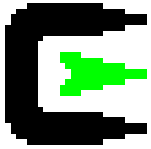
C

make

install

http://www.cygwin.com/

중좌측 하단부 클릭



Install or update
now!

Devel
Package
autoconf
automake
gcc-core: C compiler
make: The GNU version of the ‘make’ utility

간단한 컴파일

설치 된 cygwin의 홈 디렉토리에
make_test 폴더를 만든다.

폴더 안에 main.c 을 만든다.

```
/* main.c */
#include <stdio.h>
int main(void)
{
    printf("Hello, world₩n");
    return 0;
}
```

gcc 간단 명령

gcc main.c

-> 실행 파일을 만든다. a.exe 파일 생성

gcc -c main.c

-> main.o 파일만 만든다.

gcc -o run main.c

-> .o 파일은 만들지 않고

바로 run.exe 파일만 만든다.

실행방법

./a

-dumpmachine : 컴파일러 타겟 cpu
-save-temps : 컴파일 도중 생기는 중간파일 생성
-time : 컴파일 각 과정의 수행 시간
-pipe : 중간파일을 사용하지 않고 파이프사용
-da : 최적화 과정의 파일( )

-O0 : 기능끄기
-O1 : 컴파일시간 코드크기 실행시간
-O2 :
-O3 :
-Os : 코드사이즈 최소화

디렉토리 검색

-I dir : 표준 헤더와 사용자 헤더 지정 디렉토리 검색
-I- : -I 설정 시 검색헤더종류의 제한
(전처리기기가 현재 디렉토리에서 .h를 안 찾음)

-I -I- : 사용자 헤더 파일만 검색
-I- -I : 사용자 헤더 파일과 시스템 헤더 파일 모두 검색
/* 위는 i 의 대문자 아래 l은 L의 소문자*/

-L dir : 라이브러리 디렉토리 추가
-l dir : 라이브러리 파일 목록 추가 (-L의 기능보완)

-nostdinc : 표준헤더파일이 있는 디렉토리 안 사용

링크

-lname : 소문자 L 링크시 name인 라이브러리 찾기

ar rcs libmy.a myfile.o

gcc main.o libmy.a –o run.exe
gcc main.o –L. -lmy –o run.exe

-l 옵션은 순서적이다

Zoo.o –lmy boo.o (boo.o에서 libmy.a를 참조)

gcc 옵션 중

-ansi
표준과 충돌하는 GNU 확장안을 취소하며 ANSI/ISO C 표준을 지원한다. 이 옵션은 ANSI 호환 코드를 보장하지 않는다.

-pedantic
ANSI/ISO C 표준에서 요구되는 모든 경고를 나타낸다.

-pedantic-errors
ANSI/SIO C 표준에서 요구되는 모든 에러를 나타낸다.

-MM
적절한 형태로 종속성 리스트를 만들어 낸다.
소스파일과 헤더파일의 다양한 조합 속에서 ...

-std=c99 -std=c89
가변배열, // 단일주석 테스트해보자

추론법칙(inference rule)

```
/* foo.c */
#include<stdio.h>
int main(void)
{
    puts("main");
}
```

Makefile이 없는 상태에서

```
make foo
```

추론법칙(inference rule)

make foo

cc foo.c –o foo
make 가 컴파일러를 호출 하는 방법을 안다.

4개의 파일로 다시 변경

```
/* inc.h */
#ifndef _inc_h_
#define _inc_h_
#include<stdio.h>
int zooo();
int booo();
#endif
```

```
/* zooo.c */
#include " inc.h "
int zooo()
{
        printf("func zooo. ₩n");
        return 0;
}
```

```
/* main.c */
#include " inc.h "
int main(void)
{
        zooo();
        booo();
        puts("main end");
        return 0;
}
```

```
/* booo.c */
#include " inc.h "
int booo()
{
        printf("func booo. ₩n");
        return 0;
}
```

불편한 컴파일

gcc –o main.exe main.c zooo.c booo.c 혹은

gcc –ansi –pedantic –o main.exe main.c zooo.c booo.c

매번 파일 바꿔 줄 때 마다 실행해야 한다.

혹은 헤더 파일 변경시 반복 실행해야 한다.

make Utility(파일 관리 유틸리티)

작은 프로그램 -> 편집 -> 컴파일

큰 프로그램 -> 편집 -> 컴파일(문제발생)

- 모든 파일을 새로 컴파일 해야 한다.

