 보기로 하며, 확장자 규칙을 한번 적용해 보기로 한다.

Makefile예제 12

```
-------------------------------------------------------------------------

.SUFFIXES = .tex .dvi

TEX = latex <- TEX 매크로를 재정의

PSFILE = make.ps
DVIFILE = make.dvi

$(PSFILE) : $(DVIFILE)
        dvips $(DVIFILE) -o

make.ps : make.dvi
make.dvi : make.tex

-------------------------------------------------------------------------
```

예제 12 에서는 .tex 와 .dvi 를 처리하는 루틴이 자동적으로 동작을 하게 된다. Makefile 을 한번 동작시켜 보자.

% make

```
latex make.tex
....
dvips make.dvi -o <- OK
```

예제 11 과 예제 12 는 하는 일은 같다. 하지만 예제 12는 매크로를 사용함으로써 나중에 내용을 바꿀 때 예제 11보다 편하다는 것을 이해하였으므로...

4. Makefile를 작성할 때 알면 좋은 것들

Makefile을 작성할 때 기본적으로 알고 있으면 유익한 것들을 기술한다. 이전 강좌의 내용을 대체로 이해하고 있다면 좋은 팁이 될 것이다. 메뉴얼에 나오는 광범위한 내용은 다루지 않고 기본적인 것들에 관심을 두기로 한다.

4.1 긴 명령어를 여러 라인으로 표시하기

Makefile을 작성할 때 명령어가 한 줄을 넘어간다고 가정하자. 이때 그냥 줄줄이 적는다면 읽기도 힘들고, 작성하는 사람도 조금 찜찜하다. 이때 '\' 문자를 이용해서 여러 라인으로 나타낼 수 있다. 이미 C언어에 익숙한 사람이라면 낯익은 기호일 것이다. 아래의 예제를 보자.

예제 13

```
---------------------------------------------------------------------------
OBJS = shape.o \
rectangle.o \
circle.o \
line.o \
bezier.o
---------------------------------------------------------------------------
```

위의 예제는 OBJS = shape.o rectangle.o circle.o line.o bezier.o 라는 문장을 여러 라인으로 표시한 것이다. 보기에도 깔끔해 보이지 않은가.

4.2 확장자 규칙의 이용 (Use suffix rule !!)

두번째 장에서 확장자 규칙에 대해서 많이 설명을 했다. Makefile을 작성할 때 C, C++, tex 등의 파일은 이미 정의되어 있는 규칙을 이용하면 간단하고, 깔끔한 Makefile을 작성할 수 있다. 두번째 장에서 직접 우리가 규칙을 간단히 구현해 보기도 했는데, 이것은 확장자 규칙의 개념을 설명하기 위함이었다.

어떤 파일들이 이미 규칙으로 정해져 있는지 한번 살펴보기로 한다. 아래에 열거된 파일들은 특별히 따로 정의하지 않은 상태에서 바로 이용할 수 있는 것들이다.(GNU Make 매뉴얼에 바탕을 두고 작성되었다.)

C 컴파일 (XX.c -> XX.o)
C++ 컴파일 (XX.cc 또는 XX.C -> XX.o)
Pascal 컴파일 (XX.p -> XX.o)
Fortran 컴파일 (XX.f 또는 XX.F -> XX.o)
Modula-2 컴파일 (XX.def -> XX.sym)
(XX.mod -> XX.o)
assembly 컴파일 (XX.s -> XX.o)
assembly 전처리 (XX.S -> XX.s)
single object file 의 링크 (XX.o -> XX)
Yacc 컴파일(?) (XX.y -> XX.o)
Lex 컴파일(?) (XX.l -> XX.o)
lint 라이브러리 생성 (XX.c -> XX.ln)
tex 파일 처리 (XX.tex -> XX.dvi)
texinfo 파일처리 (XX.texinfo 또는 XX.texi -> XX.dvi)
RCS 파일 처리 (RCS/XX,v -> XX)
SCCS 파일처리 (SCCS/XX.n -> XX)

위에 정의된 파일만이 make에서 처리할 수 있는 것은 아니다. 그 밖의 파일에 대해서는 사용자가 직접 정의해 주면 얼마든지 make를 사용할 수 있다.

그럼 이젠 위와 같은 파일들을 처리하기 위한 명령어는 어떤 매크로로 정의되어 있는지 알아보자. 이미 말했듯이 아래에 열거된 매크로는 재정의 가능하다. 가령 TEX = tex 이지만 대부분 TEX = latex로 재정의 되어야 할 것이다.

AR = ar (Archive maintaining program)
AS = as (Assembler)
CC = cc (= gcc , C compiler)
CXX = g++ (C++ compiler)
CO = co (extracting file from RCS)
CPP = $(CC) -E (C preprocessor)
FC = f77 (Fortran compiler)
LEX = lex (LEX processor)
PC = pc (Pascal compiler)
YACC = yacc (YACC processor)
TEX = tex (TEX processor)
TEXI2DVI = texi2dvi (Texiinfo file processor)
WEAVE = weave (Web file processor)
RM = rm -f (remove file)

이미 두번째 장에서 밝혔지만 위의 명령어에서 사용될 FLAG(옵션)에 정의한 매크로에 대해서도 알아보기로 한다.

ARFLAGS = (ar achiver의 플래그) *
ASFLAGS = (as 어셈블러의 플래그)
CFLAGS = (C 컴파일러의 플래그) *
CXXFLAGS = (C++ 컴파일러의 플래그) *
COFLAGS = (co 유틸리티의 플래그)
CPPFLAGS = (C 전처리기의 플래그)
FFLAGS = (Fortran 컴파일러의 플래그)
LDFLAGS = (ld 링커의 플래그) *
LFLAGS = (lex 의 플래그) *
PFLAGS = (Pascal 컴파일러의 플래그)
YFLAGS = (yacc 의 플래그) *

위애서 '*'표시한 것은 자주 쓰이게 될 플래그이다. 위에서 표시한 여러 가지 매크로들을 무조건 재정의 하라는 배려에서인지, 대부분 값이 설정되어 있지 않다. 가령 C프로그램을 짤 때 CFLAGS를 재정의 해야 할 것이다.

4.3 매크로 치환 (Macro substitution)

매크로를 지정하고, 그것을 이용하는 것을 이미 알고 있다. 그런데, 필요에 의해 이미 매크로의 내용을 조그만 바꾸어야 할 때가 있다. 매크로 내용의 일부만 바꾸기 위해서는 $(MACRO_NAME:OLD=NEW)과 같은 형식을 이용하면 된다.

MY_NAME = Michael Jackson
YOUR_NAME = $(NAME:Jack=Jook)

위의 예제에서는 Jack이란 부분이 Jook으로 바뀌게 된다. 즉 YOUR_NAME 이란 매크로의 값은 Michael Jookson 이 된다. 아래의 예제를 하나 더 보기로 한다.

OBJS = main.o read.o write.o
SRCS = $(OBJS:.o=.c)
위의 예제에서는 OBJS에서 .c가 .o로 바뀌게 된다. 즉 아래와 같다.

SRCS = main.c read.c write.c

위의 예제는 실제로 사용하면 아주 편할 때가 많다. 가령 .o 파일 100개에 .c 파일이 각각 있을 때 이들을 다 적으려면 무척이나 짜증나는 일이 될 것이다.

4.4 자동 의존 관계 생성 (Automatic dependency)

일반적인 make의 구조는 아래와 같이 target, dependency, command가 연쇄적으로 정의되어 있는 것과 같다고 하였다.

```
target : dependency
        command
        ...
```

그런데 위에서 command가 없이 타겟과 의존 관계만 표시가 되면 이는 타겟이 어느 파일에 의존하고 있는지 표시해 주는 정보의 역할을 한다. 이런 정보는 Makefile을 작성할 때 없어서는 안되는 부분이다. (이 부분이 없으면, make는 정말 바보처럼 행동합니다.)

그런데 일일이 이런 정보를 만든다는 것은 쉬운 일이 아니다. 파일이 1000개라고 할 때 이것을 어케 다 표시하누...