Make 자체로는 응용프로그램을 빌드 하는 방법을 알지 못함

따라서 빌드법을 명시하는 파일 Makefile 이 있어야 한다.

make Utility

- 프로그램 그룹 중 새롭게 컴파일되어야 하는지를 자동적으로 판단 그들을 재컴파일 시킨다
- make –f <file_name> file_name이라는 파일을 먼저 찾고 없을 경우 GNUmakefile, makefile, Makefile 순으로 하나가 있으면 그 파일을 읽게 된다.

일반적으로 Makefile을 추천 나머지는 인식률과 가독성으로 인함

필요성

- 파일의 수가 많아 각각 컴파일 해야 할 때
- 미처 컴파일 하지 못한 경우
- 컴파일 하지 않아도 되는 파일이 있는 경우

Makefile 내부

C

- 3가지 구조로 나뉨

  목표(target), 의존 관계(dependency), 명령(command)

형태 : 세가지의 기본적인 규칙(rule)의 나열 방법

Makefile

```
target ... : dependency ...
tab문자command
        ...
        ...
        ...
```

다섯 항목의 기술파일

- 콜론(:) – 타깃과 필요항목을 구분 짓는다
- 타깃 – 콜론의 왼쪽
- 필요항목 – 종속행(dependecy line), 규칙행(rule line)
- 탭문자 – 명령 행 앞에 쓰여진다.
- 명령행 – 필요항목에서 가져와 작성한다.

최종 타깃의 하위항목을 모두 확인 한 후
타깃의 필요항목을 최신파일로 바꾼후
최종 타깃을 위한 명령 행을 실행한다

C

기술파일의 규칙

1. 명령행은 반드시 tab문자로 시작
2. 비어있는 행은 무시
3. # 한줄 주석을 표시함(개행까지 무시한다)
4. ₩ 역슬래쉬를 통해 행이 길어질 경우를 이어준다.
   단, ₩뒤에는 반드시 enter가 있어야 한다.
5. ; 종속행 항목뒤에 명령을 이어 쓸 수 있다.(탭의 예외)
   명령행을 나눌 때 사용한다. 한 라인에 같이 쓸 수 있다.
6. 필수 항목이 없는 target 설정 할 수 있다(명령 절 바로 실행)
7. 명령라인에는 어떠한 명령이 와도 된다.

명령행 앞에 tab 문자가 없을 경우

error발생

*** missing separator. Stop

cat –vte makefile 으로 탭과 개행 확인 한다.

-v : 개행에 ^M

-t : 개행에 ^M Tab에 ^I , 개행에 ^M(-t 만 사용)

-e : 개행에 ^M$

모든 명령행은 탭 문자로 시작

Makefile 작성
(파일명이 확장자없이 Makefile 이다)

```
main.exe: main.o
    gcc -W -Wall -o main.exe main.o
```

-Wall 모든 경고 메세지 출력

-W Wall에서 제외된 16가기 종류의 경고메세지 출력

Makefile 작성

```
run : zooo.o booo.o main.o
    gcc -W -Wall -o run zooo.o booo.o main.o
zooo.o : zooo.c
    gcc -W -Wall -c -o zooo.o zooo.c
booo.o : booo.c
    gcc -W -Wall -c -o booo.o booo.c
main.o : main.c
    gcc -W -Wall -c -o main.o main.c
clean :
    rm -rf *.o *.exe
```

-Wall 모든 경고 메세지 출력

-W Wall에서 제외된 16가기 종류의 경고메세지 출력

Dummy target(Phony target)

필수 항목이 없는 target 설정 할 수 있다(명령 절 바로 실행)

```
clean :
    rm -rf *.o *.exe
```

visual basic clean명령

1. make를 실행 후

2. 다시 make 를 실행 하자
   make : ‘target’ is up to date.

3. 3개의 .c 파일 중 한 개 수정해 보자

4. 다시 make
   수정된 파일만 다시 컴파일 된다.

불필요한 컴파일을 막을 때

make –t

파일의 생성 날짜를 현재 시간으로 갱신한다. (-touch)

zooo.c 로 zooo.o가 1시에 만들어졌다.

zooo.c를 수정한 시각은 2시이다.(재 컴파일 대상)

make –t 옵션을 사용할 경우

zooo.o의 시각을 현재시각으로 맞춘다(2시 이후가 됨)

zooo.c는 새롭게 컴파일 되지 않는다.