이런 단조롭고 귀찮은 일을 자동으로 해주는 좋은 유틸리티가 있다. 우선 gccmakedep가 있는지 확인해 보자. gccmakedep는 어떤 파일의 의존 관계를 자동으로 조사해서 Makefile의 뒷부분에 자동으로 붙여 주는 유틸리티이다. gccmakedep가 없다면 gcc -M XX.c 라고 해보자. 그러면 XX.c의 의존 관계가 화면에 출력됨을 알 수 있을 것이다. (gccmakedep 도 내부적으로 gcc -M 을 사용한다.)

프로그램을 설치할 때 make dep 라는 것을 친 기억이 있을 것이다. 파일들의 의존 관계를 작성해 준다는 의미이다. 그럼 우리의 Makefile에도 이런 기능을 첨가해 보기로 한다.

예제 14

```
.SUFFIXES = .c .o
CFLAGS = -O2 -g

OBJS = main.o read.o write.o
SRCS = $(OBJS:.o=.c)
test : $(OBJS)
        $(CC) -o test $(OBJS)

dep :
        gccmakedep $(SRCS)
```

위의 Makefile을 이해할 수 있다면 이제 Makefile에 대해서 어느 정도 도가 텄다고 해도 무방할 것이다. 위의 예제에서 파일들간의 의존 관계가 없다. 그럼 이제 make dep 을 써서 자동적으로 생성시켜 보자.

```
% make dep
% vi(emacs) Makefile
```

Makefile의 뒷부분에 다음과 같은 내용이 붙어 있는 것을 알게 될 것이다.

예제 15

```
# DO NOT DELETE
main.o: main.c /usr/include/stdio.h /usr/include/features.h \
/usr/include/sys/cdefs.h /usr/include/libio.h \
/usr/include/_G_config.h io.h
read.o: read.c io.h
write.o: write.c io.h
```

main.o에는 조금 자질구레한 헤더 파일까지 붙어 있다. 이것은 헤더 파일 안에서 include 하는 파일들을 다 찾다 보니까 그런 것이다. 별로 신경쓸 것은 없고... 대충 우리가 지금까지 손으로 작성해 온 것과 거의 흡사함을 알 수있다. 아니 오히려 더 정확함을 알 수 있다. (이제부터 make는 스마트하게 동작한다.)

4.5 다중 타겟 (Multiple target)

하나의 Makefile에서 꼭 하나의 결과만 만들어 내라는 법은 없다. 가령 결과 파일이 3개가 필요하다고 하자. 아래의 예제를 보기로 한다.

예제 15

```
-------------------------------------------------------------------------
.SUFFIXES = .c .o
CC = gcc
CFLAGS = -O2 -g
OBJS1 = main.o test1.o <- 각각의 매크로를 정의
OBJS2 = main.o test2.o
OBJS3 = main.o test3.o
SRCS = $(OBJS1:.o=.c) $(OBJS2:.o=.c) $(OBJS3:.o=.c)

all : test1 test2 test3 <- 요기에 주의

test1 : $(OBJS1)
        $(CC) -o test1 $(OBJS1)

test1 : $(OBJS2)
        $(CC) -o test2 $(OBJS2)

test1 : $(OBJS3)
        $(CC) -o test3 $(OBJS3)

dep :
        gccmakedep $(SRCS)
-------------------------------------------------------------------------
```

위의 프로그램은 make all 을 함으로써 동작한다. 실제로 동작시켜 보면 아래와 같은 결과가 나온다.

```
% make all (또는 make)
gcc -O2 -g -c main.c -o main.o
gcc -O2 -g -c test1.c -o test1.o
gcc -o test1 main.o test1.o <- test1 의 생성
gcc -O2 -g -c test2.c -o test2.o
gcc -o test2 main.o test2.o <- test2 의 생성
gcc -O2 -g -c test3.c -o test3.o
gcc -o test3 main.o test3.o <- test3 의 생성
```

4.6 순환 make (Recursive MAKE)

규모가 큰(?) 프로그램들은 파일들이 하나의 디렉토리에 있지 않는 경우가 많다. 여러 개의 서브시스템이 전체 시스템을 구성한다고 가정하면 각 서브시스템에 Makefile이 존재한다. (서브시스템 = 서브디렉토리) 따라서 여러 개의 Makefile을 동작시킬 필요가 있도록 Makefile을 고쳐 보자. 서브디렉토리에 있는 Makefile을 동작시키는 방법은 의외로 간단하다. 아래의 간단한 예제를 보자.