매크로 규칙

1. = 를 포함한다
2. # 주석의 시작
3. 논리라인 일 경우 ₩ 사용한다.
4. $() ${} 를 사용한다.
5. 정의 되지 않은 매크로는 null문자열로 치환
6. 중복 시 최후 정의 값으로 치환
7. 이전 매크로를 이용한 매크로 정의 가능

   NAME = string

   NAME2 = MY $(NAME) #MY string
8. 여러 대입 기법사용

   =, :=, +=, ?=

C

대입 기법

=	재귀적 확장 매크로(여러번스캔)
:=	단순 확장 매크로(위->아래 한번스캔)
+=	연결 확장 매크로(기존매크로에 공백과추가)
?=	조건 매크로(있으면 무시, 없으면 생성)

C

대입 기법 사용

```
OBJ = kbs                    #kbs
#OBJ = $(OBJ) mbc
*** recursive variable 'OBJ' references itself (eventually). Stop

SAM = file
SAM = $(KBJ) edit            #sbs edit          =   재귀적 확장 매크로

KBJ := tbc
KBJ :=  $(UNG) sbs           #sbs               := 단순 확장 매크로

UNG := tool
UNG := $(UNG) help           #tool help

KGB += ask
KGB += you                   #ask you           += 연결 확장 매크로

COM = this
COM ?=that                   #this              ?= 조건 매크로
```

C

매크로 주의

1. name = “main.c” # “main.c” 으로 인식
2. 매크로명 ‘:’, ‘=‘, ‘#’
3. 순차적이다. 치환보다 먼저 정의
4. ‘₩’, ‘>’ 과 같은 쉘 메타 문자 사용 자제

매크로

매크로 정의

```
    test = /usr/bin        # ''나 ""를 사용하지 않는다.
    $(test) , ${test}
```

내부적으로 정의 된 매크로

```
    ${CC}
```

명령행에서 매크로

```
    make OBJECT=main.o   #'',""로 묶어 줄 수 있다.
```

매크로 적용

```
CC          = gcc
CFLAGS = -W -Wall
TARGET =  run

all : $(TARGET)
$(TARGET) : zooo.o booo.o main.o
    $(CC) $(CFLAGS) -o $(TARGET) zooo.o booo.o main.o
zooo.o : zooo.c
    $(CC) $(CFLAGS) -c -o zooo.o zooo.c
booo.o : booo.c
    $(CC) $(CFLAGS) -c -o booo.o booo.c
main.o : main.c
    $(CC) $(CFLAGS) -c -o main.o main.c
clean :
    rm -rf *.o *.exe
```

make 옵션

-C dir
Makefile을 계속 읽지 말고 우선은 dir로 이동하라는 것이다. 순환 make에 사용된다.

-d
Makefile을 수행하면서 각종 정보를 모조리 출력해 준다. (-debug)
make 의 동작을 대충 이해할 수 있다.

-h
옵션에 관한 도움말을 출력한다. (-help)

-f file
file 에 해당하는 파일을 Makefile로써 취급한다. (-file)

-r
내장하고 있는 각종 규칙(Suffix rule 등)을 없는 것으로 (-no-builtin-rules)간주한다. 따라서 사용자가 규칙을 새롭게 정의해 주어야 한다.

make 옵션

-t
파일의 생성 날짜를 현재 시간으로 갱신한다. (-touch)

-v
make의 버전을 출력한다. (-version)

-p
make에서 내부적으로 세팅되어 있는 값들을 출력한다. (-print-data-base)

-k
make는 에러가 발생하면 도중에 실행을 포기하게 되는데 (-keep-going) -k는 에러가 나더라도 멈추지 말고 계속 진행하라는 뜻

기정의 매크로

make -p | grep ^[[:alpha:]]*[[:space:]]*=[[:space:]] > premacro.txt

/*** premacro.txt ***/

CO = co
WINDIR = C:₩WINDOWS
PATHEXT = .COM;.EXE;.BAT;.CMD;.VBS;.VBE;.JS;.JSE;.WSF;.WSH
USERNAME = bj
CC = gcc
INFOPATH = /usr/local/info:/usr/share/info:/usr/info:
CPP = $(CC) -E
LD = ld
YACC = yacc
CFLAGS = -W -Wall

내부적으로 미리정의 된 매크로(make -p) C

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

자동매크로

$@ : target 파일 이름 (확장자 포함)　　확장자규칙불가

$^ : target의 종속 항목 리스트 전부　　확장자규칙불가

$? : target보다 더 최근에 변경된 종속항목 파일

$< : target의 종속 항목 리스트 첫번째(확장자 포함) 확장자규칙

$* : target 파일 이름 (확장자 제외)　　확장자규칙

$% : 현재의 타깃이 라이브러리 모듈일 때 .o 파일에 대응되는 이름

main.o : $@ main.o

　　$* main

main : $@ main

　　$* 못 찾는다.//target의 확장자가 없으니..

$(@F) 타겟의 파일　　$(<F)필요항목의 파일

$(@D) 타겟의 디렉토리　　$(<D)필요항목의 디렉토리

자동매크로사용

```
CC          = gcc
CFLAGS = -W -Wall
TARGET = run

all : $(TARGET)
$(TARGET) : zooo.o booo.o main.o
    $(CC) $(CFLAGS) -o $@ $^
zooo.o : zooo.c
    $(CC) $(CFLAGS) -c -o $@ $^
booo.o : booo.c
    $(CC) $(CFLAGS) -c -o $@ $^
main.o : main.c
    $(CC) $(CFLAGS) -c -o $@ $^

clean :
    rm -rf *.o *.exe
```

종속리스트의 부족

C

```
target : main.o   #의존관계 표시시 엉킴
        gcc -o $@.exe main.o zooo.o booo.o
```

위의 예제에서 $@를 $?으로 하면?

만들질 파일의 가장 최근의 파일로 .....

taget와 실행파일(run)의 불일치

```
target : main.o zooo.o booo.o
    gcc -o run $^

clean :
    rm -rf *.o *.exe
```

바뀐 파일은 컴파일 하나
매번 최종 명령행을 실행한다.

강력한 확장자 규칙

C

파일을 확장자에 따라 적절한 연산을 수행하는 규칙

.SUFFIXES 특수타겟이다.

.SUFFIXES 의 종속항목 은 관심을 갖을 확장자를 make에게 등록해 준다.

make 에게 새로운 확장자 법칙을 추가한다.(재정의)

.SUFFIXES : .c .o //gcc -c main.c 로 된다

주의 깊게 본다 확장자를

내부 확장자 규칙

```
%.C:
%: %.C
    $(LINK.C) $^ $(LOADLIBES) $(LDLIBS) -o $@
%.o: %.C
    $(COMPILE.C) $(OUTPUT_OPTION) $<

%.cpp:
%: %.cpp
    $(LINK.cpp) $^ $(LOADLIBES) $(LDLIBS) -o $@
%.o: %.cpp
    $(COMPILE.cpp) $(OUTPUT_OPTION) $<
```

C

.o와 대응되는 .C를 발견하면

C

새로운 방식

```
.SUFFIXES : .c .o
%.o : $.c
```

오래된 방식

```
.SUFFIXES : .c .o
.c.o:
```

%.o: %.c #와일드카드 문법

```
OBJECT = zooo.c booo.c main.c
.SUFFIXED : .o .c
%.o : %.c
    $(CC) -DDEBUG -c -o$@ $<

all : run
run :  $(OBJECT)
    $(CC) -o $@ $^

clean :
    rm -rf *.o *.exe
```

make 함수 사용

SRCS = $(shell ls *.c)　　#쉘 명령의 수행

SRCS = $(wildcard *.c)　#와일드카드확장

문자열 처리 함수

$(subst 찾을 문자열, 변경할 문자열, 목표문자열)

- $(subst like , LOVE, I like you)

$(patsubst 패턴, 변경할 문자열, 목표문자열)

- OBJ = $(patsubst %.c, %.o, $(wildcard *.c))

$(매크로명: 패턴=치환문자열)

- $(OBJ:%.c=%.o)

문자열 관련 함수

$(sort $(AZ)) # 중복 문자는 하나로 통합된다.

- AZ = bbb ccc aaa ddd fff ddd

$(strip $(CC)) # ifeq ($(strip $(CC)), gcc)

- 앞뒤 공백제거 중간 공백 문자열은 한 개 공백으로

$(filter 패턴, 문자열) # $(filter %.c %.S, $(OBJECT))

- 패턴과 일치하는 문자열만 분리

$(filter-out 패턴, 문자열) # filter과 반대

문자열 관련 함수

$(findstring 찾을 문자열, 대상문자열) #찾으면 출력 아니면 null
$(words 문자열) #문자열의 단어 개수
$(word n, 문자열) #n번재 문자열
$(wordlist 시작번호, 끝번호, 문자열) #문자열내 시작~끝