예제 16

```
---------------------------------------------------------------------------
subsystem:
        cd subdir; $(MAKE) .....(1)

subsystem:
        $(MAKE) -C subdir .....(2)
---------------------------------------------------------------------------
```

위의 예제에서 (1)과 (2)는 동일한 명령을 수행한다 (1)을 기존으로 동작을 한번 묘사해 보자. 우리가 만들 시스템의 타겟이 subsystem이다. (이름은 아무래도 상관없다) 우선 subdir이라는 곳으로 가서, 거기에 있는 Makefile을 동작시키게 된다. (간단하죠.) MAKE라는 것은 그냥 make라는 명령어를 표시하는 매크로일 뿐... 그럼 완전한 예제를 한번 구성해 보기로 한다.

예제 16

```
---------------------------------------------------------------------------
.SUFFIXES = .c .o
CC = gcc
CFLAGS = -O2 -g

all : DataBase Test <- 요기에 집중.

DataBase:
        cd db ; $(MAKE) <- db 로 이동해서 make 실행

Test:
        cd test ; $(Make) <- db 로 이동해서 make 실행
---------------------------------------------------------------------------
```

위의 예제에서 db, test 디렉토리에 있는 Makefile은 지금까지 우리가 공부했던 Makefile과 거의 흡사하다고 가정하자.
그럼 위의 Makefile을 실행시켜 본다.

```
% make
cd db ; make
make[1]: Entering directory`/home/raxis/TEST/src'
gcc -O2 -g -c DBopen.c -o DBopen.o
gcc -O2 -g -c DBread.c -o DBread.o
gcc -O2 -g -c DBwrite.c -o DBwrite.o
make[1]: Leaving directory `/home/windows/TEST/src'
cd test ; make
make[1]: Entering directory `/home/raxis/TEST/test'
gcc -O2 -g -c test.c -o test.o
make[1]: Leaving directory `/home/windows/TEST/test'
```

위의 가상 실행을 보면 우선 db로 가서 거기의 Makefile을 수행시키고, 다음에는 test로 가서 Makefile을 실행시킴을 볼 수 있다.
우선은 단순하게 컴파일만 시켰는데, 다르게 한번 생각해 보자. db 디렉토리에서의 최종 타겟으로 가령 db.a을 만들어 내고
test 디렉토리에서 이를 링크 시킨다고 생각하면 꽤 괜찮은 시나리오가 될 것이다. 위에서 1이라고 나타난 것은
현재의 레벨을 의미한다. 원래 디렉토리의 레벨이 0이고, 여기서는 레벨이 하나 더 내려갔으므로 1이라고 표시된 것이다.

4.7 불필요한 재컴파일 막기

의존 관계 규칙에 의해 하나가 바뀌면 그에 영향받는 모든 파일이 바뀐다고 앞에서 말했다. 그러나 다른 파일들에게 아무 영향을 주지 않도록 수정하였는데도 재컴파일을 시도한다면 시간 낭비가 될 수도 있다. 가령 모든 .c 파일에서 include 하는 헤더 파일에서 새로운 #define PI 3.14 라고 정의를 했다고 가정하자. 그리고 PI라는 것은 아무 곳에서도 사용을 하지 않는다.

이때는 'make -t' 라고 해보자. -t 는 touch를 의미하는 옵션으로써 컴파일을 하지 않는 대신 파일의 생성 날짜만 가장 최근으로 바꾸어 놓는다. 새로 컴파일 된 것처럼 처리를 하는 것이다. touch유틸리티 명령어에 익숙한 사람이라면 이해할 것이다. touch는 파일의 생성 날짜를 현재로 바꾸어 주는 간단한 유틸리티이다.