$(firstword 문자열) # 첫번째 문자열
$(join 문자열1, 문자열2) # 문자열1과 문자열2을 순서쌍으로

C

파일 관련 함수

$(dir , 문자열)　　#문자열에서 디렉토리만 추출

- PATH = /usr/bin

$(notdir 패턴, 문자열)　　#디렉토리 아닌것만

- PATH = /usr/bin

$(SUFFIX 문자열)　　# 확장자만 출력

- PATH = /usr/bin/config.c

$(addsuffix 접미사, 문자열)　　#각 문자열 뒤에 접미사 붙여줌

C

와일드카드 매칭, 대입 참조

```
SRC = $(wildcard *.c)
OBJ = $(SRC:.c=.o)
TARGET = main

all : $(TARGET)
$(TARGET) :  $(OBJ)
    $(CC) -o $@ $^

clean :
    rm -rf *.o *.exe
```

patsubst 함수 사용

```
OBJ = $(patsubst %.c, %.o, $(wildcard *.c))
TARGET = main

all : $(TARGET)
$(TARGET) :  $(OBJ)
    $(CC) -o $@ $^

clean :
    rm -rf *.o *.exe
```

매크로 정의

```
all :
    $(TARGET)

define TARGET
@echo ----------
@echo test
@echo ----------
endef
```

C

$(subst 찾을 문자열, 변경할 문자열, 목표문자열)

```
comma:= ,
empty:=
space:= $(empty) $(empty)
foo:= a b c
bar:= $(subst $(space),$(comma),$(foo))
```

C

```
SRC:= main.c zooo.c booo.c inc.h
main: $(sources)
    cc $(filter %.c ,$(SRC)) -o main
```

```
SRC:= main.c zooo.c booo.c inc.h
main: $(sources)
    cc $(filter-out %.h ,$(SRC)) -o main
```

C

```
SRC:= main.c zooo.c booo.c inc.h
OBJ:=$(findstring nc, $(SRC))
main:
    @echo  $(OBJ)

$(dir , 문자열)                    #문자열에서 디렉토리만 추출
  $(dir bin/foo.c main.c)

$(notdir 패턴, 문자열)          #디렉토리 아닌것만
  $(notdir bin/zooo.c booo.c)
```

C

echo $?

```c
#include<stdio.h>

int main()
{
    return 0;
}
```

C

명령사용 규칙

-n 명령행의 종속관계를 출력할 뿐 실행은 되지 않는다.

-s n과 반대로 출력은 안하고 실행 만 한다.

@ make 파일 실행시 진행과정을 화면에 뿌린다. 이 때 특정 명령어만 표시되는 것을 막을 때 쓰인다.

-n 옵션으로 실행시 @ 명령은 무시된다

.SILENT : @같은 효과로 기술파일에 명시하여 실행 종속항목 수행 명령들에 에코 끄기

.IGNORE: 모드 명령어 오류 무시

$${HOME} 쉘 변수의 사용 //사용자 홈디렉토리를 의미함

특수 문자 : 표준 출력 막기 @

.SILENT :

```
CC  = gcc
INC = .
CFLAGS = -c -ansi -pedantic

run : main.o zooo.o booo.o
    @$(CC) -o run main.o zooo.o booo.o
main.o: main.c a.h
    @$(CC) -I$(INC) $(CFLAGS) main.c
zooo.o: zooo.c a.h b.h
    @$(CC) -I$(INC) $(CFLAGS) zooo.c
booo.o: booo.c b.h c.h
    @$(CC) -I$(INC) $(CFLAGS) booo.c
clean :
    rm -rf *.o *.exe
```

레이블로 사용

C

```
clean :
        rm -rf $(OBJECT)  $(TARGET).exe
```

echo 에 의하여 완료되었음을 표시한다.

```
clean :
        rm -rf *.o *.exe
        @ echo "Done"
```

특수 문자 : 결과 무시 -

```
CC  = gcc
INC = .
CFLAGS = -c -ansi -pedantic

run : main.o zooo.o booo.o
    @$(CC) -o run main.o zooo.o booo.o
main.o: main.c a.h
    @$(CC) -I$(INC) $(CFLAGS) main.c
zooo.o: zooo.c a.h b.h
    @$(CC) -I$(INC) $(CFLAGS) zooo.c
booo.o: booo.c b.h c.h
    @$(CC) -I$(INC) $(CFLAGS) booo.c
clean :
    -rm -rf *.o *.exe
```

.IGNORE:

순환 make

sub_system :

	cd sub; $(MAKE)

혹은 cd sub && $(MAKE)

혹은 make –C sub

new :

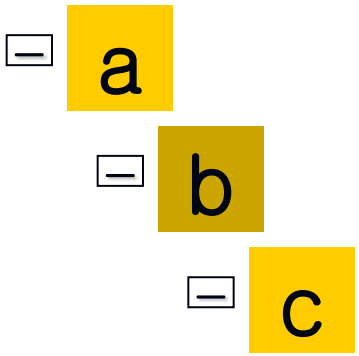
	$(MAKE) clean

	$(MAKE)

; sub 디렉토리가 없다면?

```
sub_system :
    cd  sub; $(MAKE)
    혹은
    $(MAKE) -C sub
```

예제



a폴더 파일 : Makefile zoo1.c zoo2.c zoo3.c

b폴더 파일 : Makefile(이 파일을 실행하여 나머지 폴더의 *.c 파일을 삭제한다.)

c폴더 파일 : Makefile boo1.c boo2.c boo3.c

b폴더에 위치하고 있는 Makefile을 실행 함으로써

1번 3번 Makefile의 *.c 파일을 삭제해주는
make clean 명령에 의해 실행하도록 한다.

재귀2

```
TARGET := run
    SUBDIRS := zooo booo main
    OBJS := zooo/built.o booo/built.o ₩
                        main/built.o
all : compile $(OBJS)
      $(CC) $(OBJS) -o $(TARGET)
compile:
      @for dir in $(SUBDIRS); do ₩
      make -C $$dir || exit $?;₩
      done
clean :
      @for dir in $(SUBDIRS); do ₩
      make -C $$dir clean;₩
      done
      rm -rf *.o *.i *.s $(TARGET)
```

```
TARGET := built.o
    OBJS := $(patsubst %.c, %.o, ₩
                    $(wildcard *.c))
    CFLAGS := -I../include

all : $(OBJS)
      $(LD) -r $(OBJS) -o $(TARGET)
clean :
 rm -rf *.o $(TARGET)
```

C

VPATH 매크로

VPATH = c:/bin c:/include

pattern 상관없이 현재 디렉토리와 등록된 디렉토리 까지 찾는다.

재귀2와는 다르게 파일명에 종속적이다.

파일명(앞의 built.o처럼)이 같아서는 안된다.

실행

```
VPATH = zooo main booo
OBJS = zooo.o main.o booo.o
CFLAGS = -I./include

run : $(OBJS)
	$(CC) -o $@ $^
clean :
	rm -rf *.o run
```

서브 Makefile에 매크로 전달

```
TARGET := run
SUBDIRS := zooo booo main
OBJS := zooo/built.o booo/built.o ₩
                    main/built.o
export CC := gcc
all : compile $(OBJS)
      $(CC) $(OBJS) -o $(TARGET)
compile:
      @for dir in $(SUBDIRS); do ₩
      make -C $$dir || exit $?;₩
      done
clean :
      @for dir in $(SUBDIRS); do ₩
      make -C $$dir clean;₩
      done
      rm -rf *.o *.i *.s $(TARGET)
```

```
TARGET := built.o
    OBJS := $(patsubst %.c, %.o, ₩
                    $(wildcard *.c))
    CFLAGS := -I../include

all : $(OBJS)
      $(LD) -r $(OBJS) -o $(TARGET)
clean :
 rm -rf *.o $(TARGET)
```

조건부 수행

```
ifeq ($(CC), gcc)      ←→ ifneq
    @echo “GNU gcc”
else
    @echo “$(CC)”
endif

ifdef CC #CC가 정의 되었을 때 ←=> ifndef
    @echo “GNU gcc”
else
    @echo “$(CC)”
endif
```

1. 최상위 Makefile

```
.EXPORT_ALL_VARIABLES:
TARGET          := run
TOPDIR          := $(shell /bin/pwd)
SUBDIRS         := main zooo booo
##############################
include $(TOPDIR)/Config.mk
all : compile $(OBJS)
    $(CC) $(OBJS) $(addsuffix /built.o, $(SUBDIRS)) -o $(TARGET)
include $(TOPDIR)/Rules.mk
```

C

2. 최상위 Config.mk

```
CC       := gcc
LD       := ld
INCLUDES :=    -I. -I$(TOPDIR) /include
DEFINES          := -DDEBUG
CFLAGS           := -O2 -W -Wall $(INCLUDES) $(DEFINES)
############################
.SUFFIXED : .o .c .S
%.o : %.c
        $(CC) $(CFLAGS) -c $< -o $@
%.o : %.S
        $(CC) $(CFLAGS) -c $< -o $@
############################
```