6. Makefile 작성의 가이드라인

make를 많이 써 본 사람은 어느 정도 자신만의 Makefile을 작성하는 일정한 스타일 같은 것이 있다. 프로그램이 짜는 사람마다 다르듯이 Makefile도 각각이다. 여기서는 그냥 가장 일반적인 가이드라인을 제시하기로 한다. 다음 장에서 Makefile의 여러 예제를 살펴보면서 다시 한번 자세히 설명할 것이다.

매크로를 잘 사용하면 Makefile이 깔끔해질 뿐 아니라, 내용의 수정도 용이하다. 조금 과장해서 말한다면, 최대한 매크로를 많이 사용하라고 말하고 싶다. Makefile내에서 두번 이상 나오는 것들은 매크로로 정의해 두면 편하다. 자신의 프로그램 특성에 따라서 기존의 매크로를 재정의 하는 것도 좋다.

make에서 정의되어 있는 규칙들을 최대한 이용한다. 확장자 규칙은 무조건 이용하기를 권한다. 기존의 규칙들을 자기가 정의하는 것도 좋지만, 억지로 이럴 필요는 없다.

대체로 아래와 같이 Makefile을 구성한다.
매크로 정의 부분
타겟을 얻기 위한 명령어 부분
의존 관계 부분

예제 17

```
-------------------------------------------------------------------------
.SUFFIXES = .c .o       --+
CFLAGS = -g               |
                          |
OBJS = main.o \           |
read.o \                  | 매크로 정의 부분
write.o                   |
SRCS = $(OBJS:.o=.c)      |
                          |
TARGET = test             --+

$(TARGET): $(OBJS)                        --+
                $(CC) -o $@ $(OBJS)         |
dep :                                       |
                gccmakedpend $(SRCS)        |
new :                                       | 명령어 정의 부분
                touch $(SRCS) ; $(MAKE)     |
clean :                                     |
                $(RM) $(OBJS) $(TARGET) core  --+

- 여기부터 의존관계 부분
-------------------------------------------------------------------------
```

위의 예제는 최대한 매크로를 많이 이용하려고 했기 때문에 독해(?)하기 어려울 수도 있다.

7. Makefile의 실제 예제

지금까지 강좌를 진행하면서 Makefile의 여러 가지 예제들을 제시하였다. 강좌에 나온 예제들을 조금만 바꾸면 자신의 Makefile로써 사용할 수 있다. 여기에서는 여러 가지 Makefile들의 기본틀(template)들을 소개하고자 한다.

7.1 프로그램 제작에 쓰일 수 있는 Makefile

여기서는 우선 가장 많이 사용되는 C와 C++에서의 Makefile을 소개하기로 한다. 여러 개의 파일들을 컴파일해서 하나의 실행 파일을 만드는 예제 틀이 바로 예제 7.1이다.

예제 7.1

```
-------------------------------------------------------------------------
.SUFFIXES = .c .o
CC = gcc

INC = <- include 되는 헤더 파일의 패스를 추가한다.
LIBS = <- 링크할 때 필요한 라이브러리를 추가한다.
CFLAGS = -g $(INC) <- 컴파일에 필요한 각종 옵션을 추가한다.

OBJS = <- 목적 파일의 이름을 적는다.
SRCS = <- 소스 파일의 이름을 적는다.

TARGET = <- 링크 후에 생성될 실행 파일의 이름을 적는다.

all : $(TARGET)

$(TARGET) : $(OBJS)
        $(CC) -o $@ $(OBJS) $(LIBS)

dep :
        gccmakedep $(INC) $(SRCS)

clean :
        rm -rf $(OBJS) $(TARGET) core

new :
        $(MAKE) clean
        $(MAKE)
-------------------------------------------------------------------------
```

예제 7.1 에서 바꿔야 할 부분은 표시를 해 두었다. 자신의 파일들로 적당히 고쳐 준 다음 make dep 을 수행시켜 본다. 그러면 자동으로 의존 관계가 생성된다.

```
% make dep <- 자동으로 의존 관계 생성
% make <- make 동작
```

지금까지의 강좌를 이해하고 있다면 위의 Makefile의 독해란 어렵지 않을 것이다. 개략적인 사항만 설명하기로 한다.

.SUFFIXES = .c .o
make 내부에서 정의된 확장자 규칙을 이용하기 위한 것이다. make는 자동적으로 .c와 .o로 끝나는 파일들간에 정의된 규칙이 있는지 찾게 되고 적당한 규칙을 찾아서 수행하게 된다.

CFLAGS = -g $(INC)
CFLAGS 매크로를 재정의 하고 있다. -g 는 디버그 정보를 추가하라는 것이고, $(INC)는 컴파일할때 필요한 include 패스를 적어 두는 곳이다.

all : $(TARGET)
make는 Makefile을 순차적으로 읽어서 가장 처음에 나오는 규칙을 수행하게 된다. 여기서 all 이란 더미타겟(dummy target)이 바로 첫 번째 타겟으로써 작용하게 된다. 관습적으로 all이란 타겟을 정의해 두는 것이 좋다. 결과 파일이 많을 때도 all의 의존관계(dependency)로써 정의해 두면 꽤 편리하다.

dep : gccmakedep $(INC) $(SRCS)
의존 관계를 자동적으로 생성해 주기 위한 것이다. 헤더 파일의 패스까지 추가되어야 한다는 것에 주의하기 바람. 이것은 내부적으로 gcc가 작동되기 때문이다.

예제 7.1 을 C++ 파일에 이용하기 위해서는 .SUFFIEX , CC, CFLAGS 그리고 타겟에 대한 명령어를 아래의 내용과 같이 바꾸면 된다.

예제 7.2
```
-------------------------------------------------------------------------
.SUFFIXES = .cc .o

CXX = g++
CXXFLAGS = -g $(INC)

$(TARGET) : $(OBJS)
$(CXX) -o $@ $(OBJS) $(LIBS)
-------------------------------------------------------------------------
```

물론 각자의 취향에 따라서 Makefile을 만들어도 된다. 여기서 제시하고 있는 Makefile은 어디까지나 필자의 관점에서 만든 Makefile을 소개하고 있는 것뿐이니까...

7.2 라이브러리와의 링크가 필요한 필요한 Makefile
라이브러리를 만들기 위한 Makefile은 어떤 차이가 있을까. 예제 7.1과 거의 흡사하다. 다만 TARGET 이 실행 파일이 아니고 라이브러리라는 것뿐. 그리고 라이브러리 만드는 방법을 알고 있어야 한다는 것...

=> 참고: 라이브러리 만드는 방법을 소개하기로 한다. read.o, write.o를 libio.a로 만들어 보자. 라이브러리를 만들기 위해서는 ar 유틸리티와 ranlib 유틸리티가 필요하다. (자세한 설명은 man 을 이용)

```
% ar rcv libio.a read.o write.o

a - read.o <- 라이브러리에 추가 (add)
a - write.o

% ranlib libio.a <- libio.a의 색인(index)을 생성
```

그럼 위의 과정을 Makefile로 일반화시킨다면 어떻게 될까? 아주 조금만 생각하면 된다. 예제 7.1에서 TARGET를 처리하는 부분만 아래와 같이 바꾸어 보자.

예제 7.3
```
-------------------------------------------------------------------------
TARGET = libio.a

$(TARGET) : $(OBJS)
        $(AR) rcv $@ $(OBJS) <- ar rcv libio.a read.o write.o
        ranlib $@ <- ranlib libio.a
-------------------------------------------------------------------------
```

ELF 기반에서 동적 라이브러리(dynamic library, shared library)를 만들어 보기로 하자. ELF 상에서는 동적 라이브러리 만드는 방법이 이전에 비해 아주 간단해 졌다. (옛날에 모티프 소스 가지고 동적 라이브러리 만든다고 고생한 것에 비하면 세상이 너무 좋아진 것 같음) BSD계열의 유닉스를 사용해 본 사람이라면 비슷하다는 것을 느낄 것이다. 그럼 read.c write.c를 컴파일해서 libio.so.1을 만들어 보자.(so는 shared object를 의미, 뒤의 .1은 동적 라이브러리의 버전을 의미)