3. 최상위 Rules.mk

```
compile:
	@for dir in $(SUBDIRS); do ₩
	make -C $$dir || exit $?;₩
	 done
clean :
	@for dir in $(SUBDIRS); do ₩
	make -C $$dir clean;₩
	done
	rm -rf *.o *.i *.s $(TARGET)
```

4. 각 서브디렉토리 main zooo booo

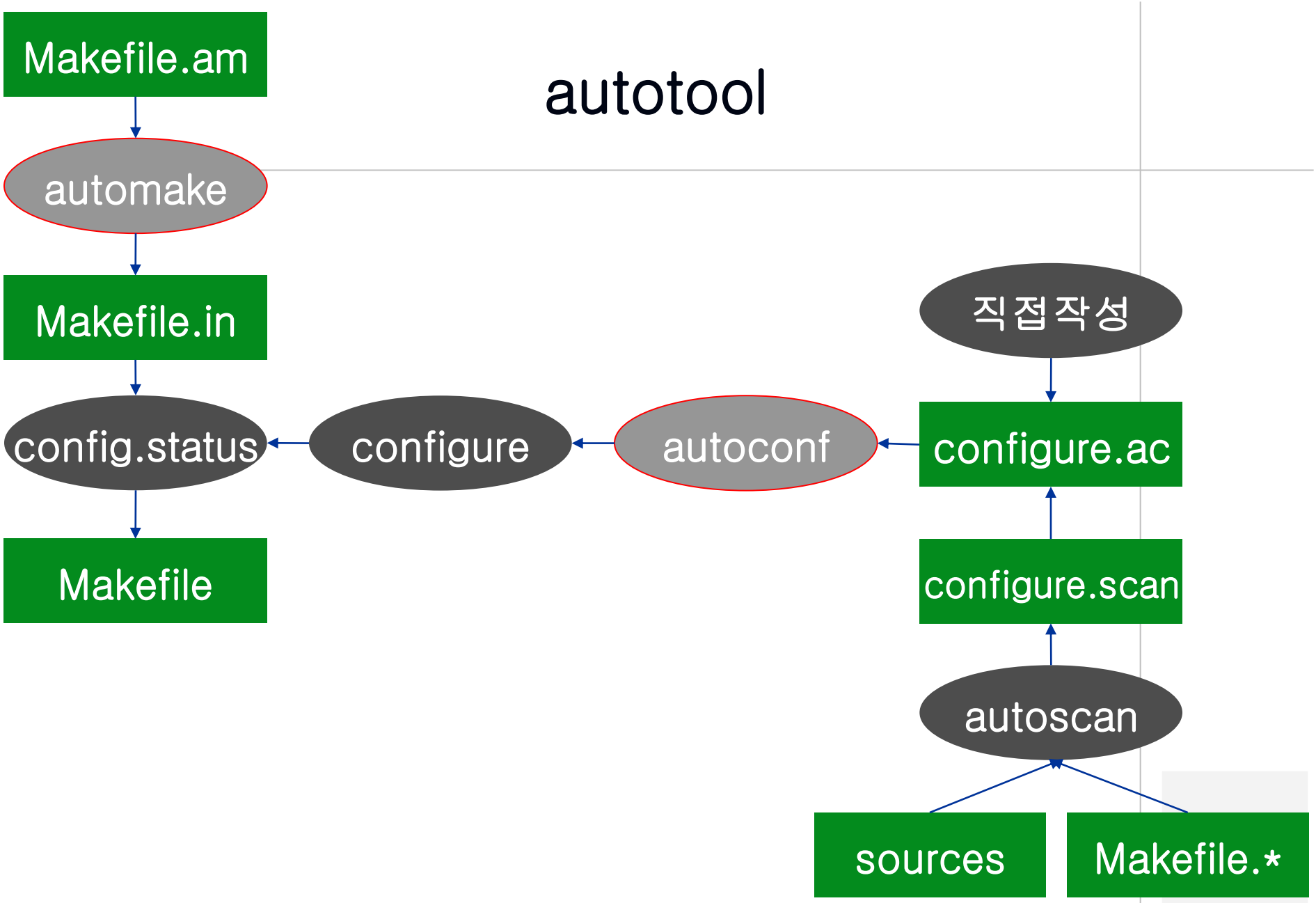
```
ifndef TOPDIR
TODIR := ..
endif

include $(TOPDIR)/Config.mk

TARGET := built.o
SUBDIRS :=
OBJS := $(patsubst %.c, %.o,  $(wildcard *.c))
OBJS += $(patsubst %.S, %.o,  $(wildcard *.c))
all : compile $(OBJS)
    $(LD) -R  $(addsuffix / $(TARGET), $(SUBDIRS))  $(OBJS) -o $(TARGET)
include $(TOPDIR)/Rules.mk
```