% gcc -fPIC -c read.c <- -fPIC을 추가해서 컴파일한다.
% gcc -fPIC -c write.c

% gcc -shared -Wl,-soname,libio.so.1 -o libio.so.1 read.o write.o

동적 라이브러리를 만들기 위한 옵션

위와 같이 하면 libio.so.1 가 생성된다. 사용자가 만든 동적 라이브러리를 사용하는 방법에 대해서는 간단히만 언급하기로 한다. 우선 libio.so.1 을 /usr/lib로 옮겨서 ldconfig -v 해서 라이브러리 설정을 갱신해 주던지, 아니면 LD_LIBRARY_PATH를 지정해 두면 된다. 아래는 test.c를 libio.so.1과 링크 시키는 예제이다.

% gcc -c test.c
% gcc -o test test.o -L. -lio <- 현재 디렉토리에 있다고 가정

예제 7.1를 약간 고쳐서 동적 라이브러리를 자동적으로 만들어 보자. 이번엔 완전한 내용의 Makefile을 소개한다.

예제 7.4

```
.SUFFIXES = .c .o

CC = gcc

INC =
LIBS =
CFLAGS = -g $(INC) -fPIC <- -fPIC 추가

OBJS = read.o write.o
SRCS = read.c write.c

TARGET = libio.so.1 <- libio.so.1이 최종 파일

all : $(TARGET)
        $(TARGET) : $(OBJS)
        $(CC) -shared -Wl,-soname,$@ -o $@ $(OBJS)

dep :
        gccmakedep $(INC) $(SRCS)

clean :
        rm -rf $(OBJS) $(TARGET) core
```

아주 조금밖에 바뀌지 않았다. 따라서 이 글을 읽는 여러분은 이제 각자의 목적에 맞게 Makefile을 구성할 수 있을 것이다. 대부분 확장자 규칙과 최종 파일을 생성해 내기 위한 명령어가 무엇인지 알고 있으면 Makefile을 자기 개성껏 꾸밀 수 있을 것이다.

7.3 LaTeX에서 쓰일 수 있는 Makefile

Makefile이 쓰일 수 있는 다른 예로써 가장 대표적인 것이 latex를 사용할 때이다. 이미 이전 강좌에서 여러 차례 소개된 적도 있다. 그럼 doc.tex를 doc.ps로 만들어 보기로 하자. 약간 어렵게 하기 위해서 doc.tex는 내부적으로 intro.tex 와 conclusion.tex를 포함하고 있다고 가정한다. (논문 같은 것을 작성할 때는 이렇게 .tex 파일이 많아지게 된다.)

```
% latex doc.tex <- doc.dvi의 생성
% dvips doc.dvi -o <- doc.tex의 생성
```

위와 같은 일을 수행하는 Makefile을 한번 살펴보자.

예제 7.5

```
-------------------------------------------------------------------------
.SUFFIXES = .tex .dvi <- 확장자 규칙

TEX = latex

OBJ = doc.dvi
SRC = doc.tex

TARGET = doc.ps <- 결과 파일

all : $(TARGET)

$(TARGET) : $(OBJ)
        dvips $(OBJ) -o <- dvips doc.dvi -o

new : <- 강제적으로 다시 make
        touch $(SRC) ; $(MAKE)

doc.tex : intro.tex conclusion.tex <- 의존 관계 설정
-------------------------------------------------------------------------
```

=> 참고로 gccmakedep는 latex 파일은 지원을 하지 않는 것 같다. 따라서 의존 관계 같은 것은 우리가 직접 적어 주어야 한다.

8. make 수행 시에 나타나는 에러들

make를 수행하게 되면 이상한 에러에 당황을 하게 되는 경우가 많아, 도대체 어디가 틀렸는지 감을 못잡는 경우가 허다하다. 그런데 make 매뉴얼에도 에러에 대한 종류와 그 대처 방안에 대해서는 거의 언급이 없는 관계로 이 부분은 필자의 경험에 의거해서 작성한다. (에러의 원인, 대처 방안이 모두 다 틀렸을 수도 있다는 것을 염두에 두기 바랍니다.)