C

autotool


Makefile.am
automake
Makefile.in
config.status
configure
autoconf
직접작성
configure.ac
Makefile
configure.scan
autoscan
sources
Makefile.*

autoconf

1. configure.ac 작성
2. autoconf 명령 (configure.ac -> configure생성)
3. Makefile.in 작성
4. ./configure 명령 (configure.ac 으 내용을 바탕으로 정보수집 하여 Makefile.in -> Makefile을 생성
5. make

/* configure.ac */

AC_INIT(show, 1.2.1, win.pros@daum.net)

AC_PROG_CC

AC_CHECK_LIB([m], [sin])

AC_CHECK_HEADERS([libintl.h])

AC_CHECK_FUNCS([setlocale])

AC_CONFIG_FILES([Makefile])
AC_OUTPUT

/* Makefile.in */

```
CC =@CC@
CFLAGS = @CFLAGS@



run : zooo.c booo.c main.c
        $(CC) $(CFLAGS) -o $@ $^

clean :
        @rm -rf *.o run.exe configure
config.status config.log
autom4te.cache
```

C

configure.ac

AC_INIT(show, 1.2.1, win.pros@daum.net)

AC_PROG_CC #c 컴파일러와 CFLAGS 옵션 체크

AC_CHECK_LIB([m], [sin])#libm.a 라이브러리 유무 체크

AC_CHECK_HEADERS([libintl.h])# 헤더파일 유무체크

AC_CHECK_FUNCS([setlocale])#setlocale 함수 유무체크

AC_CONFIG_FILES([Makefile])# ./configure 스크립트싱행후 생성 파일 지정

AC_OUTPUT

automake

1. configure.ac 작성

   AM_INIT_AUTOMAKE(show,[1.2.1]) #automake 사용알림

2. Makefile.am 작성
3. aclocal (configure.ac 로 부터 aclocal.m4 생성)
4. autoconf 명령 (configure.ac -> configure생성)
5. automake --foreign --add-missing --copy
   (Makefile.am 으로부터 Makefile.in 생성)
6. ./configure 명령 (configure.ac 으 내용을 바탕으로 정보수집 하여 Makefile.in -> Makefile을 생성
7. make

/* configure.ac */

AC_INIT(show, 1.2.1, win.pros@daum.net)

AM_INIT_AUTOMAKE(main,[0.0.1]) #append code

#AC_CONFIG_HEADER([inc.h])
#AC_PROG_MAKE_SET
#AC_PROG_RANLIB

AC_PROG_CC

AC_CHECK_LIB([m], [sin])

AC_CHECK_HEADERS([libintl.h inc.h])

AC_CHECK_FUNCS([setlocale])

AC_CONFIG_FILES([Makefile])
AC_OUTPUT

/* Makefile.am */

```
CFLAGS= @CFLAGS@ -I../ -I./
#SUBDIRS          = main zoo boo
bin_PROGRAMS = main
main_SORUCES     = inc.h main.c zooo.c booo.c

clean:
        rm -rf *.o *.a main.exe
```

C

다중 타겟

어플리케이션 install 시

1. enterprise
2. standard
3. personal
4. customer

과 같은 형태의 install을 선택할 수 있다.

main 함수에서 함수를 호출할 때

함수의 이름과 형태는 같지만

내용이 다를 수 있다.

이를 위한 다중 타겟 make파일과 각 함수들을
작성하세요.

예) make standard 일 경우 "install standard" 출력

C

라이브러리

정적 라이브러리(static library)

1. 마지막 실행파일에 라이브러리의 복사본 포함
2. .a(archive) .sa(static a)
3. 완전한 프로그램
4. 속도향상(참조결정시간비포함)
5. 크기증가, 재컴파일

공유 라이브러리(shared library)

동시접근가능

.so(shared object)

외부환경의존도

루틴의 참조시간 필요

동적 적재 라이브러리(dynamically loaded library)

C