Makefile:17: *** missing separator. Stop.
Makefile을 작성할 때 명령어(command)부분은 모두 TAB 문자로 시작해야 한다고 첫 번째 장부터 강조하였다. 위의 에러는 TAB 문자를 쓰지 않았기 때문에 make가 명령어인지 아닌지를 구별 못하는 경우이다.

대처: 17번째 줄(근처)에서 명령어가 TAB 문자로 시작하게 바꾼다.

make: *** No rule to make target `io.h', needed by `read.o'. Stop.
위의 에러는 의존 관계에서 문제가 발생했기 때문이다. 즉 read.c가 io.h에 의존한다고 정의되어 있는데, io.h를 찾을 수 없다는 에러이다.

대처: 의존 관계에서 정의된 io.h가 실제로 존재하는지 조사해 본다. 없다면 그 이유를 한번 생각해 본다. make dep를 다시 실행시켜서 의존 관계를 다시 생성시켜 주는 것도 하나의 방법이다.

Makefile:10: *** commands commence before first target. Stop.
위의 에러는 '첫 번째 타겟이 나오기 전에 명령어가 시작되었다'는 애매한 에러 메시지이다. 필자가 경험한 이 에러의 원인은 주로 긴 문장을 여러 라인에 표시를 하기 위해서 '\'를 사용할 때, 이를 잘못 사용했기 때문인 것 같다. 즉 '\'부분은 라인의 가장 끝문자가 되어야 하는데 실수로 '\'뒤에 스페이스를 몇 개 집어넣으면 여지없이 위의 에러가 발생한다.

대처: 10번째 줄(근처)에서 '\'문자가 있거든 이 문자가 라인의 가장 끝문자가 되도록 한다. 즉 '\'문자 다음에 나오는 글자(스페이스가 대부분) 는 모조리 없애 버린다.

make를 수행시키면 의도했던 실행 파일은 안생기고 이상한 행동만 한다. 가령 make clean 했을 때와 같은 행동을 보인다.
make는 천재가 아니라는 점을 생각해야 한다. make는 Makefile의 내용을 읽다가 첫 번째 타겟으로 보이는 것을 자신이 생성시켜야 할 결과 파일이라고 생각한다. 따라서 clean 부분을 Makefile의 첫번째 타겟으로 정해 버리면 위와 같은 결과가 나타나게 된다.

대처: 예제 7.1에서 all 이라는 필요 없는 타겟을 하나 만들어 두었다. 이것은 make가 all 을 첫 번째 타겟으로 인식시키기 위함이었다. 따라서 자신이 생성시키고 싶은 결과 파일을 첫 번째 타겟이 되게 하던지, 아니면 예제 7.1처럼 all과 같은 더미 타겟(dummy target)을 하나 만들어 둔다. 그리고 make clean, make dep 같은 부분은 Makefile의 끝부분에 만들어 두는 것이 안전하다.

이미 컴파일했던 파일을 고치지 않았는데도 다시 컴파일한다.
이 행동은 make가 의존 관계를 모르기 때문이다. 즉 사용자가 의존 관계를 설정해 주지 않았다는 말이 된다. 따라서 make는 무조건 모든 파일을 컴파일해서 실행 파일을 만드는 일이 자신이 할 일이라고 생각하게 된다.

대처: 목적 파일, 소스 파일, 헤더 파일들의 의존 관계를 설정해 주어야 한다. gccmakedep *.c 라고 하면 Makefile의 뒷부분에 자동적으로 의존 관계를 만들어 준다. 그외의 다른 파일들에 대해서는 사용자가 적절하게 의존 관계를 설정해 주어야 한다.

main.o : main.c io.h
read.o : read.c io.h
write.o : write.c io.h

위의 예제는 첫 번째 장에서도 제시했던 건데... TARGET : DEPENDENCY의 형식으로 의존 관계를 작성한 것이다. (make에게 의존 관계를 알려주는 방법이죠)

그 외의 경우에 대해서는 각자가 한번 원인과 결과를 알아보기 바란다. 그리고 팁의 형식으로 글을 올린다면 다른 사람에게도 많은 도움이 될 것이다. 일단 make에서 에러를 내기 시작하면 초보자는 원인조차 모르는 경우가 많기 때문이다.